Engineering Better Software at Microsoft

Jason Yang
jasony@microsoft.com

Principal Development Lead
Windows Engineering Desktop
Microsoft Corporation

Who we are

Windows Engineering Desktop – Analysis Technologies Team

Develops and supports some of the most critical compile-time program analysis tools and infrastructures used at Microsoft.

Source-level

- PREfix ← global analyzer
- PREfast/Esp ← local analyzer
- SAL ← source code annotation

Binary-level

- Vulcan ← binary instrumentation
- Magellan ← code coverage

A primer on SAL

An introduction to program analysis

A glimpse at the engineering process in Windows

good APIs + annotations + analysis tools

↓

significantly fewer code defects

3,631,361 *

* number of annotations in Windows alone

more secure and reliable products

Why SAL?

Manual Review
too many code paths to think about

Massive Testing
inefficient detection of simple programming errors

Global Analysis
long turn-around time

Local Analysis
lack of calling context limits accuracy

SAL
light-weight specifications make implicit intent explicit

Evolution of Source Code Annotation Language (SAL)

2002
SAL 1.0
Focuses on buffer overrun

2003
PREfast & PREfix
Starts to support annotations

2004
EspX
Buffer overrun checker deployed for Windows Vista

2005
VS 2005
SAL aware compiler shipped with Visual Studio

2007
EspC
Concurrency checker deployed for Windows 7

2007
PFD
PREfast for Drivers shipped with DDK

2010
SAL 2.0
Improves coverage and usability

SAL

For industrial strength C/C++

Tailored for compile-time analysis

Target critical problem areas

```
_Post_ _Notnull_ void *
foo(_Pre_ _Notnull_ int *p)
{ … }
```

vs.

C0 Contracts

For a subset of C

Current enforcement entirely based on runtime analysis

May handle full functional specification

```
void * foo(int *p)
//@requires p != NULL;
//@ensures \result != NULL;
{ … }
```

```
_Pre_satisfies_(p>q) ←→ //@requires p>q;
_Post_satisfies_(p>q) ←→ //@ensures p>q;
```

What do these functions do?

```
void * memcpy(
    void *dest,
    const void *src,
    size_t count
);

wchar_t *wmemcpy(
    wchar_t *dest,
    const wchar_t *src,
    size_t count
);
```

memcpy, wmemcpy

msdn

Visual Studio 2010 | Other Versions

Copies bytes between buffers. More secure versions of these functions are available; see memcpy_s, wmemcpy_s.

Copy

```
void *memcpy(
   void *dest,
   const void *src,
   size_t count
);
wchar_t *wmemcpy(
   wchar_t *dest,
   const wchar_t *src,
   size_t count
);
```

Remarks

memcpy copies count bytes from src to dest; wmemcpy copies count wide characters (two bytes). If the source and destination overlap, the behavior of memcpy is undefined. Use memmove to handle overlapping regions.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

Remarks

memcpy copies count bytes from src to dest; wmemcpy copies count wide characters (two bytes). If the source and destination overlap, the behavior of memcpy is undefined. Use memmove to handle overlapping regions.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

For every buffer API there's usually a wide version.
Many errors are confusing "byte" vs. "element" counts.

Remarks

memcpy copies count bytes from src to dest; wmemcpy copies count wide characters (two bytes). If the source and destination overlap, the behavior of memcpy is undefined. Use memmove to handle overlapping regions.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

For every buffer API there's usually a wide version.
Many errors are confusing "byte" vs. "element" counts.

Remarks

memcpy copies count bytes from src to dest; wmemcpy copies count wide characters (two bytes). If the source and destination overlap, the behavior of memcpy is undefined. Use memmove to handle overlapping regions.

Security Note Make sure that the destination buffer is the same size or larger than the source buffer. For more information, see Avoiding Buffer Overruns.

Vital property for avoiding buffer overrun.

SAL speak

```
void * memcpy(
     _Out_writes_bytes_all_(count) void *dest,
     _In_reads_bytes_(count) const void *src,
     size_t count
);

wchar_t *wmemcpy(
     _Out_writes_all_(count) wchar_t *dest,
     _In_reads_(count) const wchar_t *src,
     size_t count
);
```

- ✓ Captures programmer intent.
- ✓ Improves defect detection via tools.
- ✓ Extends language types to encode program logic properties.

Precondition: function can assume `p` to be non-null when called.

```
_Post_ _Notnull_ void * foo(_Pre_ _Notnull_ int *p);
```

Postcondition: function must ensure the return value to be non-null.

```
struct buf {
    int n;
    _Field_size_(n) int *data;
};
```

Invariant: property that should be maintained.

What: annotation specifies program property.

Where: `_At_` specifies annotation target.

```
_At_(ptr, _When_(flag != 0, _Pre _Notnull_))
void Foo(
    int *ptr,
    int flag);
```

When: `_When_` specifies condition.

Type

Types are used to describe the representation of a value in a given program state.

Enforced by compiler via type checking.

Each execution step in a type-safe imperative language preserves types, so types by themselves are sufficient to establish a wide class of properties without the need for program logic.

vs.

Program Logic

Program logic describes transitions between program states.

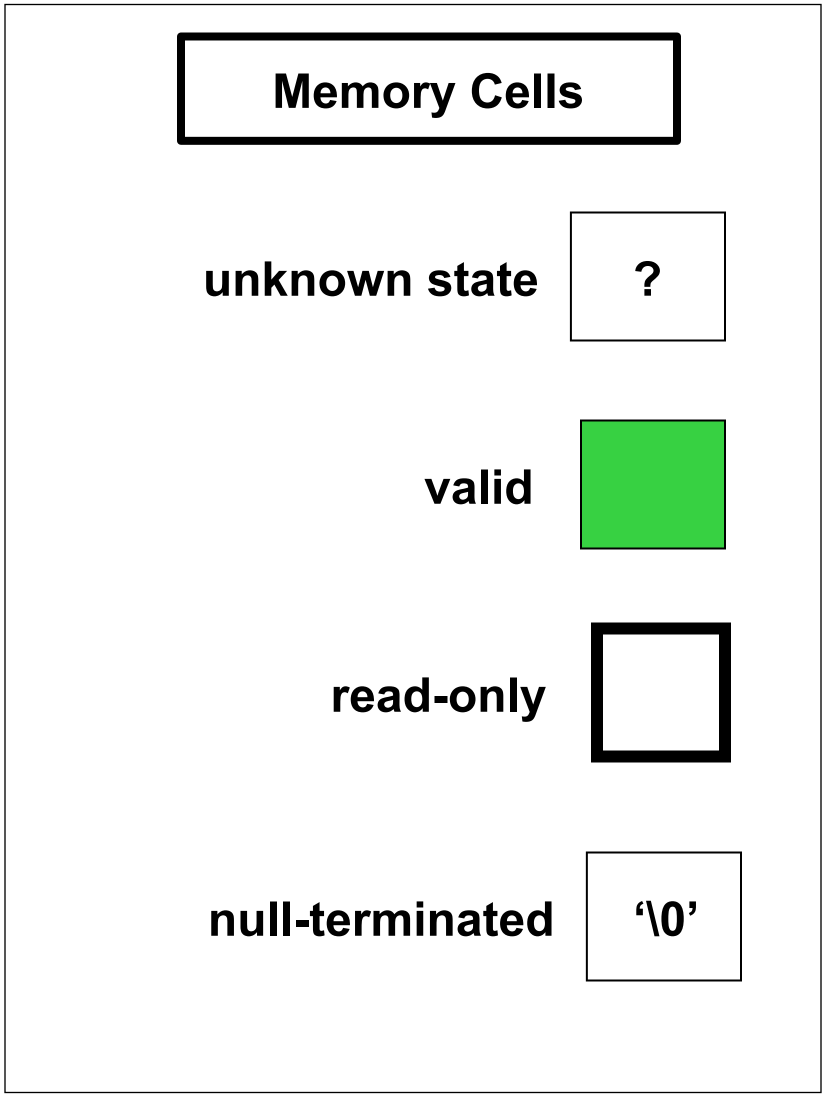
Programming errors can be detected by static analysis.

