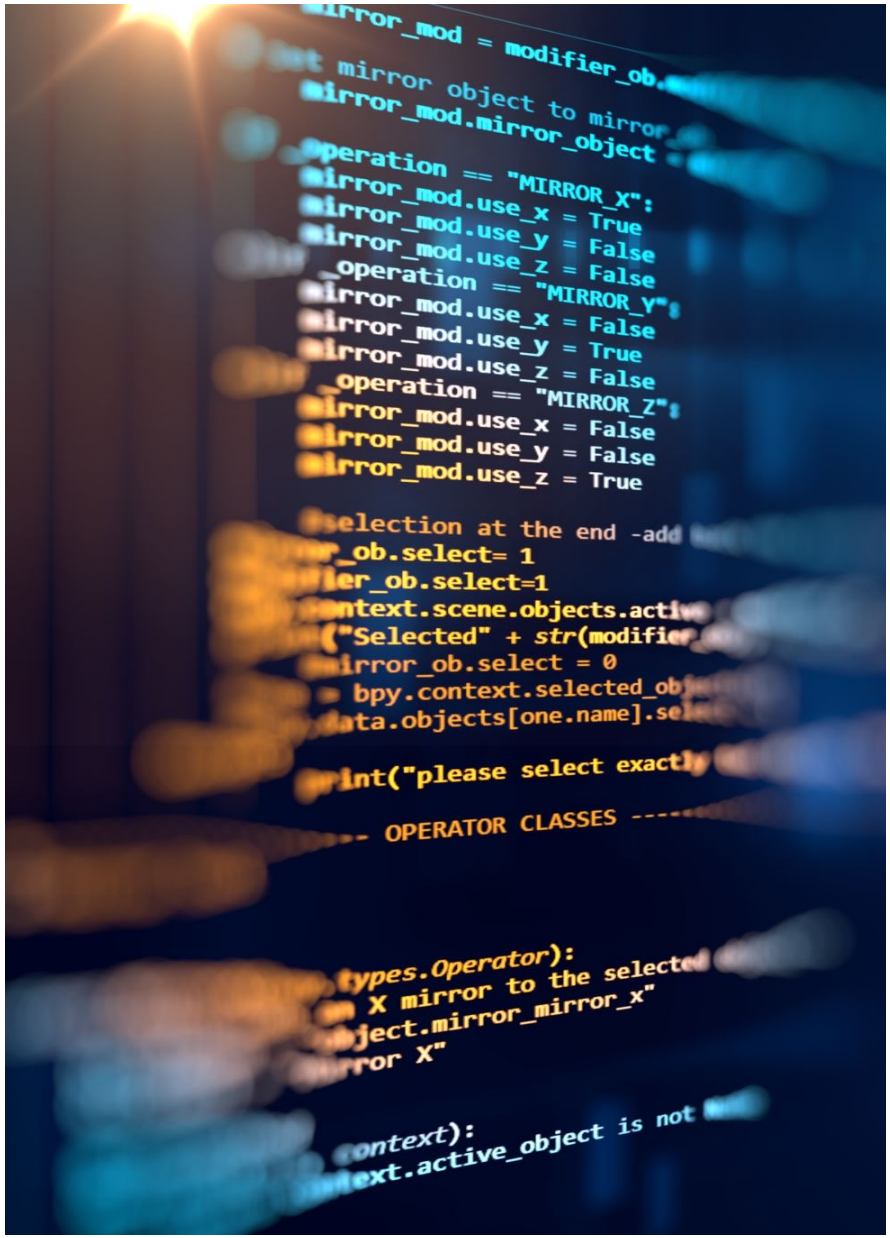

A (small) programming language tour
Major security vulnerabilities



Security vulnerabilities

- Different programming languages come with their own security risks.
- **Factors Influencing Security Vulnerabilities**
  - Design Choices
  - Implementation Decisions
  - Memory Management Models
- **Today's Focus:**
- An overview of key security vulnerabilities categorized by popular programming languages.

LANGUAGE SYNTAX

- The grammar can be found inside the language “reference manual”. So, no possible programmer/compiler misunderstood, everything is fine.

However:

- many examples of (very) bad syntactic choices those effects are
  - to confuse the programmer
  - to confuse the code reviewers . . .
- This opens the way to potential vulnerabilities . .

EXAMPLE: THE C PROGRAMMING LANGUAGE

- The C programming language: Assignments and Booleans.
- In the C language: assignment operator is denoted by **=**
  - **assignment is an expression (it returns a value)**
- no Booleans (in C), integer value 0 interpreted as "false"
  - (well-known) trap for C beginners . . .

≠0 is true in C

Shortcut circuit rule: in C if your first evaluation of AND is false we skip checking the remaining values

EXAMPLE: THE C PROGRAMMING LANGUAGE

```
if ((options==(__WCLONE|__WALL))  && (current->uid=0))
        retval = -EINVAL ;
/* uid is 0 for root */
```

result of assignment is the value computed by expression, which is always 0.

EXAMPLE: THE C PROGRAMMING LANGUAGE

```
if ((options==(__WCLONE|__WALL))  && (current->uid=0))
        retval = -EINVAL ;
/* uid is 0 for root */
```

What it seems to be doing at first glance, is:
"if a particular pair of options are set, and the user is root, then the call is invalid."

THE C CODE

```
if ((options == (__WCLONE|__WALL)) &&
                (current->uid = 0))
retval = -EINVAL;
```

Note the single equals sign in the second half of the if test; it's a single rather than a double equals, assignment rather than equality.

THE C IMPLEMENTATION: CONSEQUENCES

- Short-circuit evaluation. The If test has two parts; the second is only executed if the first is true.
- In C, the return value of an assignment is the value assigned. This is what allows: a = b = c = 3; to be valid C; it sets a, b and c to 3.
- C allows assignment within a logical test. So setting **current->uid to 0** within the if test is perfectly valid C.
- The Unix user_id for root is 0, which is logical false.
- Overall: the "retval = -EINVAL;" code will never be executed, and the function continues on its way, having set the user to root.

Idea: for visibility of variables. I have to address what are the variables that are visible at a certain point. For (1), depends on the syntactic structure of a language, that provides the way to determine the visible variables. The syntax helps. The 2nd you do that at run time and it depends on execution.

VARIABLE SCOPING

- simple mechanism available in the mainstream programming languages
  - Static scoping (an identifier always refers to its declaration environment)
  - Dynamic scoping (an identifier refers to the binding associated with the most recent environment in the execution flow)
  - Dynamic scoping has disappeared from recent programming languages. The exception is exception handling! That essentially uses dynamic scoping to associate a handler with an exception.
- static scoping information is helpful
  - for the compiler to define an appropriate mapping or to apply possible optimisations.

    ↳ This allows for not using variable names. All info about variables is there at compile time. So you can also do stuff like type checking. All because of syntax choices.

ChatGPT

With static scoping, a variable's scope is determined at compile time based on the structure of the code. This means that when a function is defined, the compiler knows which variables it has access to by looking at the surrounding blocks of code.

