

TOWARDS SAFETY & SECURITY IN C++

C-00000291*.sys

SOFTWARE VULNERABILITIES

- security and safety
- safety in our code,
or lack thereof
- existing solutions for
improvement
- new features in C++26
- options for the future



ABOUT ME

- Electrical engineer
- I build computers and create software for more than 40 years
- I develop hardware and software in the field of applied digital signal processing for more than 35 years
- I'm a member of the C++ committee (learning novice) for 5 years (EWG, SG15)



SOFTWARE VULNERABILITIES

- About 70 percent of Microsoft common vulnerabilities and exposures (CVEs) are memory safety vulnerabilities (based on 2006-2018 CVEs).⁸
- About 70 percent of vulnerabilities identified in Google's Chromium project are memory safety vulnerabilities.⁹
- In an analysis of Mozilla vulnerabilities, 32 of 34 critical/high bugs were memory safety vulnerabilities.¹⁰
- Based on analysis by Google's Project Zero team, 67 percent of zero-day vulnerabilities in 2021 were memory safety vulnerabilities.¹¹

The Case for Memory Safe Roadmaps, 2023

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	56.92	3	+1
2	CWE-787	Out-of-bounds Write	45.20	18	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	35.88	4	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	19.57	0	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	12.74	4	+3
6	CWE-125	Out-of-bounds Read	11.42	3	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.30	5	-2
8	CWE-416	Use After Free	10.19	5	-4
9	CWE-862	Missing Authorization	10.11	0	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.03	0	0
11	CWE-94	Improper Control of Generation of Code ('Code Injection')	7.13	7	+12
12	CWE-20	Improper Input Validation	6.78	1	-6
13	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	6.74	4	+3
14	CWE-287	Improper Authentication	5.94	4	-1
15	CWE-269	Improper Privilege Management	5.22	0	+7
16	CWE-502	Deserialization of Untrusted Data	5.07	5	-1
17	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	5.07	0	+13
18	CWE-863	Incorrect Authorization	4.05	2	+6
19	CWE-918	Server-Side Request Forgery (SSRF)	4.05	2	0
20	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	3.69	2	-3
21	CWE-476	NULL Pointer Dereference	3.58	0	-9
22	CWE-798	Use of Hard-coded Credentials	3.46	2	-4
23	CWE-190	Integer Overflow or Wraparound	3.37	3	-9
24	CWE-400	Uncontrolled Resource Consumption	3.23	0	+13
25	CWE-306	Missing Authentication for Critical Function	2.73	5	-5

Rank	ID	Name	Score	CVEs in KEV	Rank Change vs. 2023
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	56.92	3	+1
2	CWE-787	Out-of-bounds Write	45.20	18	-1
3	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	35.88	4	0
4	CWE-352	Cross-Site Request Forgery (CSRF)	19.57	0	+5
5	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	12.74	4	+3
6	CWE-125	Out-of-bounds Read	11.42	3	+1
7	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	11.30	5	-2
8	CWE-416	Use After Free	10.19	5	-4
9	CWE-862	Missing Authorization	10.11	0	+2
10	CWE-434	Unrestricted Upload of File with Dangerous Type	10.03	0	0
11	CWE-94	Improper Control of Generation of Code ('Code Injection')	7.13	7	+12
12	CWE-20	Improper Input Validation	6.78	1	-6
13	CWE-77	Improper Neutralization of Special Elements used in a Command ('Command Injection')	6.74	4	+3
14	CWE-287	Improper Authentication	5.94	4	-1
15	CWE-269	Improper Privilege Management	5.22	0	+7
16	CWE-502	Deserialization of Untrusted Data	5.07	5	-1
17	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	5.07	0	+13
18	CWE-863	Incorrect Authorization	4.05	2	+6
19	CWE-918	Server-Side Request Forgery (SSRF)	4.05	2	0
20	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	3.69	2	-3
21	CWE-476	NULL Pointer Dereference	3.58	0	-9
22	CWE-798	Use of Hard-coded Credentials	3.46	2	-4
23	CWE-190	Integer Overflow or Wraparound	3.37	3	-9
24	CWE-400	Uncontrolled Resource Consumption	3.23	0	+13
25	CWE-306	Missing Authentication for Critical Function	2.73	5	-5





now
@pony.social

memory unsafety may cause a lot of software crashes, but it also helps a lot of people jailbreak their game consoles, so, it;s impossible to say if its bad or not,

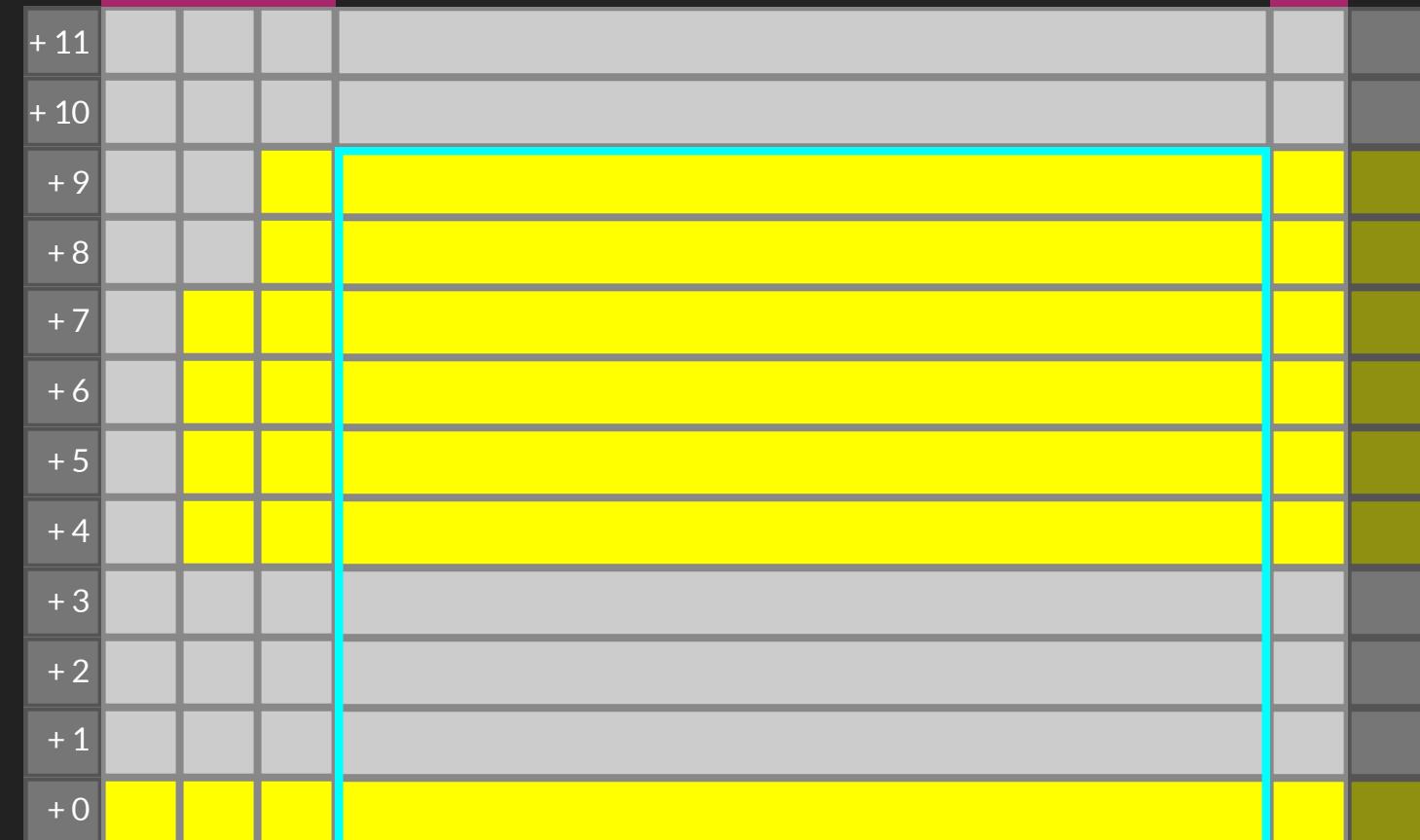
OH MY, ALL THOSE 'SAFETIES' !

- Language Safety
- Spatial Safety
- Temporal Safety
- Type Safety
- Bounds Safety
- Memory Safety
- Lifetime Safety
- Thread Safety
- Functional Safety
- Occupational Safety
- Traffic/Road Safety
- Industrial Safety
- Undefined Behaviour
- Correctness
- Regulation
- Security
- etc...

DEFINITIONS

spatial

(over-)allocated memory region exists

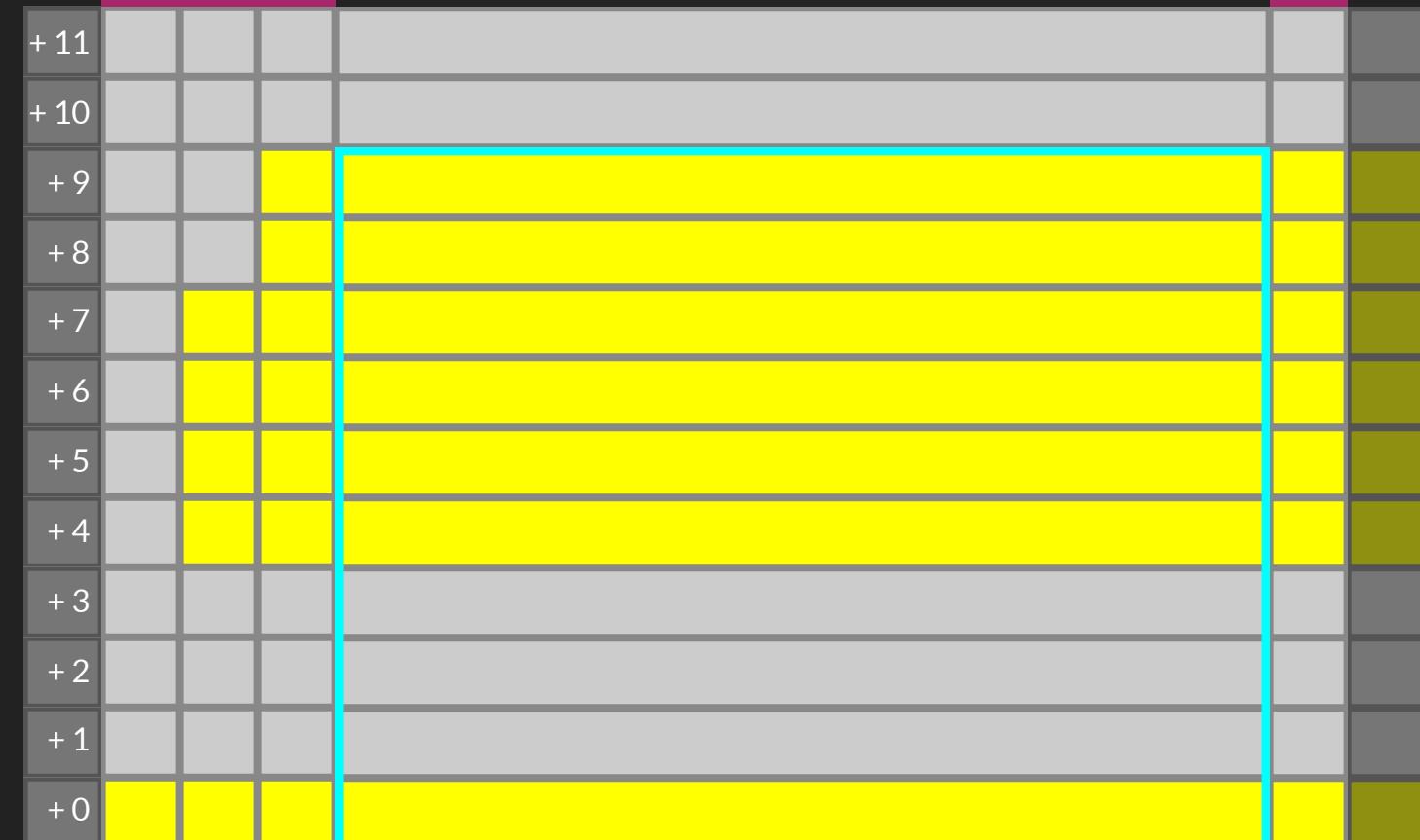


```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

temporal

spatial

(over-)allocated memory region exists



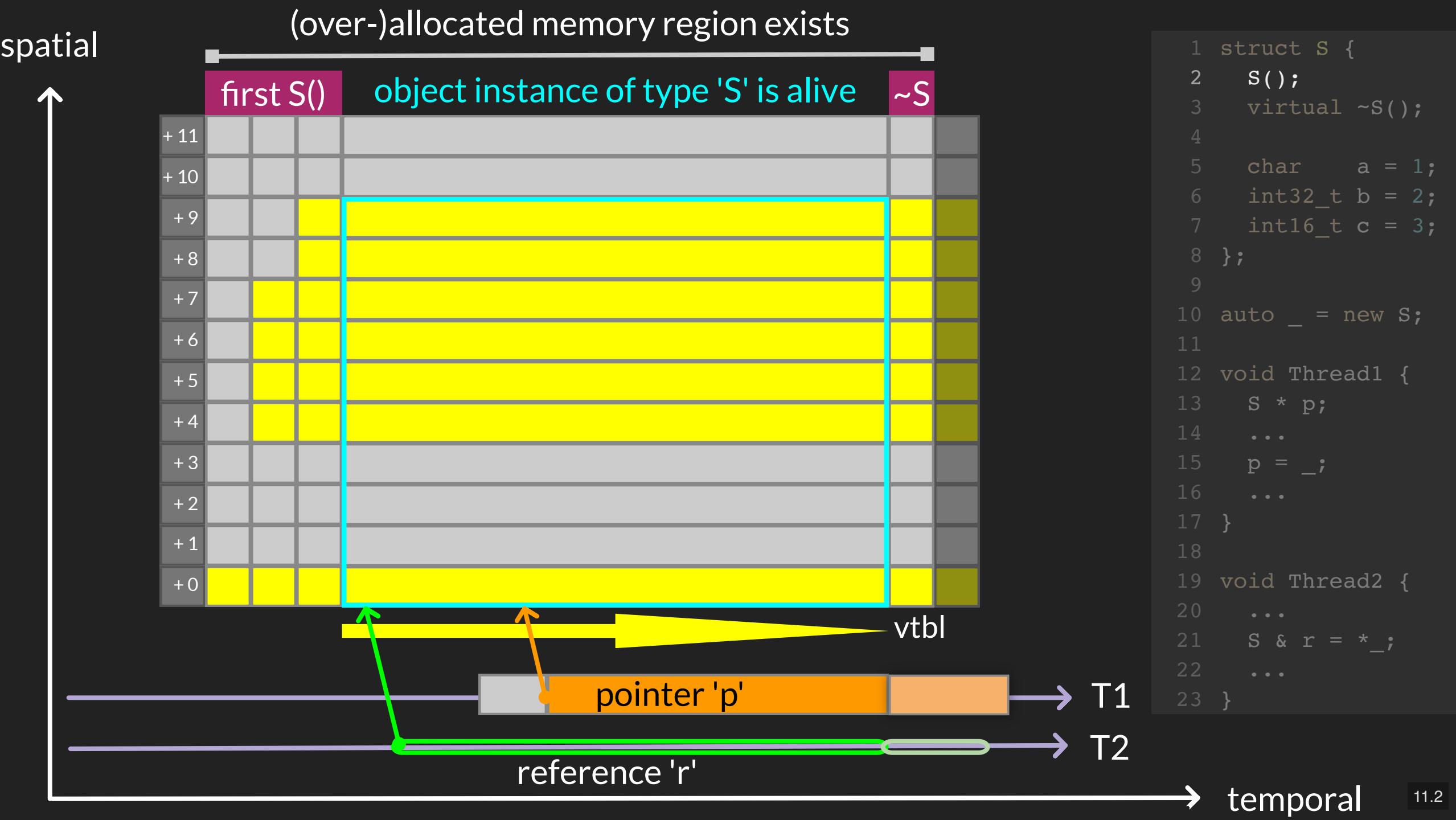
```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

reference 'r'