Types are often not descriptive enough to avoid errors because knowledge about program logic is often implicit.

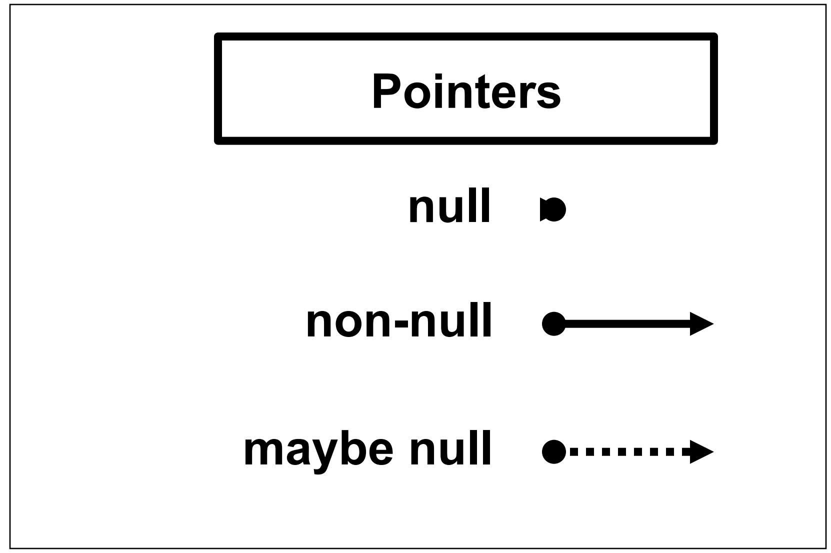
Memory cell semantics

unknown	?	Memory allocated and can be written to but nothing is known about its contents, for example the result of malloc() (that does not zero init the returned buffer)
valid		Object has a “well-formed” value: initialized + type specific invariants (if any)
read-only		Memory is read-only
null-terminated	‘\0’	Buffer is null-terminated

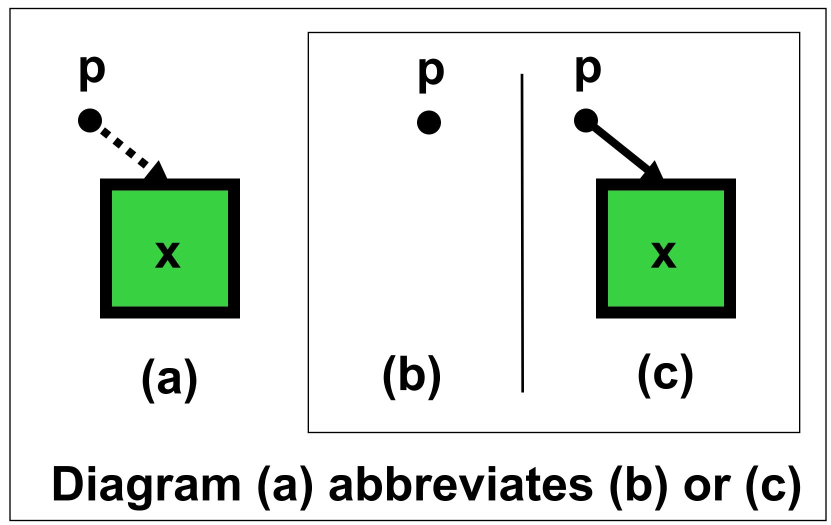
Legend



Memory Cells
unknown state
?
valid
read-only
null-terminated
'\0'

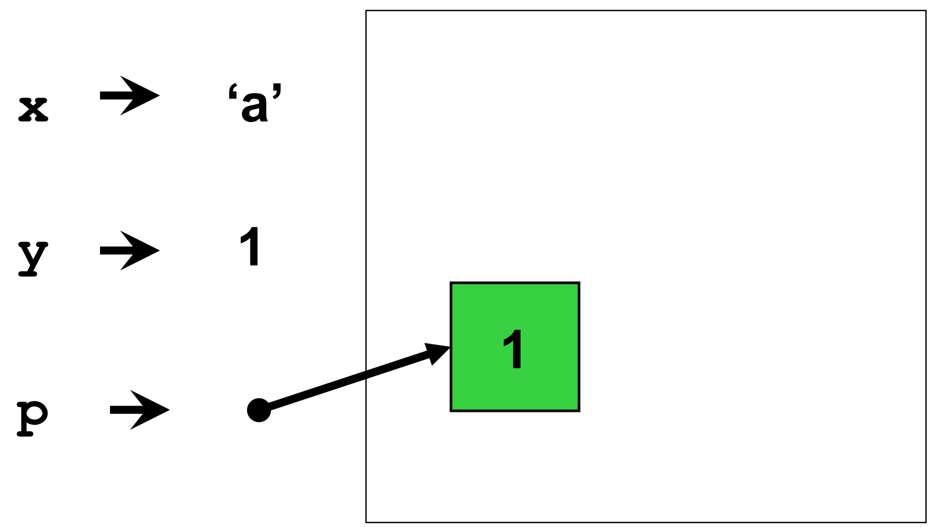


Pointers
null
non-null
maybe null

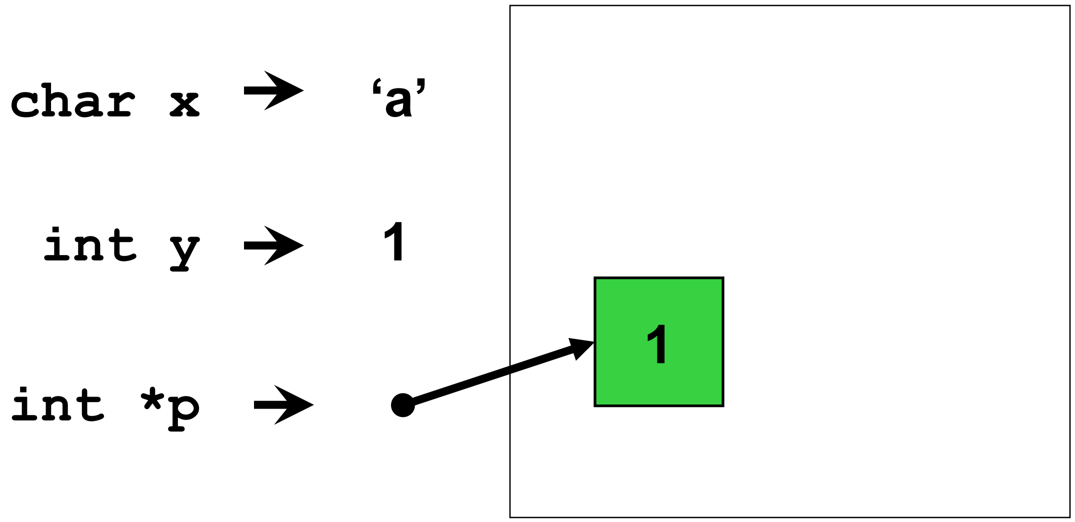


p
x
(a)
p
(b)
p
x
(c)
Diagram (a) abbreviates (b) or (c)

