TOWARDS SAFETY & SECURITY IN C++

Daniela Engert - Stockholm 2025

C-00000291*.sys

- 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
- Building computers and creating software for more than 40 years
- Developing hardware and software in the field of applied digital signal processing for more than 35 years

- Member of the C++ committee (a.k.a. WG21) 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

2024 CWE Top 25

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

2024 CWE Top 25

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

[redacted] now

[redacted]

(@[redacted]@pony.social

[redacted]

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,

Somewhere on Mastodon...

IN REAL LIFE ...


SECURITY


SECURITY


GMH Prüftechnik


FUNCTIONAL
SAFETY


FUNCTIONAL
SAFETY


GOING DOWN ...

```cpp
// contents of f.cpp

#include "f.h"

S s;
U u;

constexpr int f() {
    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

S s;
U u;

constexpr int f() {
    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

S s;
U u;

constexpr int f() {
    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

S s;
U u;

constexpr int f() {
    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

S s;
U u;

constexpr int f() {
    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

S s;
U u;

constexpr int f() {
    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s;
    U u;

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

https://godbolt.org/z/Tc13f7YbT

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s;          // objects 's', 'u' uninitialized
    U u;          // in automatic or dynamic memory

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st); // GCC may complain about UB
     // erroneous behaviour (EB) 🚀 in C++26 💥🛡️
    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s;          // objects 's', 'u' uninitialized
    U u;          // in automatic or dynamic memory

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st); // GCC may complain about UB
     // erroneous behaviour (EB) 🚀  in C++26  💥🛡️
    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s;          // objects 's', 'u' uninitialized
    U u;          // in automatic or dynamic memory

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st); // GCC may complain about UB
     // erroneous behaviour (EB) 🚀  in C++26  💥🛡️
    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s{};          // initialized now, {} from C++11
    U u{};          // recommended best practice 👍
                    // ⇒ no longer UB or EB ⇒ 🎉
    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s{};              // no EB or UB ⇒
    U u{};              // 👍 at compile-time

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    constexpr int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

https://godbolt.org/z/Trzo7rcPG

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s{};              // no EB or UB ⇒
    U u{};              // 👍 at compile-time

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st);

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    constexpr int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

https://godbolt.org/z/Trzo7rcPG

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s{};              // initialized
    U u{};              // initialized

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st); // UB: out of bounds read

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    constexpr int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s{};              // initialized
    U u{};              // initialized

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st); // UB: out of bounds read

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut);

    return 0;
}

int main() {
    constexpr int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s{};              // initialized
    U u{};              // initialized

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st); // UB: out of bounds read

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut); // UB: reads from
                        //     inactive members
    return 0;
}

int main() {
    constexpr int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

```cpp
// contents of f.cpp

#include "f.h"

constexpr int f() {
    S s{};              // initialized
    U u{};              // initialized

    int *sp = s.b;
    int *sq = &s.b[0];
    int *sr = &s.b[4];
    int *st = &s.c;

    assert( sp ==  sq);
    assert( sr ==  st);
    assert(*sr == *st); // UB: out of bounds read

    int *up = u.b;
    int *uq = &u.b[0];
    int *ut = &u.c;

    assert( up ==  uq);
    assert( uq ==  ut);
    assert(*uq == *ut); // UB: reads from
                        //     inactive members
    return 0;
}

int main() {
    constexpr int rc = f();
    return rc;
}
```

```cpp
// contents of f.h

// product type

struct S {
    int a;
    int b[4];
    int c;
};

// sum type

union U {
    int a;
    int b[4];
    int c;
};
```

The current C++ standard document mentions "undefined behaviour" (UB) ≈ 90 times in the language-specific part.


This is fine.

Constant evaluation

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

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

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

CONSTEXPR

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

CONSTEXPR

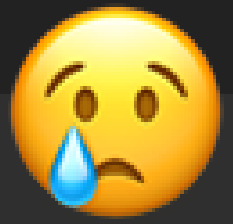
- since C++11
- major improvements until C++20
- minor language improvements in C++23, but huge library improvements

new in C++26:

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

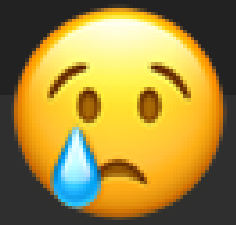

MORE CODE ...

```
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);
```

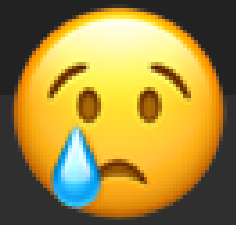


```
1  constexpr int f(const int *array, bool a, bool b) {
2      unsigned index = 1;
3
4      if (a) --index;
5      if (b) --index;
6
7      // implicit contract: index lies within the referenced array
8      return array[index];
9  }
10
11 constexpr int array[4]{};
12
13 static_assert(f(array, false,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

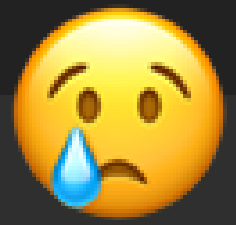
```
1  constexpr int f(const int *array, bool a, bool b) {
2      unsigned index = 1;
3
4      if (a) --index;
5      if (b) --index;
6
7      // implicit contract: index lies within the referenced array
8      return array[index];
9  }
10
11 constexpr int array[4]{};
12
13 static_assert(f(array, false,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



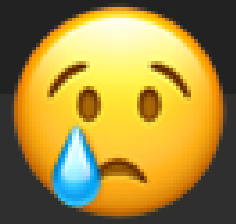
```
1  constexpr int f(const int *array, bool a, bool b) {
2      unsigned index = 1;
3
4      if (a) --index;
5      if (b) --index;
6
7      // implicit contract: index lies within the referenced array
8      return array[index];
9  }
10
11 constexpr int array[4]{};
12
13 static_assert(f(array, false,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



```
 1 constexpr int f(const int *array, 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);
```



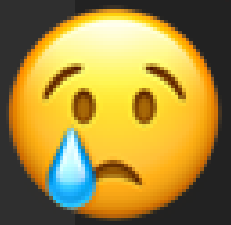
```cpp
constexpr int f(const int *array, bool a, bool b) {
    unsigned index = 1;

    if (a) --index;
    if (b) --index;

    // implicit contract: index lies within the referenced array
    return array[index];
}

constexpr int array[4]{};

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);
```

```cpp
template <std::size_t N>
constexpr int f(const int (&array)[N], bool a, bool b) {
    unsigned index = 1;

    if (a) --index;
    if (b) --index;

    // perform a range check here!!
    return array[index];
}

constexpr int array[4]{};

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);
```



```cpp
#include <span>  // "Contemporary C++" -> use strong types

constexpr int select(std::span<const int> sequence,
                     SomeBooleanOption a, AnotherBooleanOption b) {
    unsigned index = 1;

    if (a) --index;
    if (b) --index;

    // !!! perform a range check here !!!
    return sequence[index];
}

constexpr int array[4]{};

static_assert(select(array, option_a, option_b) == 0);
```

```cpp
#include <span>  // "Contemporary C++" -> use strong types

constexpr int select(std::span<const int> sequence,
                     SomeBooleanOption a, AnotherBooleanOption b) {
    unsigned index = 1;

    if (a) --index;
    if (b) --index;

    // !!! perform a range check here !!!
    return sequence[index];
}

constexpr int array[4]{};

static_assert(select(array, option_a, option_b) == 0);
```

```cpp
#include <span>

constexpr int select(std::span<const int> sequence,
                     SomeBooleanOption a, AnotherBooleanOption b) {
    unsigned index = 1;

    if (a) --index;
    if (b) --index;

    // range checking is internally done
    // in a 'hardened' standard library!
    return sequence[index];
}

constexpr int array[4]{};

static_assert(select(array, option_a, option_b) == 0);
```

P3471

Standard library hardening

P3471

Standard library hardening

Expect this even before C++26 !

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_contract_violation (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_contract_violation (with P2900)

C++ standard library implementations may go beyond P3471

- more containers
- more classes
- more functions

- destructor pointer tombstones → resist "use after free" 😎 🛡️

C++ standard library implementations may go beyond P3471

- more containers
- more classes
- more functions

- destructor pointer tombstones → resist "use after free" 😎 🛡️

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

```cpp
import std;

int main(int argc, char*[]) {
    std::println("hardened STL: {}", static_cast<bool>(_MSVC_STL_HARDENING));

    std::vector v(std::max(argc, 1), argc);
    std::println("v[3]  = {}", v[3]);
    std::println("v[-1] = {}", v[-1]);
}
```

MS-STL version 202502

just as ever

with hardening

```cpp
import std;

int main(int argc, char*[]) {
    std::println("hardened STL: {}", static_cast<bool>(_MSVC_STL_HARDENING));

    std::vector v(std::max(argc, 1), argc);
    std::println("v[3]  = {}", v[3]);
    std::println("v[-1] = {}", v[-1]);
}
```

MS-STL version 202502

just as ever

with hardening

```
> cl test.cpp
> test.exe
hardened STL: false
v[3]  = 9673
v[-1] = -1811929271

// program ended with return value 0
```

```cpp
import std;

int main(int argc, char*[]) {
    std::println("hardened STL: {}", static_cast<bool>(_MSVC_STL_HARDENING));

    std::vector v(std::max(argc, 1), argc);
    std::println("v[3]  = {}", v[3]);
    std::println("v[-1] = {}", v[-1]);
}
```

MS-STL version 202502

just as ever

```
> cl test.cpp
> test.exe
hardened STL: false
v[3]  = 9673
v[-1] = -1811929271

// program ended with return value 0
```

with hardening

```
> cl test.cpp
> test.exe
hardened STL: true

// program execution stopped here
// with error code 0xC0000409
// 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  // somewhere in the program source
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  // somewhere in the program source
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  // somewhere in the program source
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 }
```

```
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 💥🛡️) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

```cpp
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 ✴🛡) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

```cpp
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 ✴ 🛡 ) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

```cpp
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 ✴ 🛡 ) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

```cpp
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 ✴ 🛡) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

```cpp
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 💥🛡️) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

```cpp
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 ✴ 🛡 ) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

```cpp
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 ✴🛡) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

```cpp
// somewhere in the program source

int func(int x)           // function parameters are *immutable* in pre/post assertions!
    pre (x != 1)          // a precondition assertion
    post(r : r != x)      // a postcondition assertion
                          //   'r' names the result object of 'func'
{
    // observable checkpoint (P1494 💥🛡️) right before evaluating the assert-argument
    contract_assert(x != 3);   // an assertion statement, e.g. check invariants
    // observable checkpoint right after returning from the handler in 'observe' semantics
    return x | 2;
}

void call_func() {
    func(0);    // no contract violation
    func(1);    // violates precondition assertion of 'func'
    func(2);    // violates postcondition assertion of 'func'
    func(3);    // violates assertion statement within 'func'
    func(4);    // no contract violation
}

// ------------------------------------------------------------------------------

// in global namespace, global module, invisible to users

// called on any contract violation
// system-provided or optionally user-replaceable

void handle_contract_violation(std::contracts::contract_violation cvo) {
    // handle contract violation according to
    //   -  the evaluation semantics as indicated in 'cvo'
    //   -  the rest of the info in 'cvo'
}
```

CONTRACT EVALUATION SEMANTICS

Semantics	check condition	fail → invoke handler	terminate
ignore (zero overhead)	❌	❌	❌
observe	✅	✅ [1]	❌
enforce	✅	✅	✅ [2]
quick_enforce	✅	❌	✅ [3]

[1] runtime: proceed after calling the handler / 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] runtime: proceed after calling the handler / 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


SAFE C++ ?