T1

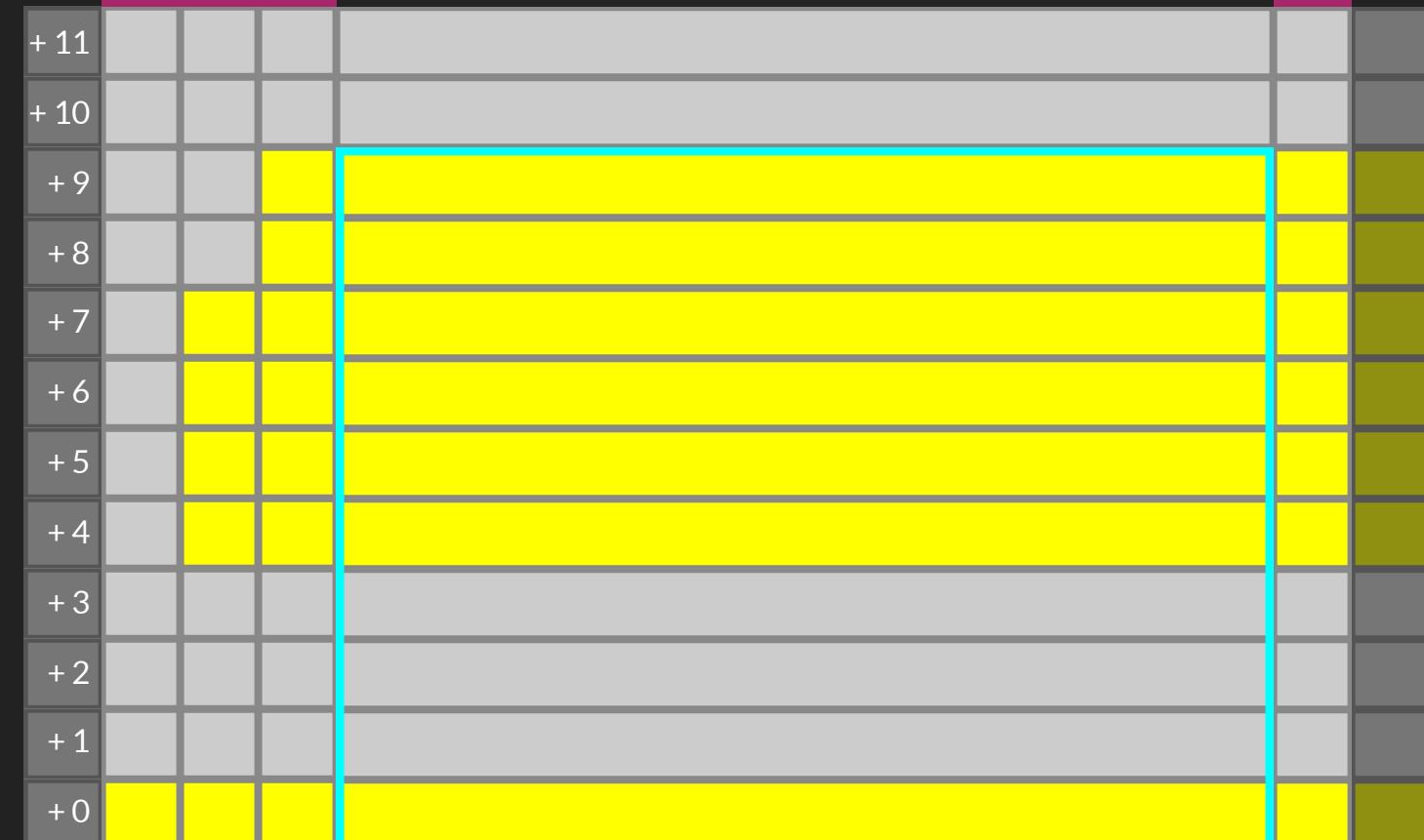
T2

temporal



spatial

(over-)allocated memory region exists



```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

reference 'r'

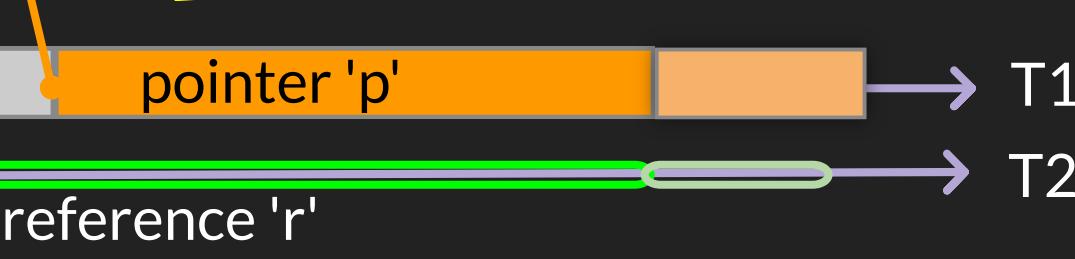
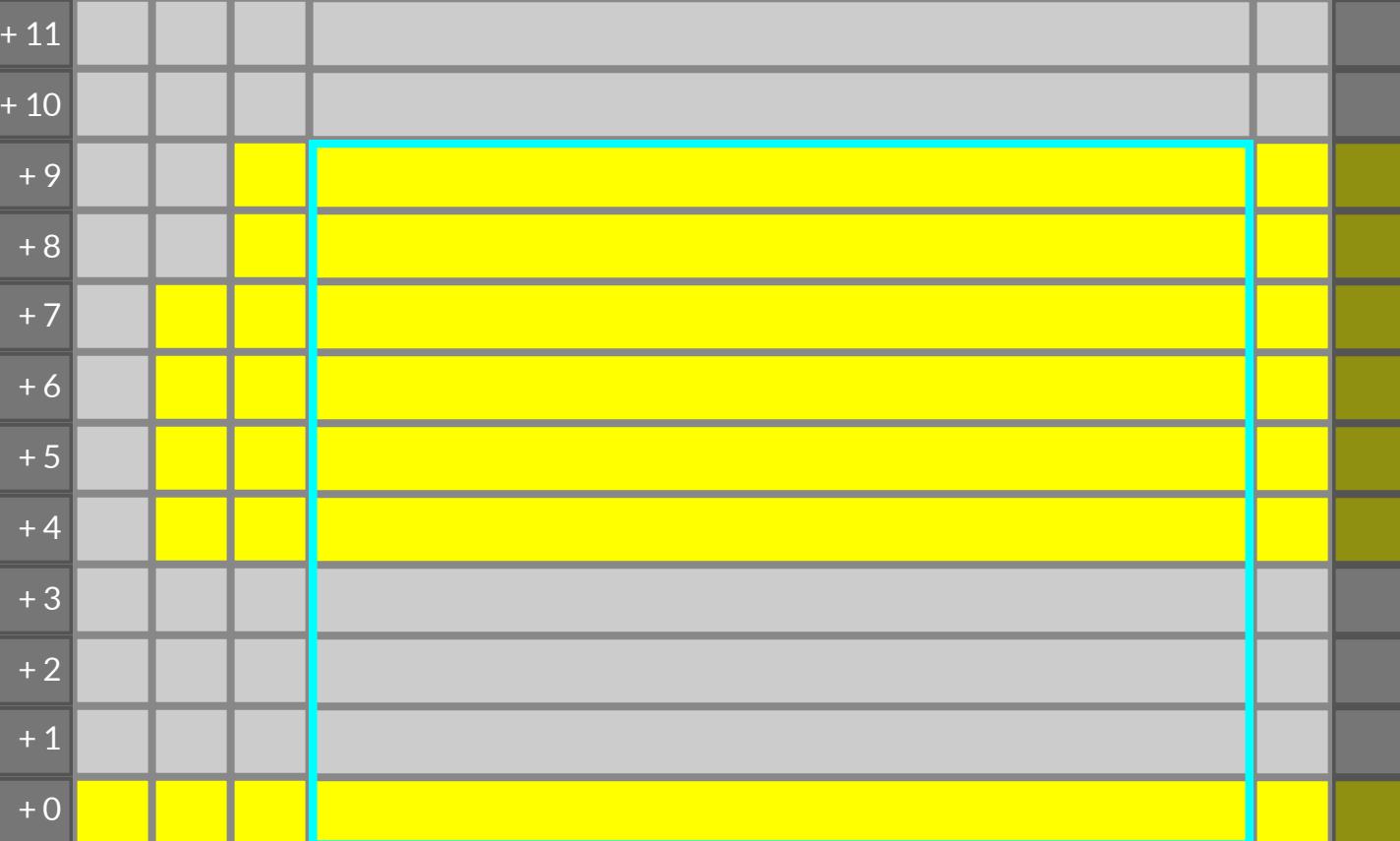
T1

T2

temporal

spatial

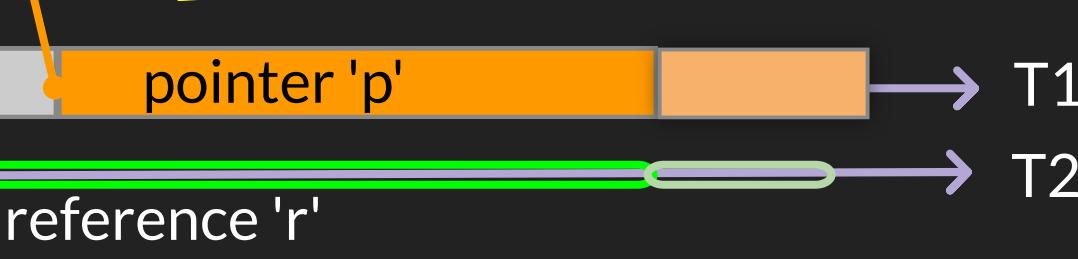
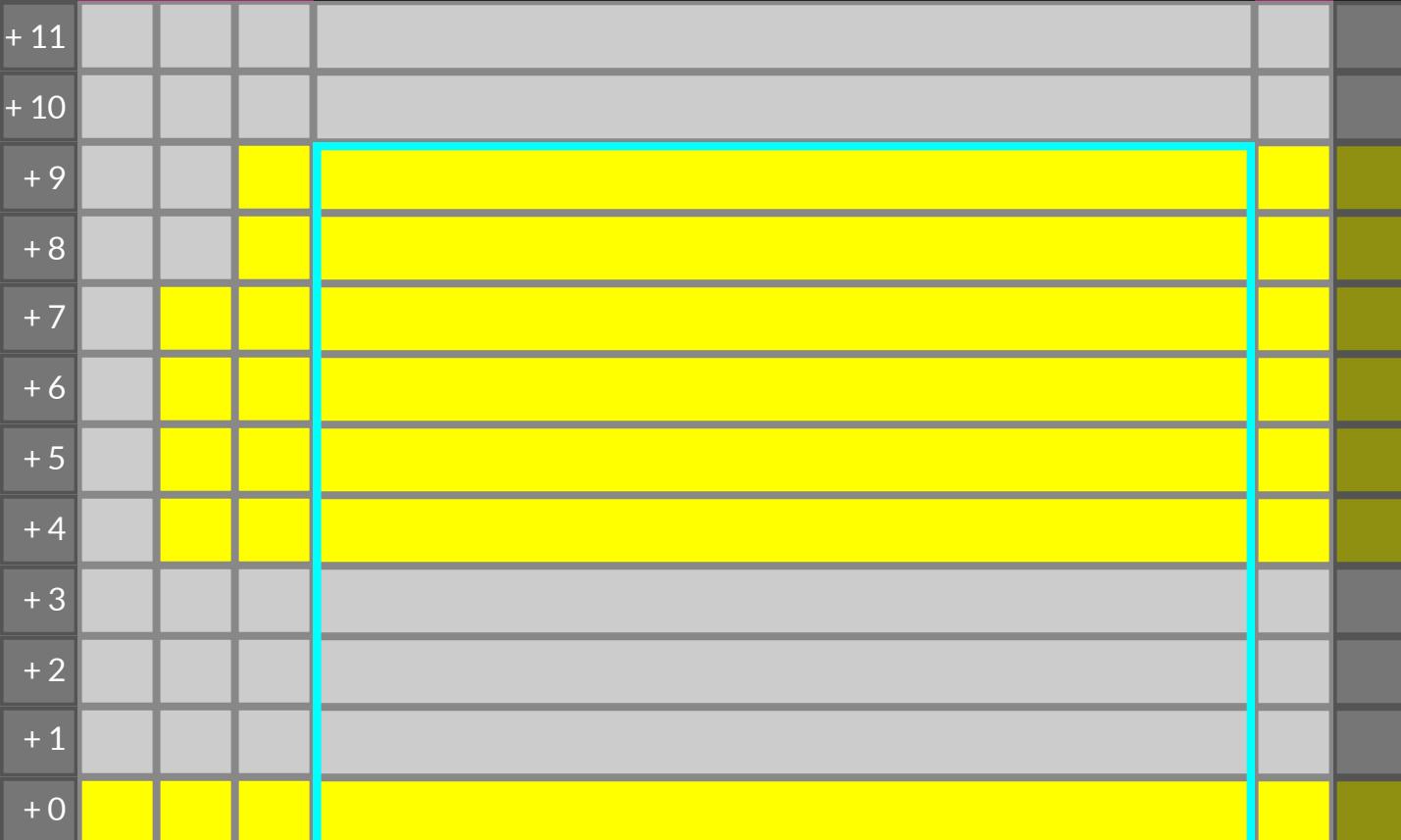
(over-)allocated memory region exists