Program state



Well-typed program state



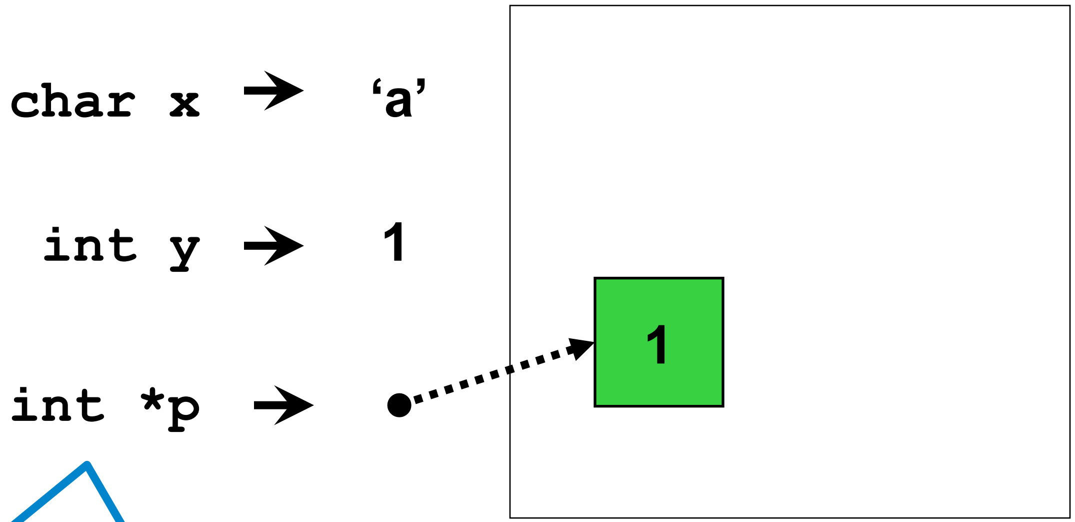
Well-typed program state

`char x` → **'a'**

`int y` → **1**

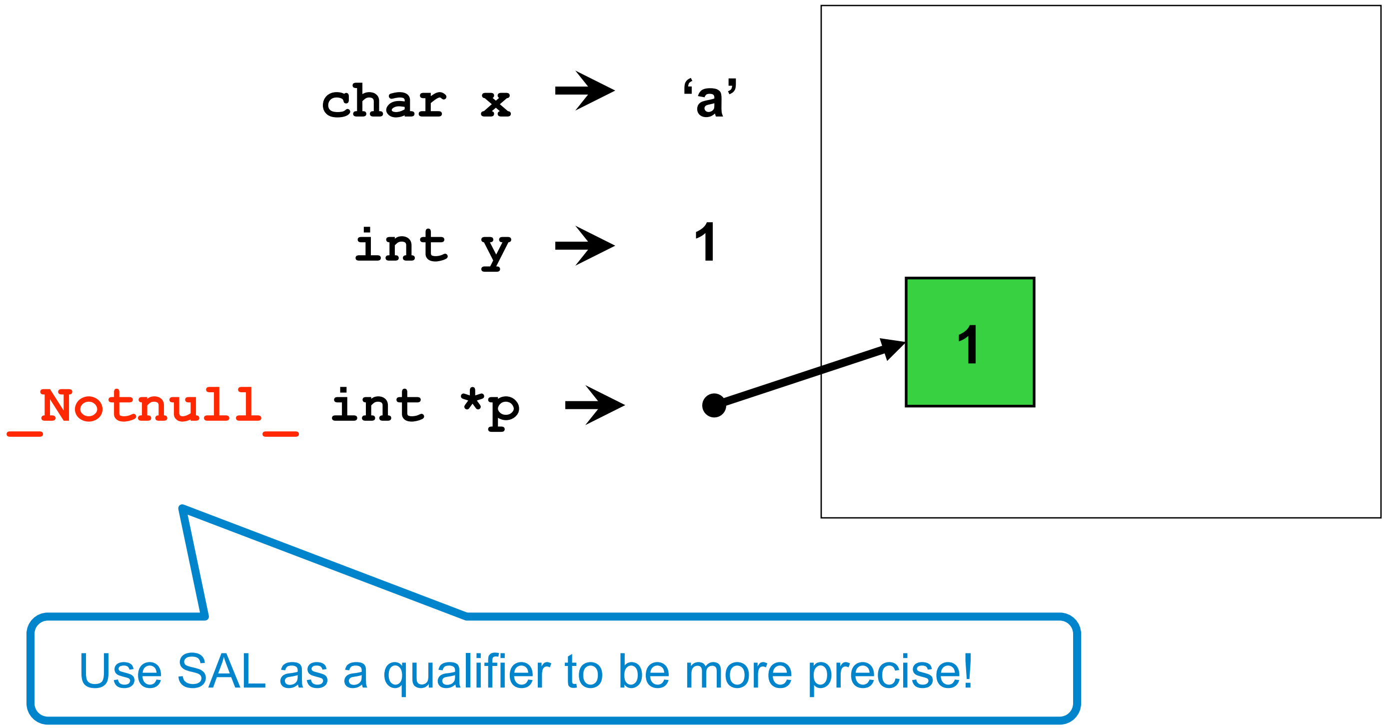
`int *p` → •

Well-typed program state



C type is not descriptive enough to avoid errors.

Program state with qualified type



Qualified type is not always sufficient

```
void foo(_Notnull_ _Writable_elements_(1) int *p)
{
    *p = 1;
}
```

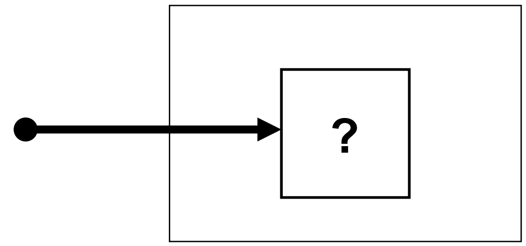
```
void foo(_Notnull_ _Valid_ void *p)
{
    *p = 1;
}
```

Which one is right?

Problem: types don't capture state transitions!

Pre/post conditions make up a contract

```
_Notnull_ _Writable_elements_(1)
                    int *p →
```

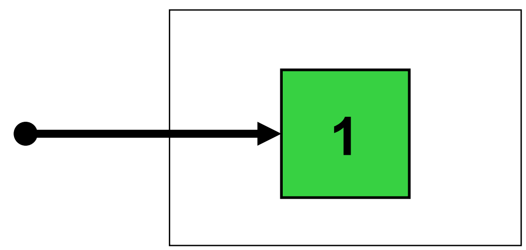


Precondition

`foo(&a);`

Postcondition

```
_Notnull_ _Valid_ int *p →
```



Contract for program logic

```
void foo(
    _Pre_ _Notnull_ _Pre_ _Writable_elements_(1)
    _Post_ _Notnull_ _Post_ _Valid_
    int *p)
{
    *p = 1;
}
```

`_Post_ _Notnull_` can be removed because C is call by value.

Simplified, but still cumbersome to use!

```
void foo(
    _Pre_ _Notnull_ _Pre_ _Writable_elements_(1)
    _Post_ _Valid_
    int *p)
{
    *p = 1;
}
```

C preprocessor macros to the rescue

```
#define _Out_ \
_Pre_ _Notnull_ _Pre_ _Writable_elements_(1) \
_Post_ _Valid_
```

```
void foo(_Out_ int *p)
{
    *p = 1;
}
```

See how simple the user-visible syntax is!

