CSCI 4907/6545 Software Security

Fall 2025

Instructor: Jie Zhou

Department of Computer Science

George Washington University

Slides materials are partially credited to Erik Poll of Radboud University.

Outline

- Review: Address Sanitizer & Pointer-based Memory Safety
- Type Safety
- Memory-safe Programming Languages

Stronger memory safety:
Catch errors when they occur.

ASan Algorithm

- Map regular memory to shadow memory
  - Each byte is mapped.
  - Regular memory includes valid memory objects and redzones.
- Shadow memory indicates the validity of mapped regular memory.
- Before each memory access, check the target memory's validity by querying its mapped shadow memory's status.

Shadow Memory

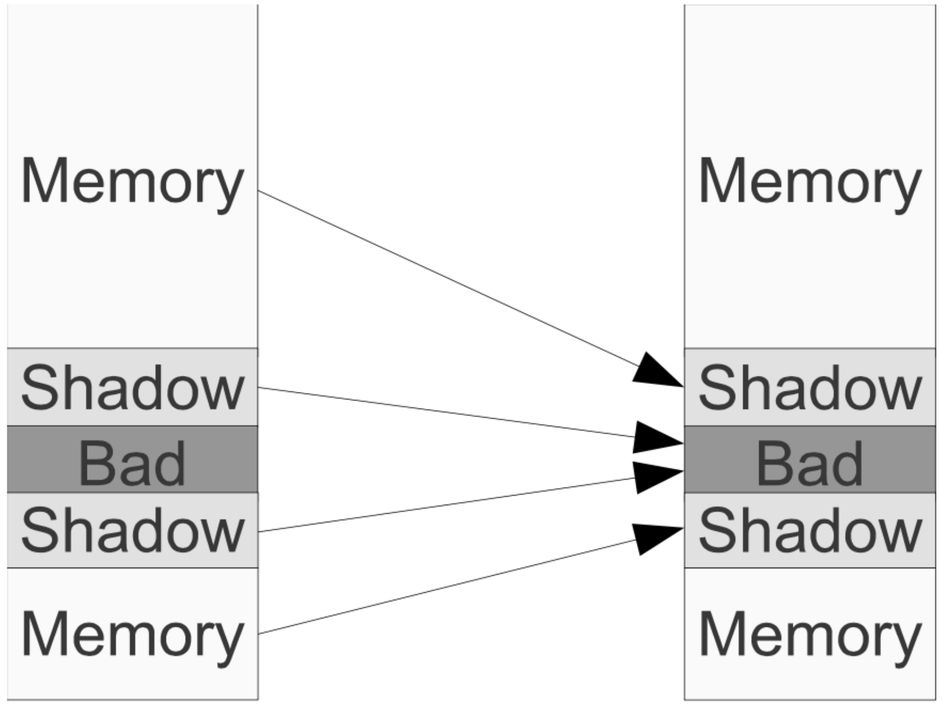
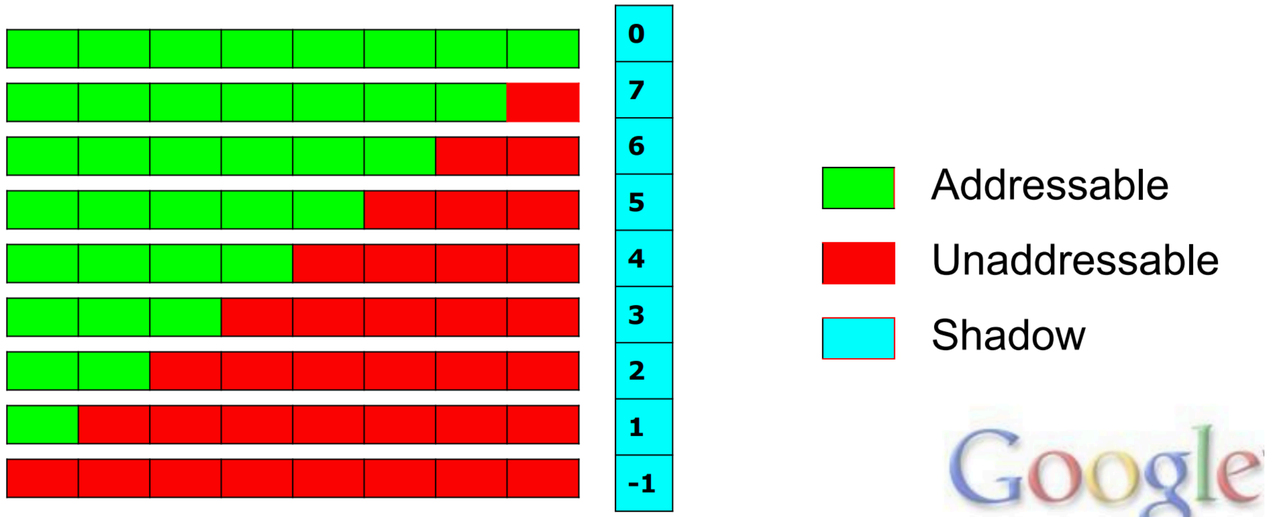


Figure 1: AddressSanitizer memory mapping.

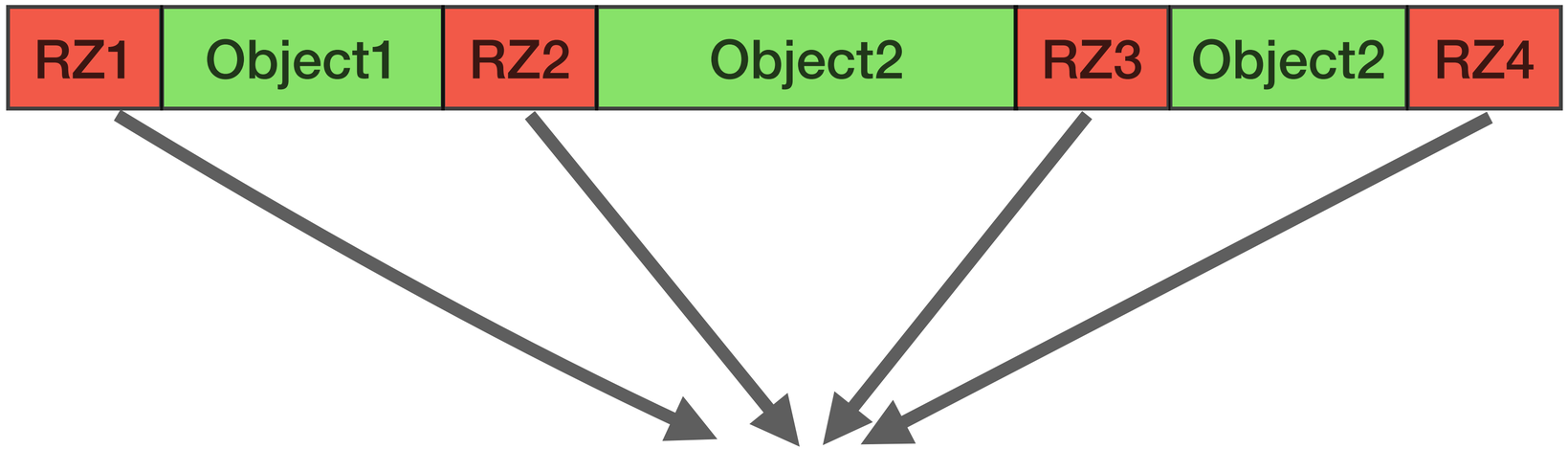
Shadow Memory Mapping

- Newly allocated memory heap objects are typically aligned at a 8-byte boundary.
- Any aligned 8-byte of memory is in one of 9 states:
  - The first *k* (0 <= *k* <= 8) bytes are addressable.
  - The remaining (8 - *k*) bytes are not.
- These 9 states can be encoded into one byte.