P3081

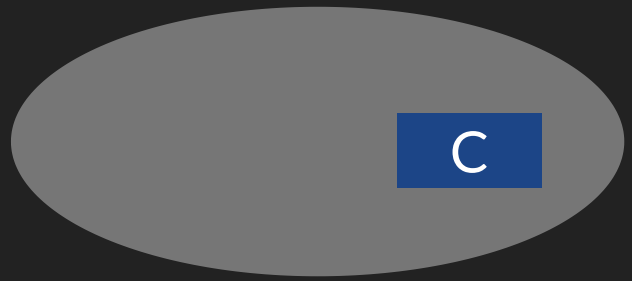
Core safety profiles for C++26

P3081

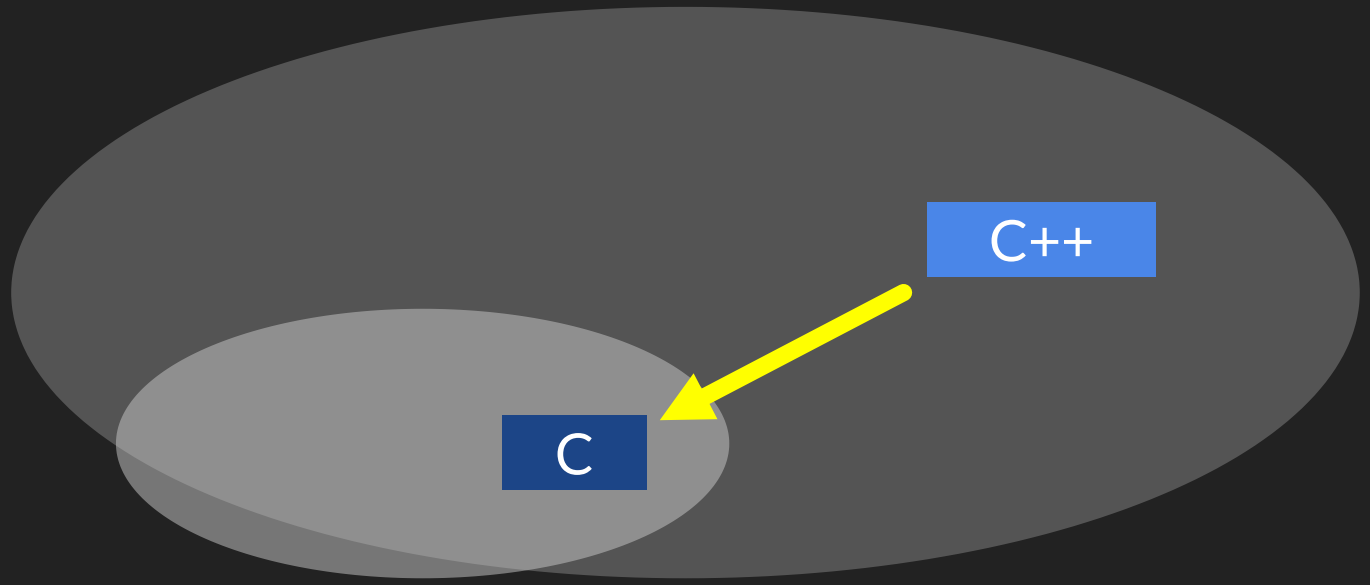
Core safety profiles for C++26

Not in C++26 ! 😬

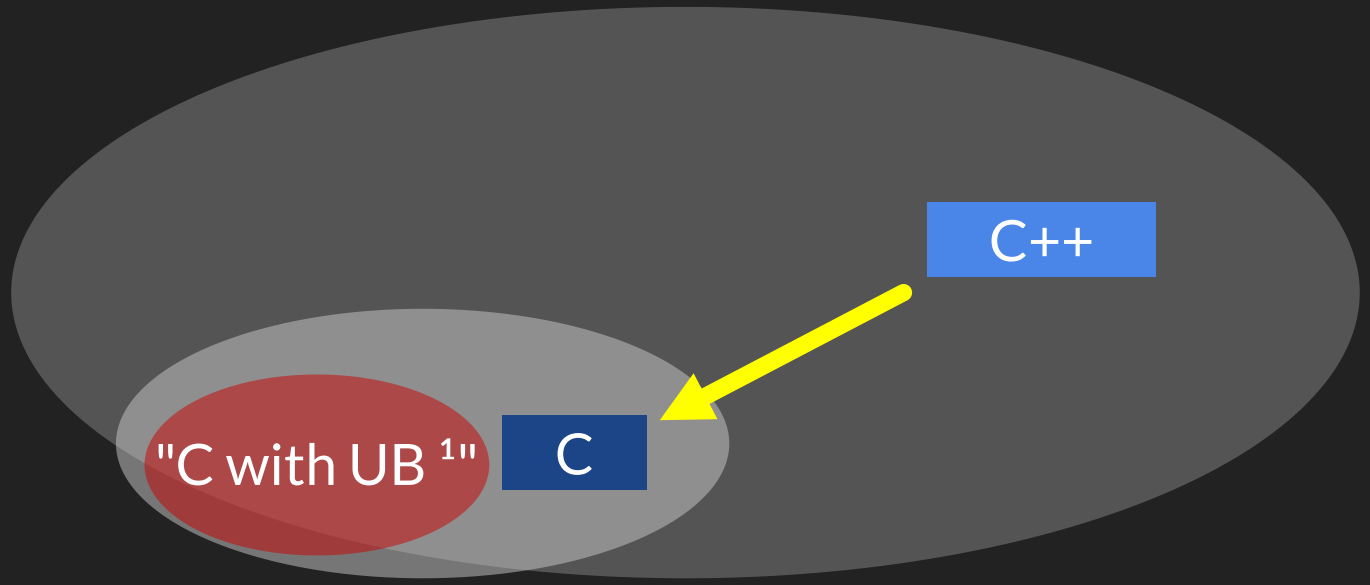
THE IDEA



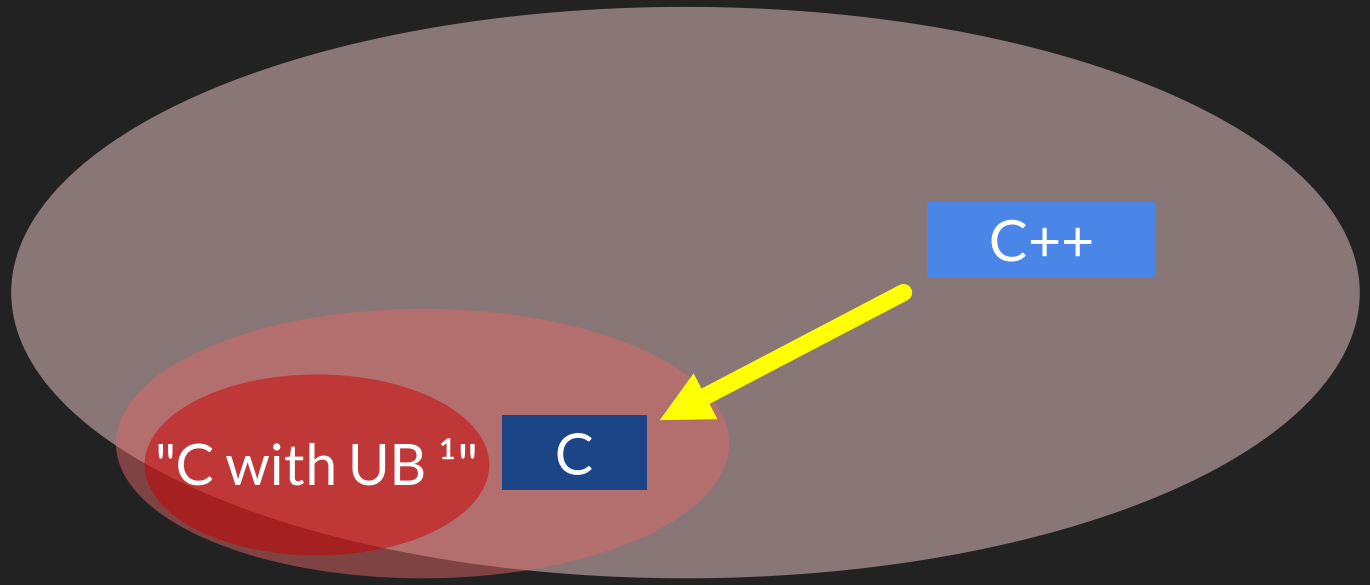
THE IDEA



THE IDEA



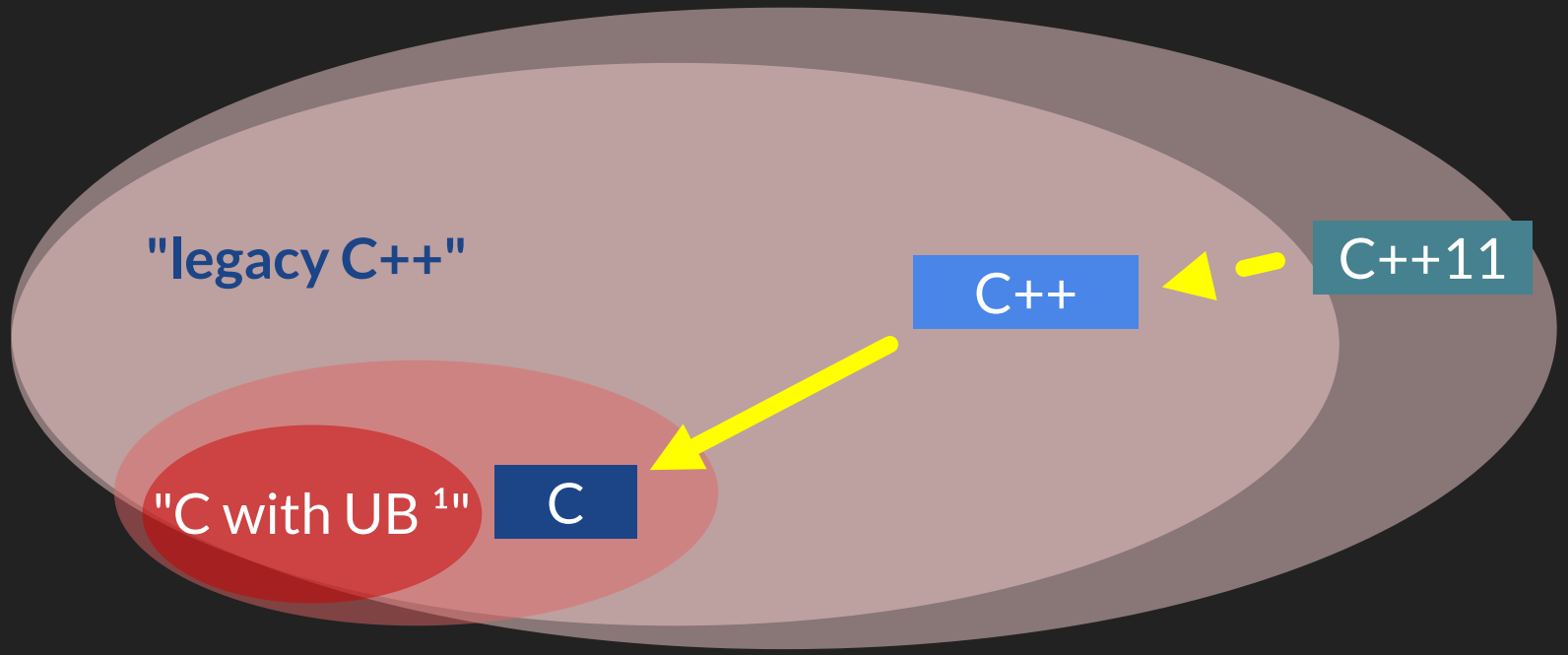
THE IDEA



[1] WG14 N3453: C contains

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

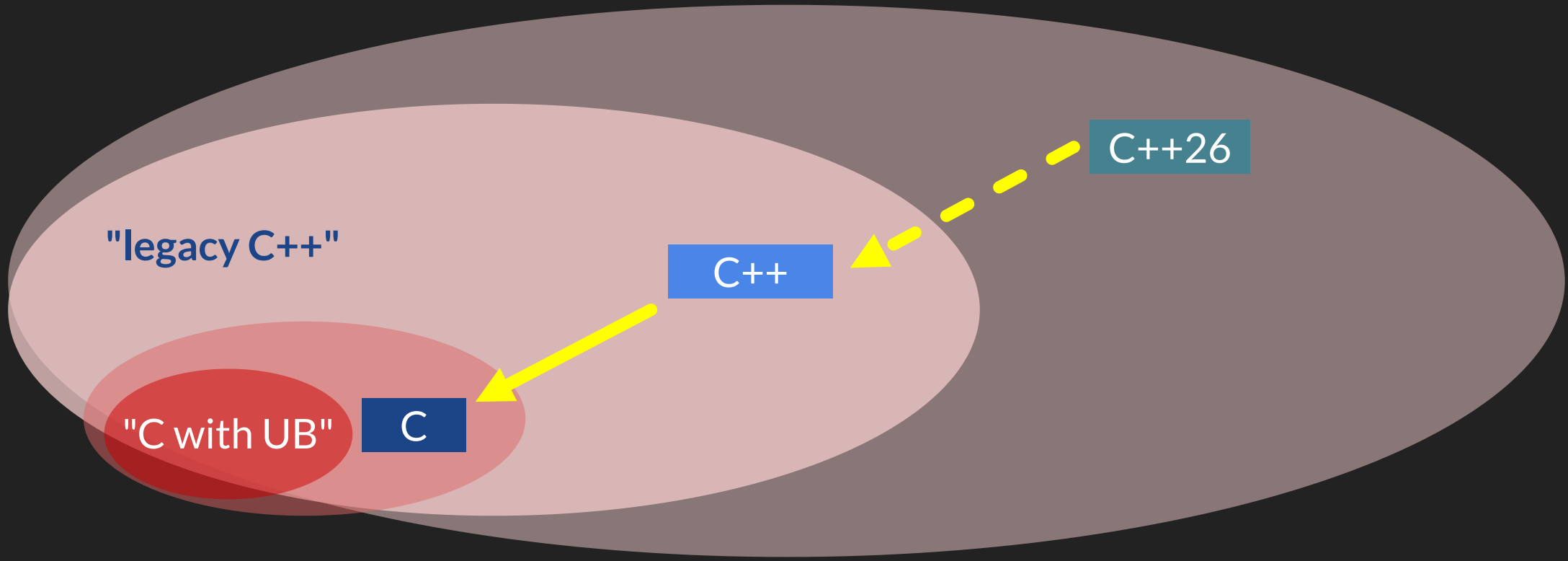
THE IDEA



[1] WG14 N3453: C contains

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

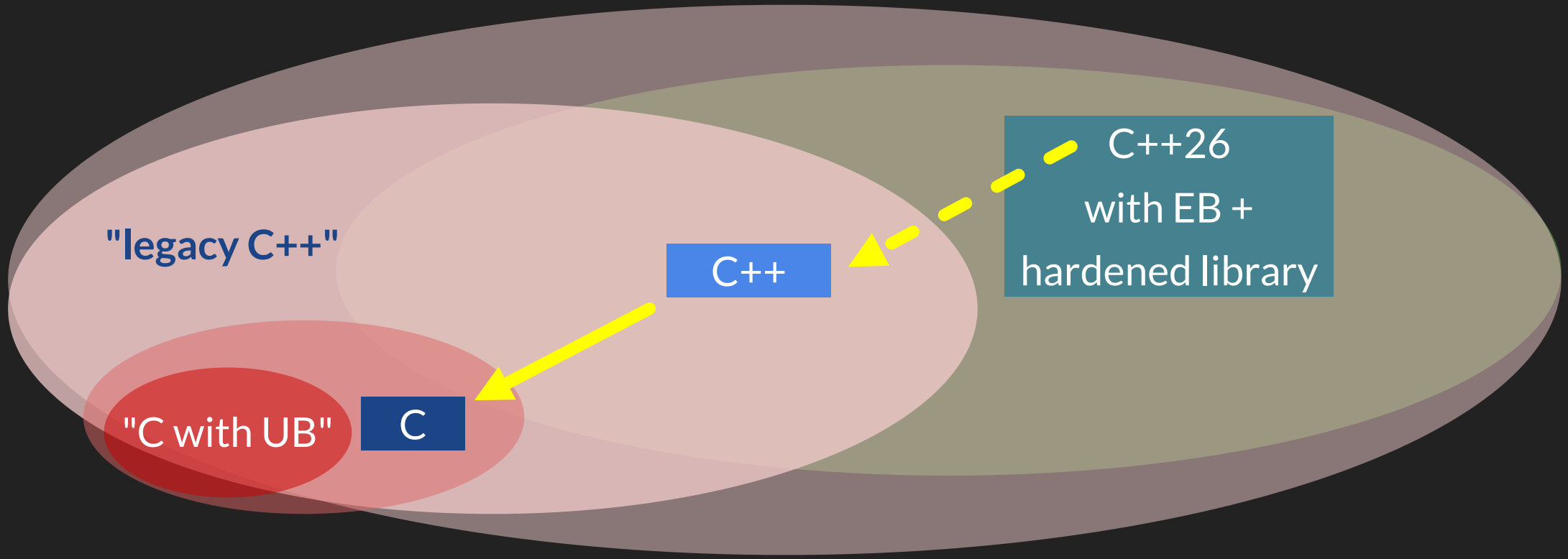
THE IDEA



Inherited UB taints the entire language

C++ adds its own UB