Under the hood—two implementations

```
#define _Out_ \
[SA_Pre(Null=SA_No, WritableElementsConst=1)] \
[SA_Post(Valid=SA_Yes)]
```

← attributes

```
#define _Out_ \
__declspec("SAL_pre") __declspec("SAL_notnull") \
__declspec("SAL_pre") \
__declspec("SAL_writableTo(elementCount(1))") \
__declspec("SAL_post") __declspec("SAL_valid")
```

← declspecs

Historically, there are some key differences between the two mechanisms.

With the Visual Studio 2010 compiler, the gap is (almost) eliminated.

A consistent user-visible language makes the choice transparent.

Basic Properties

validity

`_Valid_`
`_Notvalid_`

const-ness

`_Const_`

string termination

`_Null_terminated_`
`_NullNull_terminated_`

buffer size

`_Readable_elements_`
`_Writable_elements_`
`_Readable_bytes_`
`_Writable_bytes_`

null-ness

`_Null_`
`_Notnull_`
`_Maybenull_`

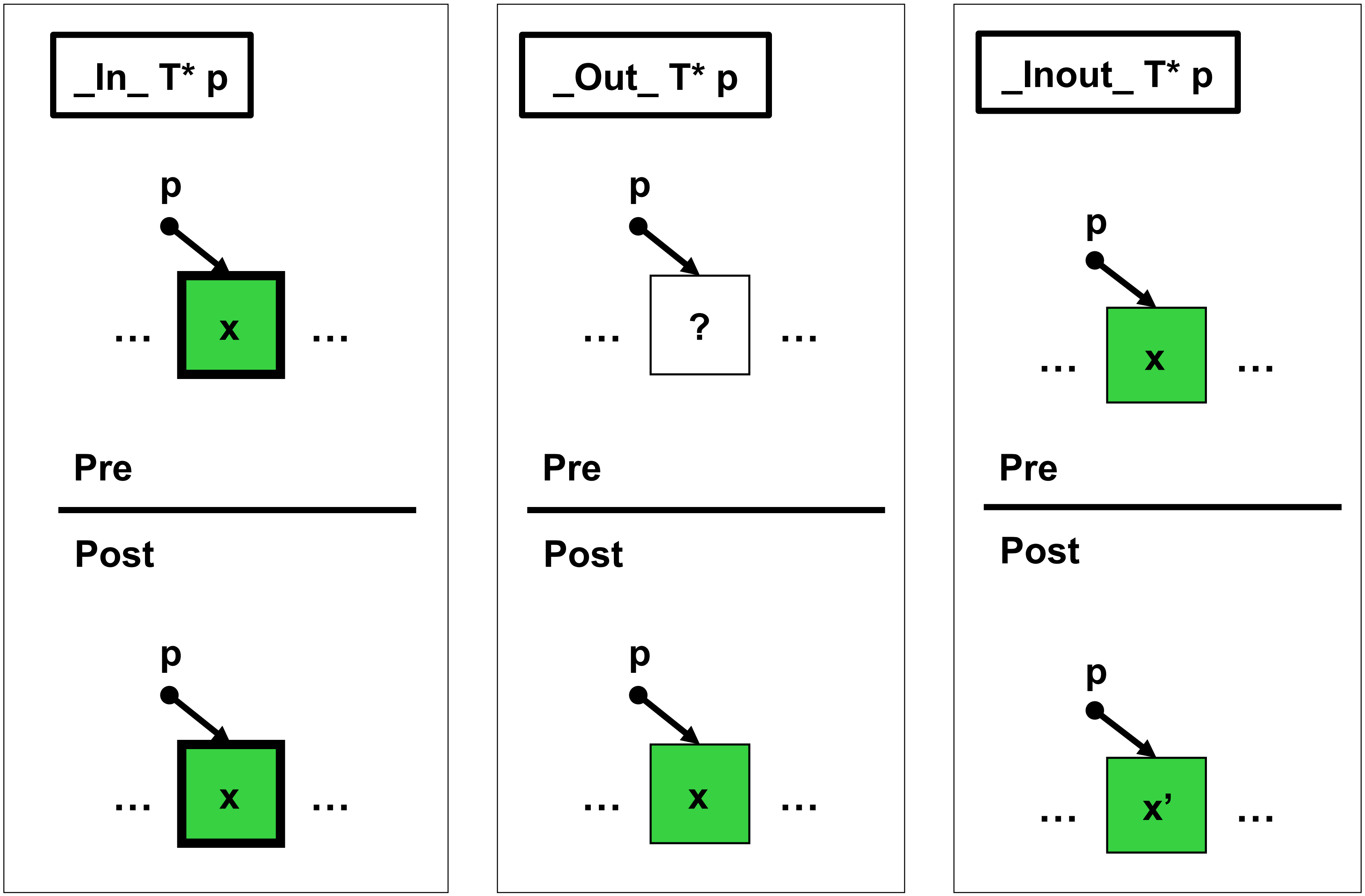
Examples

"Pointer **`ptr`** may not be null."
"String **`str`** is null terminated."
"Length of string **`str`** is stored in **`count`**."
"Object **`obj`** is guarded by lock **`cs`**."