Redzones Between Valid Memory Objects

- Page-level redzones are too coarse-grained for memory objects.
- ASan uses small redzones between each valid memory object.
  - Minimum: 32 bytes; default: 128 bytes
  - Larger redzones enable higher probability of detecting buffer overflows.



Mapped to shadow memory indicating
they are not addressable.

Size of Redzone

- Larger redzones enables higher probability of detecting buffer overflows.
- However, higher performance overhead
  - More memory consumption
  - Slower execution time
    - Larger redzones mean more memory writes to their shadow memory.

Considerations for Engineering Sanitizers

- What types of memory bugs to detect?
- What kinds of operations to instrument?
- What metadata to maintain?
- What data structures to use to manage metadata?
- What is the performance overhead budget?
- What optimizations can we do?
- How is the compatibility with un-sanitized code?

Summary of ASan

- Using shadow memory to sanitize *every* memory access
- Detecting both spatial and temporal memory safety
- Implementation: Compiler instrumentations + run-time library support
- High performance and memory overhead
- Incomplete bug detection ability

Pointer-based Memory Safety

- For each pointer, maintain information, called *metadata*, about the pointed memory object
- Use the metadata to do validity checking for pointer dereference
- Ideally, we would like a mechanism that is
  - Comprehensive, i.e., catching all memory safety errors
  - Efficient, i.e., low execution time and memory overhead
  - Automatic, i.e., minimal effort from programmers
  - Backward-compatible, i.e., running smoothly with unchanged legacy code

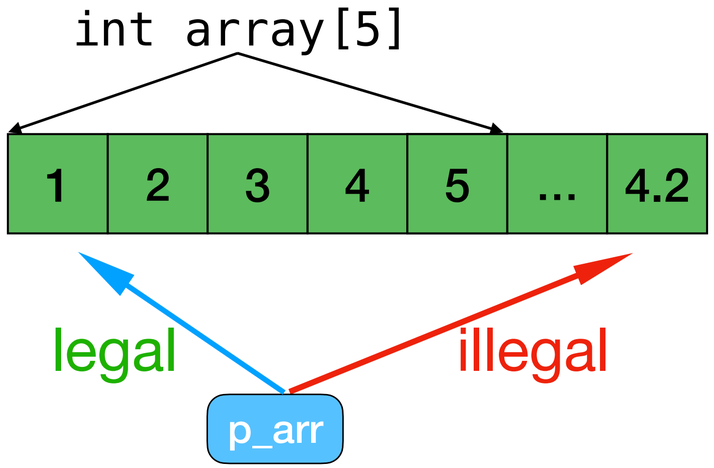
Core Question:
How to manage safety metadata?

Managing Pointer's Metadata

- What metadata is needed?
- How is a pointer mapped to / associated with its metadata?
- How to propagate metadata during pointer propagation (e.g., assignment)?
- How to update the metadata?
- How to perform memory safety checks using the metadata?

Spatial Memory Safety Bugs: Buffer Overflows

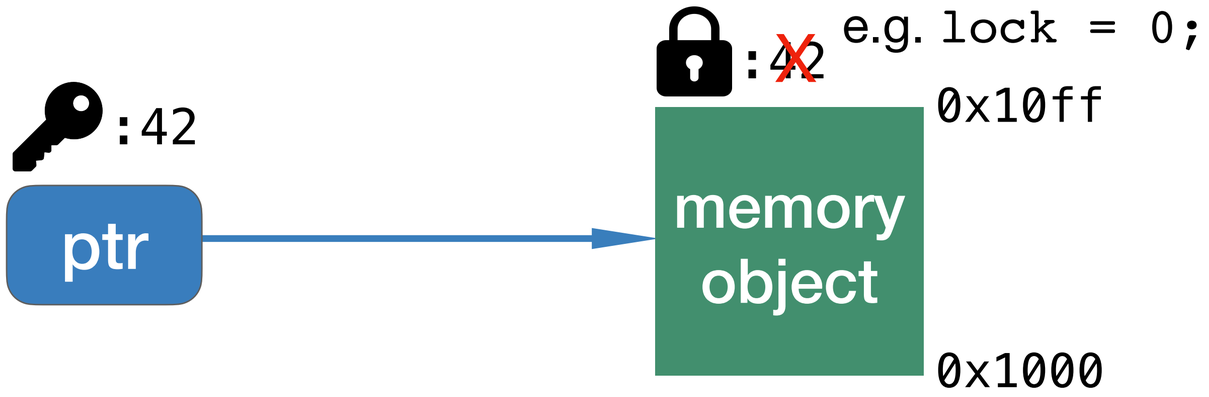
Reading/writing a buffer out of its bounds.



Essential information:

- Starting address (base/lower bound)
- Ending address (upper bound)
- Object size

Key-lock Checking for Temporal Memory Safety



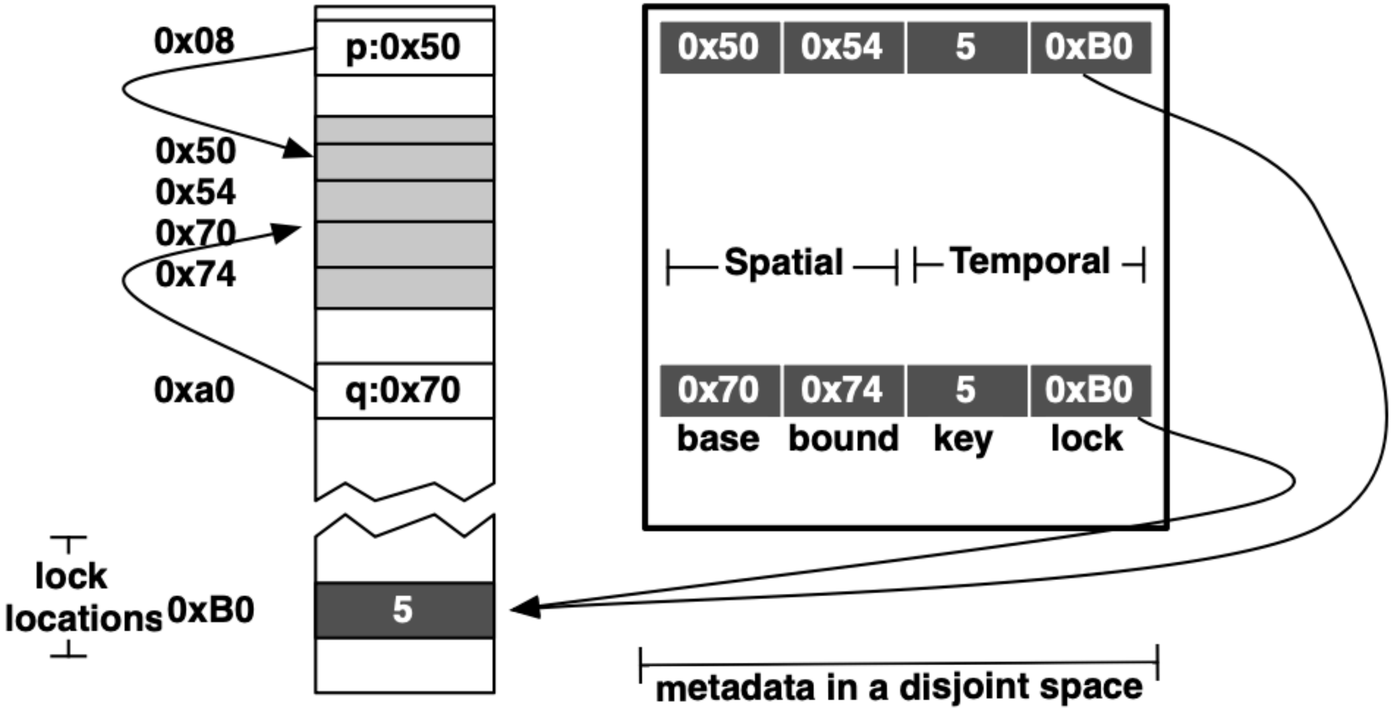

```
check_if_key_matches_lock(ptr);
ptr->num = 30;
```

- Assign memory object a *lock* and pointer a *key*
- Initialize key and lock to the same value
- Invalidate lock upon memory deallocation
- Dynamically check if key matches lock

Charles Fischer and Richard LeBlanc. *The Implementation of Run-Time Diagnostics in Pascal*.
1980 IEEE Transactions on Software Engineering

SoftBoundCETS' Metadata Management

- Record metadata in a disjoint memory region
- E.g., p and q points to different sub-fields in the same memory object, so they maintain different base-upper bounds but the same key-lock.