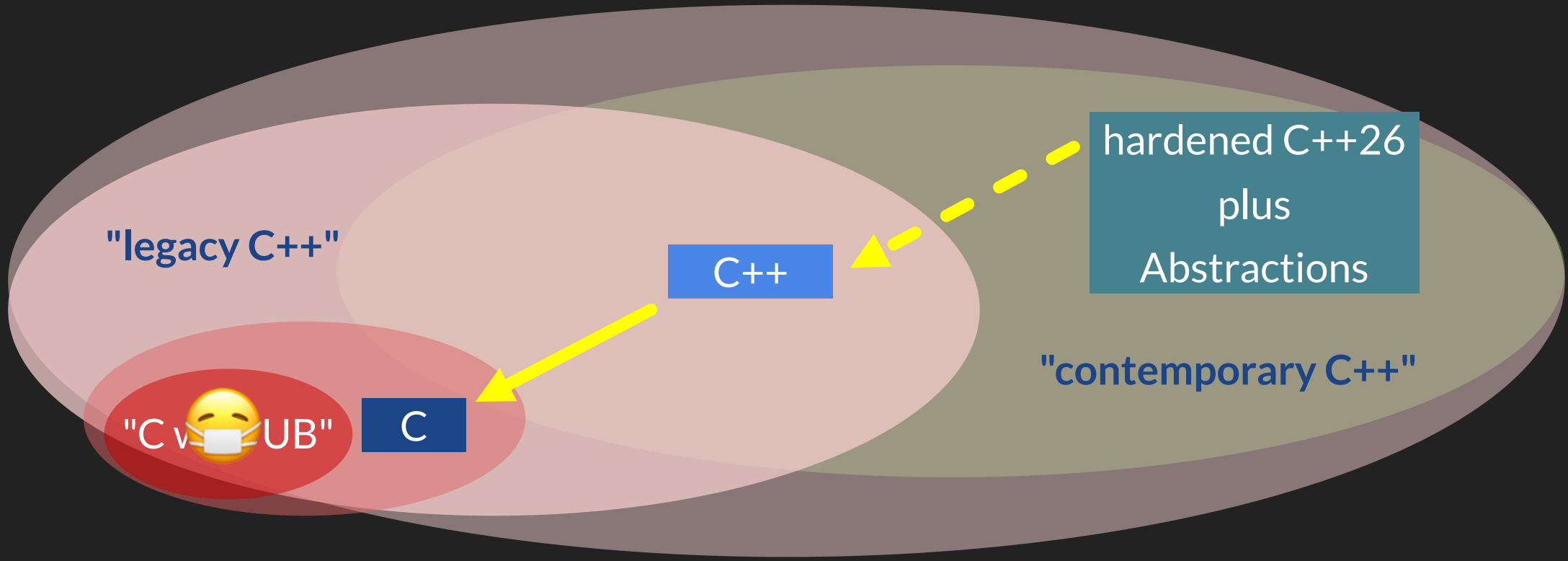
THE IDEA



Inherited UB taints the entire language

C++ adds its own UB

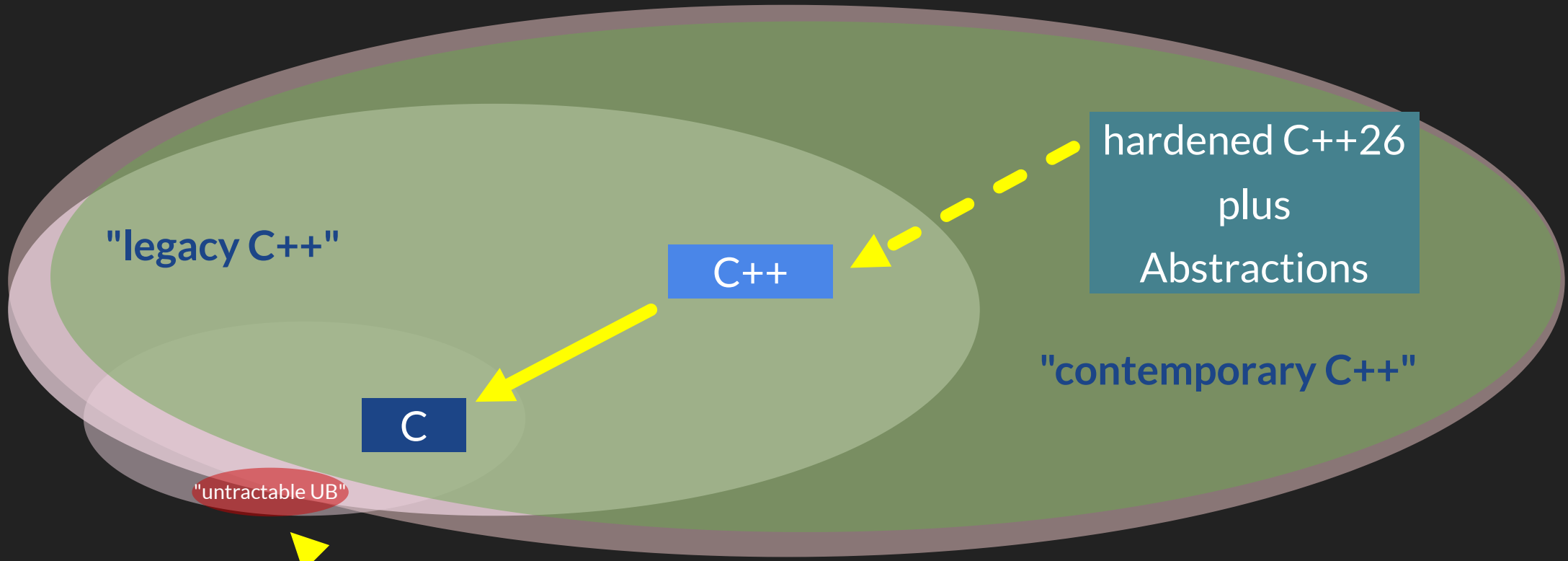
THE IDEA



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

This will be pursued in a Whitepaper along with EB, Contracts, Profiles

SUMMARY

- why this perceived push towards security?
- 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 Committee Draft)
- 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

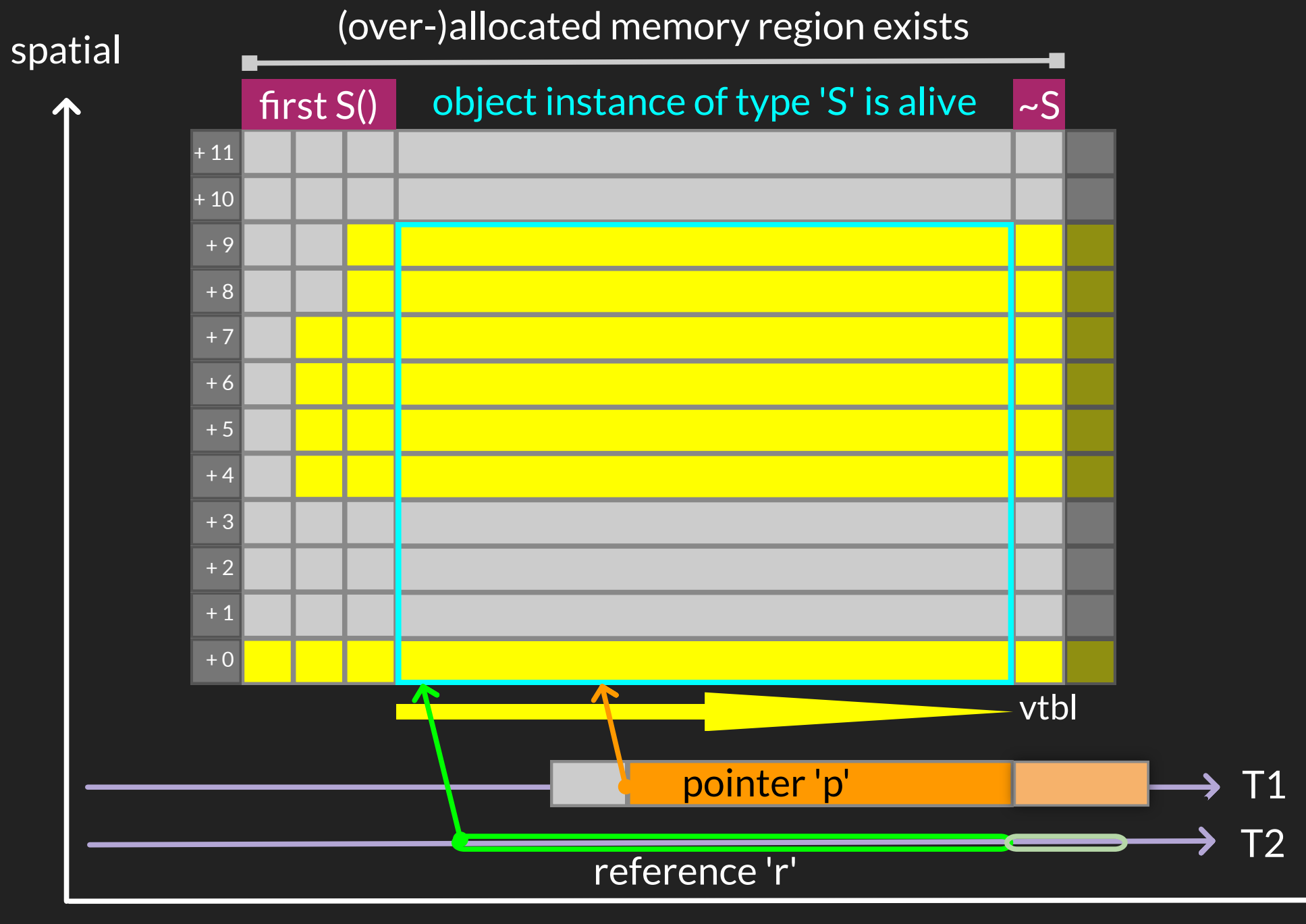
DanielaE

Images: courtesy of Matúš Chochlík

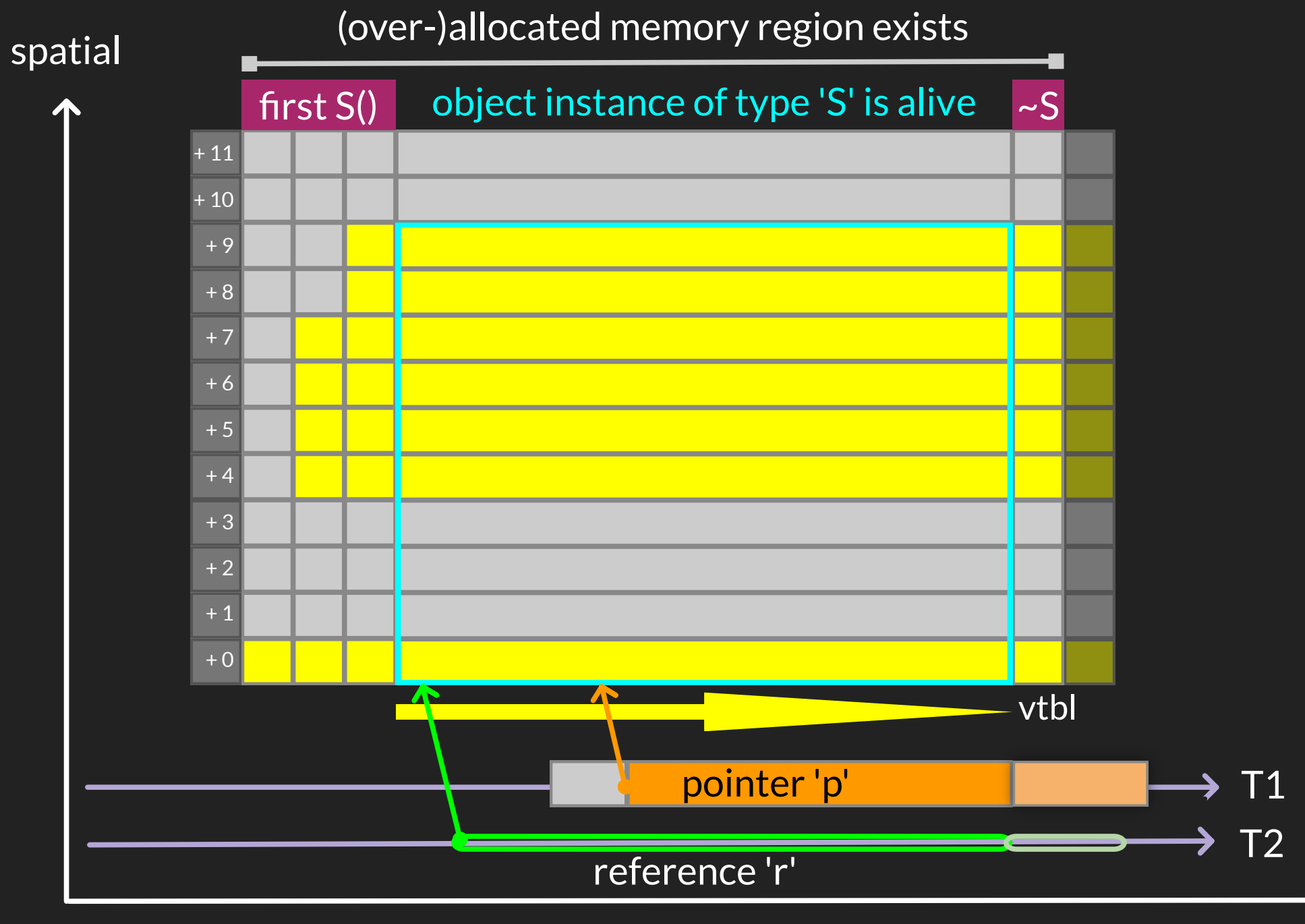
source: WikiMedia Commons, public domain plus Unsplash & GIPHY