```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

spatial

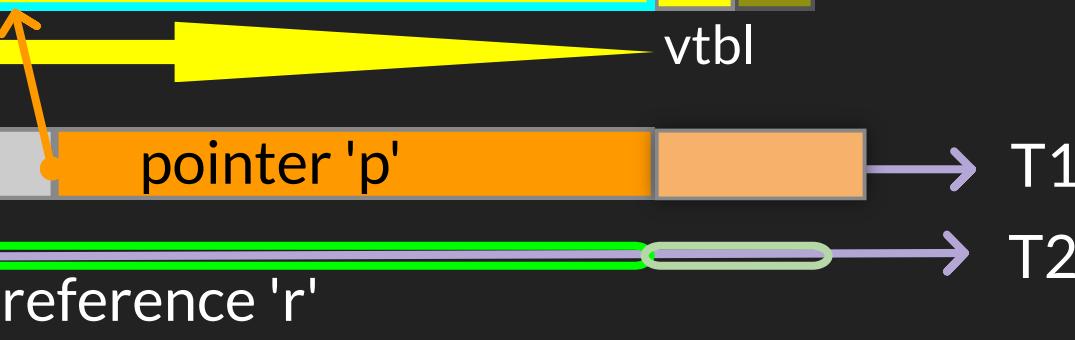
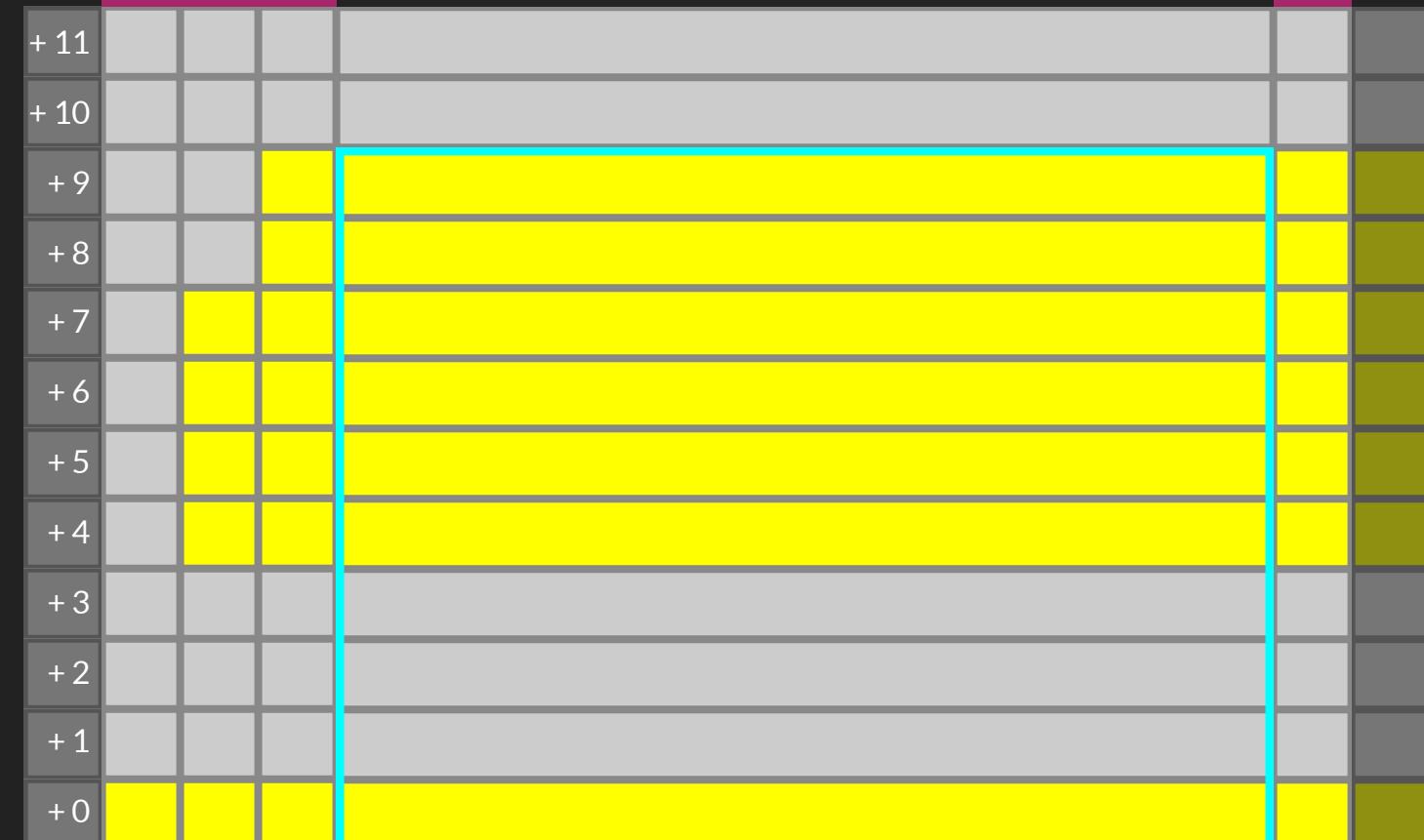
(over-)allocated memory region exists



```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

spatial

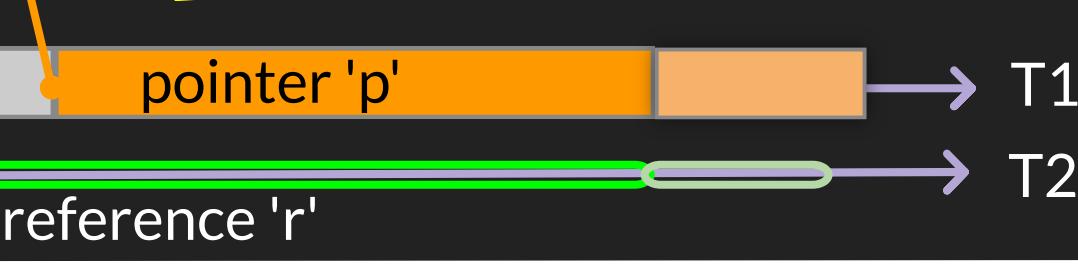
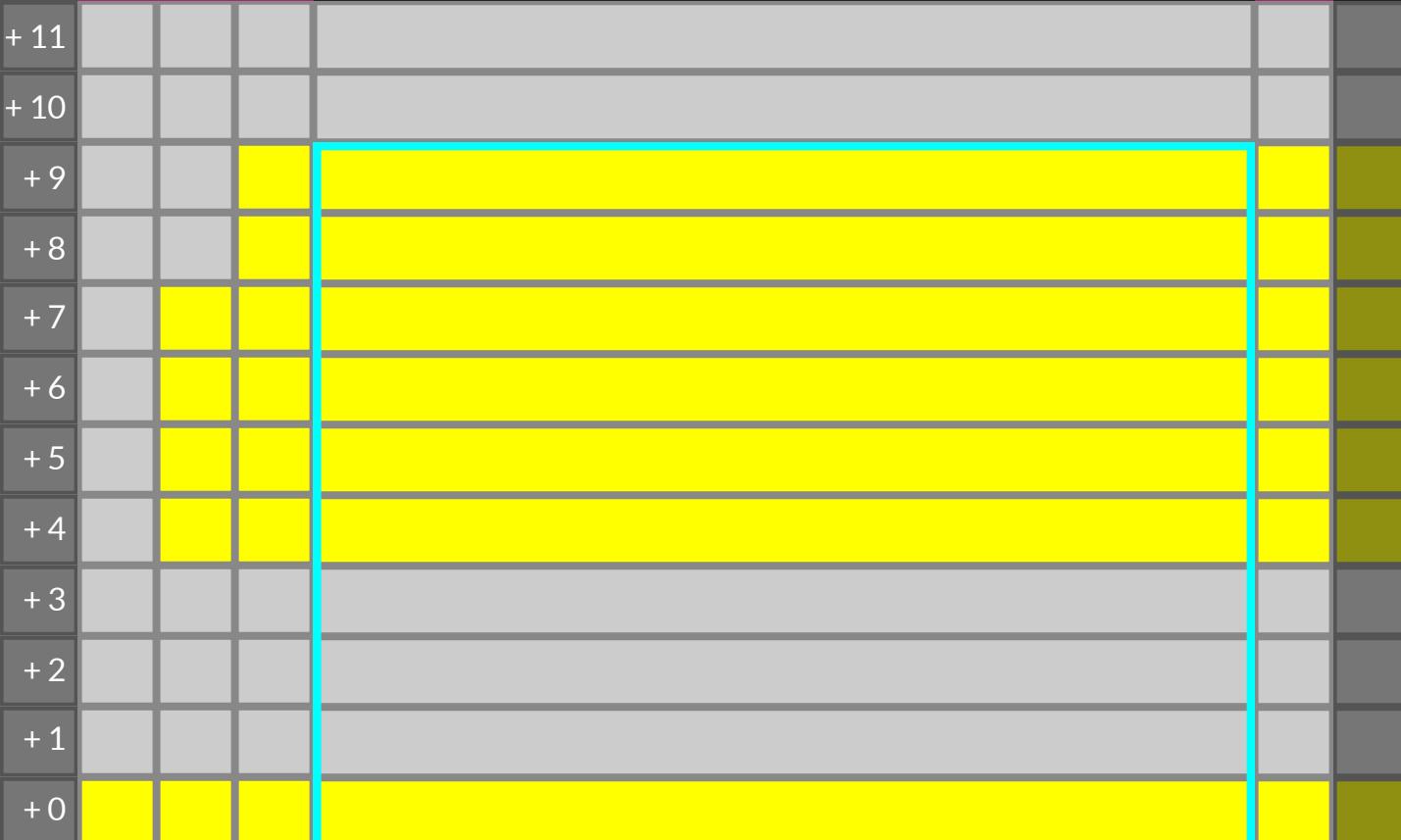
(over-)allocated memory region exists



```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

spatial

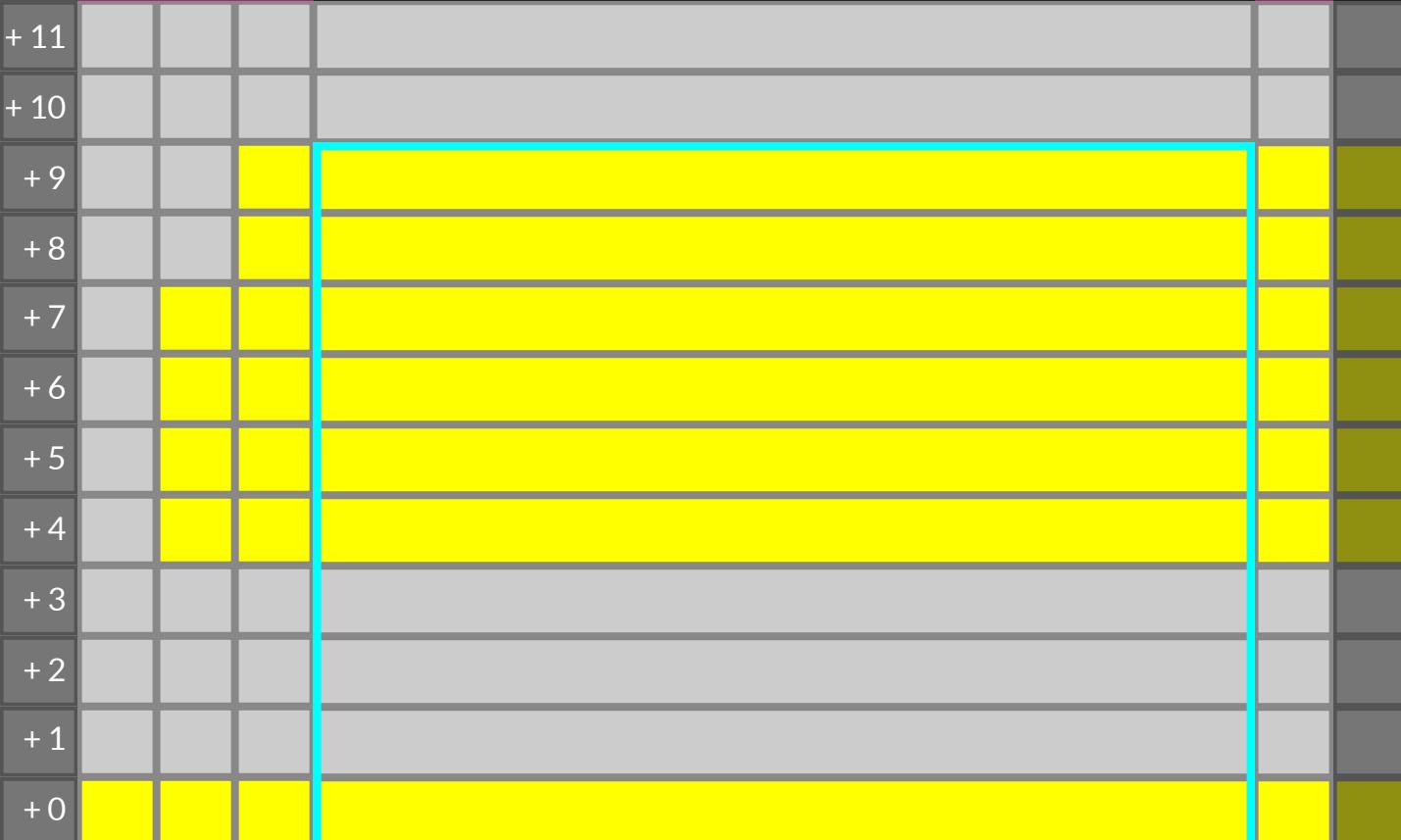
(over-)allocated memory region exists



```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

spatial

(over-)allocated memory region exists

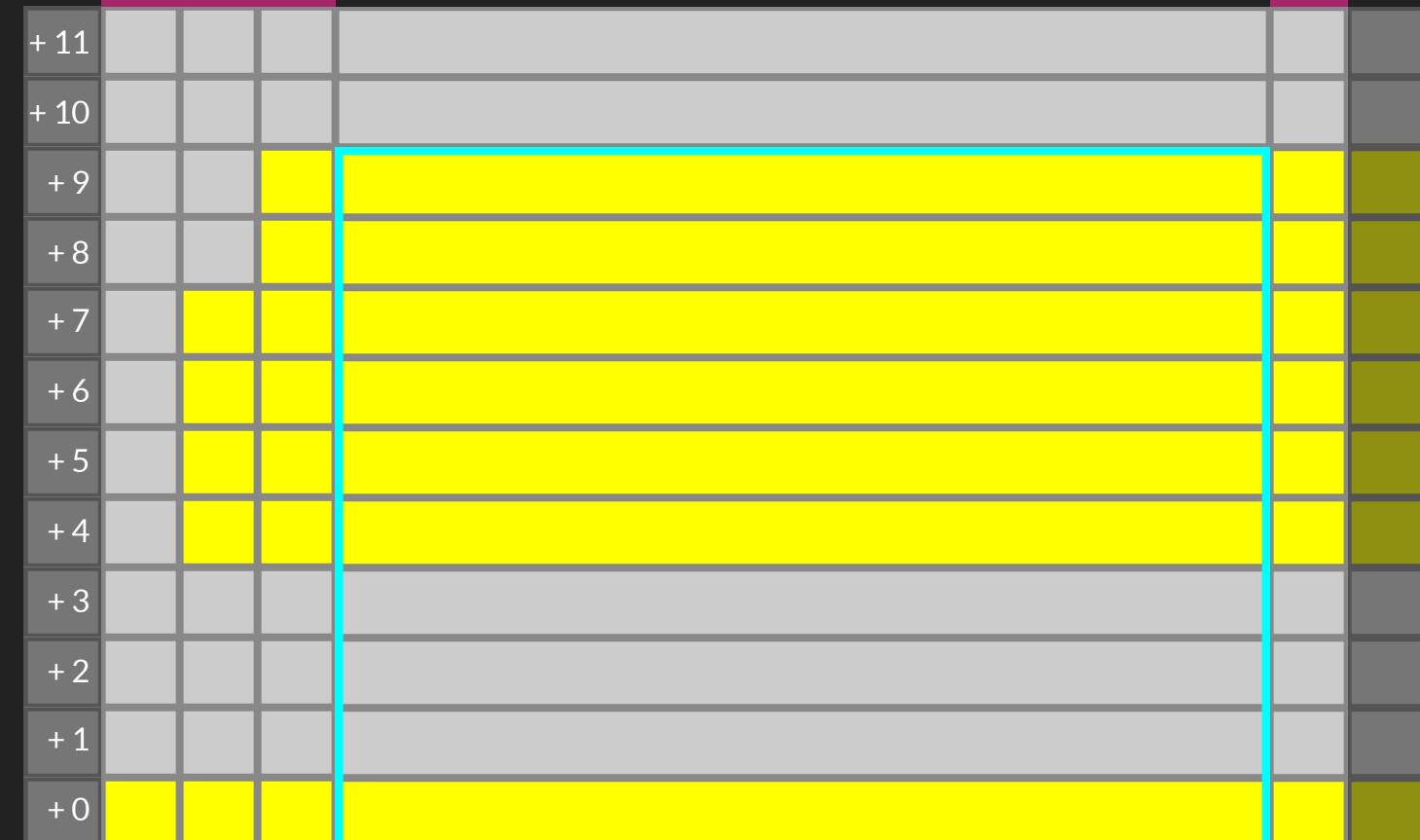


vtbl
pointer 'p'
reference 'r'
T1
T2
temporal

```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

spatial

(over-)allocated memory region exists

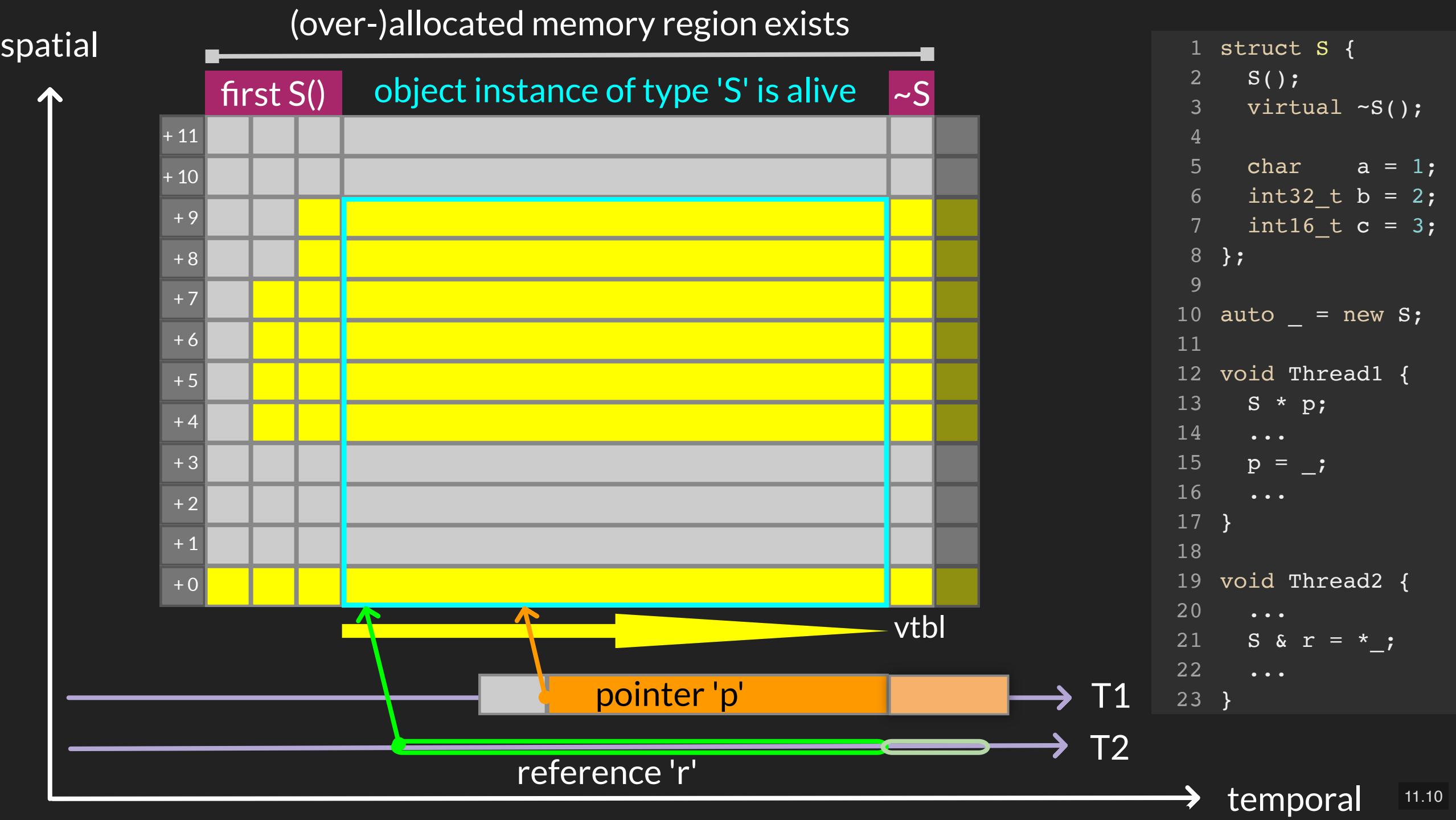


```
1 struct S {  
2     S();  
3     virtual ~S();  
4  
5     char    a = 1;  
6     int32_t b = 2;  
7     int16_t c = 3;  
8 };  
9  
10 auto _ = new S;  
11  
12 void Thread1 {  
13     S * p;  
14     ...  
15     p = _;  
16     ...  
17 }  
18  
19 void Thread2 {  
20     ...  
21     S & r = *_;  
22     ...  
23 }
```