SoftBoundCETS

- Achieves full memory safety with good backward-compatibility
- However, very high performance overhead.
  - ~75% on SPEC CPU2006, reported in 2015, based on LLVM-3.4.
  - 140% on 8 C SPEC CPU2017 benchmarks, reported in 2024, based on LLVM-12

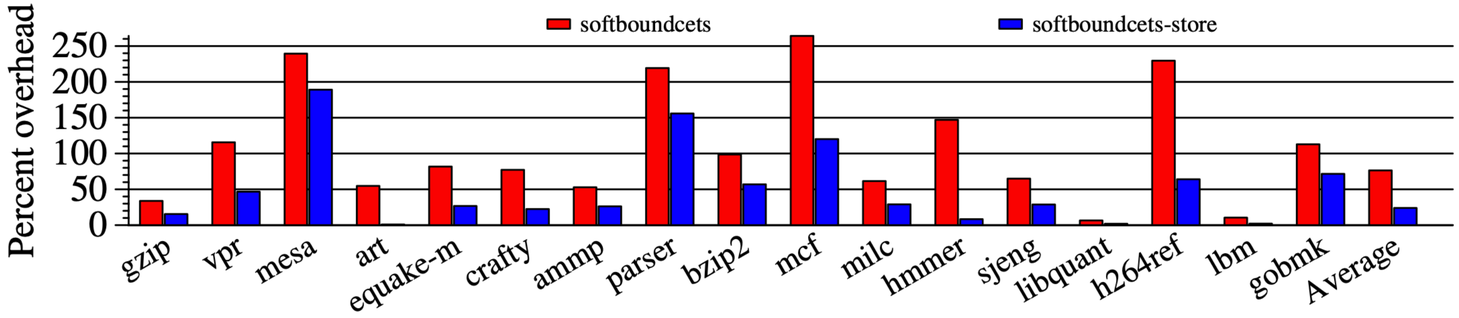


Figure 6 Runtime execution time overheads of the SoftBoundCETS compiler prototype with comprehensive memory safety checking (left bar of each stack) and while checking only stores (right bar of each stack) on a Intel Haswell machine. Smaller bars are better as they represent lower runtime overheads.

Why is this approach slow?

SoftBoundCETS' Metadata Management

- Use a two-level lookup trie to locate the metadata
  - Use a pointer's address as the initial lookup index

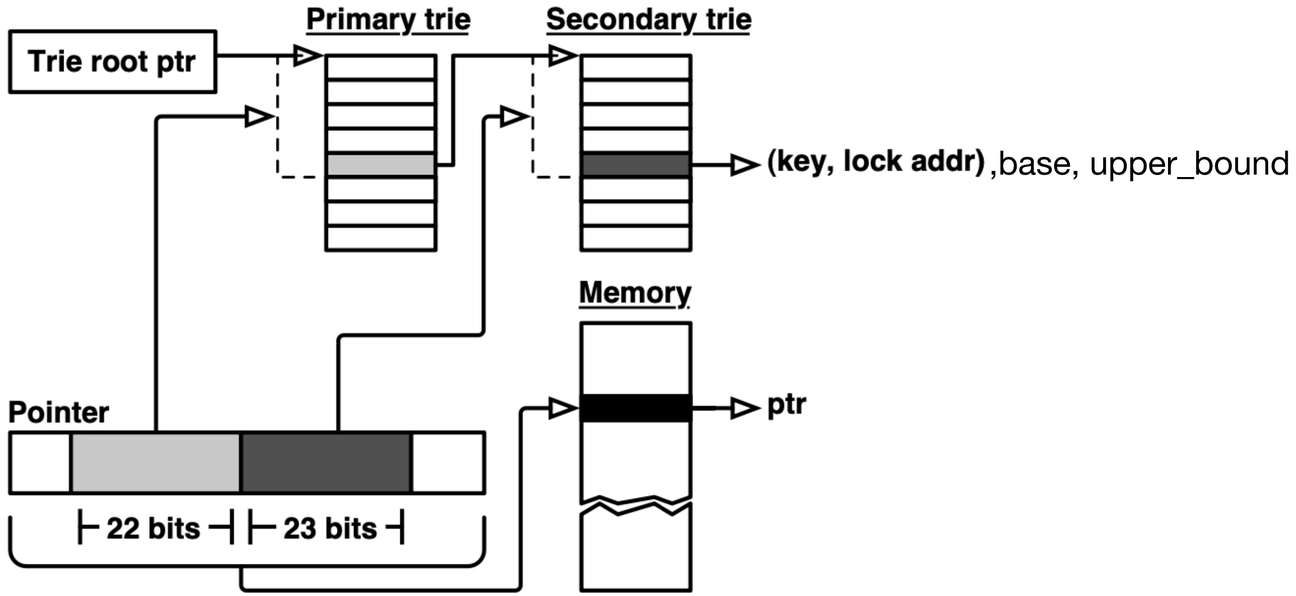


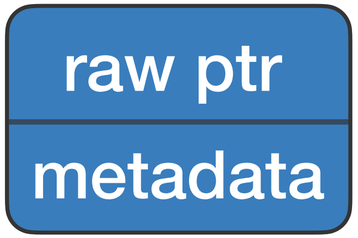
Figure 3. Organization of the trie for pointer metadata

- Loading metadata:14 x86-64 instructions
- Storing metadata: 16 x86-64 instructions

Fat Pointers

- Pointer representation carries metadata besides the raw pointer (address).

Fat pointer:



- One implementation

```
struct _safe_ptr {
    void *raw_ptr;
    void *base_addr;
    void *upper_bound;
    uint64_t key;
    void *lock_addr;
};
```

How do fat pointers interact with legacy library code?

Legacy libraries are unaware of the new fat pointers.