Example (Python - Static Scoping)

Python Copy code

```python
x = 10  # Global variable

def outer():
    x = 20  # Enclosed variable
    def inner():
        print(x)  # Refers to x in outer() due to static scoping
    inner()

outer()  # Output: 20
```

In this example:

- `inner()` looks for `x` in its own scope first.
- Since `x` is not found in `inner()`, it checks `outer()`, where `x = 20`.
- The global `x = 10` is ignored because `x = 20` is found first in the lexical (static) hierarchy.

With dynamic scoping, a variable's scope is determined at runtime by looking at the call stack. The most recent (or nearest) variable definition in the call chain is used, regardless of the actual structure of the code.

Example (Emulated in Python - Dynamic Scoping)

Python does not support dynamic scoping directly, but it can be emulated using function parameters or specific hacks like modifying `globals()` or `locals()`. However, in languages like Lisp or Bash, dynamic scoping is possible.

Example in **Bash**:

Bash Copy code

```bash
x=10

function outer() {
    local x=20
    inner
}

function inner() {
    echo $x  # Finds x in the calling function (outer)
}

outer  # Output: 20
```

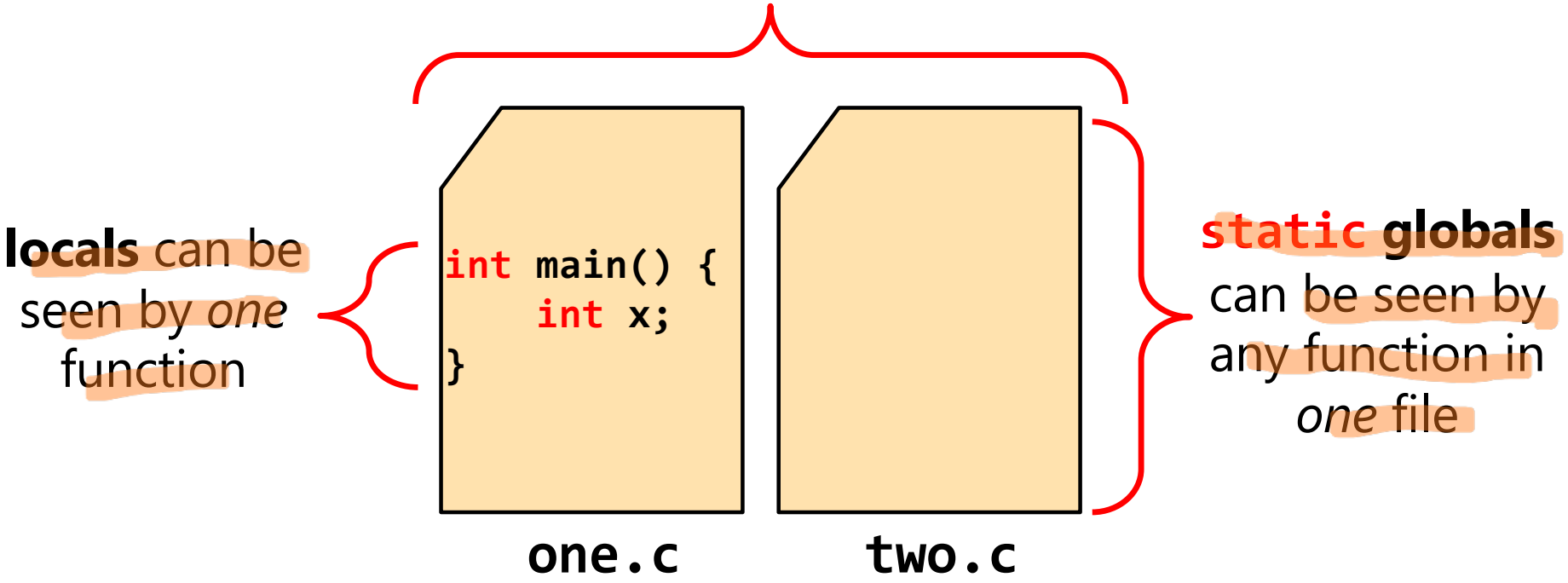
Why Do Exceptions Use Dynamic Scoping?

Unlike variables, exceptions are **not resolved based on where they are written in the code**. Instead, they are **propagated up the call stack at runtime** until they reach a handler that can catch them. This means that the exception is handled by the most recent applicable `catch` or `except` block in the **execution sequence**, not based on where the function was defined.

Scope

- **scope** is "where a name can be seen"
- C has three levels of scope

globals can be seen by *any* function in *any* file



How is the Scope of a Variable important to Security

```
unsigned int count = 0;

void counter() {
  if (count++ > MAX_COUNT) return;
  /* ... */

}
```

Assuming that the variable count is only accessed from the function Counter
What is the security issue in this simple program?

How is the Scope of a Variable **IS** important to Security

```
unsigned int count = 0;

void counter() {
  if (count++ > MAX_COUNT) return;
  /* ... */

}
```

Assuming that the variable count is only accessed from the function Counter
What is the security issue in this simple program?

Global variables are "alive" during the entire program's execution, making them "susceptible to attack"

A better solution

```
void counter() {
  static unsigned int count = 0;
  if (count++ > MAX_COUNT) return;
  /* ... */

}
```

The static modifier, when applied to a local variable, fixes the scope of the variable so that it persists for as long as the program does and does not disappear between invocations of the function.

The static modifier also prevents reinitialization of the variable.

Well … a counter … is not enough

```
public class MySimulator {
    private static MySimulator simulator = null;
    // rep invariant: there should never be more than one MySimulator
    //           object created

    private MySimulator() {
        System.out.println("created a MySimulator object");
    }

    // factory that returns the sole MySimulator object,
    // creating it if it doesn't exist
    public static MySimulator getInstance() {
        if (simulator == null) {
            simulator = new MySimulator();
        }
        return simulator;
    }
}
```

↳ globally known inside compilation unit

If you have several threads without synchronization, you might get multiple instances of Simulator.

```
public class MySimulator {

    private static MySimulator simulator = null;
    // rep invariant: there should never be more than one MySimulator
    //           object created

    private MySimulator() {
        System.out.println("created a MySimulator object");
    }

    // factory that returns the sole MySimulator object,
    // creating it if it doesn't exist
    public static MySimulator getInstance() {
        if (simulator == null) {
            simulator = new MySimulator();
        }
        return simulator;
    }
}
```

This class has a race in the `getInstance()` method – two threads could call it at the same time and end up creating two copies of the `MySimulator` object, which we don't want.