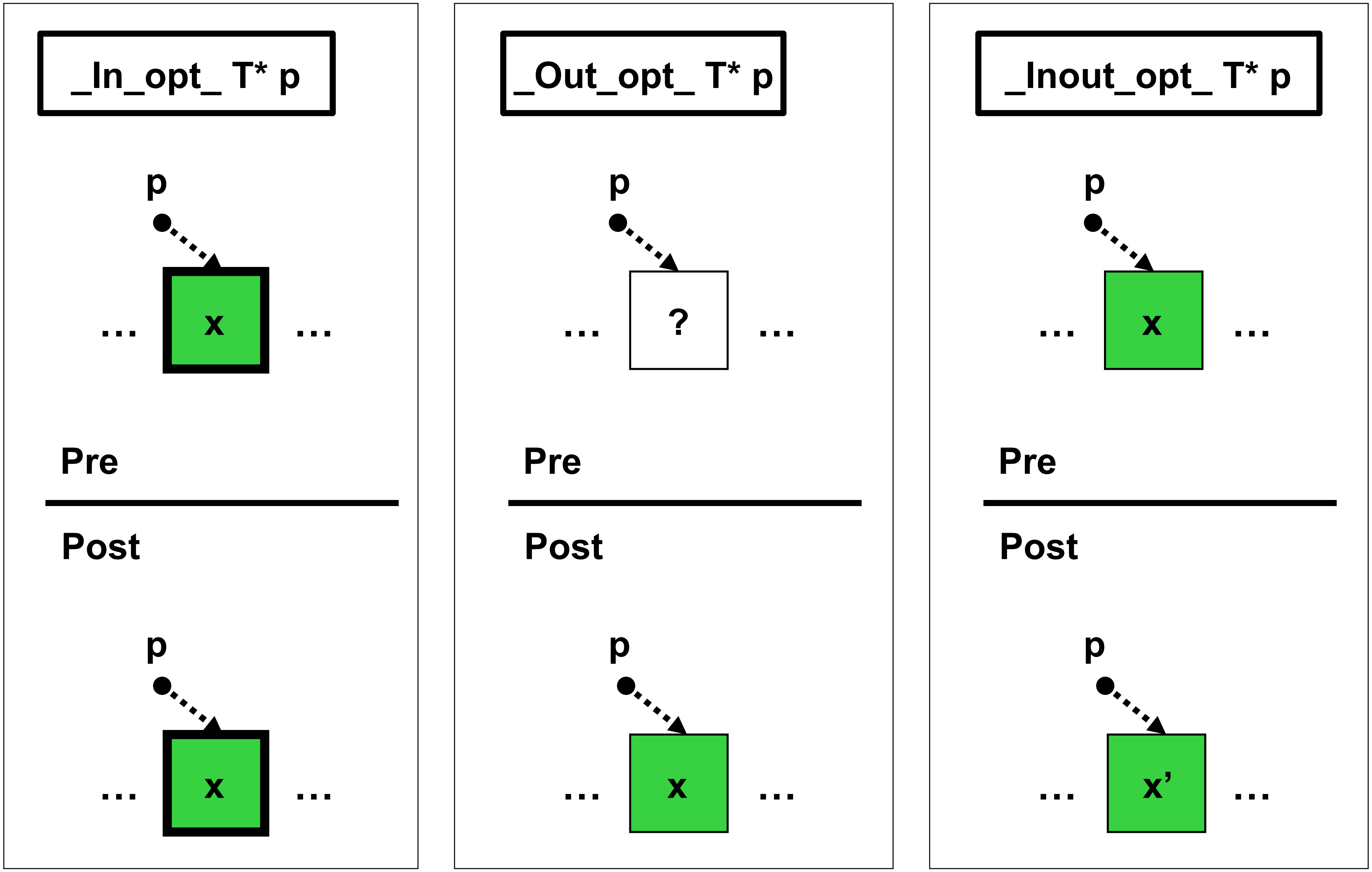
Popular annotations in Windows

SAL	Count
_In_	1961906
Out	381083
_In_opt_	253496
Inout	185008
Outptr	99447
_In_reads_(size)	71217
_Out_opt_	63749
_Out_writes_(size)	56330
_In_reads_bytes_(size)	43448
_Out_writes_bytes_(size)	19888
_Inout_opt_	18845
_In_z_	17932
_Inout_updates_(size)	14566
_Out_writes_opt_(size)	12701
_In_reads_opt_(size)	12247
_Outptr_result_maybenull_(size)	12054
_Outptr_result_buffer_(size)	9597
_In_reads_bytes_opt_(size)	9138
_Outptr_result_bytebuffer_(size)	7693
_Out_writes_bytes_opt_(size)	7667
_Outptr_opt_	6231
_Out_writes_to_(size, count)	5498