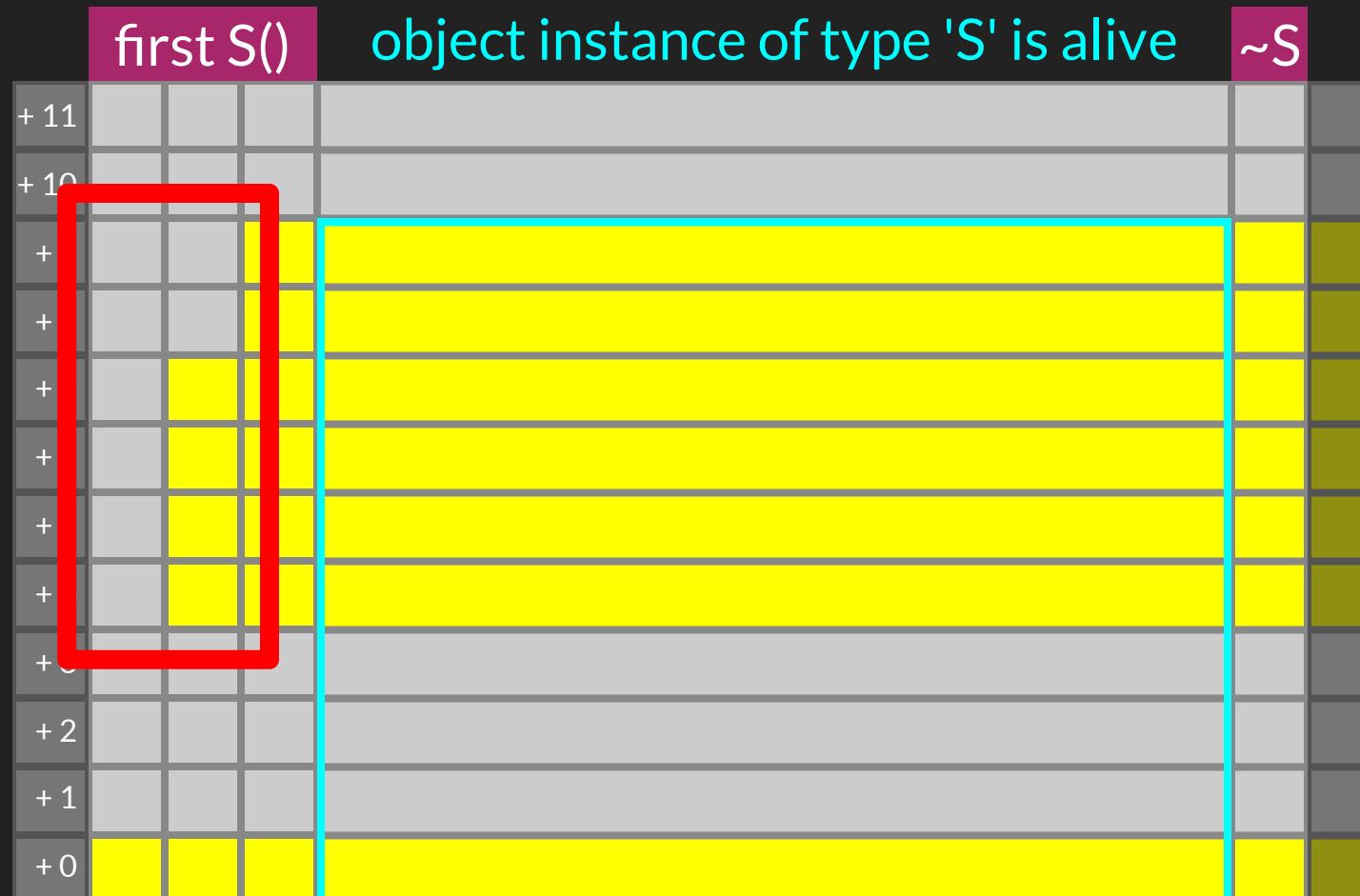
reference 'r'

T1
T2

temporal



spatial



LANGUAGE-SAFETY

reference 'r'

pointer 'p'