The C programming language

Understanding array in C

- **Array Data Type**
  - An **array** is a **contiguous block of memory** used to store multiple elements of the **same data type**.
  - Each element is stored at a **specific index** and can be accessed using an **index-based notation**
- **Example**
  - `int numbers[5] = {1, 2, 3, 4, 5}; // Declare & initialize`

numbers[5]

Index	Value
0	1
1	2
2	3
3	4
4	5

Intuitive representation

members[3] = 4

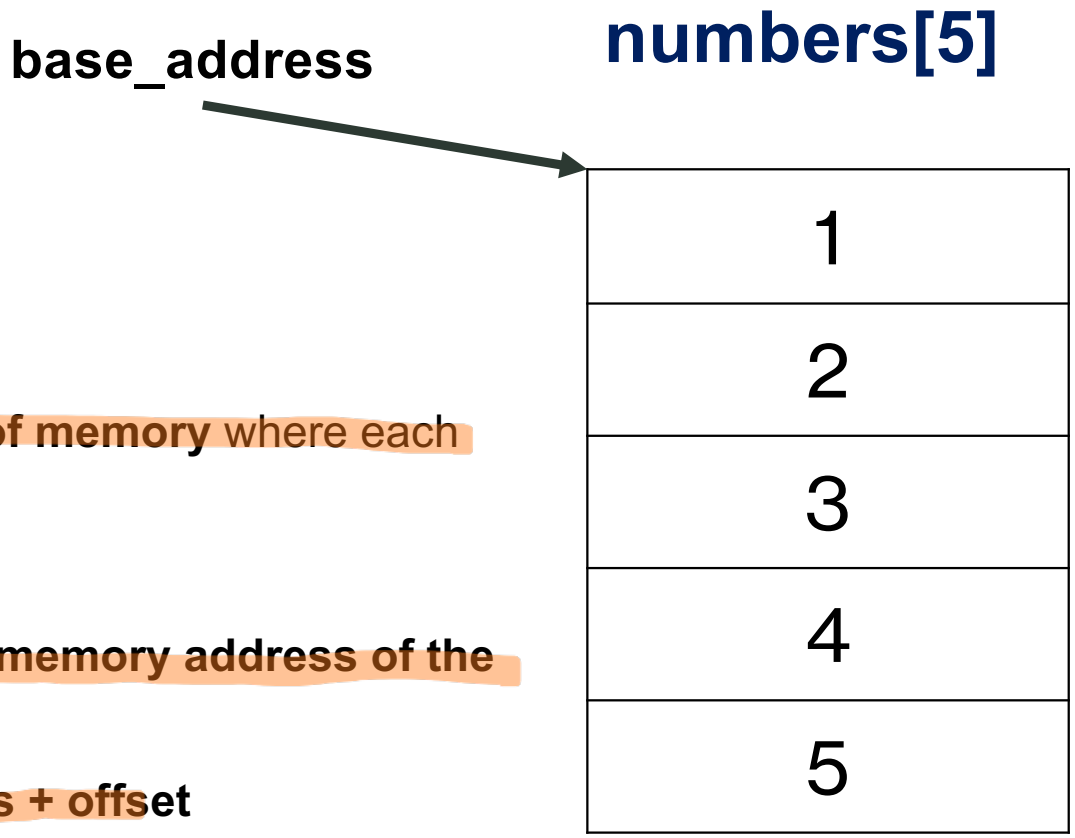
Memory representation

An array is stored as a **single block of memory** where each element is allocated sequentially.

The **base address** of the array is the **memory address of the first element**.

Accessing elements via **base address + offset**

numbers[2] = base_address + 2*sizeof(type_elem)



In C, when an array is declared, its name acts as an **implicit pointer** to the **first element**.

The name of an array does not store an address explicitly, but it is used as a pointer to the first element.

```
int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    printf("Address of arr: %p\n", arr);
 // Base address (same as &arr[0])
    printf("Address of arr[0]: %p\n", &arr[0]);
// Explicit address of first element
    printf("Value at arr[0]: %d\n", *arr);
// Dereferencing gives first element
    printf("Value at arr[1]: %d\n", *(arr + 1));
// Pointer arithmetic

    return 0;
}
```

In C, when an array is declared, its name acts as an **implicit pointer** to the **first element**.

The name of an array does not store an address explicitly, but it is used as a pointer to the first element.

```
int main() {
    int arr[5] = {10, 20, 30, 40, 50};

    printf("Address of arr: %p\n", arr);
 // Base address (same as &arr[0])
    printf("Address of arr[0]: %p\n", &arr[0]);
// Explicit address of first element
    printf("Value at arr[0]: %d\n", *arr);
// Dereferencing gives first element
    printf("Value at arr[1]: %d\n", *(arr + 1));
// Pointer arithmetic

    return 0;
}
```

Address of arr: 0x1000
Address of arr[0]: 0x1000
Value at arr[0]: 10
Value at arr[1]: 20

Example

int arr[3] = {10, 20, 30};

Assuming int is **4 bytes** and 0x1000 as base_address the memory layout (on a **32-bit system**), of the array **arr** might look like:

Index (arr[i])	Value	Address
arr[0]	10	0x1000
arr[1]	20	0x1004
arr[2]	30	0x1008

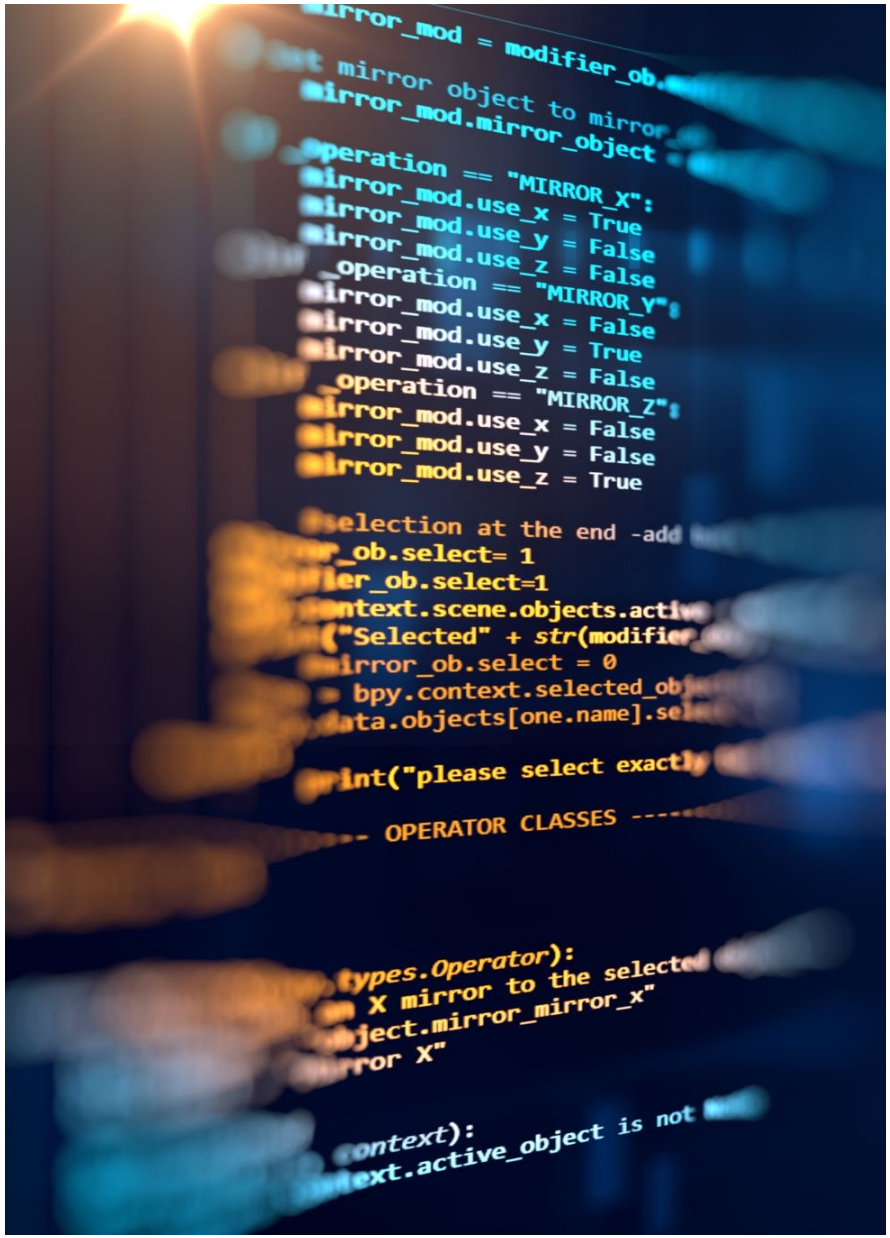
Address of arr[i]=Base Address+(i×Size of Data Type)