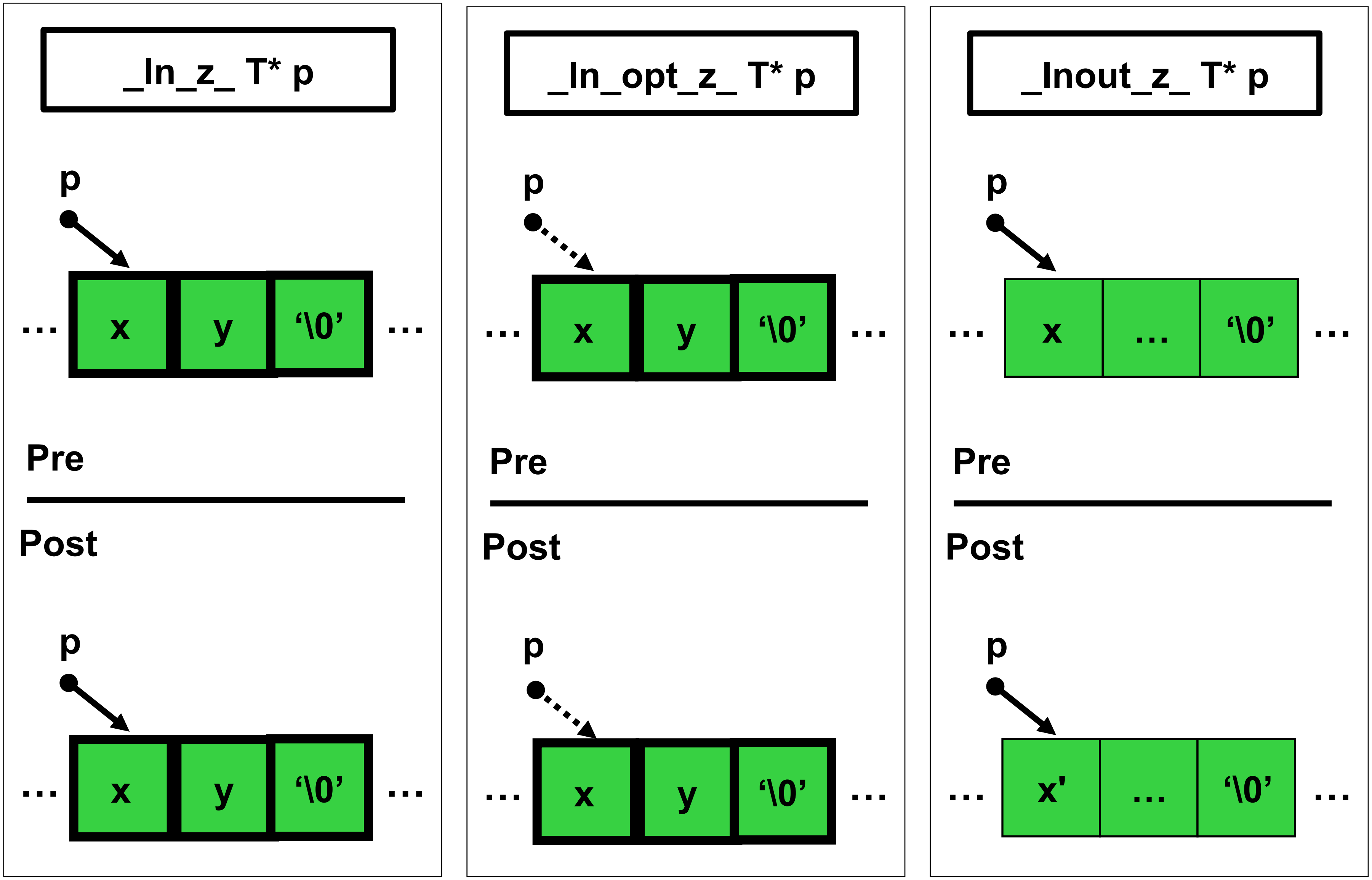
Single element pointers



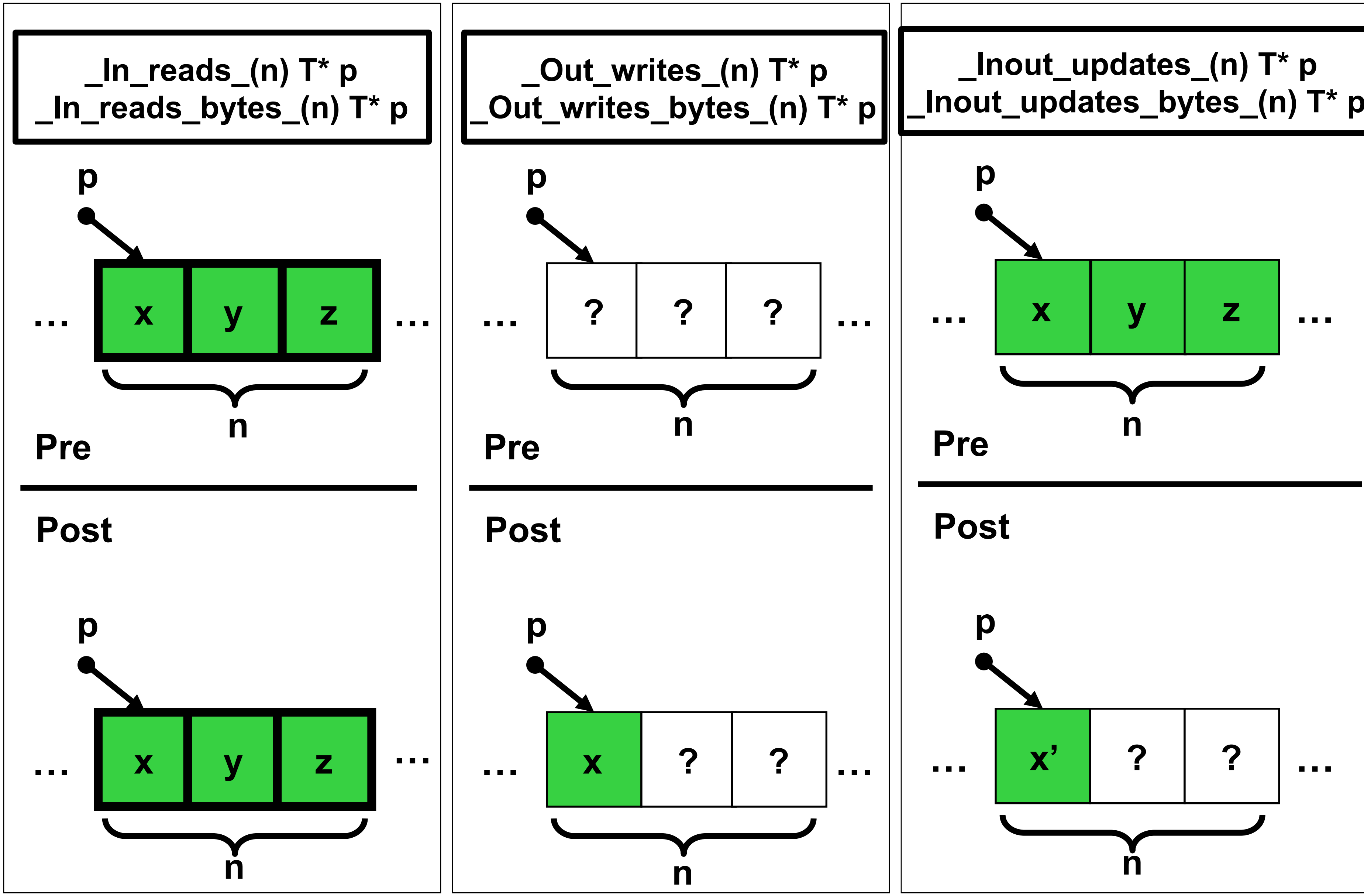
Single element pointers that might be null



Null-terminated strings



Buffers


_In_reads_(n) T* p
_In_reads_bytes_(n) T* p
_Out_writes_(n) T* p
_Out_writes_bytes_(n) T* p
_Inout_updates_(n) T* p
_Inout_updates_bytes_(n) T* p
p
x y z
? ? ?
n
Pre
Post
x ? ?
x’ ? ?
…

Shared-memory concurrency

A critical technique for improving application responsiveness.

Lock-based multithreaded programming is (still) the most dominant paradigm.

Threads are notoriously hard to get right, and the Multi-core, Many-core trend is likely to exacerbate the problem.

⬇

We need tools to help developers write reliable multithreaded code.

Concurrency annotations

_Acquires_lock_(cs)	← Postcondition: lock count increased by 1
_Releases_lock_(cs)	← Postcondition: lock count reduced by 1
_Requires_lock_held_(cs)	← Precondition: lock held when called
_Requires_lock_not_held_(cs)	← Precondition: lock not held when called
_Guarded_by_(cs) T data;	← Invariant: data protected by lock

A primer on SAL

An introduction to program analysis

A glimpse at the engineering process in Windows

What is program analysis?

Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), type checking, abstract interpretation, constraint solving, instrumentation, alias analysis, dataflow analysis, binary analysis, dependency analysis, code coverage, automated debugging, fault isolation, fault injection, testing, symbolic evaluation, model checking, specifications, …

code search == program analysis

program analysis == code search

Accuracy

False positive:
report is not a bug.

vs.

Completeness

False negative:
bug is not reported.

don't miss any bug + report only real bugs == mission impossible

We need to deal with partial programs and partial specifications.

Any of the inputs could trigger a bug in the program.

- No false negative—we have to try all of the inputs.
  If we do the inputs in bunches, we'll have noise.
- No false positive—we have to try the inputs one by one.
  But the domain of program inputs is infinite.

Dynamic Analysis		Static Analysis
Run the program.		Simulate many possible runes of the program.
Observe program behavior on a single run.	**vs.**	Observe program behavior on a collection of runs.
Apply rules to identify deviant behavior.		Apply rules to identify deviant behavior.

Local Analysis

Single-function analysis (e.g., PREfast)

Scales well enough to fit in compilers.

Example: unused local variable

```
void foo(int *q) {
    int *r = q;
    *q = 0;
}
```

vs.

Global Analysis

Cross-function analysis (e.g., PREfix)

Can find deeper bugs.

Example: null dereference due to broken contract

```
void bar(int *q) {
    q = NULL;
    foo(q);
}

void foo(int *p) {
    *p = 1;
}
```

SAL turns global analysis into local analysis!

```
void bar(int *q)
{
    q = NULL;
    foo(q); // BUG: violating _Pre_ _Notnull_ from _Out_
}
```

```
void foo(_Out_ int *p)
{
    *p = 1;
}
```

How do pre/post conditions work?

Requirement on **foo**'s callers: must pass a buffer that is **count** elements long.

```
void foo(_Out_writes_(count) int *buf, int count)
{
```

Assumption made by **foo**: **buf** is **count** elements long.

…

Local checkers: do the assumptions imply the requirements?

Requirement on **foo**: argument **buf** is **count*4** bytes long.

```
    memset(buf, 0, count*sizeof(int));
}
```

Requirement on **memset**'s callers: must pass a buffer that is **len** bytes long.

```
void *memset(
    _Out_writes_bytes_(len) void *dest,
    int c,
    size_t len);
```

EspX: checker for buffer overruns

```
void zero(_Out_writes_(len) int *buf, int len)
{
    int i;
    for(i = 0; i <= len; i++)
        buf[i] = 0;
}
```

```
assume(sizeOf(buf) == len)

for(i = 0; i <= len; i++)

    inv(i >= 0 && i <= len)

assert(i >= 0 && i < sizeOf(buf))

        buf[i] = 0;
```

```
Constraints:
    (C1) i >= 0
    (C2) i <= len
    (C3) sizeOf(buf) == len
Goal: i >= 0 && i < sizeOf(buf)
 Subgoal 1:  i >=0    by  (C1)
Subgoal 2:  i < len    FAIL
```

Warning: Cannot validate buffer access.
Overflow occurs when `i == len`

EspC: checker for concurrency rules

Requirement on **foo**'s callers: must hold **p->cs** before calling **foo**.

```
_Requires_lock_held_(p->cs)
void foo(S *p)
{
```

Assumption made by **foo**: **p->cs** is held.

EspC: does the assumption imply the requirement? Yes.