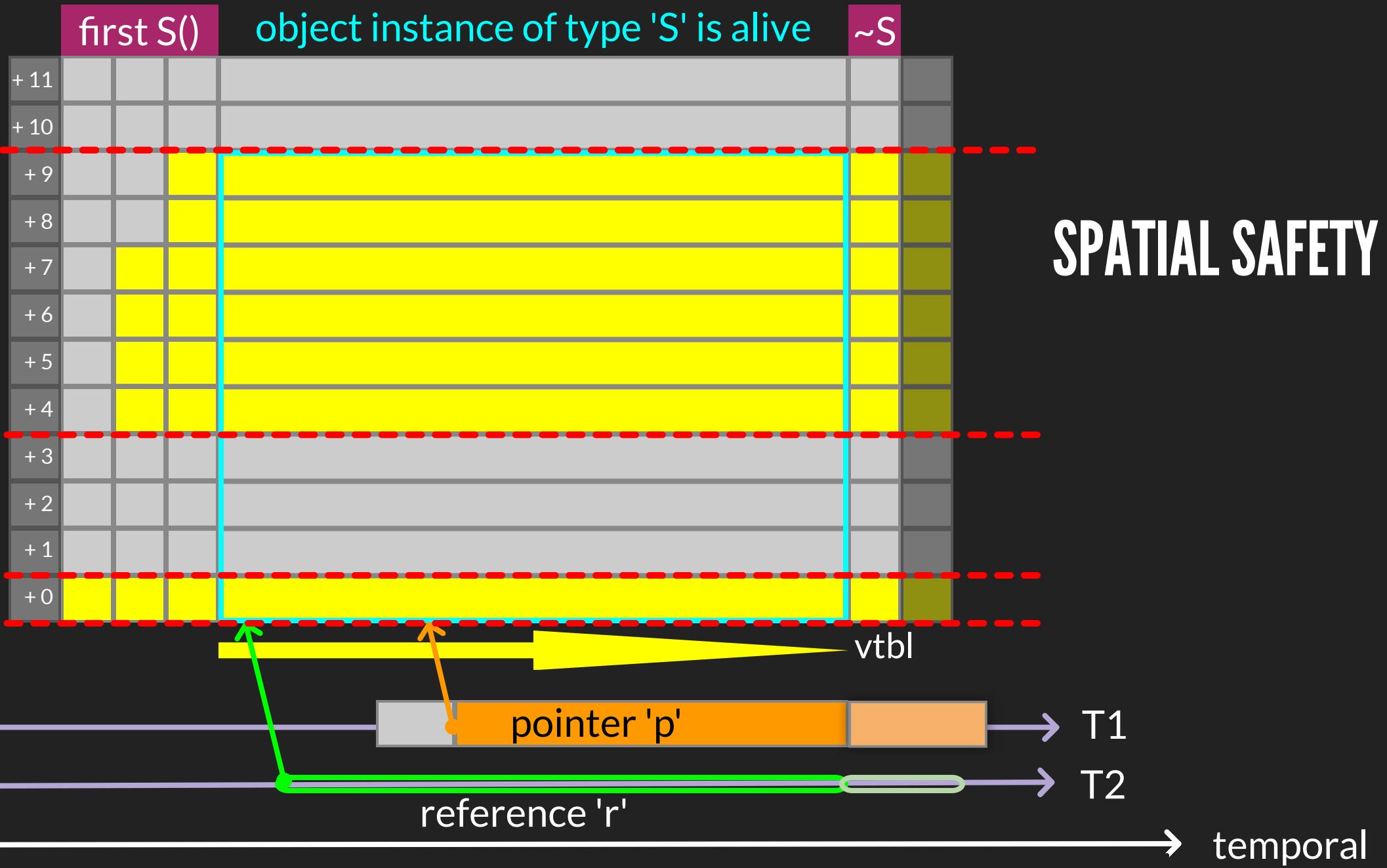
vtbl

T1

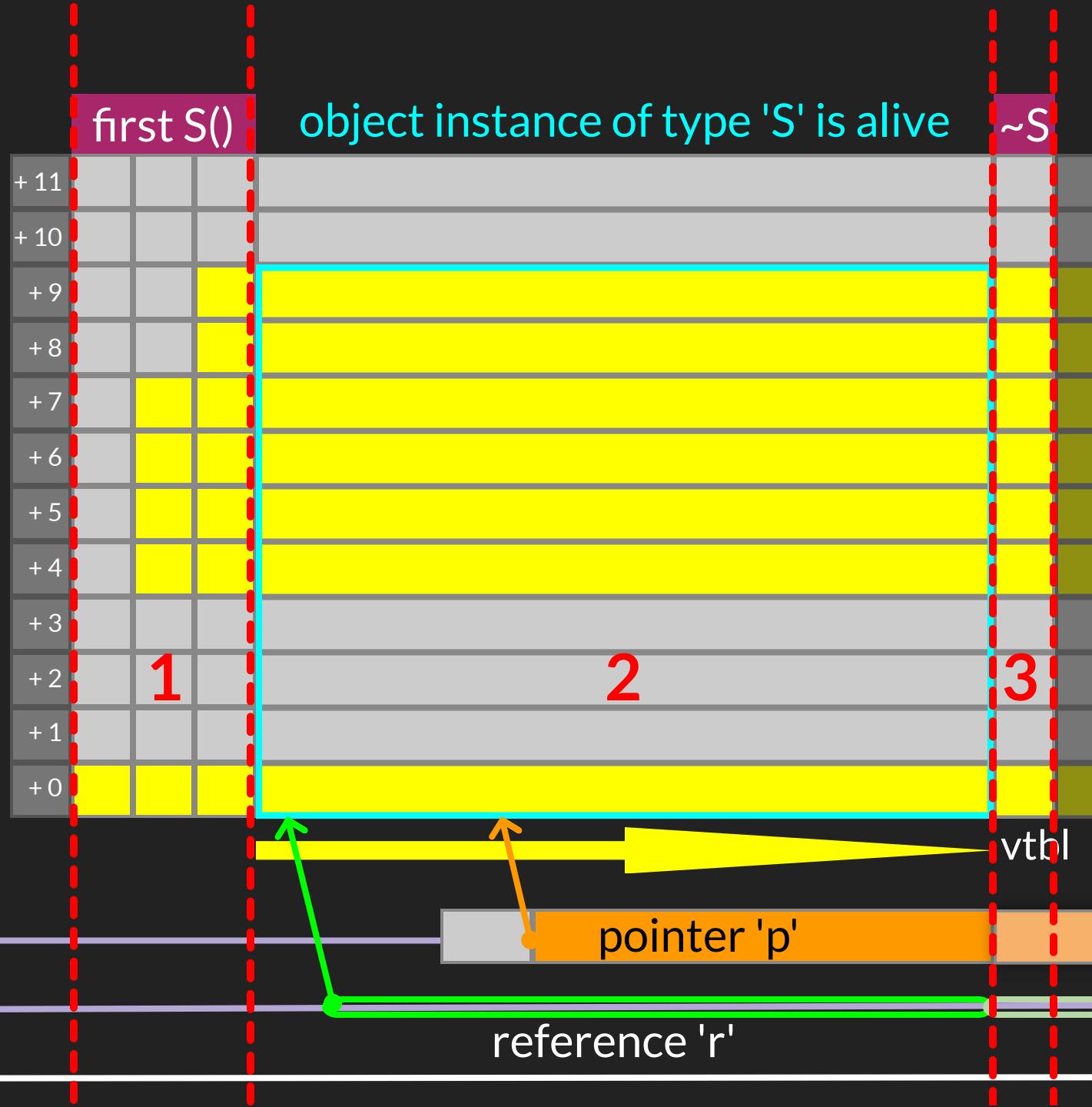
T2

temporal

spatial



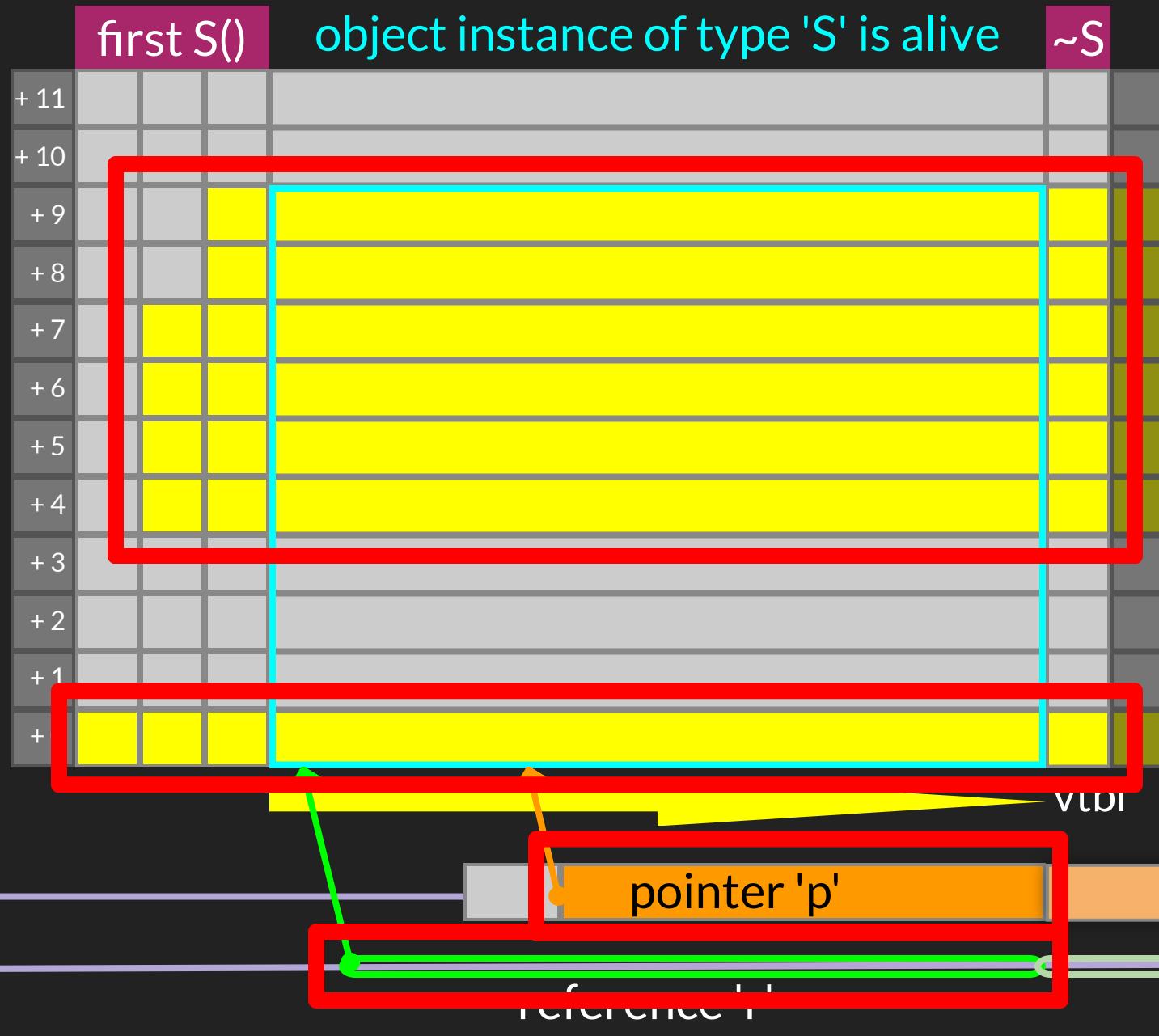
spatial

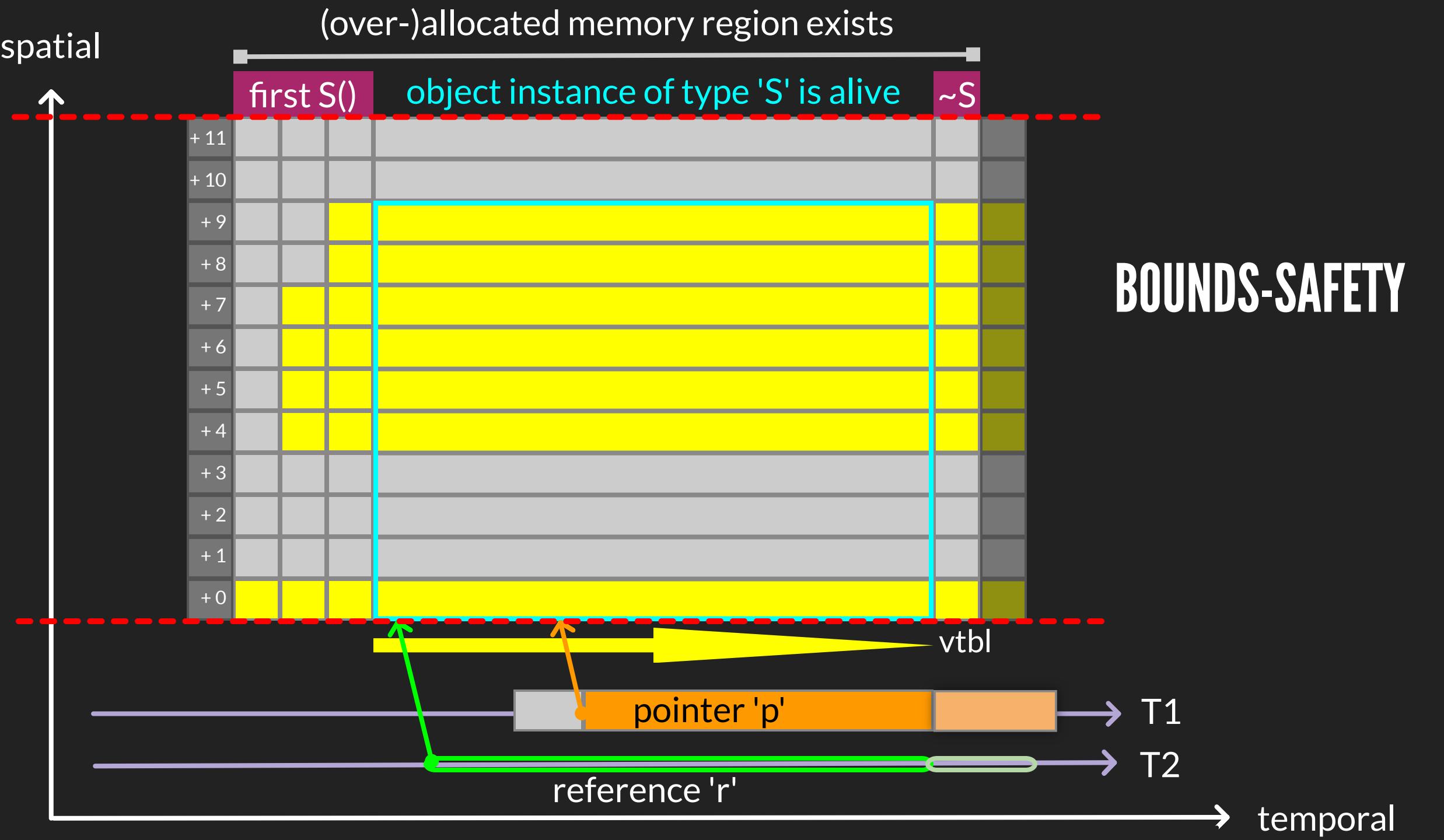


TEMPORAL SAFETY

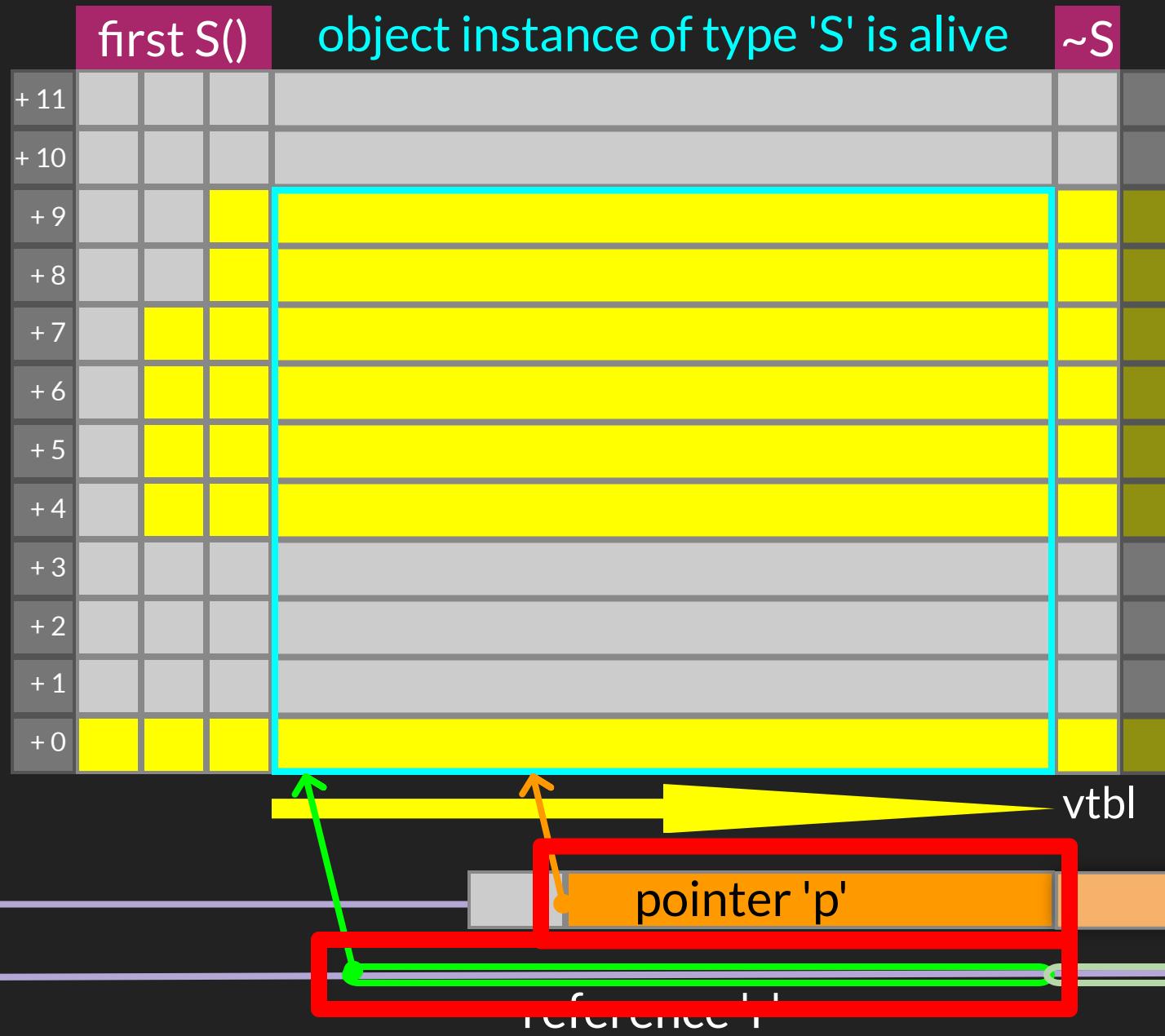
spatial

TYPE-SAFETY



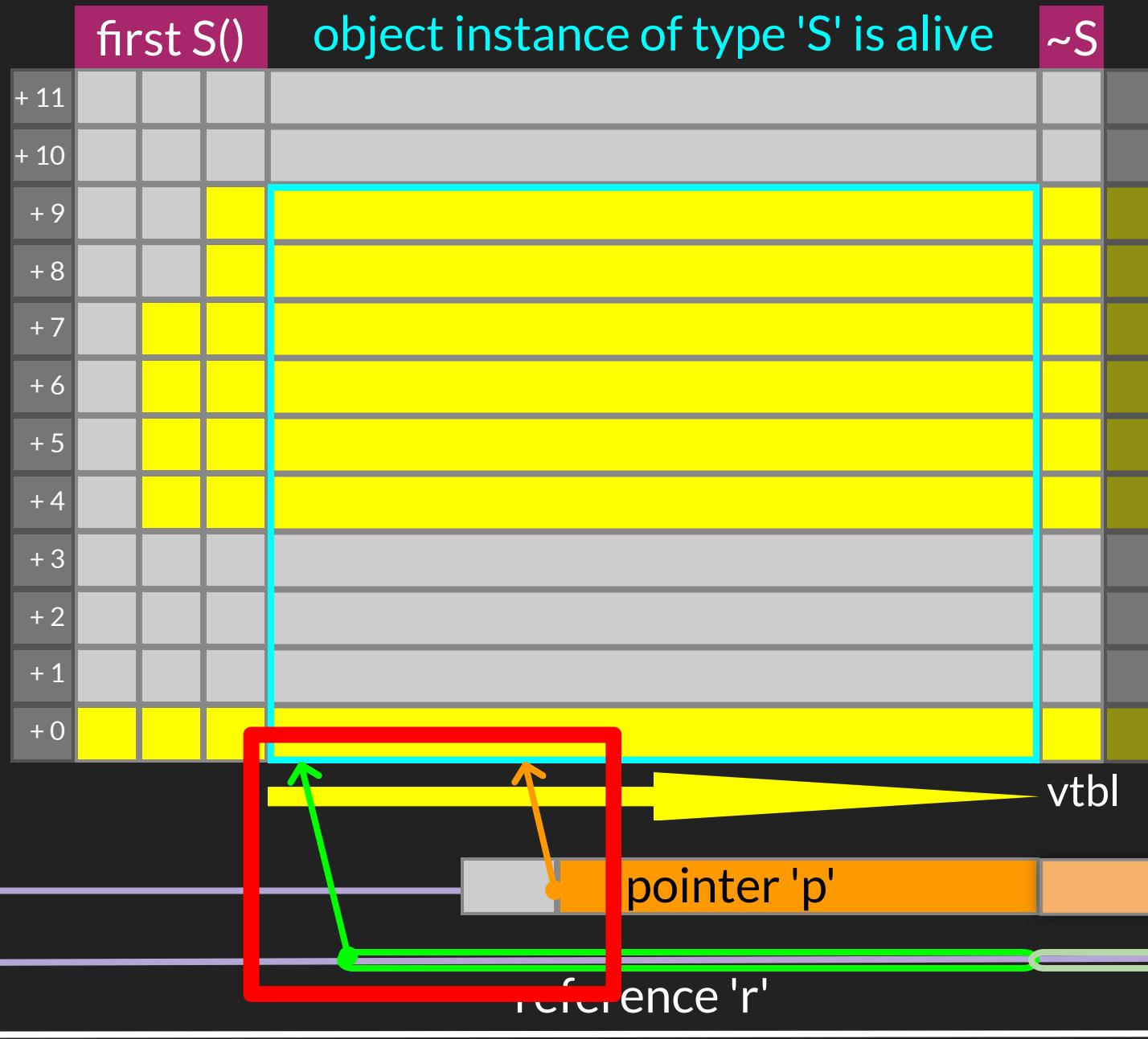


spatial



LIFETIME-SAFETY

spatial



THREAD-SAFETY

IN REAL LIFE ...





SECURITY



SECURITY



FUNCTIONAL SAFETY



FUNCTIONAL SAFETY

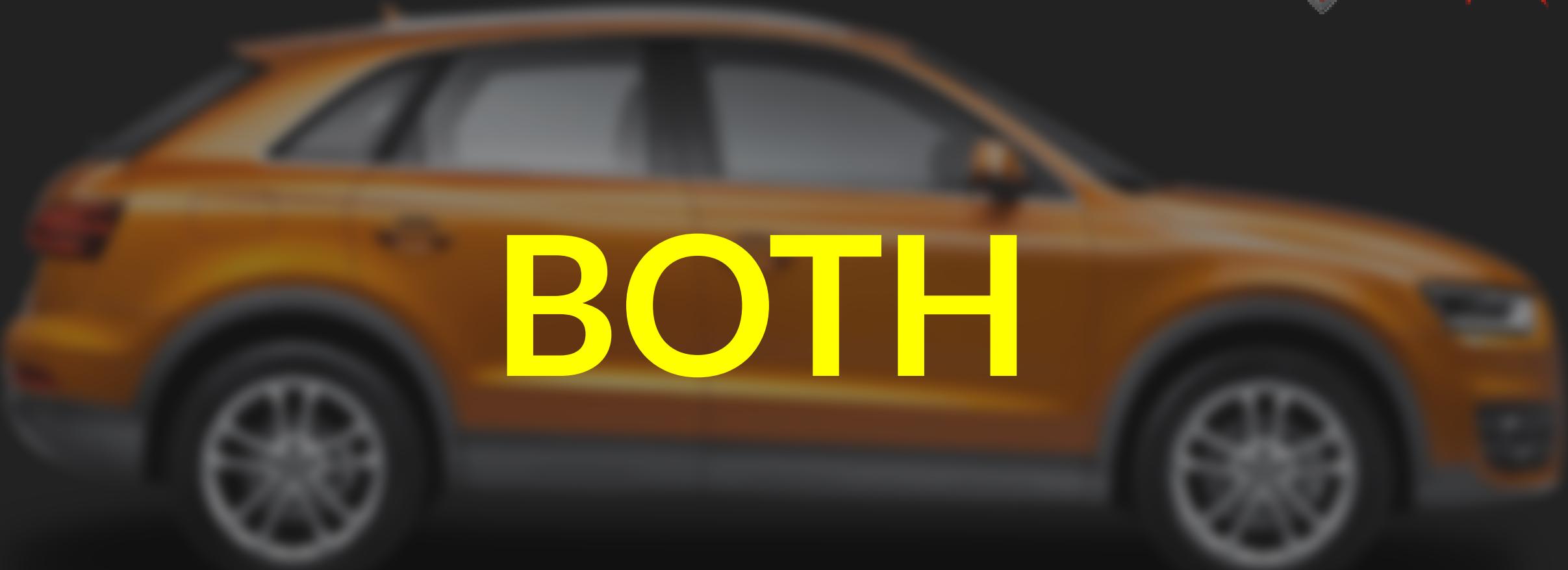


A blurry, out-of-focus image of an orange car, possibly a hatchback or SUV, centered against a dark background.

BOTH



BOTH



A close-up photograph of bare tree branches, likely from an apple or pear tree, covered in patches of bright yellow lichen. The branches are dark and twisted, creating a complex network against a blurred background of a blue sky with white clouds.

GOING DOWN...