arr[2]: 0x1000+(2×4)=0x1008

Arrays vs pointers: summary

	Arrays (arr)	Pointers (ptr)
Holds Memory Address?	Yes, but fixed	Yes, can be changed
Modifiable?	No (constant address)	Yes (can be reassigned)
sizeof() behavior	Full array size	Pointer size (e.g., 8 bytes)
Supports Arithmetic?	Yes, via pointer arithmetic	Yes, via pointer arithmetic

For type checking they are the same



C does not check array bounds

- C was designed as a programming language to develop operation systems
- Design choice: behave **as close to assembly language as possible**, allowing direct memory manipulation.
- **Why?**
  - Performace considerations: Every time an array is accessed, the compiler would need to insert additional instructions to verify whether the index is within bounds.
  - Low-Level Memory Access through ponters and pointer arithmentic is a **key feature** of X

Comparison

	C	Java	Rust
Bounds Checking	No	Yes (Runtime check)	Yes (Compile-time)
Performance Impact	Fast	Slower (extra checks)	Fast (Compile-time)
Undefined Behavior	Yes (dangerous)	No (Exception thrown)	No (Safety enforced)

Stack Memory Allocation for Local Arrays

```
void function() {
    int arr[4] = {10, 20, 30, 40};
//  Stored in stack
    printf("Address of arr: %p\n", arr);
}

int main() {
    function();
    return 0;
}
```

arr is stored in the activation recod of the function in the stack.

The activation record is **deallocated(pop) when function() exits**, making the array inaccessible after returning.

The Buffer Overflow Vulnerability

Try to access an array beyond its boundaries

Buffer overflow occurs when an array is accessed **beyond its allocated boundary**, overwriting adjacent memory.

Since C does **not check array bounds**, this can overwrite **function return addresses, variables, or security-critical data**.

This is bad for reading too

```
void vulnerableFunction() {
    char buffer[8]; // Small buffer

    printf("Enter input: ");
    gets(buffer); //  Deprecated function, allows overflow

    printf("You entered: %s\n", buffer);
}

int main() {
    vulnerableFunction();
    return 0;
}
```

Why is this a security vulnerability?

1. Stack Smashing (Overwriting Return Address)

1. The buffer is only **8 bytes**, but gets() allows unlimited input.
2. If an attacker enters **more than 8 characters**, it overwrites **adjacent memory**.
3. This can **overwrite the function return address**, redirecting execution to **malicious code**.

2.Denial-of-Service (DoS) Attack

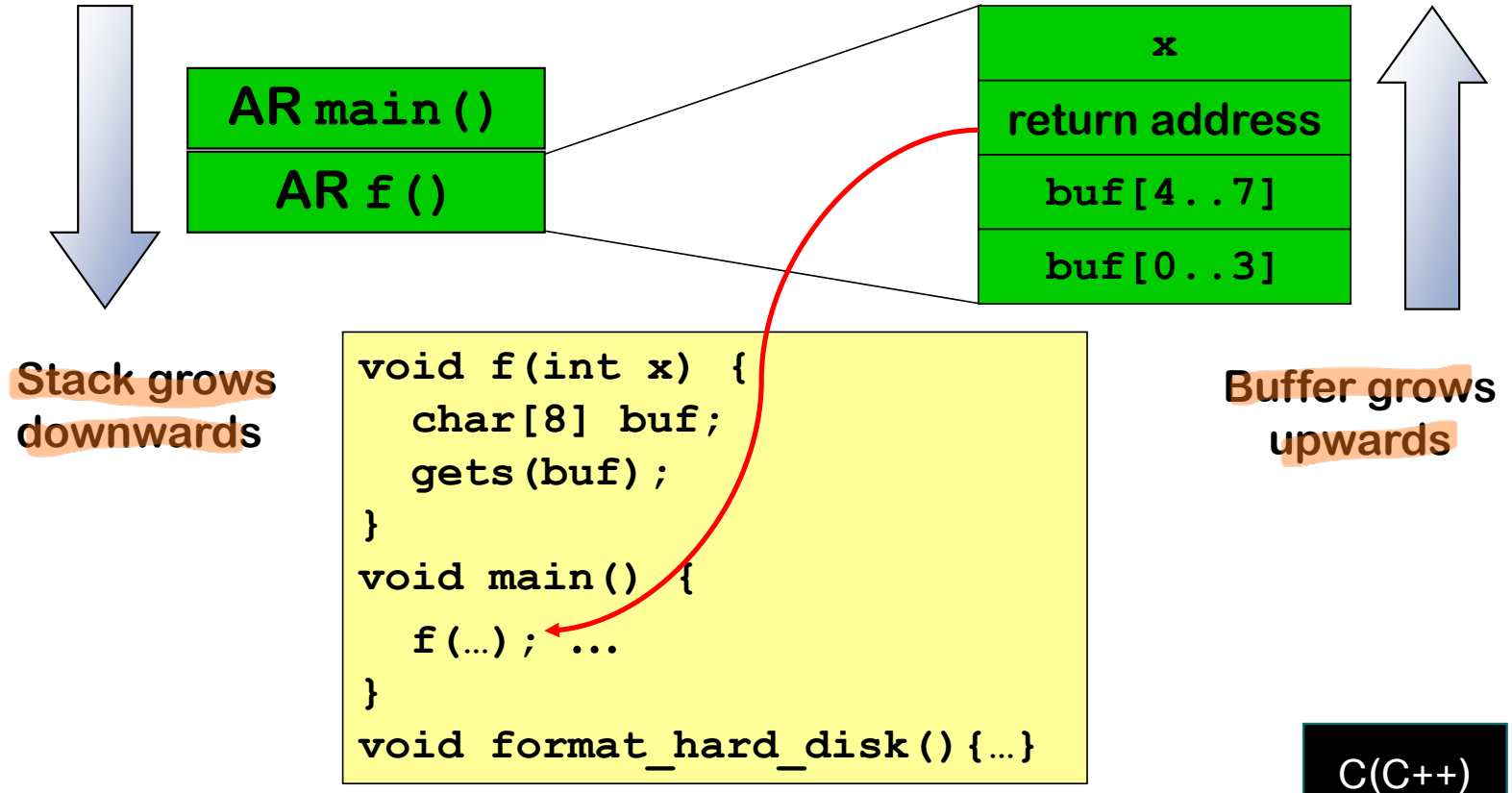
1. Buffer overflow can cause a **segmentation fault**, crashing the program.