Type Compatibility with Unchanged Code

- One implementation

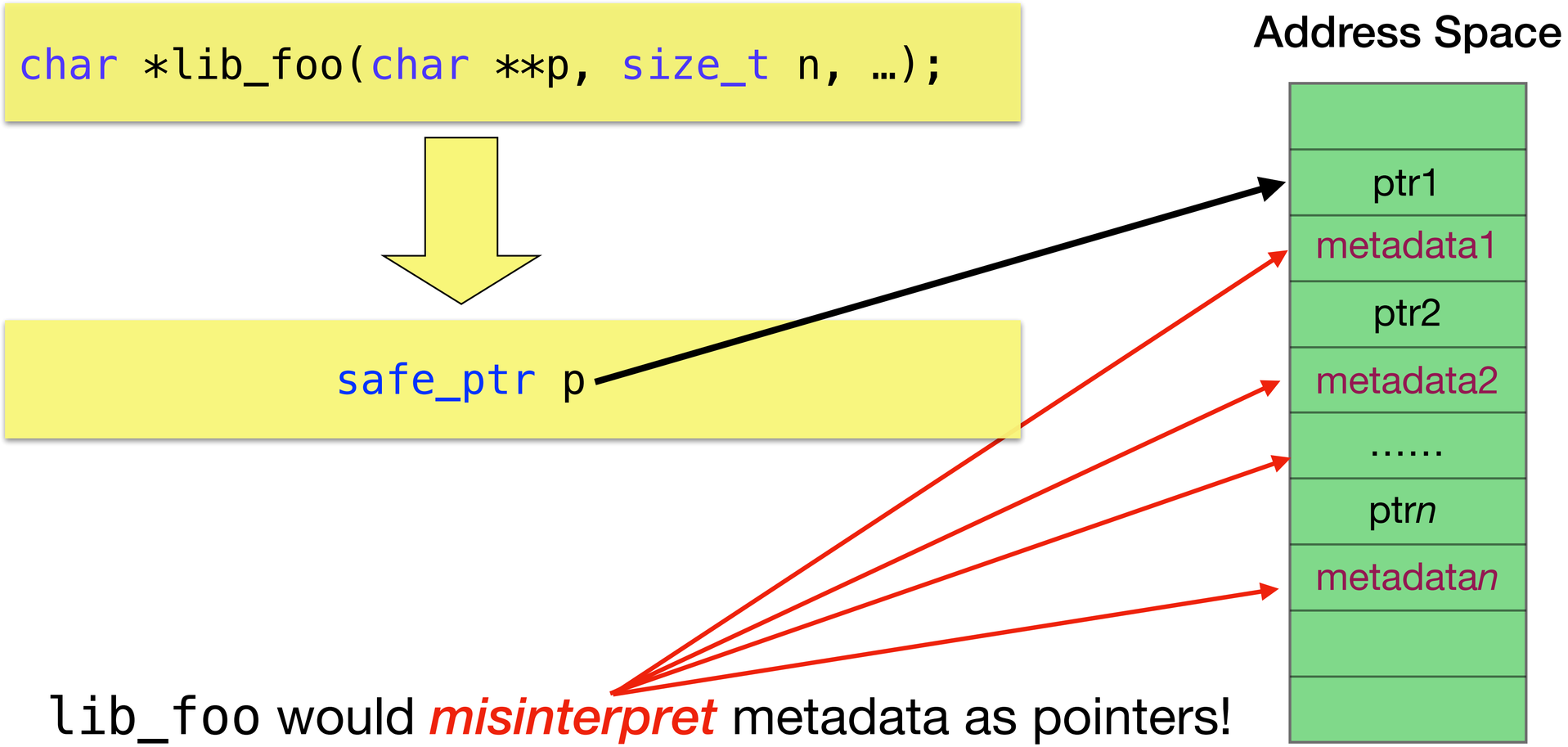
```
struct safe_ptr {
    char *raw_ptr;
    char *base_addr;
    char *upper_bound;
    uint64_t key;
    void *lock_addr;
};
```

```
char *strchr(const char *s, int c);
```

How to pass a `safe_ptr` to `strchr()`?

`strchr()` returns a raw pointers.
How to make it a safe_ptr?

Memory Layout Compatibility with Unchanged Code



Solutions to Backward Compatibility Issues

- Porting library functions using fat pointers
  - Pros: complete and secure
  - Cons: high programmer effort; not always viable
- Mixing using raw pointers and fat pointers
  - Pros: easy to implement
  - Cons: security hazards
- Hybrid of disjoint and in-place metadata
  - Pros: secure
  - Cons: complex and slow
- Data marshaling
  - How much program effort is required?
  - What is the performance overhead?

Compatibility Issues with Disjoint Metadata Scheme

```
void qsort(void *base, size_t n, size_t width, int (*compar)(const void *, const void *));
```

- `qsort`: Sort an array of `n` item, each of size `width`, using function `compar`.
  - E.g., Use a `greater_than()` function to sort an array of n integers.
- What about sorting an array of pointers?
  - SoftBoundCETS' metadata lookup procedure will not work!
    - It uses a pointer's address to index the metadata.
  - SoftBound's solution: Rewrite `qsort()` considering the associated metadata

Strengths and Weaknesses of Fat Pointers

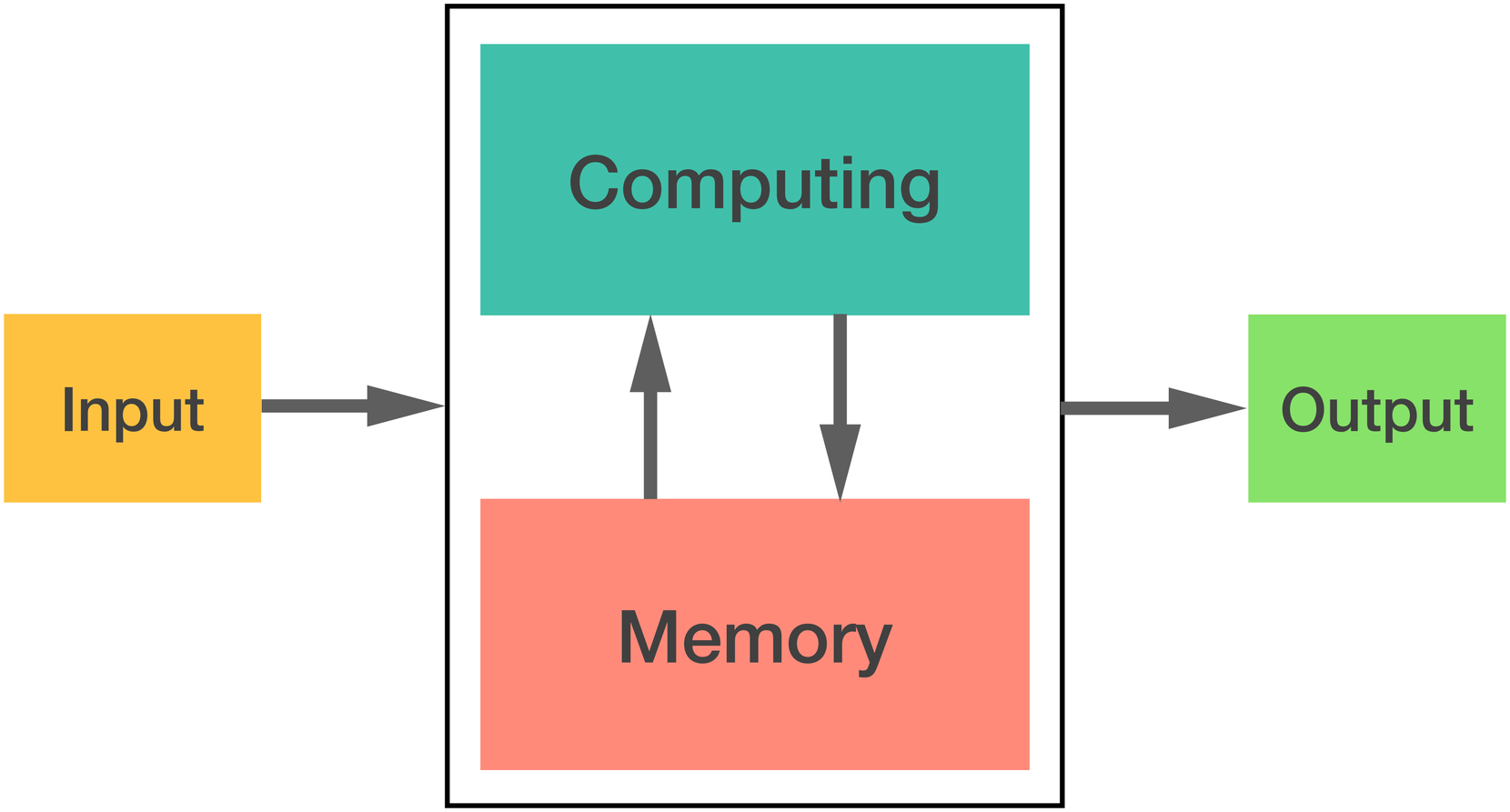
- Good performance
  - Run-time can (very) quickly finds metadata to use.
- Difficult to interoperate with libraries that do not use fat pointers
  - Need wrappers to convert fat pointers to raw pointers and vice versa
  - Need to change memory layout of data structures

“Well-typed programs cannot go wrong.”