Ceterum censeo ABI esse frangendam

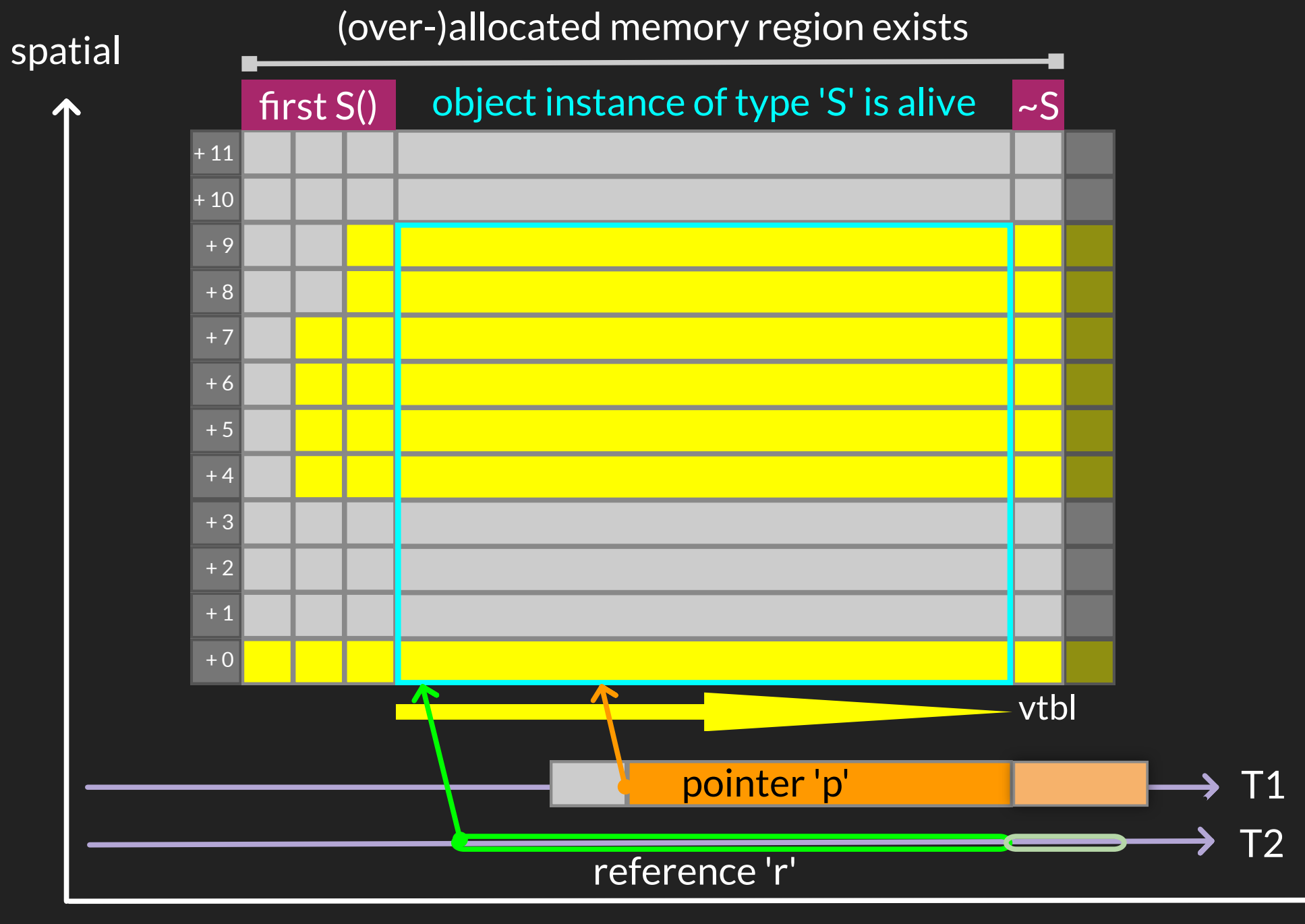

DEFINITIONS



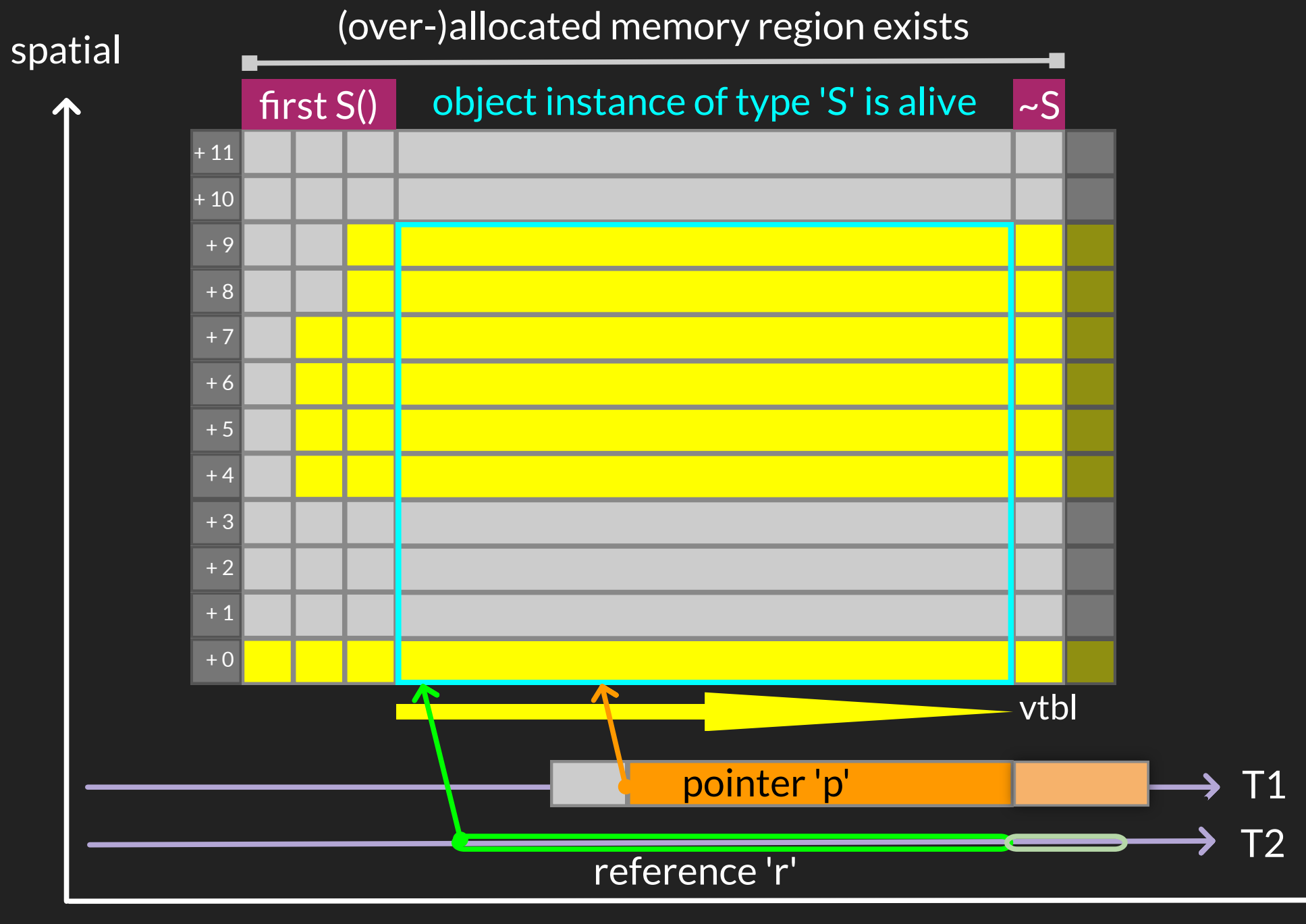

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



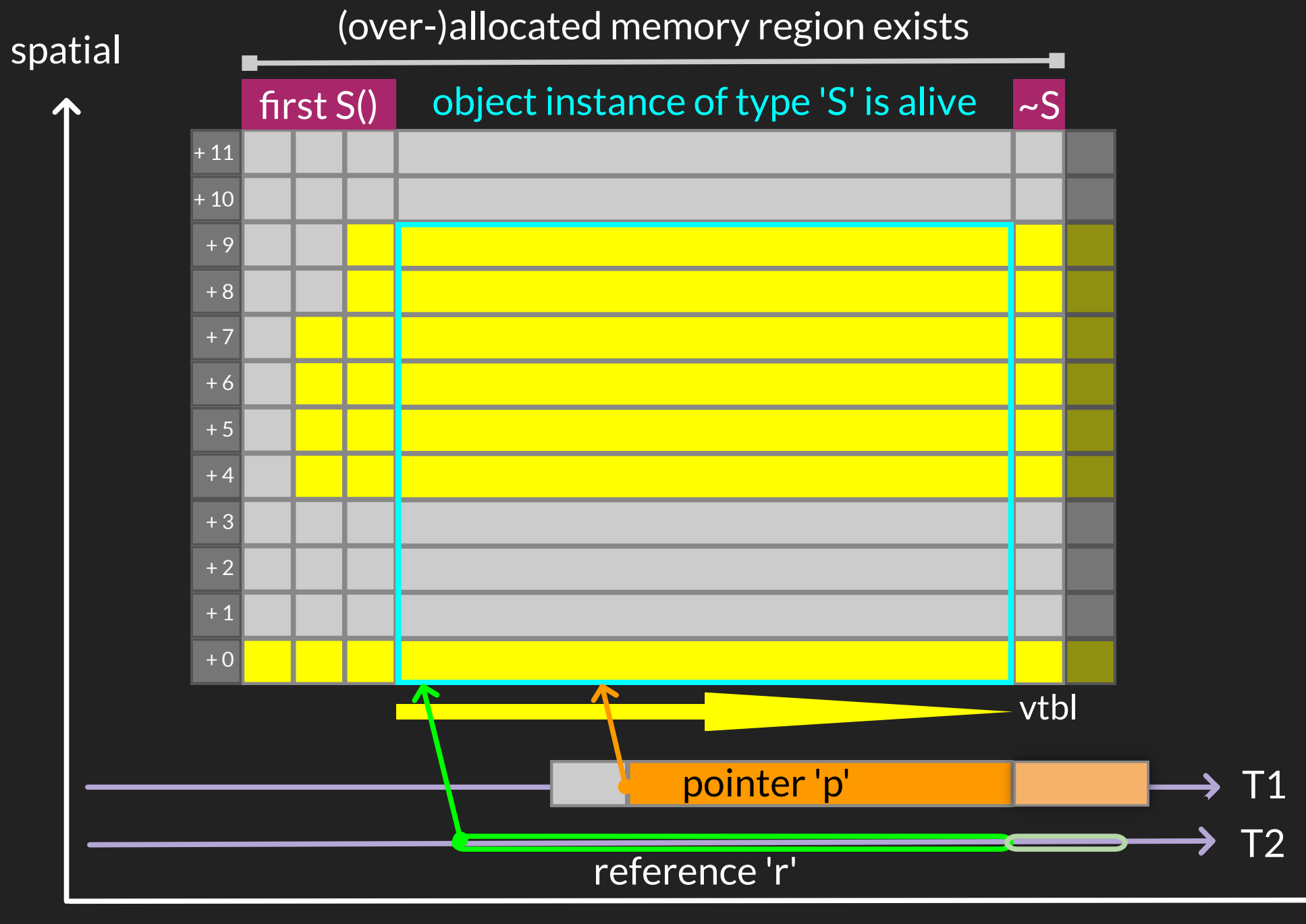

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



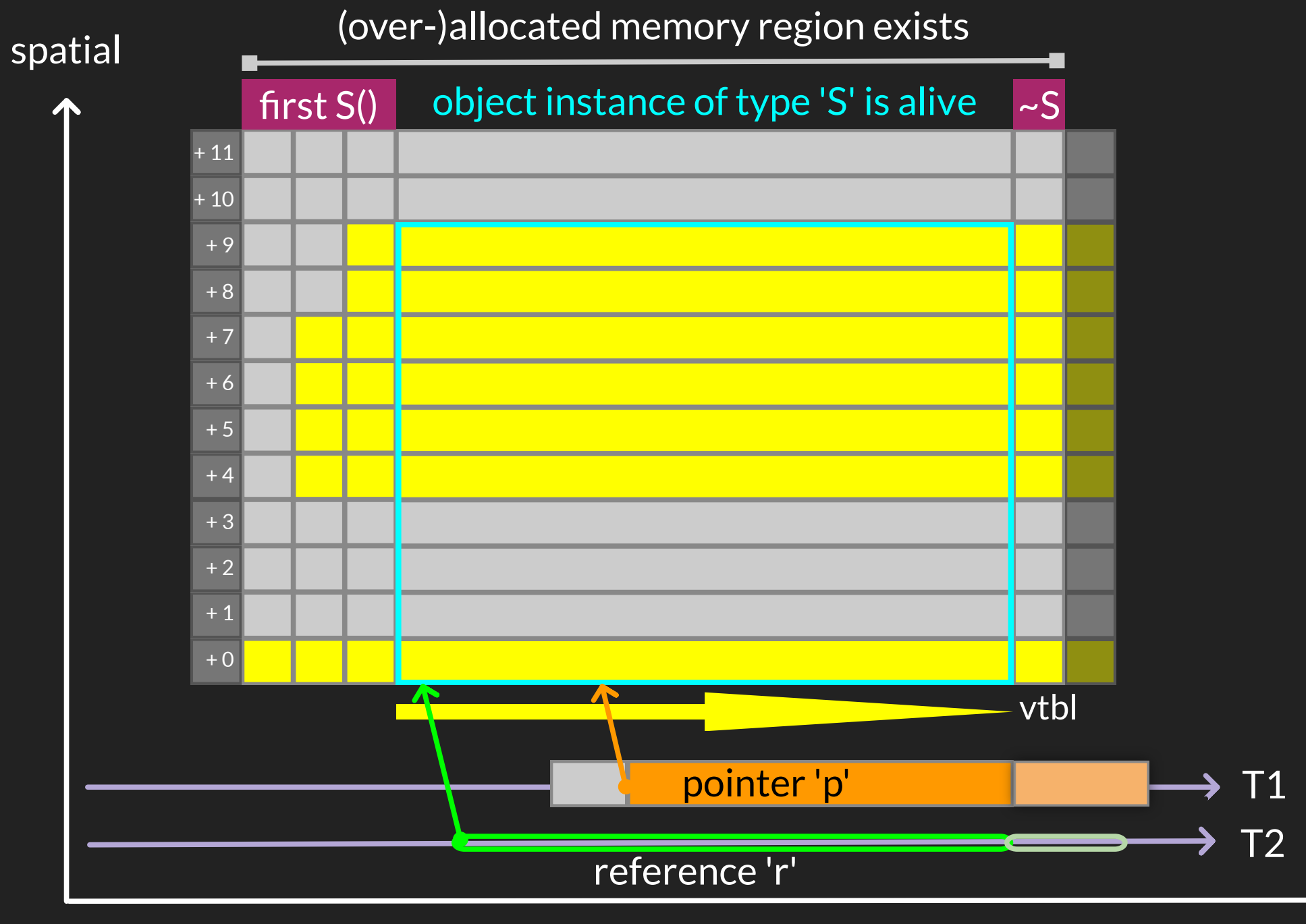

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