3.Remote Code Execution (RCE)

1. Attackers can **inject shellcode** (malicious machine code) to take control of the system.

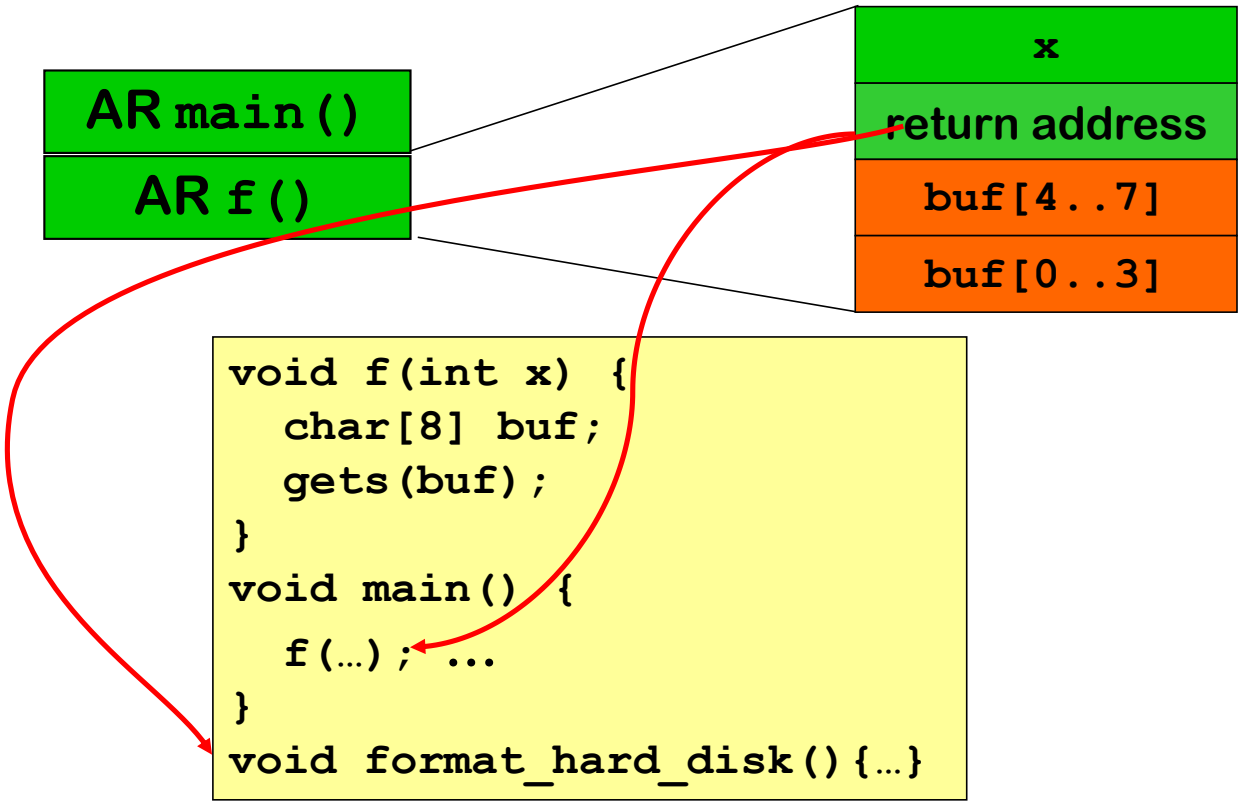
Behavior of the program is now under control of the attacker.
Consequence of design choice

The stack consists of Activation Records:



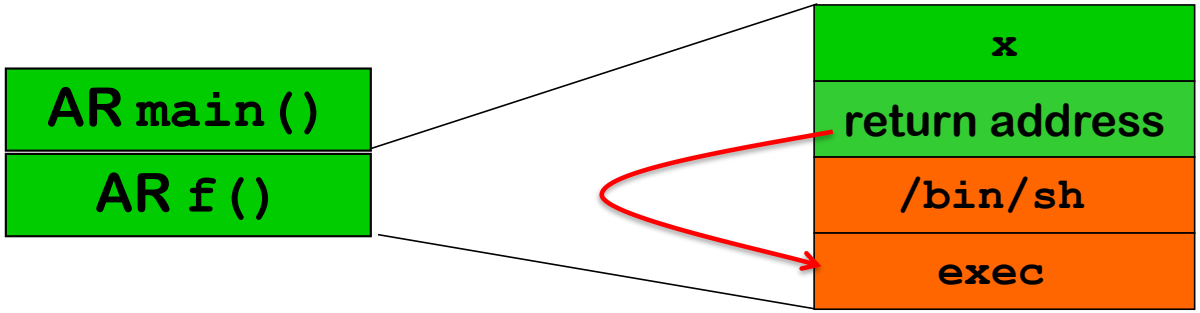
What if `gets()` reads more than 8 bytes ?

Attacker can jump to abitrary point in the code!

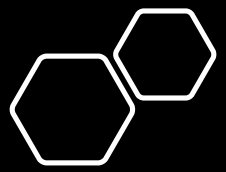


What if `gets()` reads more than 8 bytes ?