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 S s;
6 U u;
7
8 constexpr int f() {
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 S s;
6 U u;
7
8 constexpr int f() {
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 S s;
6 U u;
7
8 constexpr int f() {
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 S s;
6 U u;
7
8 constexpr int f() {
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 S s;
6 U u;
7
8 constexpr int f() {
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 S s;
6 U u;
7
8 constexpr int f() {
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s;
7     U u;
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

<https://godbolt.org/z/snPW3bYGE>

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s;           // uninitialized objects in
7     U u;           // automatic or dynamic memory
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st); // GCC complains about UB
17    // erroneous behaviour (EB) 🚶 in C++26 ⚡
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s;           // uninitialized objects in
7     U u;           // automatic or dynamic memory
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st); // GCC complains about UB
17    // erroneous behaviour (EB) 🚶 in C++26 ⚡
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s;           // uninitialized objects in
7     U u;           // automatic or dynamic memory
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st); // GCC complains about UB
17    // erroneous behaviour (EB) 🚶 in C++26 ⚡
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s{};           // initialized, best practice
7     U u{};           // {} available since C++11
8     // no more UB or EB 👍
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s{};           // initialized
7     U u{};           // no more EB
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     constexpr int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

<https://godbolt.org/z/qqsfExKdr>

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s{};           // initialized
7     U u{};           // no more EB
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st);
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     constexpr int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

<https://godbolt.org/z/qqsfExKdr>

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s{};           // initialized
7     U u{};           // initialized
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st); // UB: out of bounds read
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     constexpr int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s{};           // initialized
7     U u{};           // initialized
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st); // UB: out of bounds read
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut);
26
27    return 0;
28 }
29
30 int main() {
31     constexpr int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s{};           // initialized
7     U u{};           // initialized
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st); // UB: out of bounds read
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut); // UB: out of bounds read
26                           // UB: inactive member
27    return 0;
28 }
29
30 int main() {
31     constexpr int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s{};           // initialized
7     U u{};           // initialized
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st); // UB: out of bounds read
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut); // UB: out of bounds read
26                           // UB: inactive member
27    return 0;
28 }
29
30 int main() {
31     constexpr int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

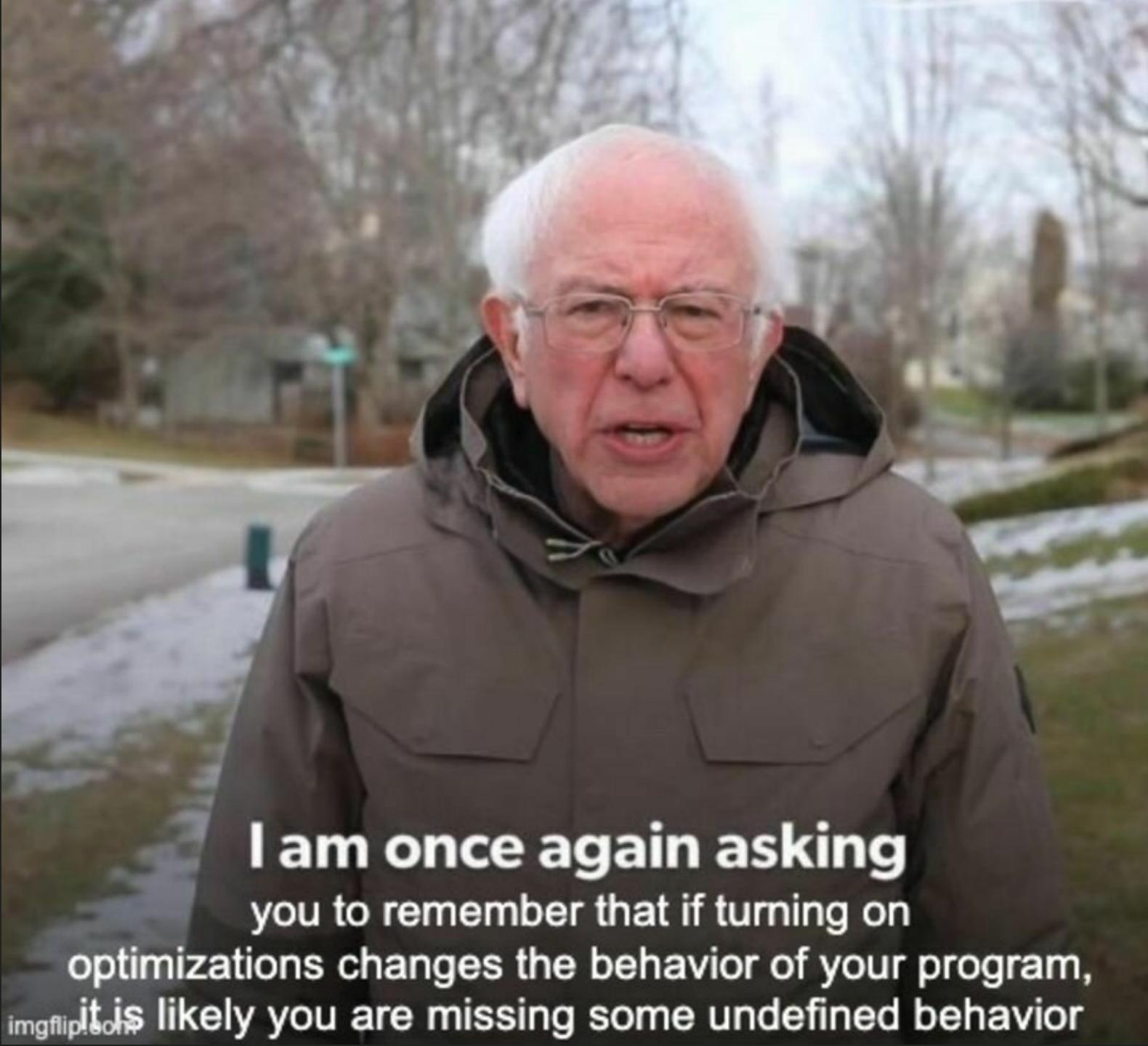
```
1 // contents of f.cpp
2
3 #include "f.h"
4
5 constexpr int f() {
6     S s{};           // initialized
7     U u{};           // initialized
8
9     int *sp = s.b;
10    int *sq = &s.b[0];
11    int *sr = &s.b[4];
12    int *st = &s.c;
13
14    assert( sp == sq);
15    assert( sr == st);
16    assert(*sr == *st); // UB: out of bounds read
17
18    int *up = u.b;
19    int *uq = &u.b[0];
20    int *ur = &u.b[4];
21    int *ut = &u.c;
22
23    assert( up == uq);
24    assert( uq == ut);
25    assert(*ur == *ut); // UB: out of bounds read
26                           // UB: inactive member
27    return 0;
28 }
29
30 int main() {
31     constexpr int rc = f();
32     return rc;
33 }
```

```
1 // contents of f.h
2
3 // product type
4
5 struct S {
6     int a;
7     int b[4];
8     int c;
9 };
10
11 // sum type
12
13 union U {
14     int a;
15     int b[4];
16     int c;
17 };
```

The current C++ standard document
mentions "undefined behaviour" (UB)
ca. 140 times in the language-specific part.







I am once again asking
you to remember that if turning on
optimizations changes the behavior of your program,
it is likely you are missing some undefined behavior

Constant evaluation

Constant evaluation
is mandated to
refuse and diagnose all occurrences of UB
during program execution.



Constant evaluation
is mandated to
refuse and diagnose all occurrences of UB
during program execution.

[defns.undefined], [expr.const]/10

spatial

first S() object instance of type 'S' is alive ~S



reference 'r'

F1
F2

temporal

```
struct S {  
    S();  
    virtual ~S();  
  
    char a = 1;  
    int32_t b = 2;  
    int16_t c = 3;  
};  
  
constexpr  
void Function1 {  
    S _;  
    S * p = &_;  
}  
  
constexpr  
void Function2 {  
    S _;  
    S & r = _;  
}
```

```
1 constexpr int f(const int *array, bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     return array[index];
8 }
9
10 constexpr int array[4] {};
11
12 static_assert(f(array, false, false) == 0);
13 static_assert(f(array, true,  false) == 0);
14 static_assert(f(array, false, true ) == 0);
15
16 int main() {
17     return
18         (f(array, false, false) == 0)
19         + (f(array, true,  false) == 0)
20         + (f(array, false, true ) == 0);
21 }
```



```
1 constexpr int f(const int *array, bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     return array[index];
8 }
9
10 constexpr int array[4]{};
11
12 static_assert(f(array, false, false) == 0);
13 static_assert(f(array, true,  false) == 0);
14 static_assert(f(array, false, true ) == 0);
15
16 int main() {
17     return
18         (f(array, false, false) == 0)
19         + (f(array, true,  false) == 0)
20         + (f(array, false, true ) == 0);
21 }
```

```
1 constexpr int f(const int *array, bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     return array[index];
8 }
9
10 constexpr int array[4]{};
11
12 static_assert(f(array, false, false) == 0);
13 static_assert(f(array, true,  false) == 0);
14 static_assert(f(array, false, true ) == 0);
15
16 int main() {
17     return
18         (f(array, false, false) == 0)
19         + (f(array, true,  false) == 0)
20         + (f(array, false, true ) == 0);
21 }
```

```
1 constexpr int f(const int *array, bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     return array[index];
8 }
9
10 constexpr int array[4]{};
11
12 static_assert(f(array, false, false) == 0);
13 static_assert(f(array, true,  false) == 0);
14 static_assert(f(array, false, true ) == 0);
15
16 int main() {
17     return
18         (f(array, false, false) == 0)
19         + (f(array, true,  false) == 0)
20         + (f(array, false, true ) == 0);
21 }
```

100% code coverage! 


```
1 constexpr int f(const int *array, bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     return array[index];
8 }
9
10 constexpr int array[4]{};
11
12 static_assert(f(array, false, false) == 0);
13 static_assert(f(array, true, false) == 0);
14 static_assert(f(array, false, true ) == 0);
15 static_assert(f(array, true, true ) == 0); // fails!
16
17 int main() {
18     return
19         (f(array, false, false) == 0)
20         + (f(array, true, false) == 0)
21         + (f(array, false, true ) == 0)
22         + (f(array, true, true ) == 0); // UB!
23 }
```

100% path coverage!

CONSTEXPR

- since C++11
- major improvements until C++20
- minor language improvements in C++23,
but huge library improvements

CONSTEXPR

- since C++11
- major improvements until C++20
- minor language improvements in C++23,
but huge library improvements

new in C++26:

- exceptions
- placement new
- structured bindings
- improved references
- lots more library functions





MORE CODE ...

```
1 constexpr int f(const int *array, bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     return array[index];
8 }
9
10 constexpr int array[4] {};
11
12 static_assert(f(array, false, false) == 0);
```



```
1 constexpr int f(const int *array, bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     // implicit contract: index lies within the referenced array
8     return array[index];
9 }
10
11 constexpr int array[4]{};
12
13 static_assert(f(array, false, false) == 0);
```

```
1 constexpr int f(const int (&array)[4], bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     // perform a range check here!
8     return array[index];
9 }
10
11 constexpr int array[4]{};
12
13 static_assert(f(array, false, false) == 0);
```



```
1 template <std::size_t N>
2 constexpr int f(const int (&array)[N], bool a, bool b) {
3     unsigned index = 1;
4
5     if (a) --index;
6     if (b) --index;
7
8     // perform a range check here!!
9     return array[index];
10 }
11
12 constexpr int array[4]{};
13
14 static_assert(f(array, false, false) == 0);
```



```
1 constexpr int f(const int array[], std::size_t N, bool a, bool b) {
2     unsigned index = 1;
3
4     if (a) --index;
5     if (b) --index;
6
7     // perform a range check here!!!!
8     return array[index];
9 }
10
11 constexpr int array[4]{};
12
13 static_assert(f(array, std::size(array), false, false) == 0);
```



```
1 #include <span> // "Contemporary C++"
2
3 constexpr int select(std::span<const int> sequence,
4                      SomeBoolFeature a, OtherBoolFeature b) {
5     unsigned index = 1;
6
7     if (a) --index;
8     if (b) --index;
9
10    // i perform a range check here !
11    return sequence[index];
12 }
13
14 constexpr int array[4]{};
15
16 static_assert(select(array, option_a, feature_b) == 0);
```

```
1 #include <span> // "Contemporary C++"
2
3 constexpr int select(std::span<const int> sequence,
4                      SomeBoolFeature a, OtherBoolFeature b) {
5     unsigned index = 1;
6
7     if (a) --index;
8     if (b) --index;
9
10    // i perform a range check here !
11    return sequence[index];
12 }
13
14 constexpr int array[4]{};
15
16 static_assert(select(array, option_a, feature_b) == 0);
```



```
1 #include <span>
2
3 constexpr int select(std::span<const int> sequence,
4                      SomeBoolFeature a, OtherBoolFeature b) {
5     unsigned index = 1;
6
7     if (a) --index;
8     if (b) --index;
9
10    // range checking is internally done
11    // in a 'hardened' standard library!
12    return sequence[index];
13 }
14
15 constexpr int array[4]{};
16
17 static_assert(select(array, option_a, feature_b) == 0);
```



P3471

Standard library hardening

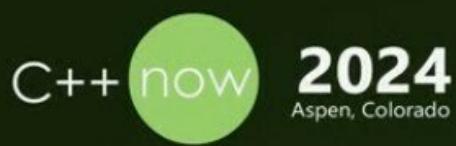
P3471



Standard library hardening



Expect this before C++26 !



CppNow.org

Video Sponsorship Provided By

millennium
think-cell



Agenda

Overview of Memory Safety

Library Undefined Behavior

Standard Library Hardening

Typed Memory Operations

Conclusions

Security in C++
Hardening Techniques From the Trenches

Louis Dionne



Louis Dionne "Security in C++", C++Now 2024

C++ library preconditions

- are either checked with termination semantics in all compilation modes,
- or are ignored → UB (status quo)
- detected contract violations
 - may stop execution instantly (without P2900)
 - detour to handle_contractViolation (with P2900)

C++ library preconditions

- are either checked with termination semantics in all compilation modes,
 - or are ignored → UB (status quo)
-
- detected contract violations
 - may stop execution instantly (without P2900)
 - detour to handle_contractViolation (with P2900)



C++ standard libraries

may go **beyond** P3471

- more containers
- more classes
- more functions
- destructor pointer tombstones → resist "use after free" 😎

C++ standard libraries

may go **beyond** P3471

- more containers
- more classes
- more functions
- destructor pointer tombstones → resist "use after free" 😎

Available in MS-STL, libc++, ... !

```
1 import std;
2
3 int main() {
4     std::println("hardened STL: {}", static_cast<bool>(_MSVC_STL_HARDENING));
5
6     std::vector v = { 1, 2, 3 };
7     std::println("v[3] = {}", v[3]);
8     std::println("v[-1] = {}", v[-1]);
9 }
```

MS-STL version 202502

just as ever

with hardening

```
1 import std;
2
3 int main() {
4     std::println("hardened STL: {}", static_cast<bool>(_MSVC_STL_HARDENING));
5
6     std::vector v = { 1, 2, 3 };
7     std::println("v[3] = {}", v[3]);
8     std::println("v[-1] = {}", v[-1]);
9 }
```

MS-STL version 202502

just as ever

with hardening

```
1 > cl test.cpp
2 > test.exe
3 hardened STL: false
4 v[3] = 9673
5 v[-1] = -1811929271
6
7 // program ended with return value 0
```



```
1 import std;
2
3 int main() {
4     std::println("hardened STL: {}", static_cast<bool>(_MSVC_STL_HARDENING));
5
6     std::vector v = { 1, 2, 3 };
7     std::println("v[3] = {}", v[3]);
8     std::println("v[-1] = {}", v[-1]);
9 }
```

MS-STL version 202502

just as ever

```
1 > cl test.cpp
2 > test.exe
3 hardened STL: false
4 v[3] = 9673
5 v[-1] = -1811929271
6
7 // program ended with return value 0
```

with hardening

```
1 > cl test.cpp
2 > test.exe
3 hardened STL: true
4
5 // program execution stopped here
6 // with error code 0xC0000409
7 // i.e. STATUS_FAIL_FAST_EXCEPTION
```



DILIGENCE



P2900

Contracts for C++



P2900

Contracts for C++

Similar to Eiffel, Ada, or D

CONTRACTS IN C++

Support Developers

CONTRACTS IN C++

Assist "Design by Contract"