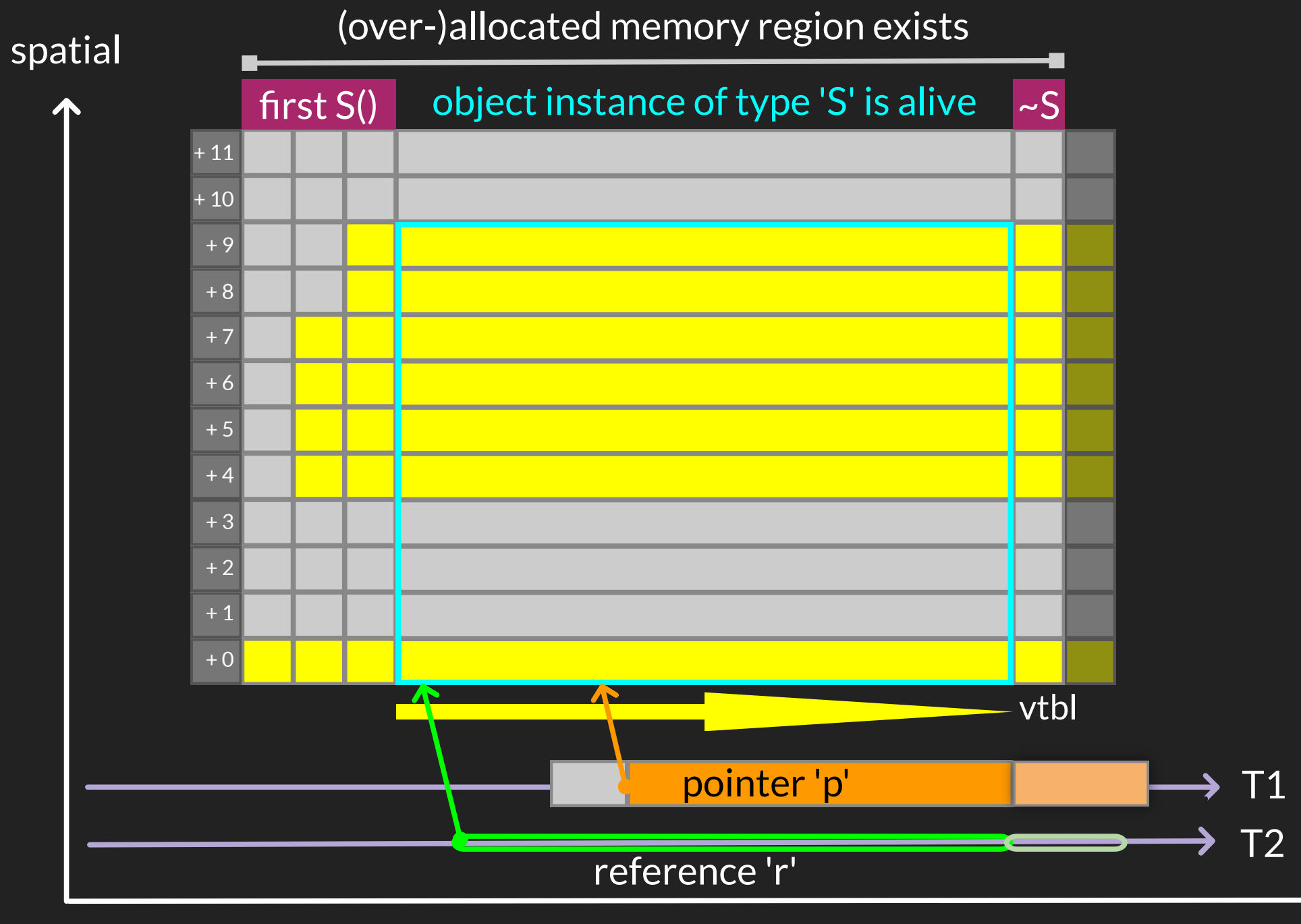

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



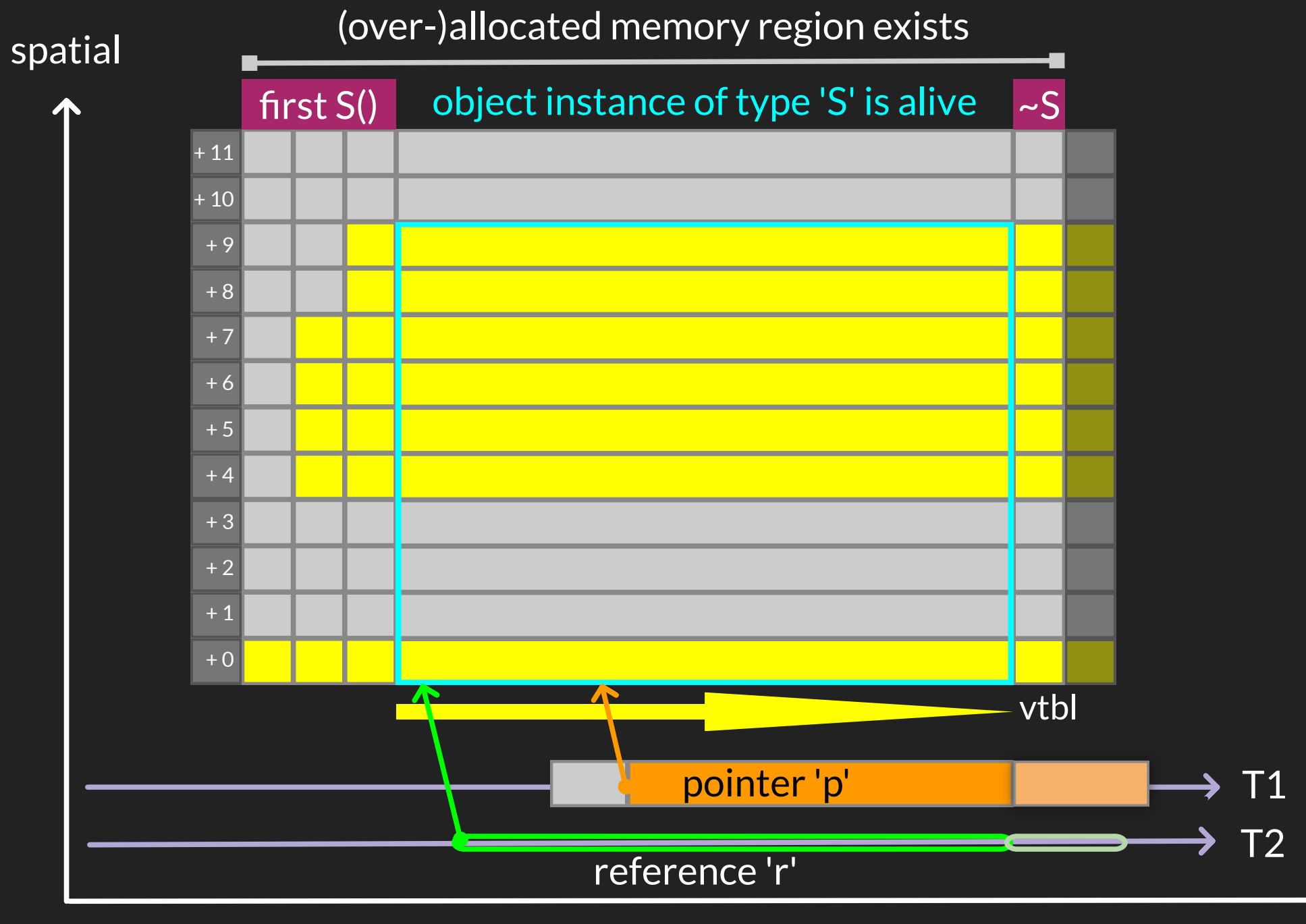

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



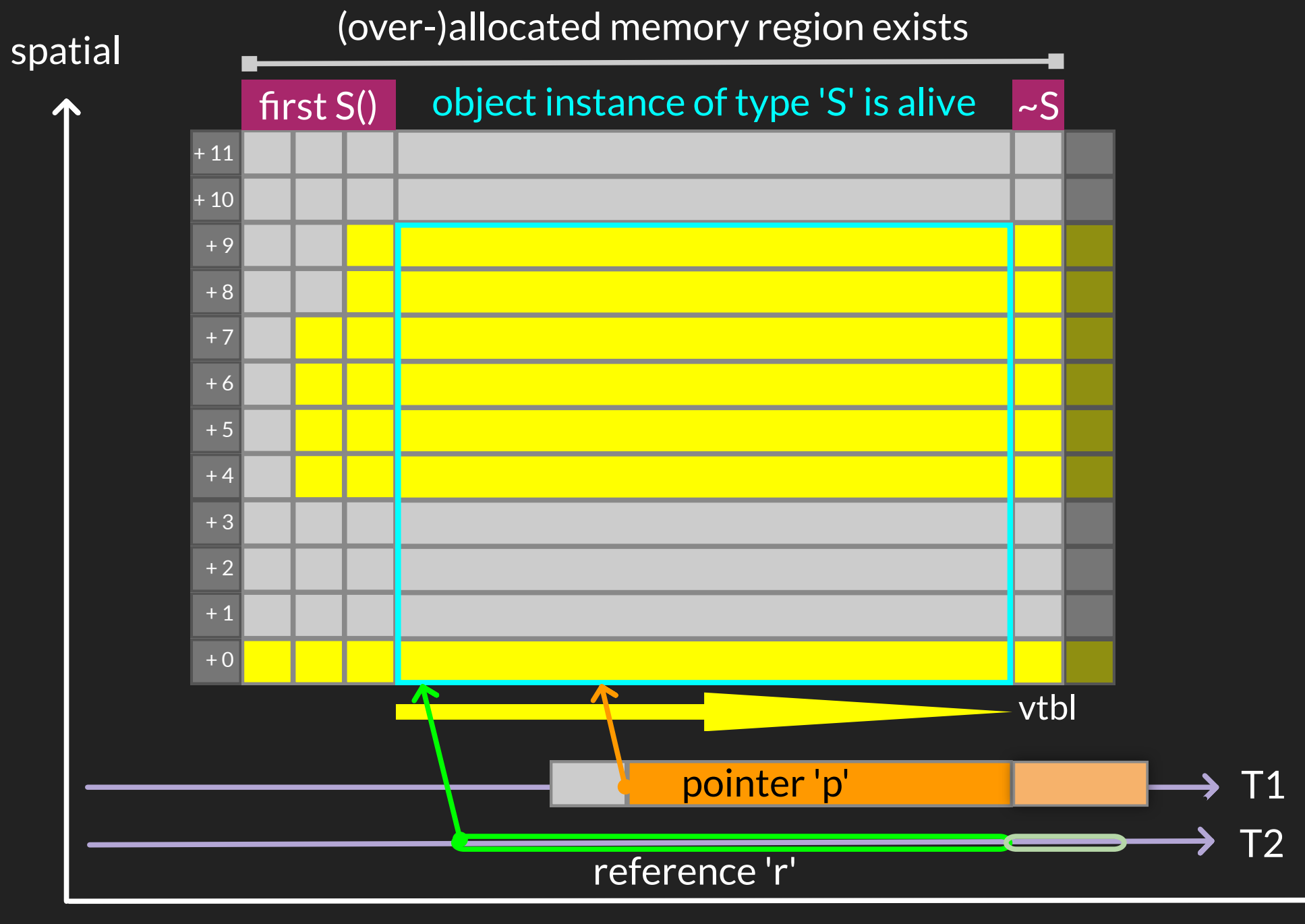

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



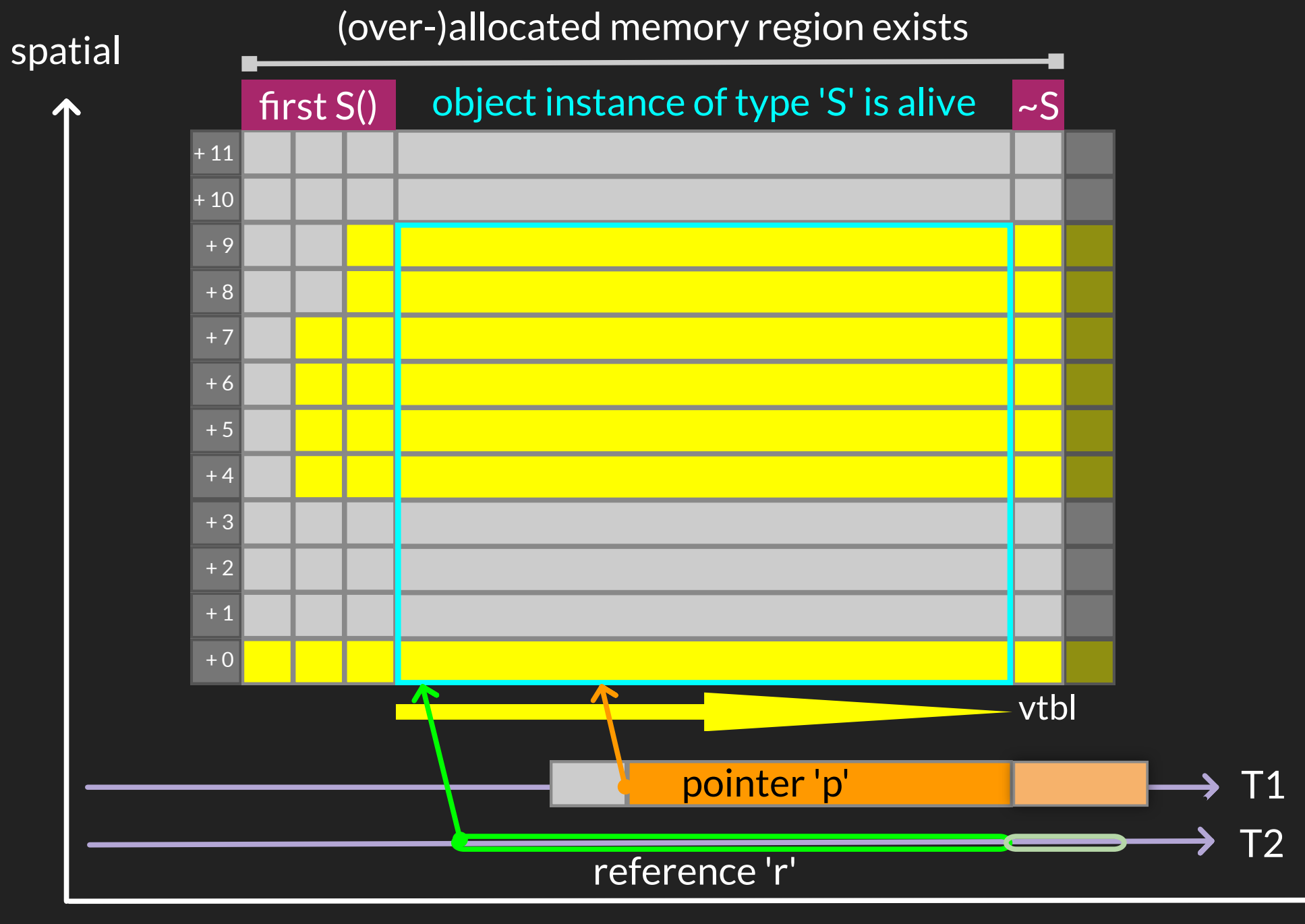