Attacker can jump to his own code (aka shell code)

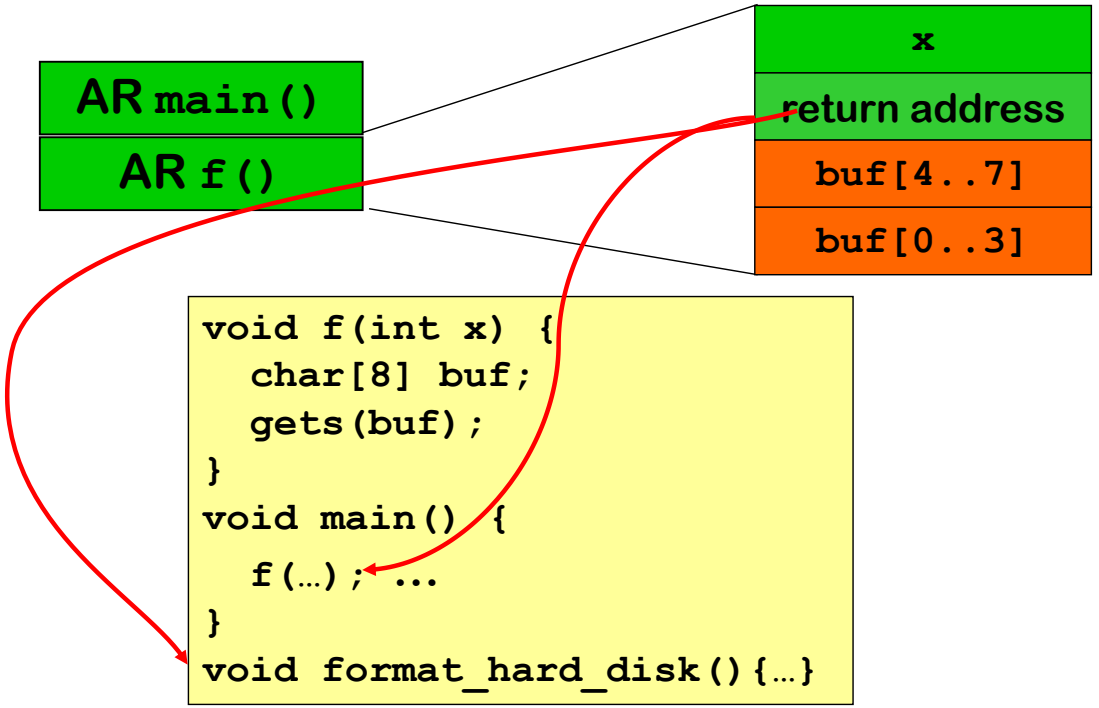


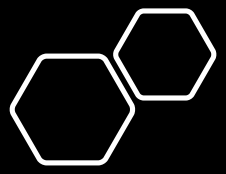

```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); ...
}
void format_hard_disk(){…}
```



code reuse attack:

the attacker corrupts return address to point to existing code (format_hard_disk)

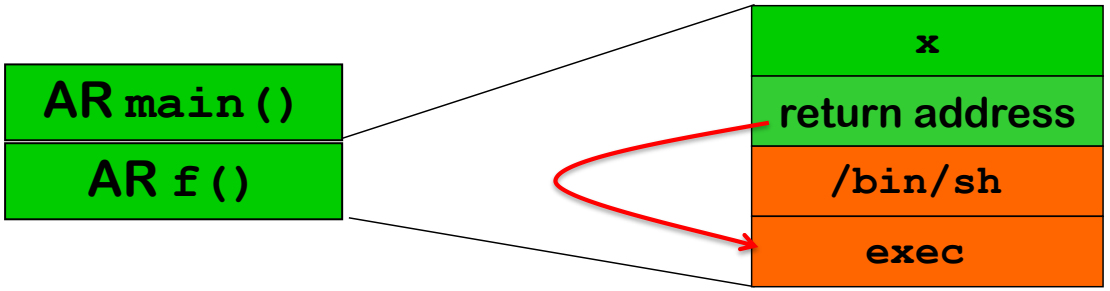




code injection attack

The attacker inserts his own shell code in a buffer and corrupts

return addresss to point to his code (**exec('/bin/sh')**)




```
void f(int x) {
  char[8] buf;
  gets(buf);
}
void main() {
  f(…); …
}
void format_hard_disk(){…}
```

A “real” example

```
void hack_me() {
   char buffer[10];
   gets(buffer);  // Allows overflow
}

int main() {
   hack_me();
   return 0;
}
```

ATTACKER INPUT

AAAAAAAAAAAAAAAA\x90\x90\x90\xEB\x0C\x90\x90\x90

A “real” example

```
void hack_me() {
   char buffer[10];
   gets(buffer);  // Allows overflow
}

int main() {
   hack_me();
   return 0;
}
```

OVERFLOW THE BUFFER

assembly jump instruction, leading to malicious shellcode execution.

ATTACKER INPUT

AAAAAAAAAAAAAAAA\x90\x90\x90\xEB\x0C\x90\x90\x90

Preventing Stack-Based Buffer Overflows

Replace **unsafe functions** (gets(), strcpy(), sprintf()) with **bounds-checking alternatives**.

```
fgets(buffer, sizeof(buffer), stdin);
// Safe alternative to gets()

strncpy(dest, src, sizeof(dest) - 1);
// Safe alternative to strcpy()
```

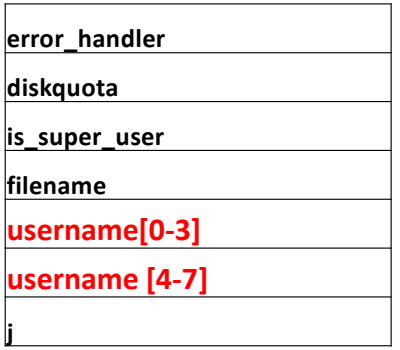
Preventing Stack-Based Buffer Overflows (2)

Compiler Protection (later)

More fun on the stack = more attacks

The threat model:The attacker has enough knowledge on run-time data layout and accesses to run-time data:

```
void f(void(*error_handler)(int),...) {
  int diskquota = 200;
  bool is_super_user = false;
  char* filename = "/tmp/scratchpad";
  char[8] username;
 int j = 12; ...
}
```



The attacker can overflow **username**

More fun on the stack = more attacks

```
void f(void(*error_handler)(int),...) {
  int diskquota = 200;
  bool is_super_user = false;
  char* filename = "/tmp/scratchpad";
  char[8] username;
 int j = 12; ...
}
```

error_handler
diskquota
is_super_user
filename
username[0-3]
username [4-7]
j

Attaccker might corrupt

- pointers (e.g. filename)

- other data on the stack (e.g. is_super_user, diskquota)

- function pointers (e.g. error_handler)

… what about the heap?

Key Characteristics of Heap Allocation in C

Managed manually: The programmer must explicitly allocate (malloc()) and free (free()) memory.

Persists beyond function scope: Unlike stack variables, heap memory is not automatically deallocated when a function exits.

Flexible size: The heap allows allocating memory **at runtime**, which is useful for variable-sized data structures like **linked lists, trees, and dynamic arrays**.

An example

```
int main() {
    int *ptr;

    // Allocate memory for 5 integers on the heap
    ptr = (int*) malloc(5 * sizeof(int));

    // Check if memory allocation was successful
    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Assign values using pointer arithmetic (Base + Offset)
    for (int i = 0; i < 3; i++) {
        *(ptr + i) = (i + 1) * 10; // Store multiples of 10
    }

:
```

Check whether or not ptr is null: if heap is full, malloc does nothing. Won't alert you. It's a very low level design.