Robin Milner

What is *type* in the context of programming languages?

Architecture of Modern Computers



What is *type* in the context of programming languages?

Type and Type System

Wikipedia

A data type (or simply type) is a collection or grouping of data **values**, usually specified by a set of possible values, a set of allowed **operations** on these values, and/or a **representation** of these values as machine types.

In computer programming, a type system is a *logical system* comprising a set of **rules** that assigns a **property** called a type (for example, integer, floating point, string) to every **term** (a word, phrase, or other set of symbols). A type system dictates the **operations** that can be performed on a term.

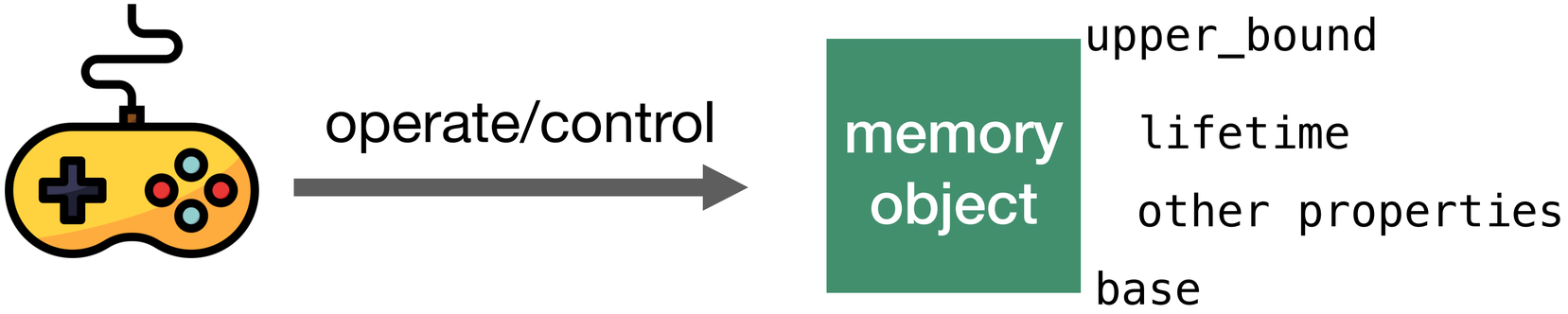
Usually the terms are various *language constructs* of a computer program, such as variables, expressions, functions, or modules.

For variables, the type system determines the allowed values of that term.

“A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data.”

Vijay Saraswat

Type Safety

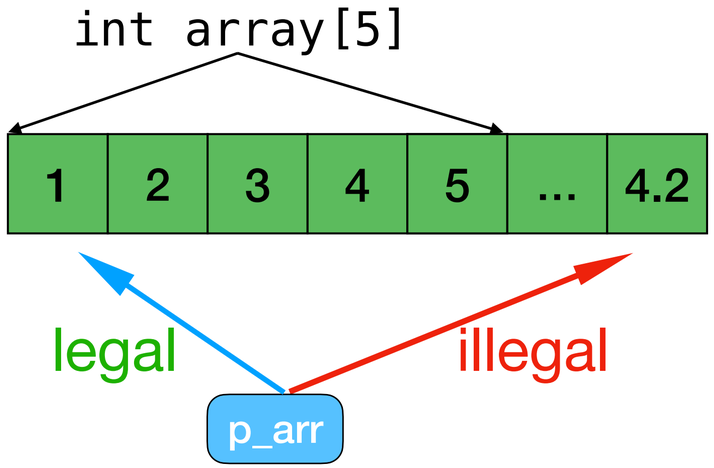


- A sound and safe type system must carefully specify what operations can be performed on the data controlled by the data handler/operator/controller, *and* what operations are allowed on the handler/operator/controller itself.
- A type-safe language has a sound and safe type system that can be respected.

Can memory safety be considered one aspect of type safety?

Spatial Memory Safety Bugs: Buffer Overflows

Reading/writing a buffer out of its bounds.



Essential information:

- Starting address (base/lower bound)
- Ending address (upper bound)
- Object size

Every programming language
has its type system.

Type and Type System

- Statically-typed vs. dynamically-typed type system / languages
  - Statically-typed: Data types are known at compile time
    - E.g., C/C++, Java, Rust
  - Dynamically-typed: Data types are known at run-time
    - E.g., Python, Javascript
- Strongly-typed vs. weakly-typed type system / languages
  - Strongly-typed: Enforcing strict type rules that cannot be broken.
    - E.g., Java, Rust, Python
  - Weakly-typed: Allowing easily breaking the type rules
    - E.g., C/C++, Javascript

Type System Can Go CRAZY!

Javascript's Array

```
var arr = [];
alert(arr.length);
alert(arr[3]);
```

"0"

"ArrayIndexOutOfBoundsException"

This slide is credited to Prof. James Mickens @Harvard

Javascript's Array

```
var arr = [];
alert(arr.length);
alert(arr[3]);
arr[3] = “hi”;
alert(arr.length);
alert(arr[3]);
alert(arr[“3”]);
delete arr[3];
alert(arr.length);
alert(arr[3]);
```

→ “0”

→ “undefined”

→ Nothing happened.

→ “4”

→ “hi”

→ “hi”

→ “4”

→ “undefined”

This slide is credited to Prof. James Mickens @Harvard

Type Casting

- Converting one type to another.
- May cause *type confusion* when used incautiously.
  - Leading to vulnerabilities, e.g., calling a function chosen by attackers.

Vulnerabilities Caused By UAF

```
 1 struct N { long usr; long pwd; int (*fn)(void); };
 2 struct O { int (*oper)(void); long u1; long u2; };
 3
 4 void foo(long uid, long secret) {
 5     struct N *p = malloc(sizeof(struct N));
 6     p->fn = __safe_function_1;
 7     p->usr = uid;
 8     p->pwd = secret;
 9     p->fn();
10 }
11
12 void bar(long user1, long user2) {
13     struct O *x = malloc(sizeof(struct O));
14     x->oper = __safe_function_2;
15     structO *q = x;
16     free(x);
17     q->oper();
18     q->u1 = user1;
19     q->u2 = user2;
20     reply("Users: %l | %l", q->u1, q->u2);
21     free(q);
22 }
```

- Write through p and read through q leads to arbitrary code execution.
  - exploit path: 16->5->7->17

Type Casting in C

- Implicit/automatic casting
  - Compiler automatically/silently converts a variable from one type to another.
  - Occurs during assignments, argument passing, and mixed-type expressions
  - E.g., Adding up two integers with different sizes

Vulnerability: Truncation Errors

```
int func(char *name, unsigned int cbBuf) {
    unsigned short bufSize = cbBuf;
    char *buf = (char *)malloc(bufSize);
    if (buf) {
        memcpy(buf, name, cbBuf);
        …
        free(buf);
        return 0;
    }
    return 1;
}
```