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



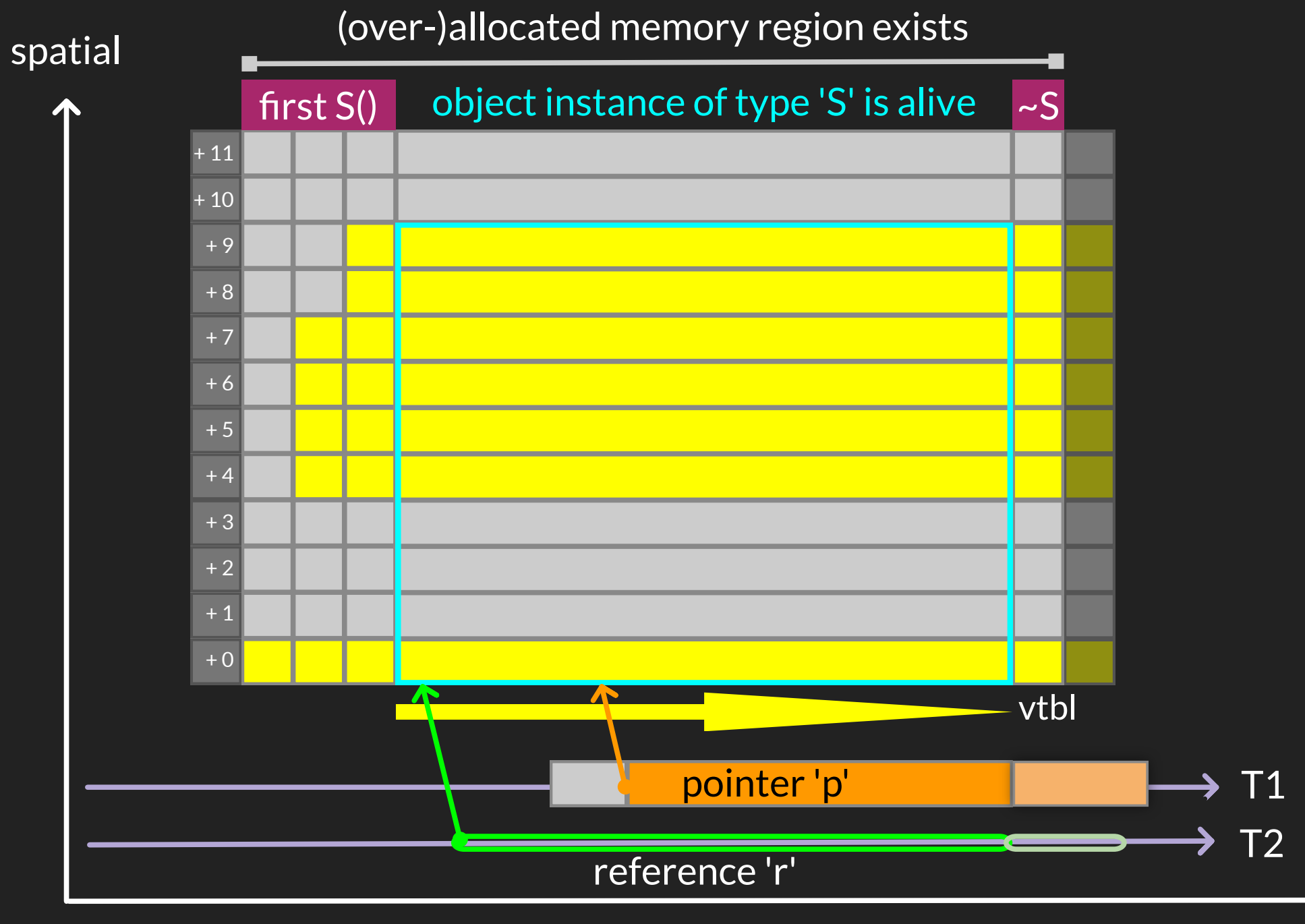

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



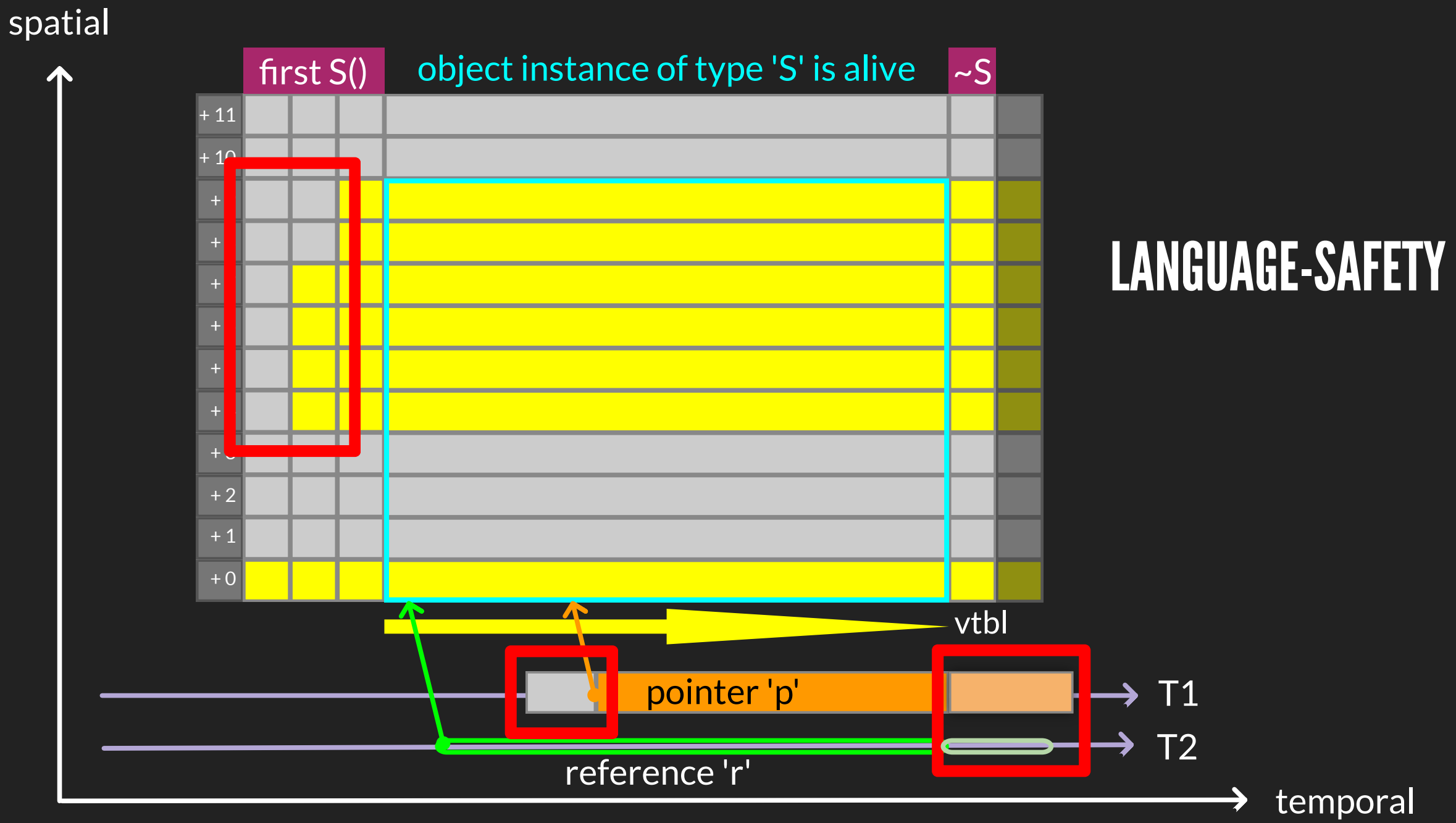

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```



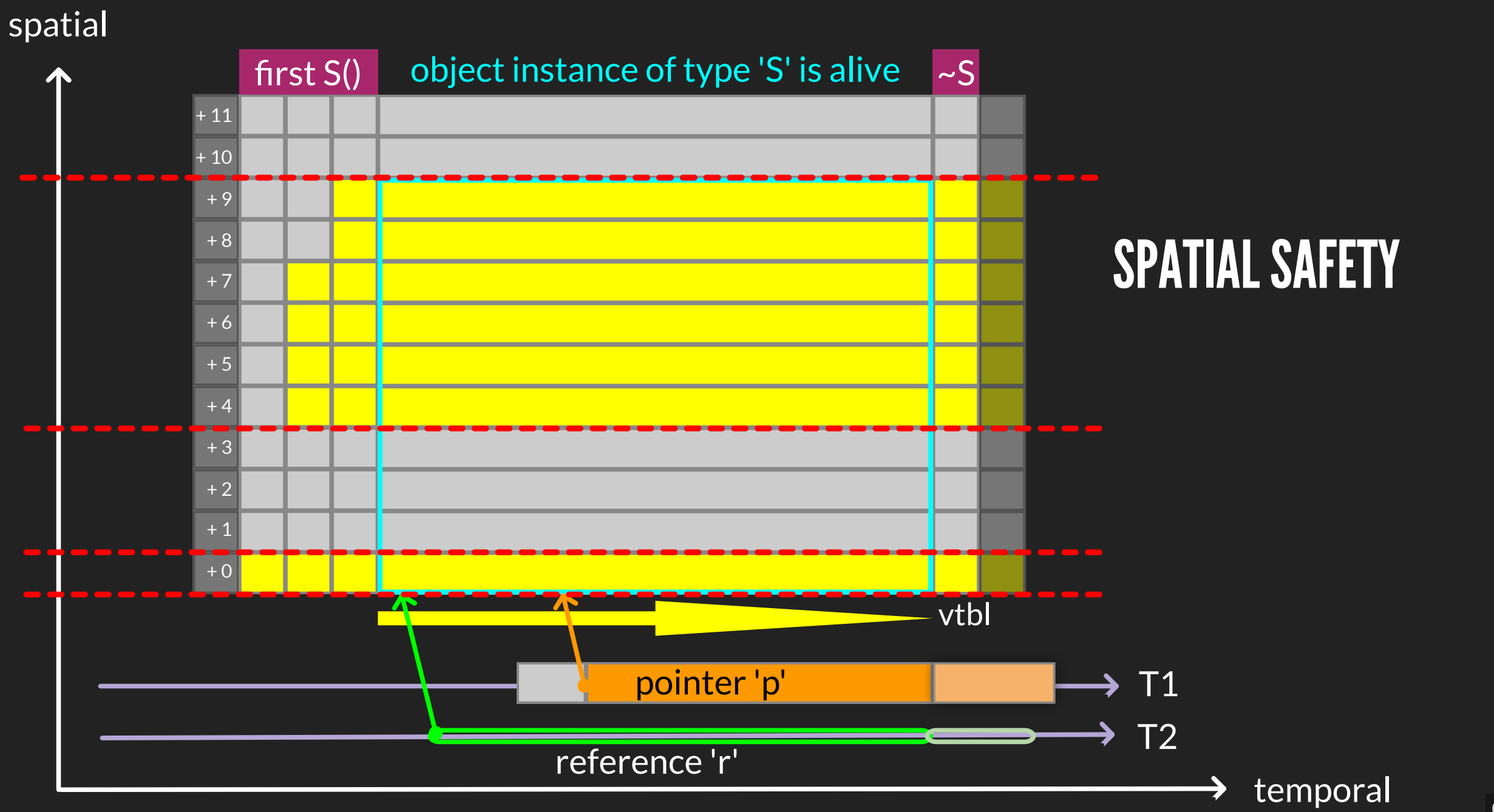

```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```

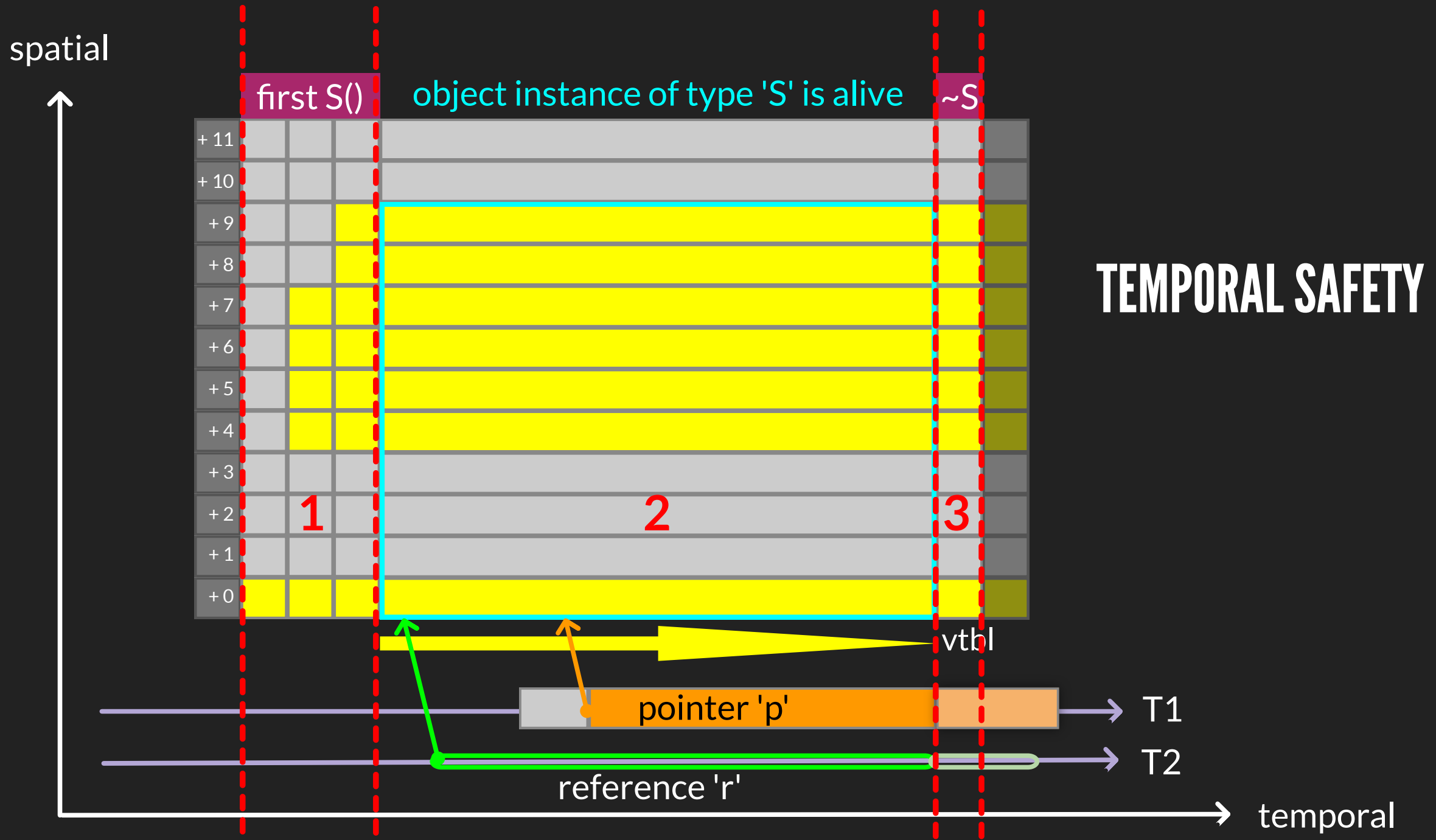