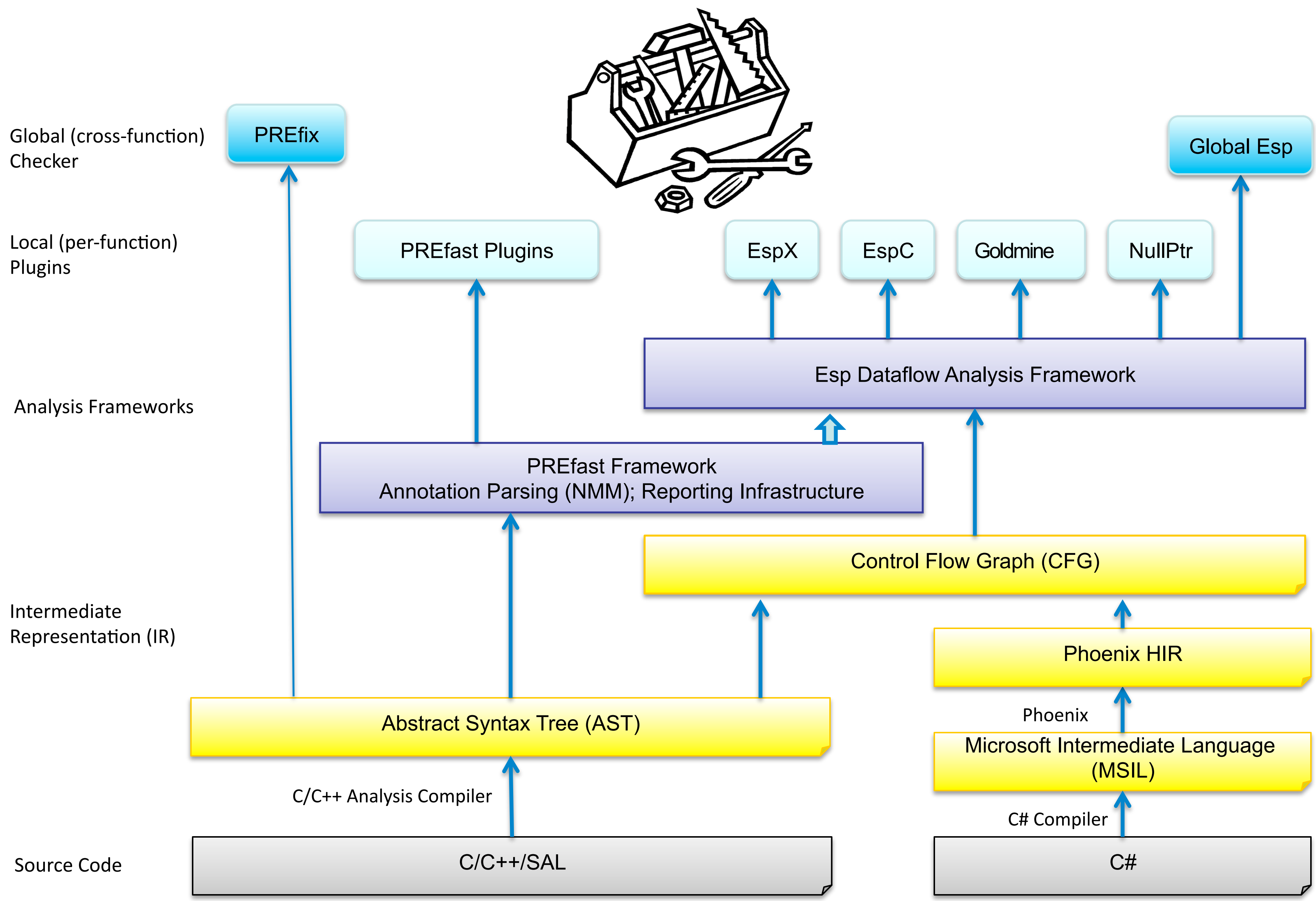
```
    …
```

Requirement on access: **p->cs** must be held.

```
    p->data = 1;
}
```

Invariant on accessing **data**: **cs** must be held.

```
typedef struct _S
{
    CRITICAL_SECTION cs;
    _Guarded_by_(cs) int data;
} S;
```


Global (cross-function) Checker
PREfix
Global Esp
Local (per-function) Plugins
PREfast Plugins
EspX
EspC
Goldmine
NullPtr
Analysis Frameworks
Esp Dataflow Analysis Framework
PREfast Framework
Annotation Parsing (NMM); Reporting Infrastructure
Control Flow Graph (CFG)
Intermediate Representation (IR)
Phoenix HIR
Phoenix
Abstract Syntax Tree (AST)
Microsoft Intermediate Language (MSIL)
C/C++ Analysis Compiler
C# Compiler
Source Code
C/C++/SAL
C#

A primer on SAL

An introduction to program analysis

A glimpse at the engineering process in Windows

The real world challenge

Code on a massive scale

Developers on a massive scale

Tight constraints on schedules

Automated program analysis tools

Code Correctness
Static tools – PREfix, PREfast, Esp

Detects buffer overrun, null pointer, uninitialized memory, leak, banned API, race condition, deadlock, …

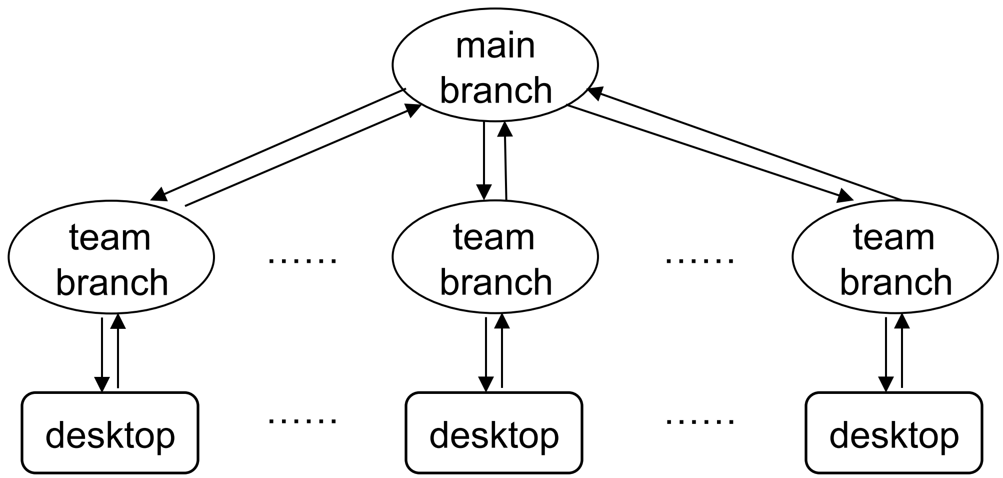
Code Coverage
Code coverage tool – Magellan (based on Vulcan)

Detects code that is not adequately tested

Architecture Layering
Dependency analysis tool – MaX (based on Vulcan)

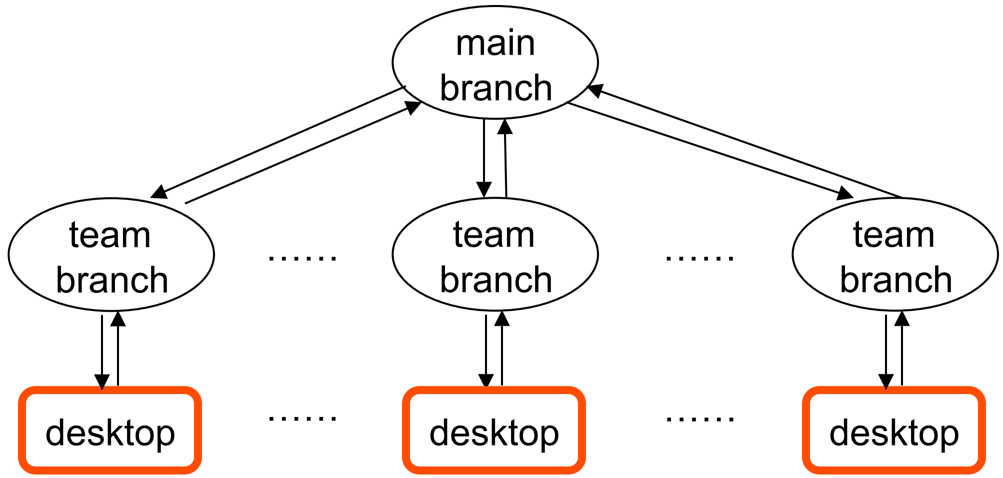
Detects code that breaks the componentized architecture of product

Build Architecture



Forward Integration (FI): code flows from parent to child branch.
Reverse Integration (RI): code flows from child to parent branch.