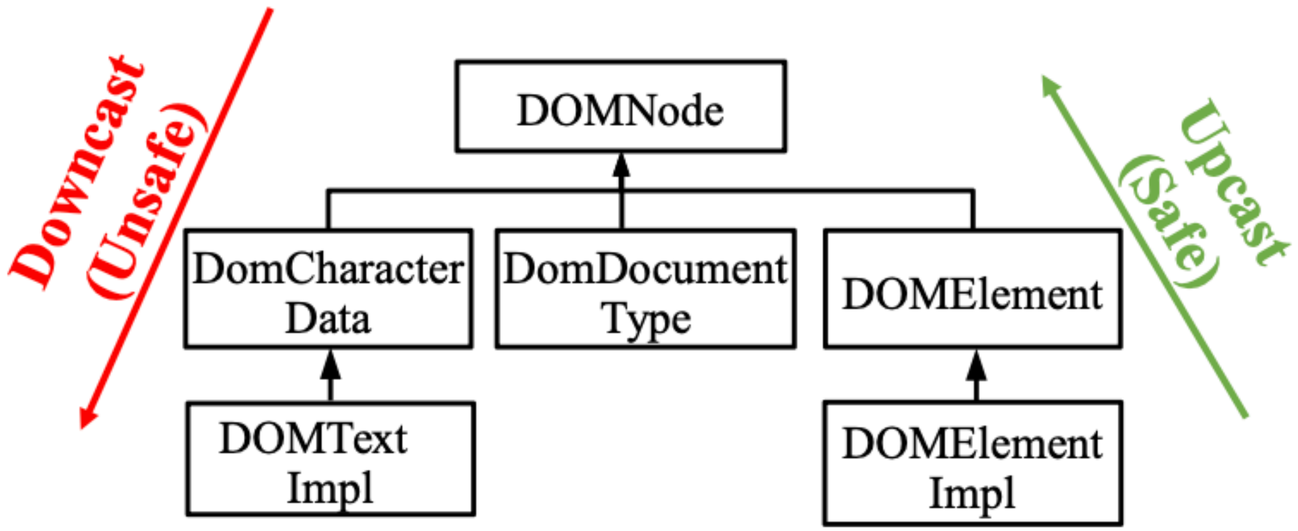
What if we call the function with `cbBuf` greater than 2^16 -1?

Type Casting in C

- Implicit/automatic casting
  - Compiler automatically/silently converts a variable from one type to another.
  - Occurs during assignments, argument passing, and mixed-type expressions
  - E.g., Adding up two integers with different sizes
- Explicit casting
  - Use a cast operator "`(type)`" to convert a variable to anther type
  - E.g., `int i = 5; float f = (float)i;`

Type Casting in C++

- C++ is an object-oriented programming language.
  - Supporting hierarchical types
- Two common types of casts: upcast and downcast.
  - Upcast is safe, but downcast may not be.



Type Casting in C++

- Four explicit type-cast operators

```
static_cast<type>(expression)
dynamic_cast<type>(expression)
reinterpret_cast<type>(expression)
const_cast<type>(expression)
```

`static_cast<target_type>(expression)`

- Explicitly converting one type to another
- Static type checking at compile time; no runtime type checking
  - Only allows casts between compatible types along the type hierarchy
    - E.g., cast between a floating-point variable to an integer
    - E.g., cast between pointers to an object and its ancestor object type
- Similar to implicit type cast (conversion)
- Weak safety guarantees, e.g., allowing downcast.

const_cast<target_type>(expression)

- Removes the constness of a reference/pointer
  - Commonly used to remove const from an object that was originally mutable
- Incurs undefined behavior if using it to modify a truly constant object

```
void changeValue(const int* ptr) {
    int* modifiablePtr = const_cast<int*>(ptr);
    *modifiablePtr = 10; // Undefined behavior if
                         // ptr points to a const object
}
```

- No type confusion issues.

reinterpret_cast<target_type>(expression)

- Conversion between any two types
- Reinterpret the underlying bits of an object as the target type.
- Commonly used to cast between pointers to incompatible types.
- Very unsafe!

```
struct A {
    int x;
    double y;
};

struct B {
    char c;
    int i;
};

int main() {
    A a{42, 3.14};
    B* b = reinterpret_cast<B*>(&a);

    // Accessing members of B via b is unsafe and can cause undefined behavior.
    std::cout << "B's i (as int): " << b->i << std::endl;
    std::cout << "B's c (as char): " << b->c << std::endl;
    return 0;
}
```

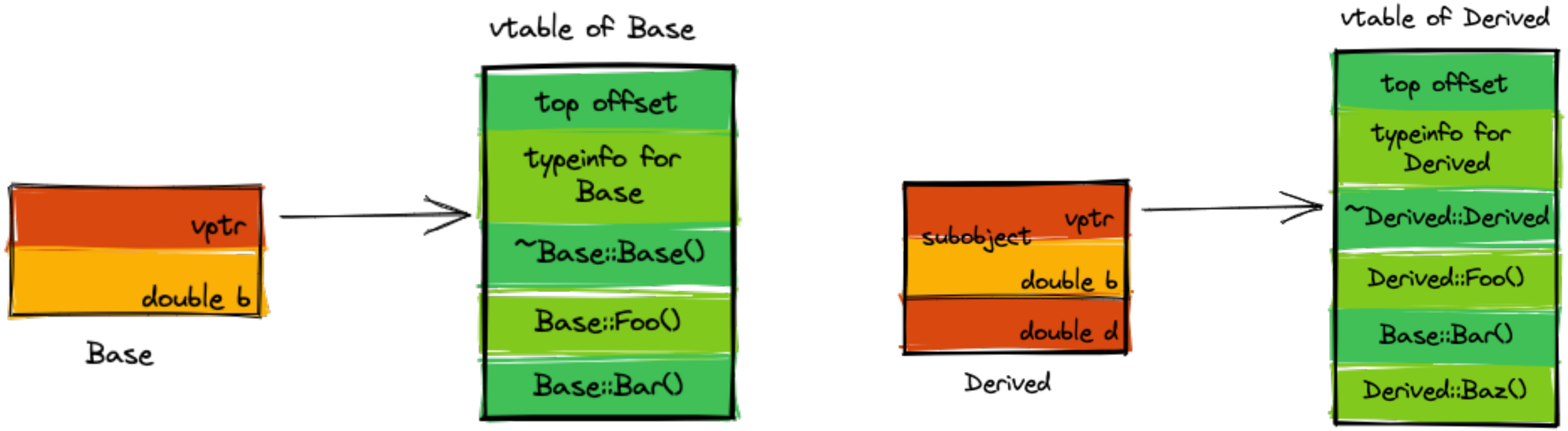
What are printed out?

`dynamic_cast<target_type>(expression)`

- Intended for safe casting within an inheritance hierarchy.
- Commonly used for downcast.
- Runtime check to ensure the cast is safe.
  - Use C++'s RTTI to determine the concrete type at runtime

C++’s Run-time Type Information (RTTI)

- A mechanism exposing a memory object’s type at run-time
- In C++, only works for classes with virtual functions
- Given a pointer to an object, RTTI uses it to query the object’s vtable, and then find the RTTI entry for the object’s type, and recursively query RTTI entries to find a compatible type for the casting target type.



`dynamic_cast<target_type>(expression)`

- Intended for safe casting within an inheritance hierarchy.
- Commonly used for downcast.
- Runtime check to ensure the cast is safe.
  - Use C++'s RTTI to determine the concrete type of pointed object at run-time
  - Follow the inheritance hierarchy to verify if this is a upcast
- For failed check, return null or a pre-defined exception
- Provide safety, but at high performance and code size cost
- Only support polymorphic objects with virtual functions

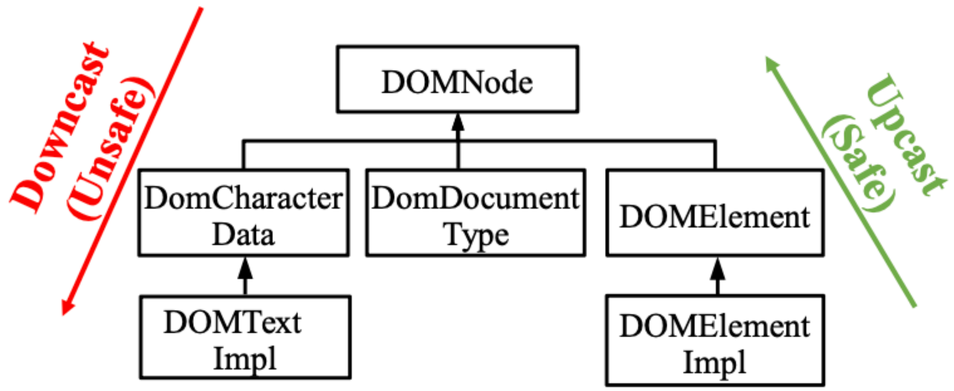
Type confusion is a major source of vulnerabilities in C/C++ programs.