An example

```
 :

// Print the values and memory addresses
    printf("Heap-allocated array:\n");
    for (int i = 0; i < 5; i++) {
        printf("ptr[%d] = %d, Address: %p\n", i, *(ptr + i), (ptr + i));
    }

    // Free the allocated memory
    free(ptr);
    return 0;
}
```

Output of the example

Assume ptr starts at location 0x55f1296c62a0

```
// Print the values and memory addresses
    printf("Heap-allocated array:\n");
    for (int i = 0; i < 5; i++) {
        printf("ptr[%d] = %d, Address: %p\n", i, *(ptr + i), (ptr + i));
    }
```

Heap-allocated array:
ptr[0] = 10, Address: 0x55f1296c62a0
ptr[1] = 20, Address: 0x55f1296c62a4
ptr[2] = 30, Address: 0x55f1296c62a8
ptr[3] = 40, Address: 0x55f1296c62ac
ptr[4] = 50, Address: 0x55f1296c62b0

} Contiguous block in the heap, like we would with an array

Base Address + Offset Calculation

Heap memory follows the **same address computation rule** as stack arrays: base Address + Offset

The principle Base + Offset applies universally in C.

↓ Used in almost all prog. languages for Heap

Security Consideration

Heap overflow attacks exploit buffer overflows in dynamically allocated memory.

Example

```
struct BankAccount {
  int number;
  char username[20];
  int balance;
}
```

Suppose attacker can overflow username

Example

```
struct BankAccount {
  int number;
  char username[20];
  int balance;
}
```

Suppose attacker can overflow username

This can corrupt other fields in the struct.

Which field(s) can be corrupted depends on the order of the fields in memory

↳ depends on compiler. Attacker should understand policy behind this

Stck vs Heap Overflow

	Stack Buffer Overflow	Heap Buffer Overflow
Memory Location	Stack (local variables)	Heap (dynamically allocated)
Impact	Overwrites **return address, +++**	Corrupts **heap structures**
Exploitation	Control flow hijacking (RCE)	Heap spraying attacks
Mitigation	Later	Later

give a new color to the value on the heap

C Data type: Integer

```
#include <stdio.h>
#include <limits.h>  // Defines
INT_MAX

int main() {
    unsigned int x = UINT_MAX; //
Maximum value for unsigned int
    printf("Before: %u\n", x);

    x = x + 1; // Overflow occurs
    printf("After %u\n", x);
    return 0;
}
```

Undefined behavior (falls outside the standard of C).

In C, **undefined behavior (UB)** refers to any operation that the C standard does not define, meaning the compiler is free to handle it however it chooses. When UB occurs, the program may produce unpredictable results, crash, or even seem to work correctly but behave inconsistently across different compilers or environments.

Examples of Undefined Behavior:

- **Dereferencing a null or uninitialized pointer**
- **Out-of-bounds array access**
- **Signed integer overflow** (e.g., `INT_MAX + 1`)
- **Modifying a variable multiple times in a single expression without sequencing** (e.g., `i = i++ + ++i;`)
- **Using uninitialized variables**

Because UB allows compilers to assume it never happens, they may optimize aggressively, sometimes removing entire sections of code if they infer that UB would occur. This is why relying on UB can lead to extremely unpredictable behavior.

C Data type: Integer

```
#include <stdio.h>
#include <limits.h>  // Defines
INT_MAX

int main() {
    unsigned int x = UINT_MAX; //
Maximum value for unsigned int
    printf("Before: %u\n", x);

    x = x + 1; // Overflow occurs
    printf("After %u\n", x);
    return 0;
}
```

Before : 4294967295
After : 0

Integer overflow and underflow

Integer Overflow occurs when a computation **exceeds the maximum limit** of an integer type, causing it to **wrap around** to a negative or unexpected value.

Integer Underflow occurs when a computation **drops below the minimum limit** of an integer type, causing a **wrap-around to a large positive number**.

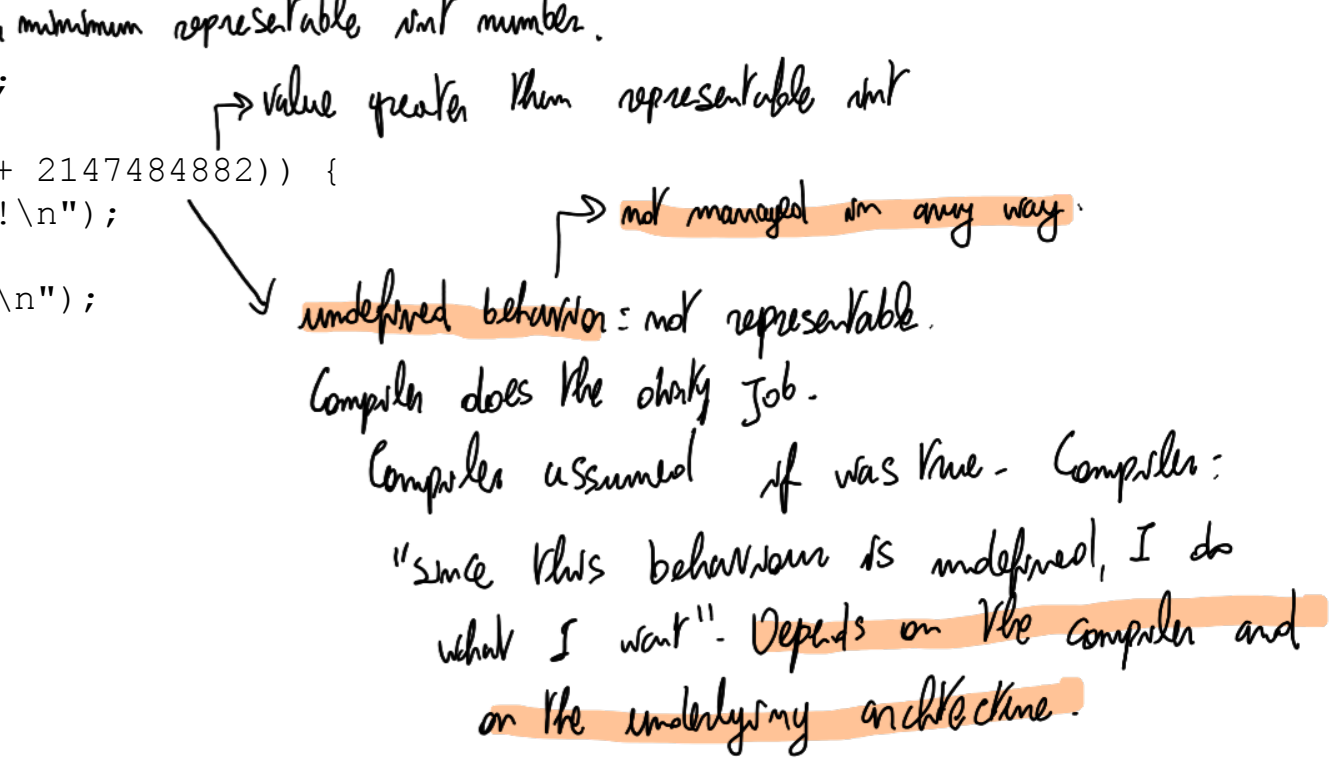
Why is this a security risk?