CONTRACTS IN C++

Check Correctness

```
1 // anywhere
2
3 int func(int x)
4 // precondition: x is not 1
5 // postcondition: result value is different from argument value
6 {
7 // ...
8     assert(x != 3); // assures an expectation
9 // ...
10    return x | 2;
11 }
12
13 void call_func() {
14     func(0); // fine
15     func(1); // oops
16     func(2); // oh my...
17     func(3); // what now?
18     func(4); // fine again
19 }
```

```
1 // anywhere
2
3 int func(int x)
4 // precondition: x is not 1
5 // postcondition: result value is different from argument value
6 {
7 // ...
8     assert(x != 3); // assures an expectation
9 // ...
10    return x | 2;
11 }
12
13 void call_func() {
14     func(0); // fine
15     func(1); // oops
16     func(2); // oh my...
17     func(3); // what now?
18     func(4); // fine again
19 }
```

```
1 // anywhere
2
3 int func(int x)
4 // precondition: x is not 1
5 // postcondition: result value is different from argument value
6 {
7 // ...
8     assert(x != 3); // assures an expectation
9 // ...
10    return x | 2;
11 }
12
13 void call_func() {
14     func(0); // fine
15     func(1); // oops
16     func(2); // oh my...
17     func(3); // what now?
18     func(4); // fine again
19 }
```

```

1 // anywhere
2
3 int func(int x)           // function parameters are *immutable* in pre/post assertions!
4     pre (x != 1)          // a precondition assertion
5     post(r : r != x)      // a postcondition assertion
6                     // 'r' names the result object of 'func'
7 {
8     // observable checkpoint (P1494 ⚡🛡️) right before evaluating the assert-argument
9     contract_assert(x != 3); // an assertion statement, e.g. check invariants
10    // observable checkpoint right after returning from the handler in 'observe' semantics
11    return x | 2;
12 }
13
14 void call_func() {
15     func(0);   // no contract violation
16     func(1);   // violates precondition assertion of 'func'
17     func(2);   // violates postcondition assertion of 'func'
18     func(3);   // violates assertion statement within 'func'
19     func(4);   // no contract violation
20 }
21
22 // -----
23
24 // in global namespace, global module, user-invisible
25
26 // called on any contract violation
27 // system-provided or optionally user-replaceable
28
29 void handle_contractViolation(std::contracts::contractViolation cvo) {
30     // handle contract violation according to
31     // - the evaluation semantics as indicated in 'cvo'
32     // - the rest of the info in 'cvo'
33 }
```



```

1 // anywhere
2
3 int func(int x)           // function parameters are *immutable* in pre/post assertions!
4     pre (x != 1)          // a precondition assertion
5     post(r : r != x)      // a postcondition assertion
6                     // 'r' names the result object of 'func'
7 {
8     // observable checkpoint (P1494 ⚡🛡️) right before evaluating the assert-argument
9     contract_assert(x != 3); // an assertion statement, e.g. check invariants
10    // observable checkpoint right after returning from the handler in 'observe' semantics
11    return x | 2;
12 }
13
14 void call_func() {
15     func(0);   // no contract violation
16     func(1);   // violates precondition assertion of 'func'
17     func(2);   // violates postcondition assertion of 'func'
18     func(3);   // violates assertion statement within 'func'
19     func(4);   // no contract violation
20 }
21
22 // -----
23
24 // in global namespace, global module, user-invisible
25
26 // called on any contract violation
27 // system-provided or optionally user-replaceable
28
29 void handle_contractViolation(std::contracts::contractViolation cvo) {
30     // handle contract violation according to
31     // - the evaluation semantics as indicated in 'cvo'
32     // - the rest of the info in 'cvo'
33 }
```



```

1 // anywhere
2
3 int func(int x)           // function parameters are *immutable* in pre/post assertions!
4     pre (x != 1)          // a precondition assertion
5     post(r : r != x)      // a postcondition assertion
6                     // 'r' names the result object of 'func'
7 {
8     // observable checkpoint (P1494 ⚡🛡️) right before evaluating the assert-argument
9     contract_assert(x != 3); // an assertion statement, e.g. check invariants
10    // observable checkpoint right after returning from the handler in 'observe' semantics
11    return x | 2;
12 }
13
14 void call_func() {
15     func(0);   // no contract violation
16     func(1);   // violates precondition assertion of 'func'
17     func(2);   // violates postcondition assertion of 'func'
18     func(3);   // violates assertion statement within 'func'
19     func(4);   // no contract violation
20 }
21
22 // -----
23
24 // in global namespace, global module, user-invisible
25
26 // called on any contract violation
27 // system-provided or optionally user-replaceable
28
29 void handle_contractViolation(std::contracts::contractViolation cvo) {
30     // handle contract violation according to
31     // - the evaluation semantics as indicated in 'cvo'
32     // - the rest of the info in 'cvo'
33 }
```

```

1 // anywhere
2
3 int func(int x)           // function parameters are *immutable* in pre/post assertions!
4     pre (x != 1)          // a precondition assertion
5     post(r : r != x)      // a postcondition assertion
6                     // 'r' names the result object of 'func'
7 {
8     // observable checkpoint (P1494 ⚡🛡️) right before evaluating the assert-argument
9     contract_assert(x != 3); // an assertion statement, e.g. check invariants
10    // observable checkpoint right after returning from the handler in 'observe' semantics
11    return x | 2;
12 }
13
14 void call_func() {
15     func(0);   // no contract violation
16     func(1);   // violates precondition assertion of 'func'
17     func(2);   // violates postcondition assertion of 'func'
18     func(3);   // violates assertion statement within 'func'
19     func(4);   // no contract violation
20 }
21
22 // -----
23
24 // in global namespace, global module, user-invisible
25
26 // called on any contract violation
27 // system-provided or optionally user-replaceable
28
29 void handle_contractViolation(std::contracts::contractViolation cvo) {
30     // handle contract violation according to
31     // - the evaluation semantics as indicated in 'cvo'
32     // - the rest of the info in 'cvo'
33 }
```

```

1 // anywhere
2
3 int func(int x)           // function parameters are *immutable* in pre/post assertions!
4     pre (x != 1)          // a precondition assertion
5     post(r : r != x)      // a postcondition assertion
6                     // 'r' names the result object of 'func'
7 {
8     // observable checkpoint (P1494 ⚡🛡️) right before evaluating the assert-argument
9     contract_assert(x != 3); // an assertion statement, e.g. check invariants
10    // observable checkpoint right after returning from the handler in 'observe' semantics
11    return x | 2;
12 }
13
14 void call_func() {
15     func(0);   // no contract violation
16     func(1);   // violates precondition assertion of 'func'
17     func(2);   // violates postcondition assertion of 'func'
18     func(3);   // violates assertion statement within 'func'
19     func(4);   // no contract violation
20 }
21
22 // -----
23
24 // in global namespace, global module, user-invisible
25
26 // called on any contract violation
27 // system-provided or optionally user-replaceable
28
29 void handle_contractViolation(std::contracts::contractViolation cvo) {
30     // handle contract violation according to
31     // - the evaluation semantics as indicated in 'cvo'
32     // - the rest of the info in 'cvo'
33 }
```

CONTRACT EVALUATION SEMANTICS

Semantics	check condition	fail → invoke handler	terminate
ignore (zero overhead)	✗	✗	✗
observe	✓	✓ ¹	✗
enforce	✓	✓	✓ ²
quick_enforce	✓	✗	✓ ³

¹ compile-time: diagnostic

² runtime: std::abort / compile-time: program is ill-formed

³ runtime: terminate abstract machine execution (i.e. no more user code)
compile-time: program is ill-formed

CONTRACT EVALUATION SEMANTICS

Semantics	check condition	fail → invoke handler	terminate
ignore (zero overhead)	✗	✗	✗
observe	✓	✓ ¹	✗
enforce	✓	✓	✓ ²
quick_enforce	✓	✗	✓ ³

¹ compile-time: diagnostic

² runtime: std::abort / compile-time: program is ill-formed

³ runtime: terminate abstract machine execution (i.e. no more user code)
compile-time: program is ill-formed



A photograph of a mountainous landscape. In the foreground, there's a set of wide, grey stone steps leading up a rocky hillside. The hillside is covered in lush green grass and small trees. In the background, there are more mountains with dense forests of tall evergreen trees under a clear blue sky.

SAFE C++ ?