HexType: A Faster and More Comprehensive Solution

- A compiler-based tool detecting type casting errors at run-time
- Maintaining a type table recording all class inheritance information
- Maintaining an object-type mapping table
- Dynamically checking if a type casting operating (e.g., `static_cast`) is safe.

HexType: Type Hierarchy Mapping

- Extract all type relationship information about classes
- Hash each type name to a string hash
- Generate a list of hash values as a global variable for each type

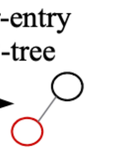


- E.g., `DOMElementImpl: H(DOMElement), H(DOMNode), ...`

HexType: Type Table for Fast Lookup and Verification

- A Hybrid data structure mapping a object's address to its type
  - Hash table + red-black tree
    - Fast lookup for frequently visited objects in the hash table
    - Slow lookup for first-time visited objects

Fast-path Slot			**Slow-path Slot** (RB-tree Ref)
Allocated Object Ref	Hashvalue for Object Name	Type Relationship Information Ref	
0x417000	2341234	0x51723D	●
0x41563C	1312321	0x51724D	_
0x41723D	7231234	0x51724D	_
_	_	_	_
0x41563E	4232123	0x51623D	●



- Instrument unsafe type casts, mainly `static_cast` and `reinterpret_cast`
  - Query the type table to find if the target type is compatible

Safe Programming Languages

Language-based Security

Enforcing security properties using programming language features and techniques.

- Safety guarantees
  - Memory safety, type-safety, thread-safety
- Forms of access control
  - Visibility/access restrictions with e.g. public, private, const
  - Sandboxing mechanisms inside programming language
    - E.g., Python's `exec()` with controlled environments
- Forms of information flow
- Examples: CFI, SFI, ASan, SoftBound, etc.
  - Retrofitting memory safety into existing software

Other Ways Programming Languages Can Help

- Offering good APIs/libraries
  - APIs with parameterized queries/prepared statements for SQL
  - More secure string libraries for C
- Making assurances of the security easier
  - Being able to understand code in a modular way
  - Only having to review the public interface, in a code review
- Offering convenient languages features
  - E.g., mechanism for exception handling

Safe Programming Languages:
Languages with Built-in Security Guarantees

Does writing in a safe language ensure secure programs?

Of Course Not!

“Safe” Programming Languages

- You can write insecure programs in ANY programming language.
  - Flaws in the program logic can never be ruled out.
- Still, some safety features can be nice.
  - Preventing *entire classes* of bugs
  - At least mitigate their impact

Safe Programming Languages

- Safe programming languages
  - impose some discipline or restrictions on the programmer
    - E.g., raw pointers are disallowed in Java.
  - offer some abstractions to the programmer, with associated guarantees
    - E.g., accessing an array in Java/Python will be enforced with bounds check.
- This takes away some freedom & flexibility from the programmer, but hopefully extra safety and easier understanding makes it worth this.

Attempts at a General Definition of Safety

- A programming language can be considered safe if
  - You can trust the *abstractions* provided by the programming language.
    In other words, the language enforces these abstractions and guarantees that they cannot be broken.
    - E.g., a boolean is either true or false, and never 23 or null.
    - Programmers do not need to care how “true/false” is represented in the machine.
- Programs have precise & well defined semantics (i.e., meaning)
  - More generally, leaving things *undefined* in any specification is asking for security trouble.

“A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data.”

Vijay Saraswat

Attempts at a General Definition of Safety

- A programming language can be considered safe if
  - You can trust the abstractions provided by the programming language. In other words, the language enforces these abstractions and guarantees that they cannot be broken.
    - E.g., a boolean is either true or false, and never 23 or null.
- Programs have a precise & well defined semantics (i.e., meaning)
  - More generally, leaving things *undefined* in any specification is asking for security trouble.
- You can understand the behavior of programs in a *modular* way.

“Safer” and “Unsafer” Languages

Warning: this is overly simplistic, as there are many dimensions of safety

Spoiler alert: functional languages such as Haskell are safe because data is immutable (no side-effects)

Dimensions & Level of Safety

- There are many dimensions of safety: memory safety, type safety, thread safety, arithmetic safety, guarantees about non-nullness, about immutability, …
- For some dimensions, there can be many levels of safety.

Eg, in increasing level of safety, going outside array bounds may:

1. **let an attacker inject arbitrary code**
2. ***possibly* crash the program (or else corrupt some data)**

'unsafe'; some undefined semactics

3. ***definitely* crash the program**
4. **throw an exception, which the program can catch to handle the issue gracefully**
5. **be ruled out at compile-time**

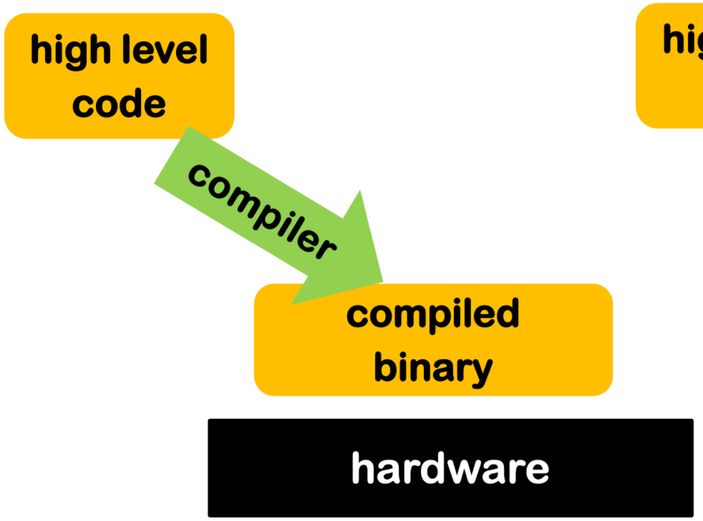
'safe'

Safety: How?

- Mechanism to provide safety include
  - Compile-time checks, e.g., type checking
  - Run-time checks, e.g., array bounds checks, null-ptr checks, etc.
  - Automated memory management using a garbage collector
  - Using an execution engine to do the things above
    - E.g., Java Virtual Machine (JVM) which performs runtime checks and periodically invokes the garbage collector, etc.

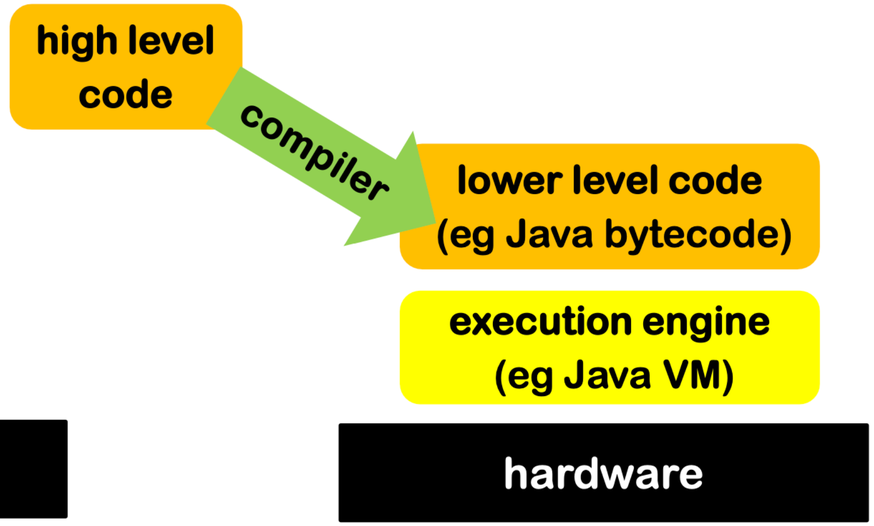
Compiled Binaries vs. Execution Engines

Compiled binary runs on bare hardware



Any defensive measures have to be compiled into the code.