```
1  struct S {
2    S();
3    virtual ~S();
4
5    char    a = 1;
6    int32_t b = 2;
7    int16_t c = 3;
8  };
9
10 auto _ = new S;
11
12 void Thread1 {
13   S * p;
14   ...
15   p = _;
16   ...
17 }
18
19 void Thread2 {
20   ...
21   S & r = *_;
22   ...
23 }
```

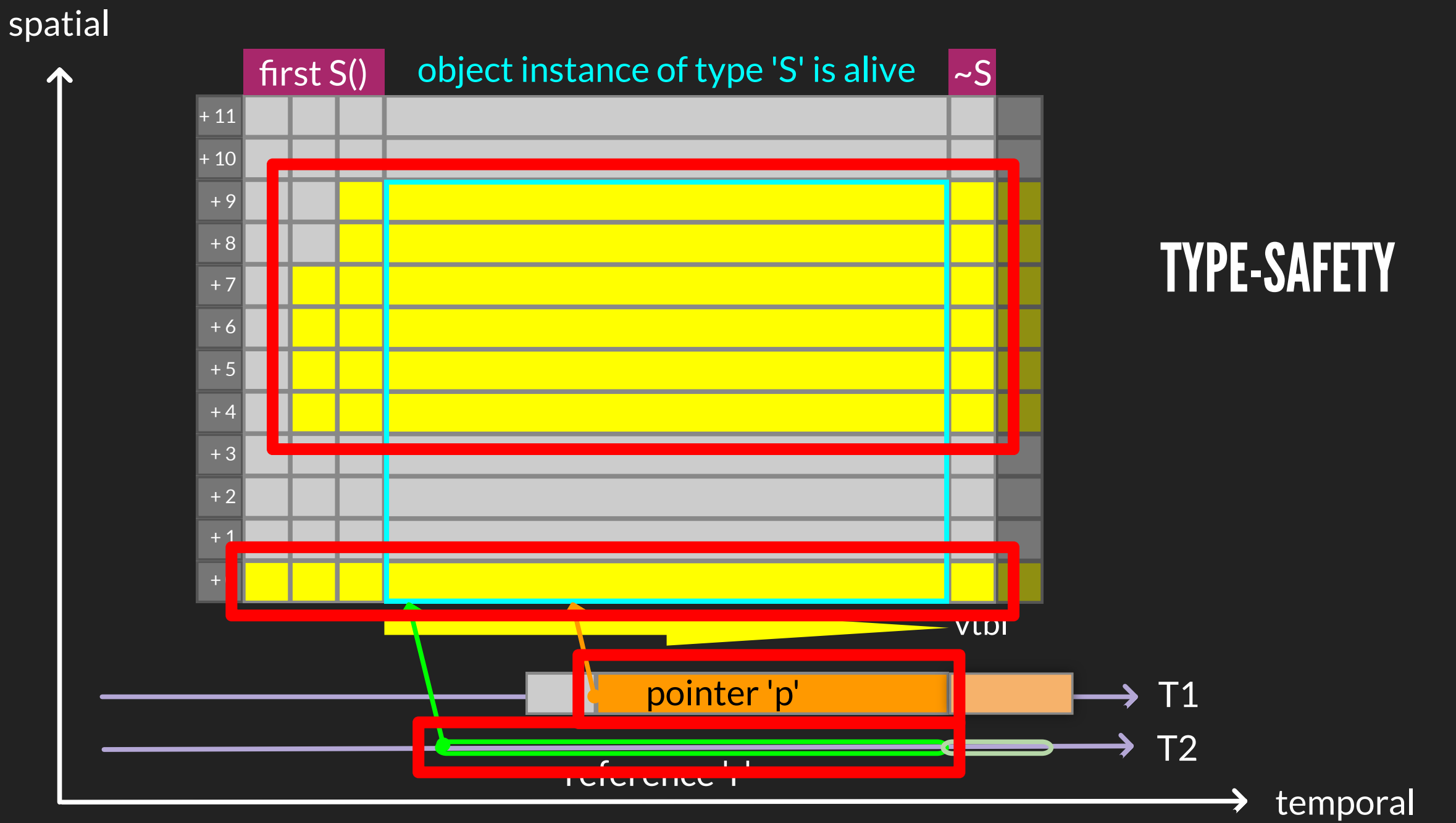


LANGUAGE-SAFETY

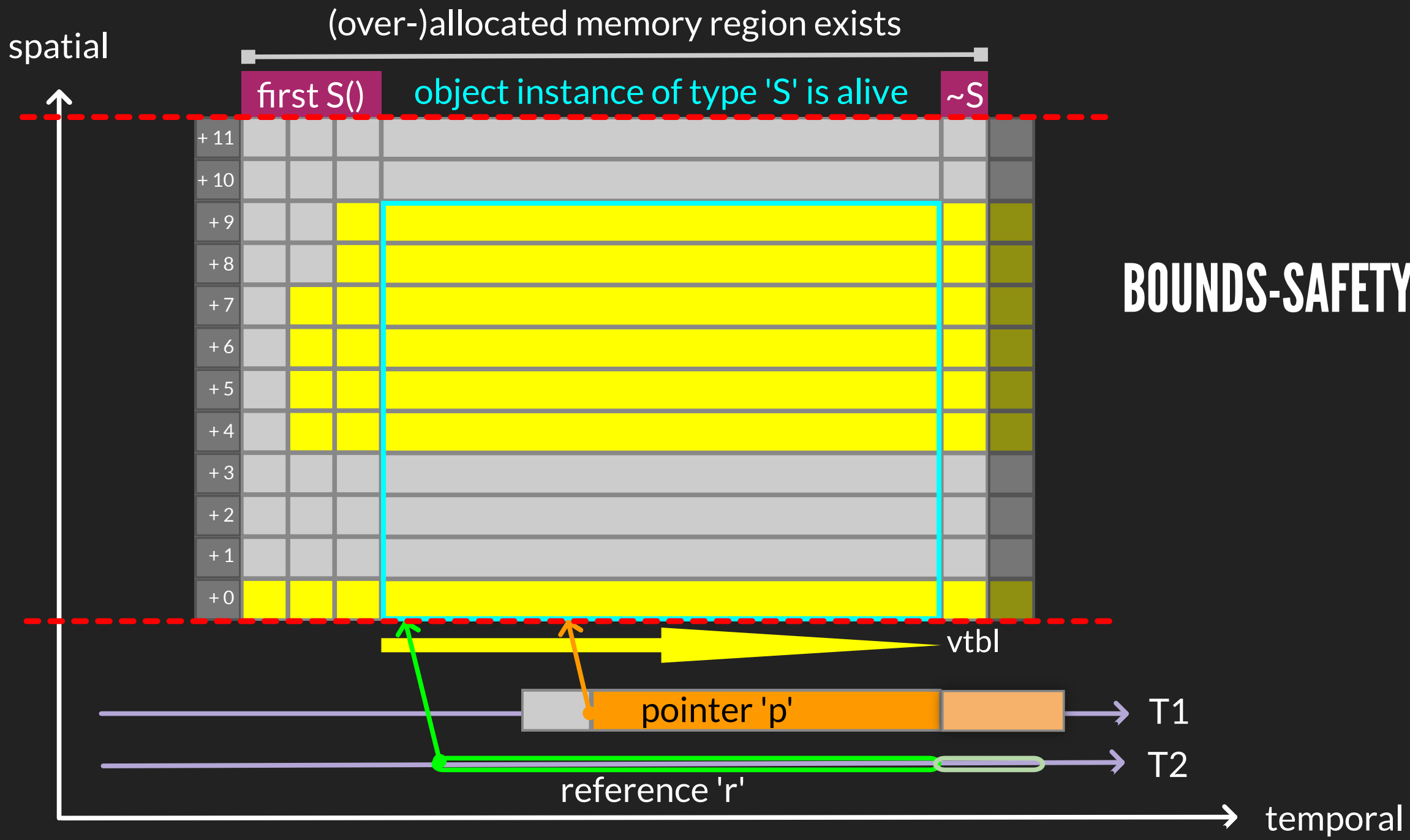

spatial
first S()
object instance of type 'S' is alive
~S
+ 11
+ 10
+ 9
+ 8
+ 7
+ 6
+ 5
+ 4
+ 3
+ 2
+ 1
+ 0