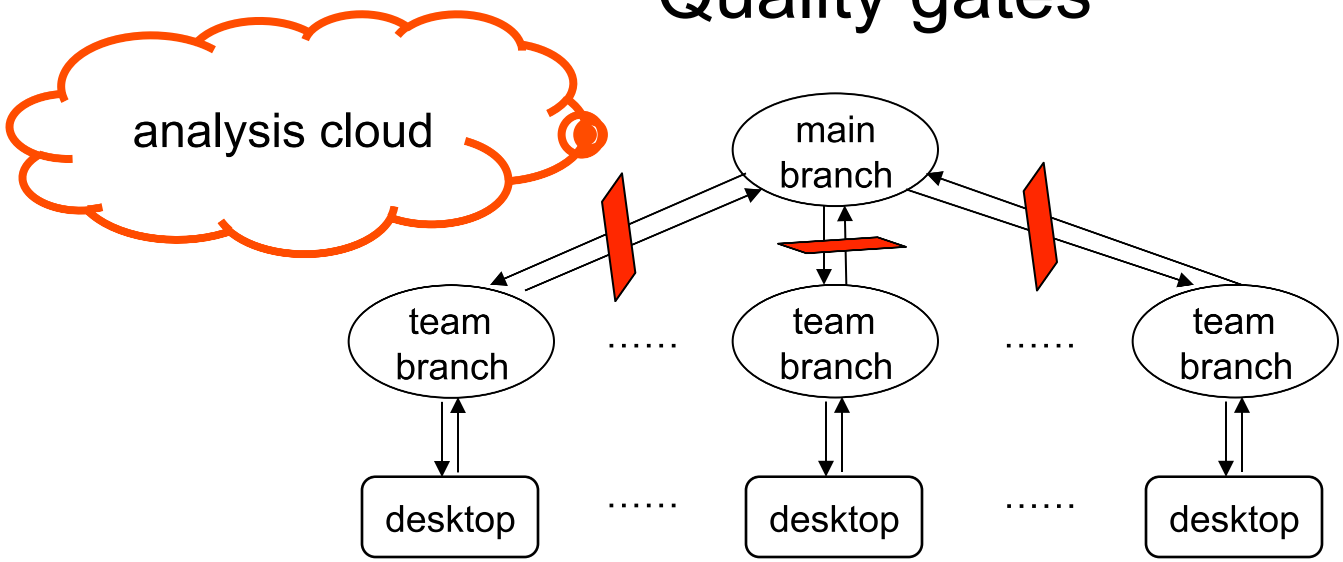
Local analysis on developer desktop



Microsoft Auto Code Review (OACR)

- runs in the background
- intercepts the build commands
- launches light-weight tools like PREfast plugins

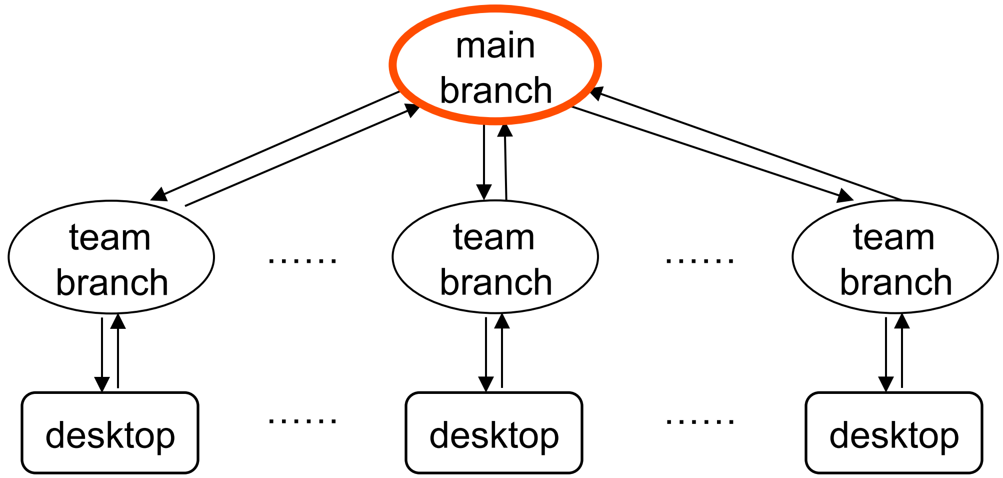
Quality gates



Quality Gates (static analysis "minimum bar")

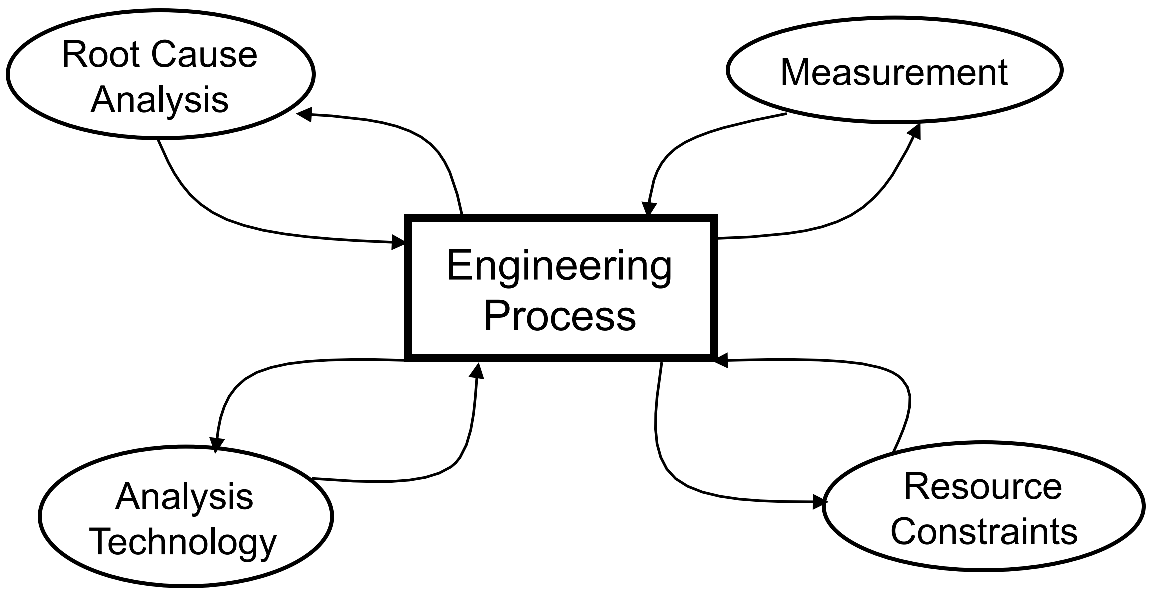
- Enforced by rejection at gate
- Bugs found in quality gates block reverse integration (RI)

Global analysis via central runs



Heavy-weight tools like PREfix run on main branch.

Methodology



Understand important failures in a deep way.

Measure everything about the process.

Tweak the engineering process accordingly.

What we've discussed

A primer on SAL

An introduction to program analysis

A glimpse at the engineering process in Windows good APIs + annotations + analysis tools

⬇

significantly fewer code defects

Automated static analysis is applied pervasively at Microsoft.

SAL annotations have been drivers for defect detection and prevention.

Learn to leverage these technologies and don't treat specifications as afterthoughts!