P3081

Core safety profiles for C++26

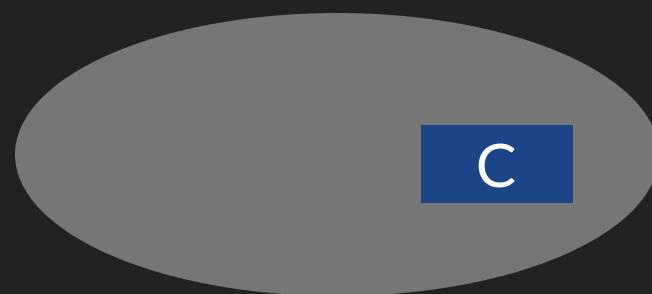


P3081

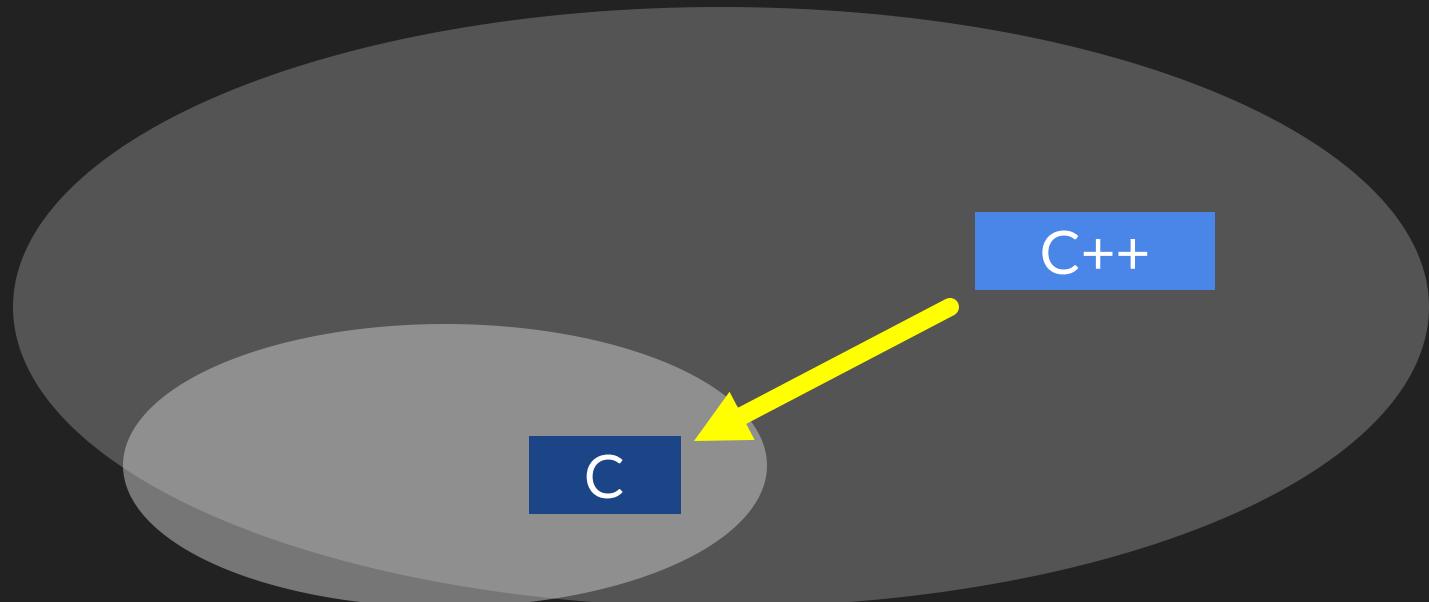
Core safety profiles for C++26

Not in C++26 ! 😬

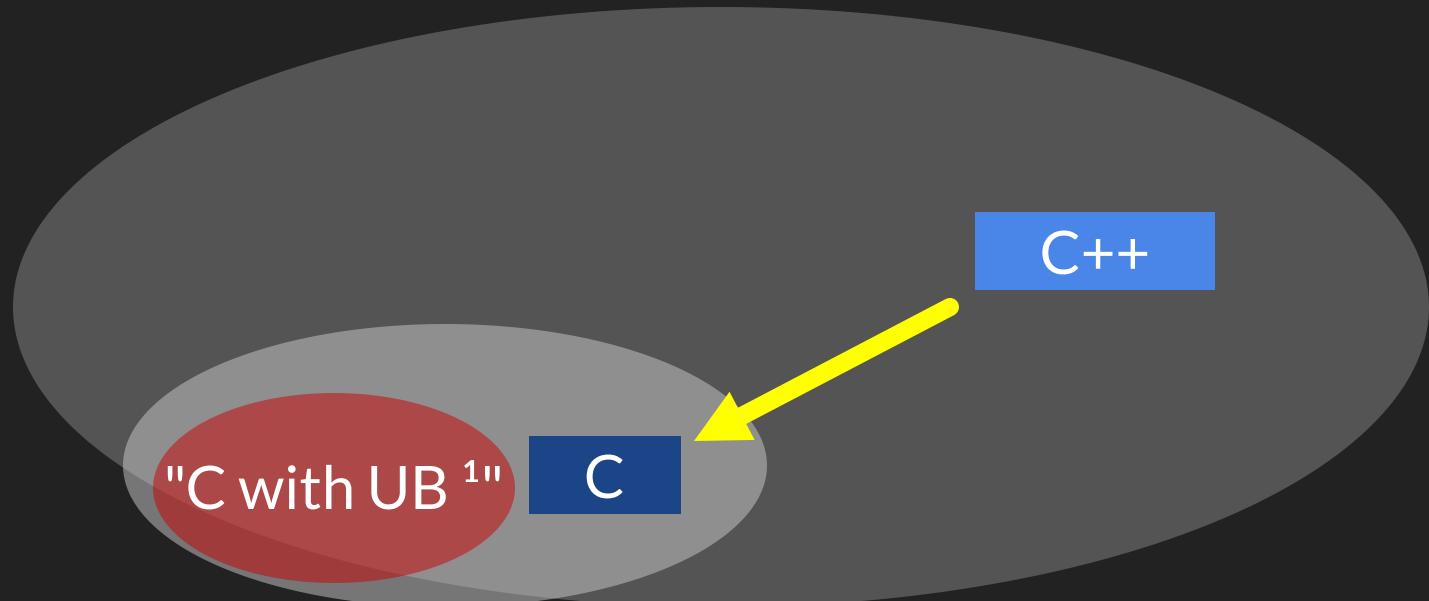
THE IDEA



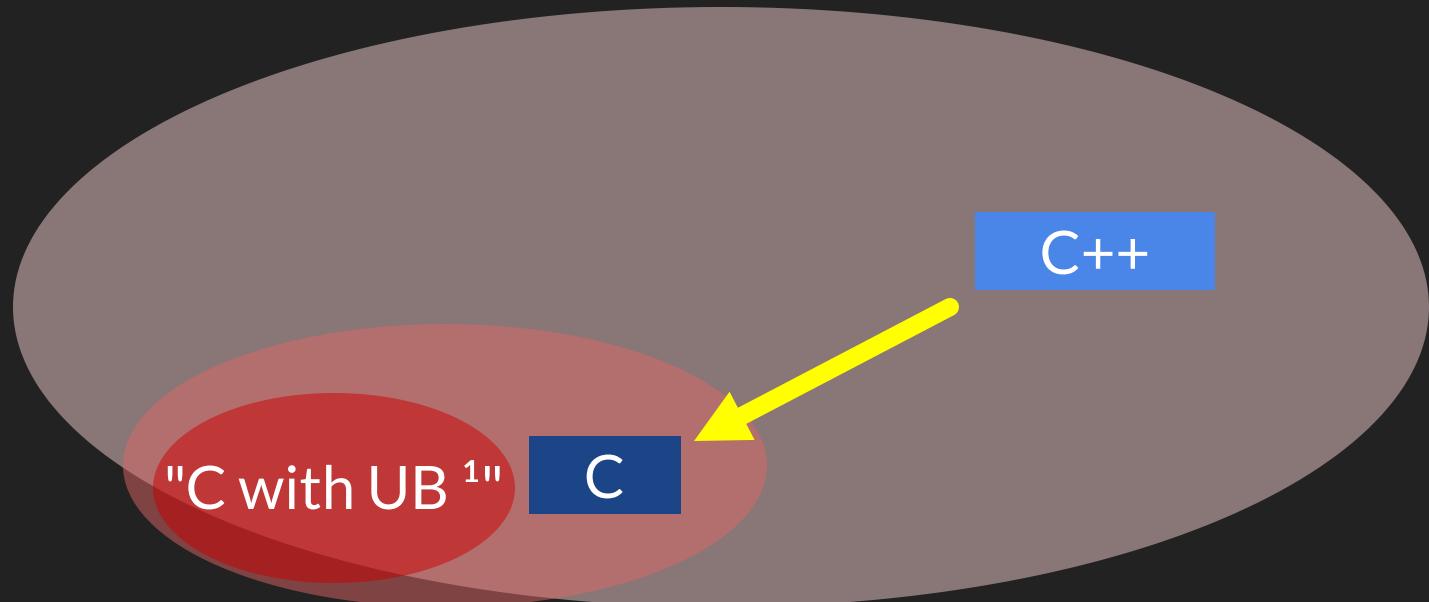
THE IDEA



THE IDEA



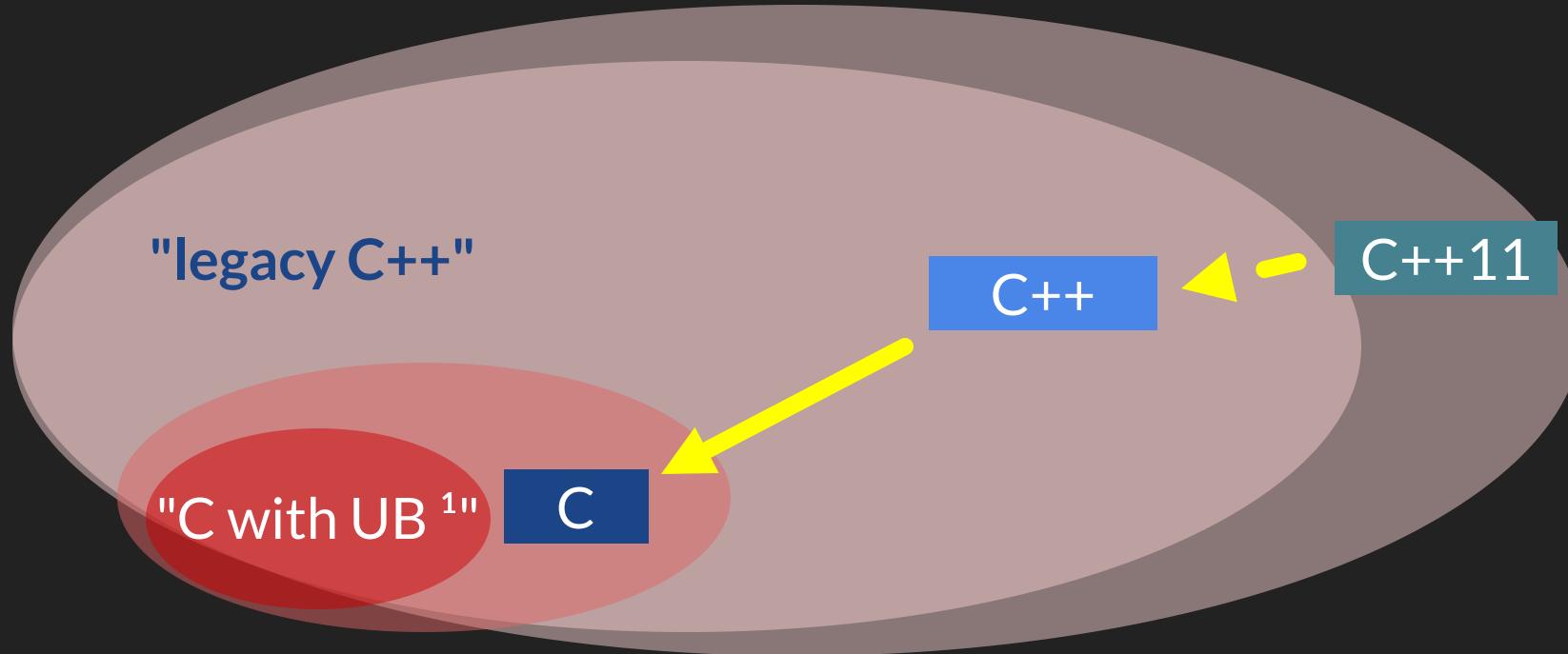
THE IDEA



¹ WG14 N3453: C contains

- 100 cases of UB in the core language
- 221 cases of UB in total

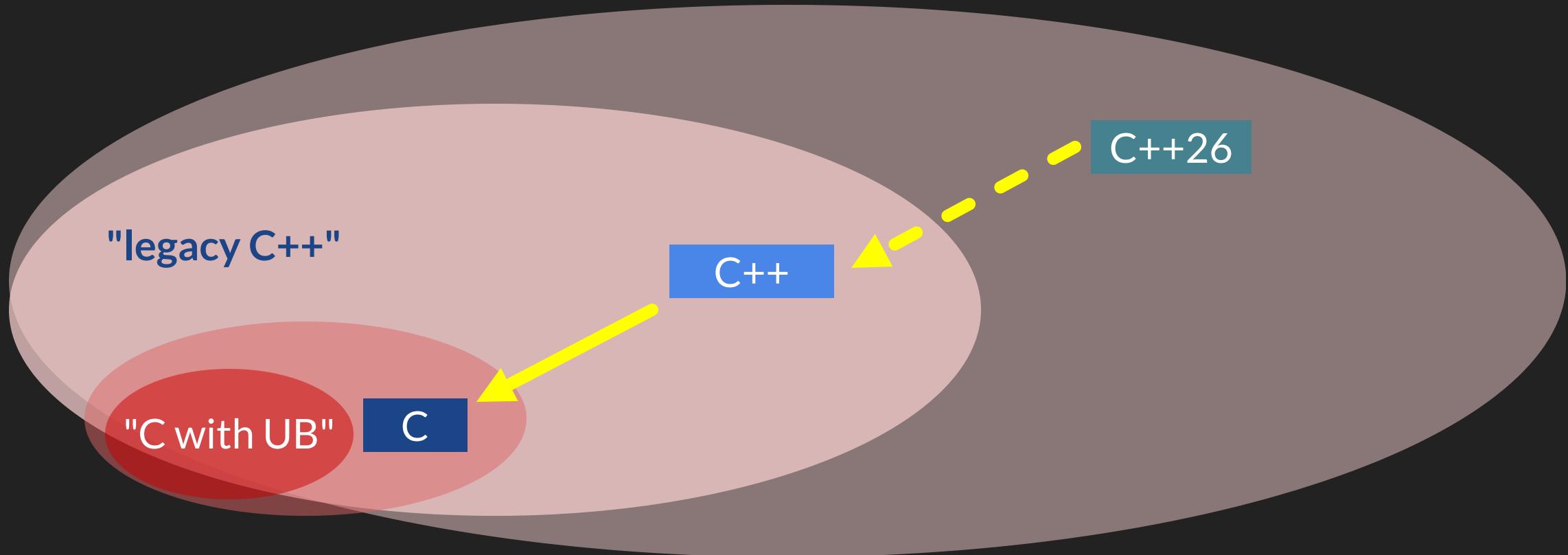
THE IDEA



¹ WG14 N3453: C contains

- 100 cases of UB in the core language
- 221 cases of UB in total

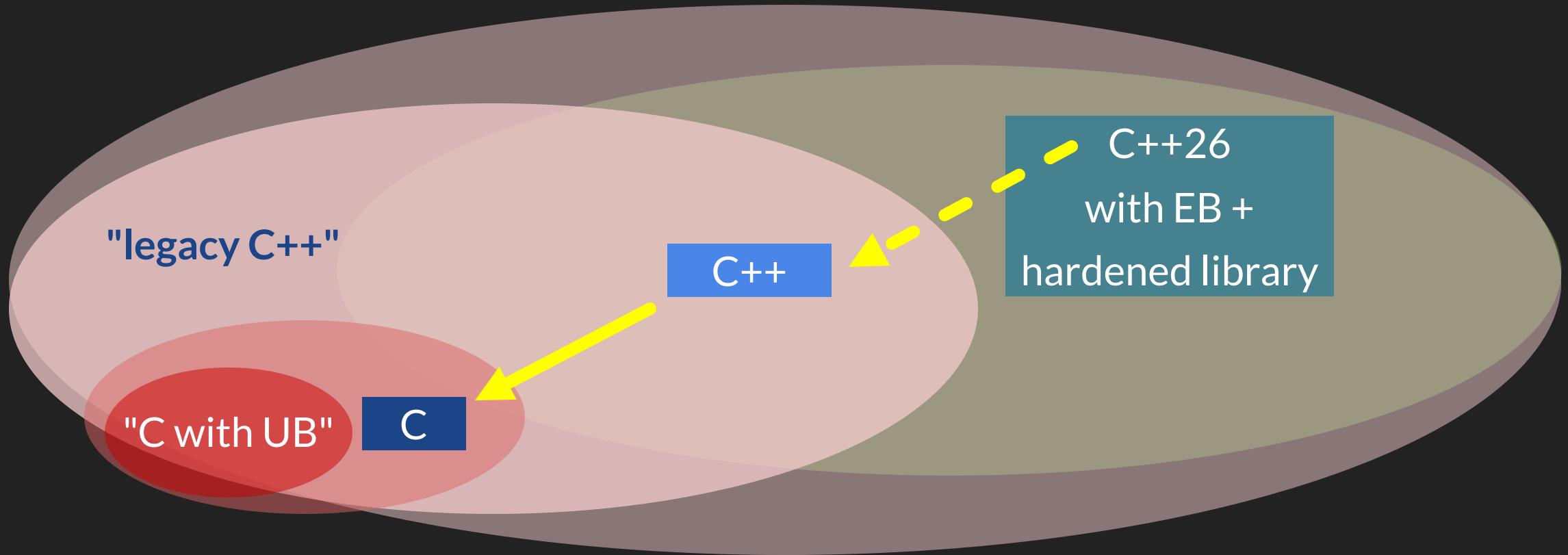
THE IDEA



Inherited UB taints the entire language

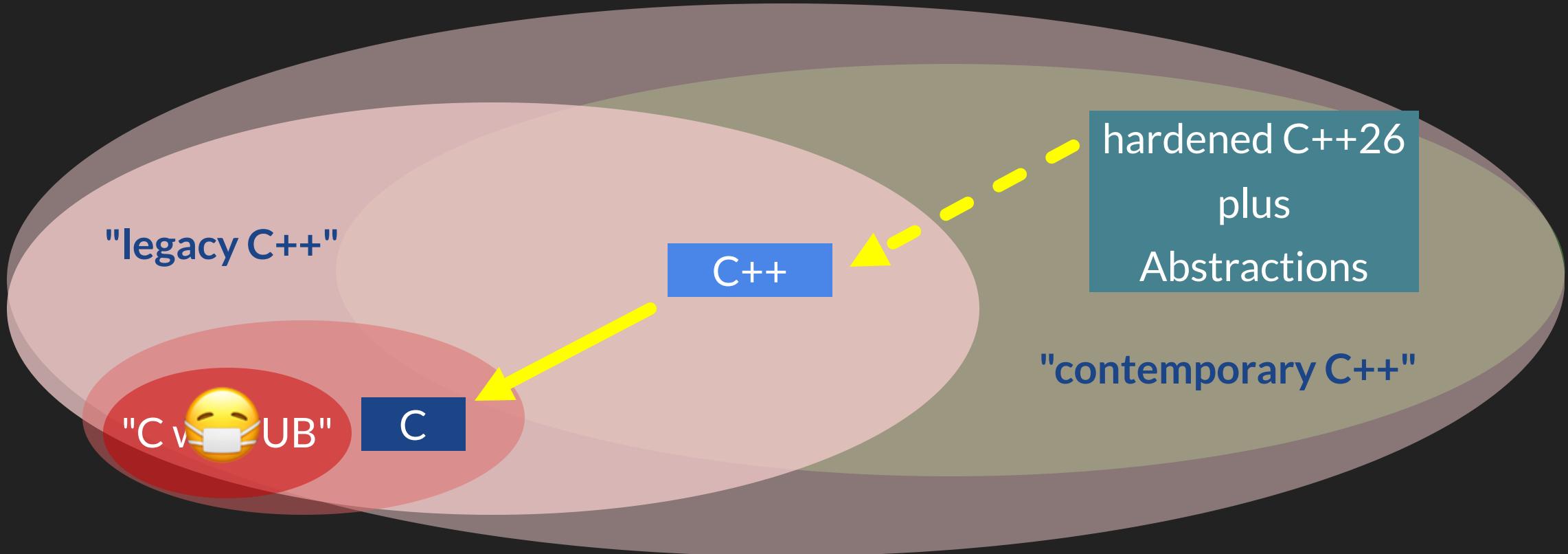
C++ adds its own UB

THE IDEA



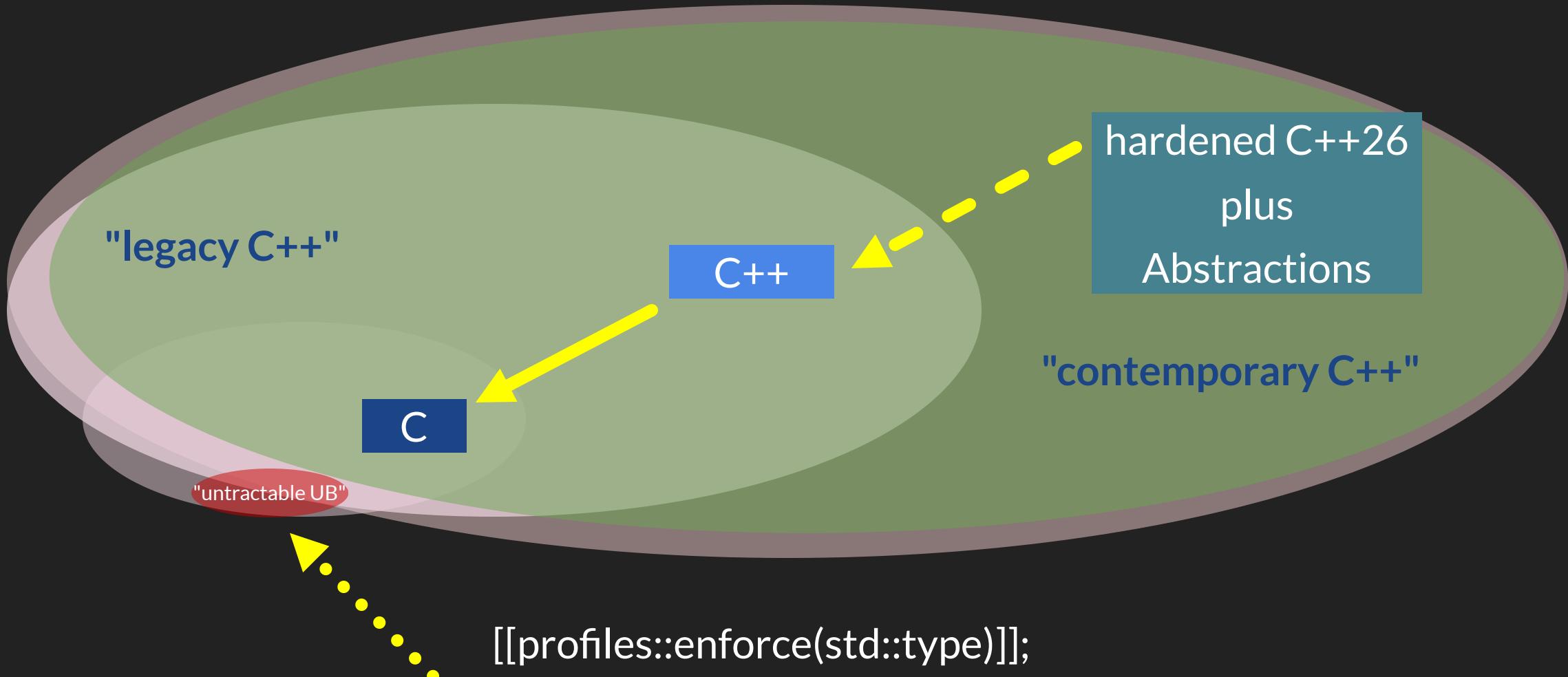
Inherited UB taints the entire language
C++ adds its own UB

THE IDEA



Inherited UB taints the entire language
C++ adds its own UB

THE IDEA



```
[[profiles::enforce(std::type)]];  
[[profiles::enforce(std::bounds)]];  
[[profiles::enforce(...)]];
```

BEYOND C++26



Tackle lifetime-safety?

Tackle lifetime-safety?

- life-time tracking and annotations like Rust (P3390 Safe C++)
 - might require major compiler efforts
 - will strongly impact the standard library

Tackle lifetime-safety?

- life-time tracking and annotations like Rust (P3390 Safe C++)
 - might require major compiler efforts
 - will strongly impact the standard library
- emphasize value semantics like Hylo (references under the hood, whole-part relationships)
 - fundamentally ingrained in C++
 - added benefit: exclusivity for free

Tackle lifetime-safety?

- life-time tracking and annotations like Rust (P3390 Safe C++)
 - might require major compiler efforts
 - will strongly impact the standard library
- emphasize value semantics like Hylo (references under the hood, whole-part relationships)
 - fundamentally ingrained in C++
 - added benefit: exclusivity for free
- pursue a third way
 - currently in development

Tackle thread-safety?

Tackle thread-safety?

- guarantee exclusivity?
 - shift object across thread boundaries
 - will require cutting cross-thread references
 - might require dataflow analysis

Tackle thread-safety?

- guarantee exclusivity?
 - shift object across thread boundaries
 - will require cutting cross-thread references
 - might require dataflow analysis
- pursue non-trivial relocation?
 - will require dataflow analysis

Tackle all
undefined behaviour?

Will be pursued in a Whitepaper
along with EB, Contracts, Profiles

SUMMARY

- why this perceived push towards security?
- clarifications on terminology
- security versus functional safety
- implications of UB at runtime and compile-time
- compile-time testing
- hardening the C++ standard library
- C++ contracts
- C++ profiles
- path into the future

"Simplicity is prerequisite
for reliability"

Edsger W. Dijkstra

RESOURCES

- Living, up-to-date C++ standard (currently at C++26 est.)
- The Case for Memory Safe Roadmaps
- P2900 Contracts for C++
- P3081 Core safety profiles for C++26
- P3589 C++ Profiles: The Framework

Contact

 dani@ngrt.de

 @DanielaKEngert@hachyderm.io

 DanielaE

Images: courtesy of Matúš Chochlík





Ceterum censeo ABI esse frangendam