SPATIAL SAFETY
vtbl
pointer 'p'
T1
T2
reference 'r'
temporal



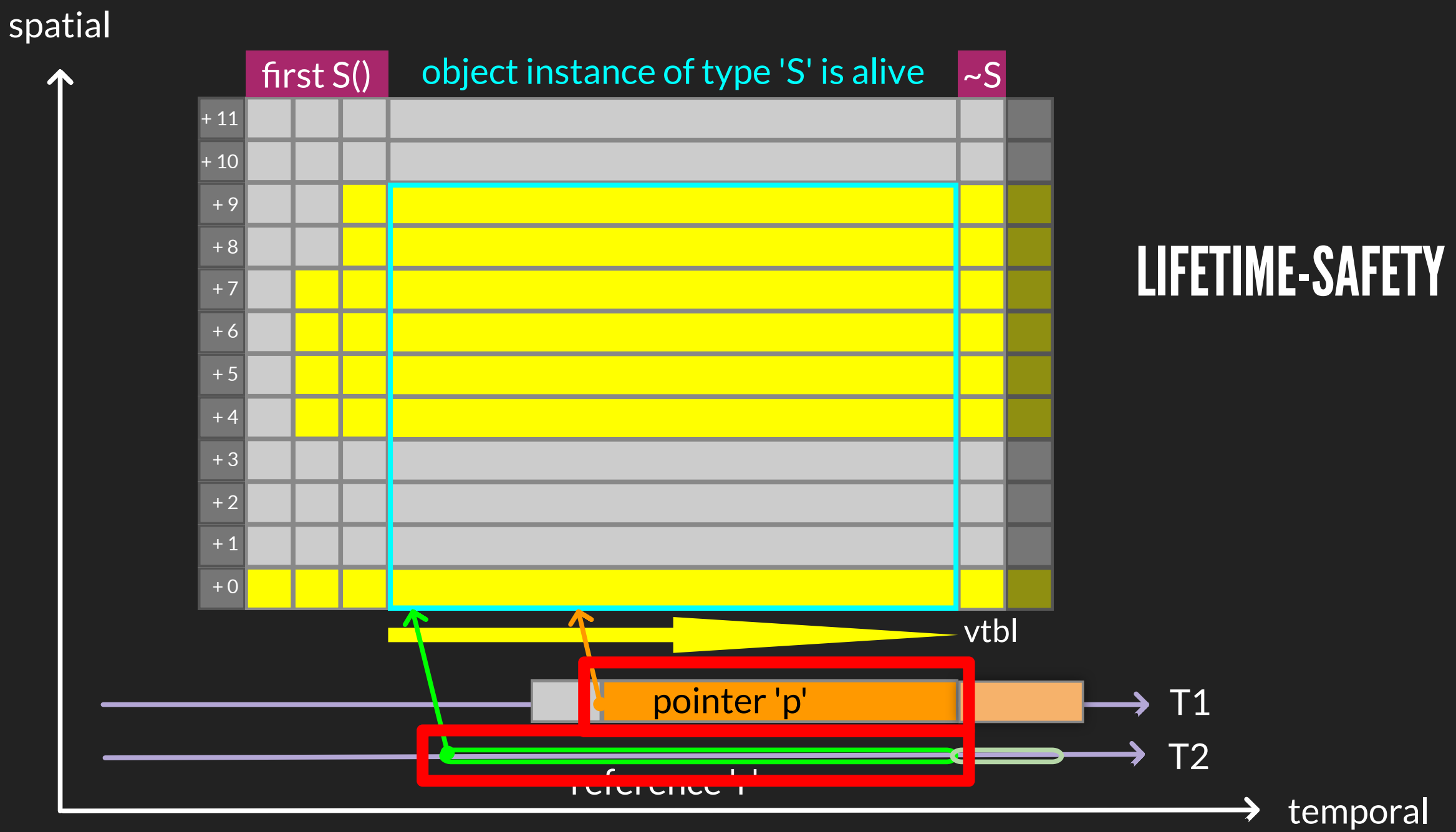
TEMPORAL SAFETY


spatial
first S()
object instance of type 'S' is alive
~S
+ 11
+ 10
+ 9
+ 8
+ 7
+ 6
+ 5
+ 4
+ 3
+ 2
+ 1
vtbl
pointer 'p'
T1
T2
temporal

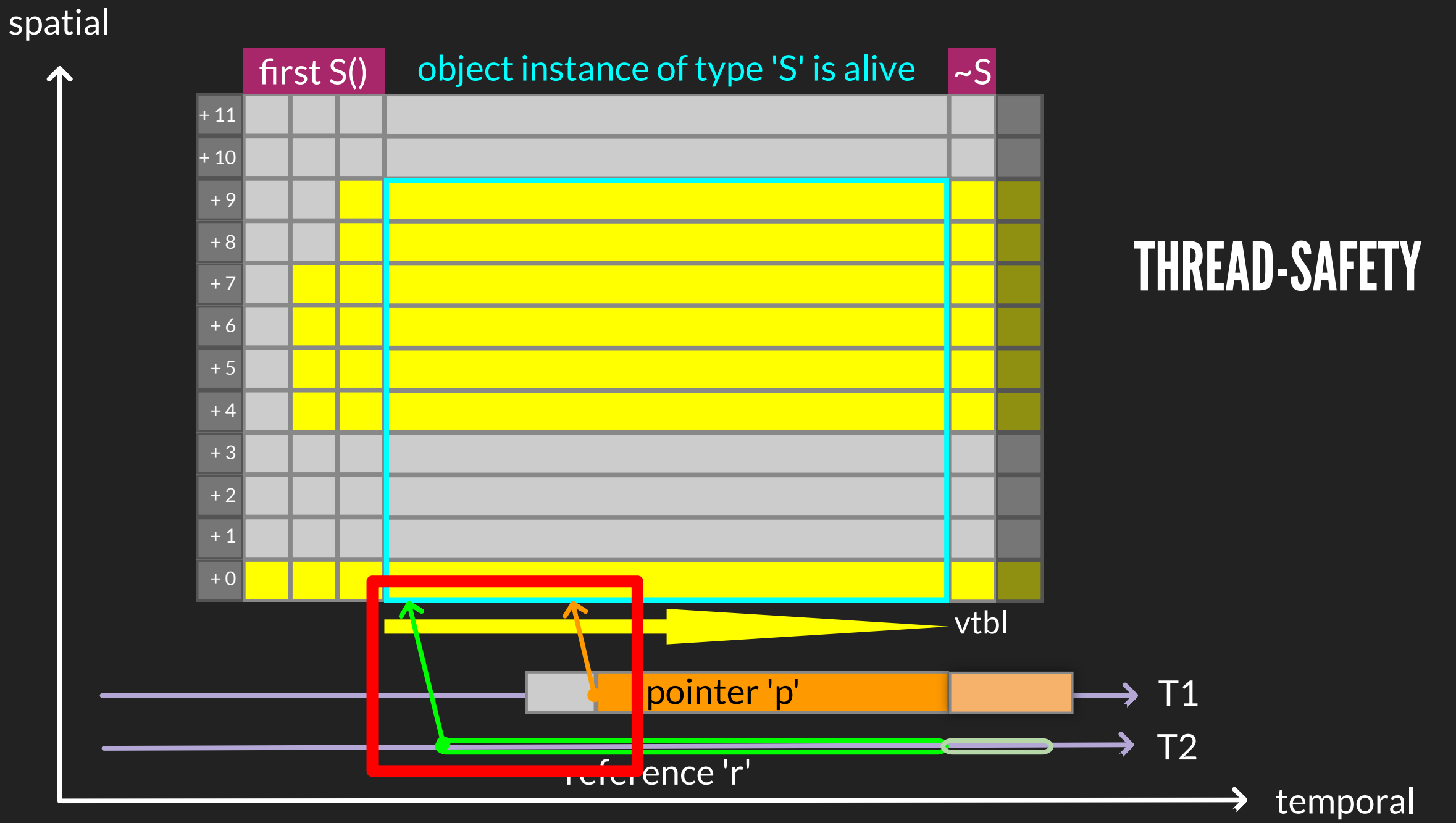
TYPE-SAFETY



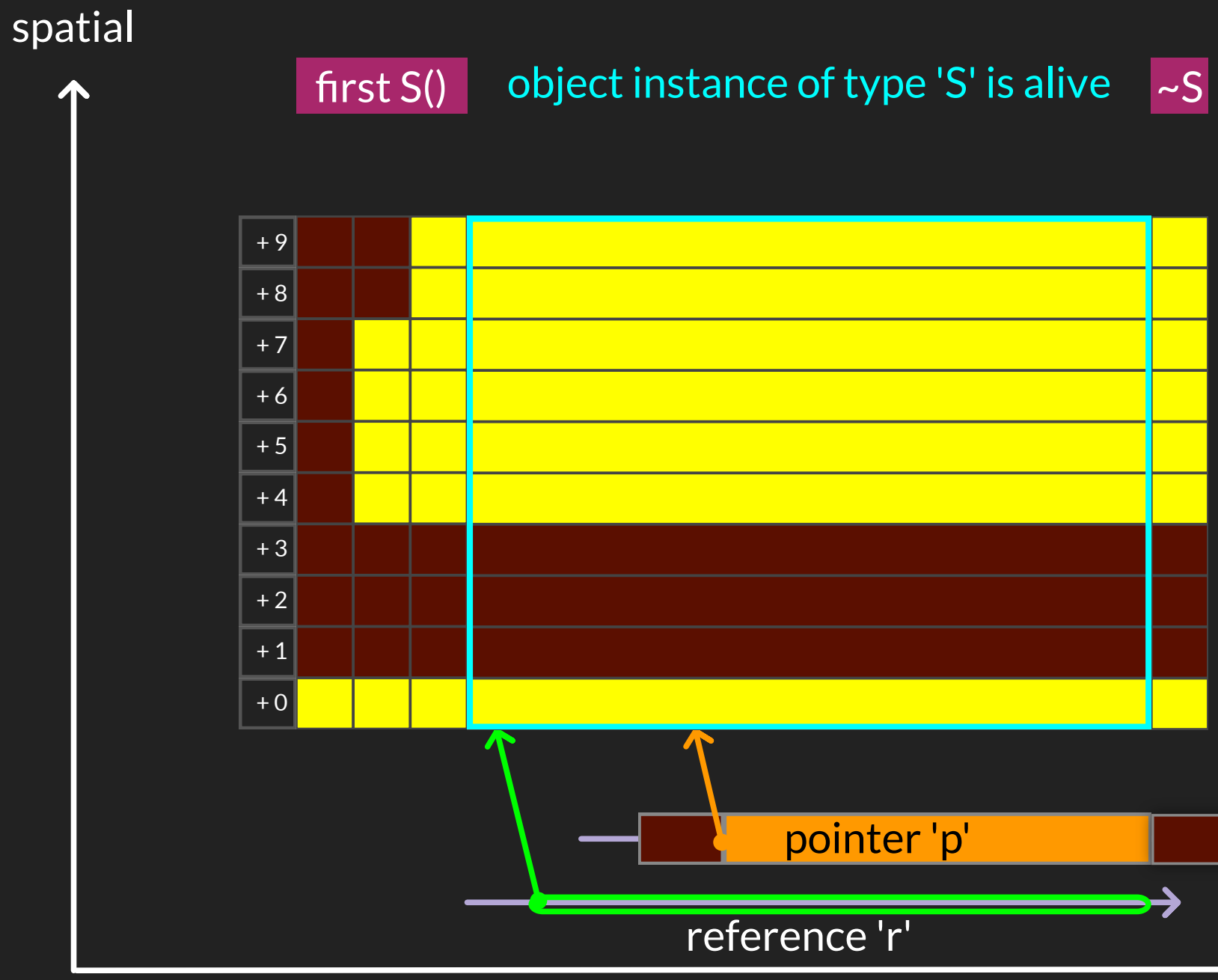
BOUNDS-SAFETY



LIFETIME-SAFETY



THREAD-SAFETY




```
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 = _;
}
```