Execution engine (aka ‘runtime') isolates code from hardware



The programming language / platform still ‘exists’ at runtime, and the execution engine can provide checks at runtime

Case Study
Checked C: A Safe Extension to C

Checked C

- Originating from Microsoft Research
- A new extension to C, aiming for memory safety and type safety
- Open-sourced from its inception

Design Principles of Checked C

- High performance
  - New checked pointer types and static checking
- Good human readability and long-term maintainability
  - Explicit checked pointers and bounds information of memory objects
- Easy incremental porting legacy C code
  - Mixing checked and legacy C code at fine granularity
  - Good backward compatibility via *bounds-safe interfaces*

Checked Pointers

- `_Ptr`<T>: pointer to a singleton object of type T
- `_Array_ptr`<T>: pointer to an array of type T
- `_Nt_array_ptr`<T>: pointer to a null-terminated (ends with ‘\0’) array of type T

Checked Pointer: _Ptr<T>

- _Ptr<T>: pointer to a singleton object of type T
  - No pointer arithmetic or subscripting allowed

```
struct Data {
    int val;
    long lval;
    ...
};
```

```
_Ptr<struct Data> p = malloc(sizeof(struct Data));
…
printf("val = %d\n", p->val);
…
p++;
```

null-pointer check inserted by compiler, if not determined at compile time

```
error: arithmetic on _Ptr type
    p++;
    ~~^
```

Checked Pointer: _Array_ptr<T>

- _Array_Ptr<T>: pointer to an array of object of type T
  - Permits pointer arithmetic and subscripting
  - Bounds declared by programmers
  - Bounds check inserted by compiler when not provable at compile time

```
_Array_ptr<int> p = malloc(sizeof(int) * BUF_LEN);
…
int i = p[5];
```

```
error: expression has unknown bounds
    int i = p[5];
            ^~~~
```

Bounds Declaration for _Array_ptr<T>

```
_Array_ptr<T> p : count(bounds_expr) = …;
```

```
#define BUF_LEN 30
_Array_ptr<int> p : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

int i = p[5];
```

no dynamic check inserted because compiler knows BUF_LEN > 5

```
int j = p[30];
```

```
error: out-of-bounds memory access
                int j = p[30];
                        ^~~~
```

```
int k = p[var];
```

null-pointer and bounds check inserted by compiler, if `var < BUF_LEN` *is not provable*

```
_Array_ptr<int> p : count(BUF_LEN / 2) = malloc(sizeof(int) * BUF_LEN);
```

allowed

```
int j = p[15]
```

```
error: out-of-bounds memory access
                int j = p[15];
                        ^~~~
```

Because p's bounds is `[p, p + 14)`

```
_Array_ptr<int> p : count(BUF_LEN + 1) = malloc(sizeof(int) * BUF_LEN);

error: declared bounds for 'p' are invalid after initialization
```

Bounds Declaration for _Array_ptr<T>

```
_Array_ptr<T> p : count(lower_bound, upper_bound) = …;
```

```
#define BUF_LEN 30
_Array_ptr<int> p0 : count(BUF_LEN) = malloc(sizeof(int) * BUF_LEN);

_Array_ptr<int> p1 : bounds(p0, p0 + BUF_LEN / 2) = p0;

int i = p1[15];
          ——————→ error: out-of-bounds memory access
                     int i = p1[15];
                             ^~~~~
```

Design Principles of Checked C

- High performance
  - New checked pointer types and static checking
- Good human readability and long-term maintainability
  - Explicit checked pointers and bounds information of memory objects
- Easy incremental porting legacy C code
  - Mixing checked and legacy C code at fine granularity
  - Good backward compatibility via *bounds-safe interfaces*

Checked Region

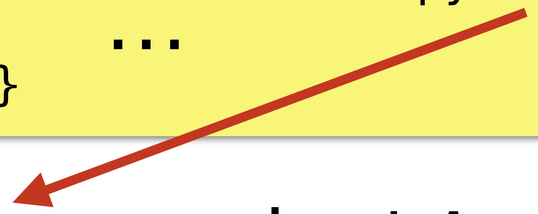
- New keyword: _Checked
  - Annotating a block of code, from a single statement to a whole source file
  - Enforcing more strict typing rules, e.g., only checked pointers allowed
  - Helping programmers to narrow down the scope of memory safety bugs:
    - Checked regions are provably blameless of causing spatial memory safety errors.
- _Unchecked regions: Force the compiler to omit checking

Backward Compatibility with Legacy C Code

- Checked pointers and raw pointers are incompatible by default.

```
char *strncpy(char *dst, const char *src, size_t len);
```

```
void foo() {
    _Array_ptr<char> s1 : count(10) = malloc(10);
    _Array_ptr<char> s2 : count(5) = malloc(5);
    ...
    dst = strncpy(s1, s2, 5);
    ...
}
```



```
error: passing '_Array_ptr<char>' to parameter of incompatible type 'const char *'
    strncpy(s1, s2, 5);
            ^~
```

Backward Compatibility with Legacy C Code

```
char *strncpy(_Array_ptr<char> dst : count(n),
              _Array_ptr<const char> src : count(n), size_t n);
```

```
void foo() {
    _Array_ptr<char> s1 : count(10) = malloc(10);
    _Array_ptr<char> s2 : count(5) = malloc(5);
    ...
    dst = strncpy(s1, s2, 5); legal
    ...
}
```

What about calling `strncpy()` *from unchecked C code?*

```
void bar() {
    char *s1 = malloc(10);
    char *s2 = malloc(5);
    ...
    strncpy(s1, s2, 5);
    ...
}
```

Can we have an mechanism working for both checked and unchecked code?

Bounds-safe Interface for `strncpy()`

```
1  char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
2                const char *src : itype(_Array_ptr<const char>) byte_count(n),
3                size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

`itype` (inter-operation type): A special type that can be either checked or unchecked type, depending on the context

1. `dst` is set to an `itype` with bounds of n bytes.
2. `src` is set to an `itype` with bounds of n bytes.
3. Return value is set to an `itype` with bounds of n bytes.

```
void bar() {
    char *s1 = malloc(10);
    char *s2 = malloc(5);
    ...
    strncpy(s1, s2, 5);
    ...
}
```

✓ *can compile and run, no bounds checks performed*

Bounds-safe Interface for `strncpy()`

```
1  char *strncpy(char *dst : itype(_Array_ptr<char>) byte_count(n),
2                const char *src : itype(_Array_ptr<const char>) byte_count(n),
3                size_t n) : itype(_Array_ptr<T>) byte_count(n);
```

```
void foo() {
    _Array_ptr<char> s1 : count(10) = malloc(10);
    _Array_ptr<char> s2 : count(5) = malloc(5);
    ...
    strncpy(s1, s2, 5);
    ...
}

   strncpy(s1, s2, 6);
```

s1 and s2 are bounds-checked at the call site.

✓ *can compile and run, as both `src` and `dst` are in bounds.*

```
error: argument does not meet declared bounds for 2nd parameter
    strncpy(s1, s2, 6);
                ^~~~
```

Design Principles of Checked C

- High performance
  - New checked pointer types and static checking
- Good human readability and long-term maintainability
  - Explicit checked pointers and bounds information of memory objects
- Easy incremental porting legacy C code
  - Mixing checked and legacy C code at fine granularity
  - Good backward compatibility via *bounds-safe interfaces*

Attempts at a General Definition of Safety

- A programming language can be considered safe if
  - You can trust the *abstractions* provided by the programming language.
    In other words, the language enforces these abstractions and guarantees that they cannot be broken.
    - E.g., a boolean is either true or false, and never 23 or null.
    - Programmers do not need to care how “true/false” is represented in the machine.
- Programs have precise & well defined semantics (i.e., meaning)
  - More generally, leaving things *undefined* in any specification is asking for security trouble.