Attackers **exploit overflow/underflow** to **bypass security checks, escalate privileges, or trigger buffer overflows**.

Common examples: **authentication systems, memory allocations, and financial calculations**.

Bypassing Security Checks

```
int authenticate(int pin) {
    int stored_pin = 1234;
    if (pin == stored_pin) {
        return 1; // Access granted
    }
    return 0; // Access denied
}
int main() {
    int user_pin = -2147483648;
// Manipulated input
    if (authenticate(user_pin + 2147484882)) {
        printf("Access Granted!\n");
    } else {
        printf("Access Denied!\n");
    }
    return 0;
}
```



Signed Integer Overflow (Undefined Behavior)

- In C, **signed integer overflow is explicitly undefined behavior (UB)** according to the standard (ISO C11 §6.5/5).
- This means the compiler **does not have to assume wraparound behavior** and is free to optimize aggressively.
- Some compilers, when optimizing, might remove code that relies on overflow, leading to unexpected results (e.g., `if (x + 1 < x)` might be removed since overflow "shouldn't happen").
- However, many modern CPUs **do** wrap around naturally when using two's complement representation (which is nearly universal), causing `INT_MAX + 1` to become `INT_MIN`, but relying on this behavior is risky because it's UB in C.

```c

#include <stdio.h>

int authenticate(int pin) {
    int stored_pin = 1234;
    if (pin == stored_pin) {
        return 1; // Access granted
    }
    return 0; // Access denied
}

int main() {
    int user_pin = -2147483648; // Manipulated input
    if (authenticate(user_pin + 2147484882)) {
        printf("Access Granted!\n");
    } else {
        printf("Access Denied!\n");
    }
    return 0;
}

```

STDIN

Input for the program (Optional)

Output:

```
Access Granted!
```

Bypassing Security Checks

```
int authenticate(int pin) {
    int stored_pin = 1234;
    if (pin == stored_pin) {
        return 1; // Access granted
    }
    return 0; // Access denied
}
int main() {
    int user_pin = -2147483648;
// Manipulated input
    if (authenticate(user_pin +
2147484882)) {
        printf("Access Granted!\n");
    } else {
        printf("Access Denied!\n");
    }
    return 0;
}
```

Overflow results in **undefined behavior** in C. The outcome depends on the compiler and system architecture.

DATA ABSTRACTION

- **Data abstraction: separation between the abstract properties of a data type and the concrete details of its implementation.**
- **The abstract properties visible to client code that makes use of the data type—the interface to the data type—while the concrete implementation is kept entirely private, and indeed can change.**
  - **changes involve no difference in the abstract behaviour.**

DATA ABSTRACTION IN OOP

- In object-oriented programming (OOP), encapsulation refers to the
  - ability of wrapping data with the methods that operate on that data,
  - restricting of direct access to some of an object's components.
- **Encapsulation is used to hide the values or state of a structured data object inside a class, preventing direct access to them by clients in a way that could expose hidden implementation details or violate state invariance maintained by the methods.**

JAVA CLASS

```
class Secret {
   private int x = 42;
 }
```

The instance variable x cannot be accessed by clients of the Secret class

Is encapsulation a security abstractions?

```
import java.lang.reflect.*;

class Secret { private int x = 42; }

public class Introspect {
  public static void main (String[] args) {
    try { Secret o = new Secret();
          Class c = o.getClass();
          Field f = c.getDeclaredField("x");
          f.setAccessible(true);
          System.out.println("x="+f.getInt(o));
        }
    catch (Exception e) { System.out.println(e); }
  }
}
```

→ meta programming package to inspect structure of a class.

provides class name as result

↳ grasping class details

```
import java.lang.reflect.*;

class Secret { private int x = 42; }

public class Introspect {
  public static void main (String[] args) {
    try { Secret o = new Secret();
          Class c = o.getClass();
          Field f = c.getDeclaredField("x");
          f.setAccessible(true);
          System.out.println("x="+f.getInt(o));
        }
    catch (Exception e) { System.out.println(e); }
  }
}
```

The printed result is x=42: introspection is used to dynamically modify the class definition and remove the private tag. You can modify the wall of incapsulation with introspection.

RIGHT ....

We forbid the use of introspection in JAVA by exploiting (the so-called) security monitor

This can be done!!

Side effects on standard libraries: serialization in JAVA uses introspection.

CALLED LIKE THIS

If you send a bytecode on the network you serialize and serialization libraries work with introspection to reconstruct bytecode.

INNER CLASSES

```
class OuterClass {
 int x = 10;
   class InnerClass {
     public int myInnerMethod() {
     return x;
    }
}
public class Main {
public static void main(String[] args) {
  OuterClass myOuter = new OuterClass();
  OuterClass.InnerClass myInner = myOuter.new InnerClass();
  System.out.println(myInner.myInnerMethod());
  }
} // Outputs 10
```

→ local class. Cannot be used outside of the outerclass.

1. **Encapsulation and Introspection**
   - **Encapsulation** in Java is about restricting direct access to object fields and methods, typically using `private` and `protected` modifiers.
   - **Introspection (Reflection)** allows bypassing encapsulation by letting code inspect and modify private fields and methods at runtime (e.g., via `java.lang.reflect`).
2. **Security Monitor Forbidding Introspection**
   - Java has a **Security Manager** (deprecated in Java 17) that could restrict reflection to enforce security policies.
   - If reflection is **forbidden**, it prevents unauthorized code from breaking encapsulation and modifying private fields dynamically.
   - This can be useful in secure environments where you don't want untrusted code tampering with objects.
3. **Side Effect: Java Serialization Uses Introspection**
   - **Serialization** in Java (`ObjectOutputStream` and `ObjectInputStream`) often uses reflection to access private fields when reconstructing objects.
   - If introspection is disabled, serialization might fail because it can't access private fields needed to restore objects.

Serialization in Java is the process of converting an object into a **byte stream**, so it can be saved to a file, sent over a network, or stored in a database. This allows objects to be **recreated later** by deserializing the byte stream back into an object.

INNER CLASSES

What about Java Inner Classes in CVE

https://cwe.mitre.org/data/definitions/492.html

CWE-492: Use of Inner Class Containing Sensitive Data

Weakness ID: 492
Abstraction: Variant
Structure: Simple

View customized information: Conceptual | Operational | Mapping Friendly | **Complete** | Custom

Description

Inner classes are translated into classes that are accessible at package scope and may expose code that the programmer intended to keep private to attackers.

Extended Description

Inner classes quietly introduce several security concerns because of the way they are translated into Java bytecode. In Java source code, it appears that an inner class can be declared to be accessible only by the enclosing class, but Java bytecode has no concept of an inner class, so the compiler must transform an inner class declaration into a peer class with package level access to the original outer class. More insidiously, since an inner class can access private fields in its enclosing class, once an inner class becomes a peer class in bytecode, the compiler converts private fields accessed by the inner class into protected fields.

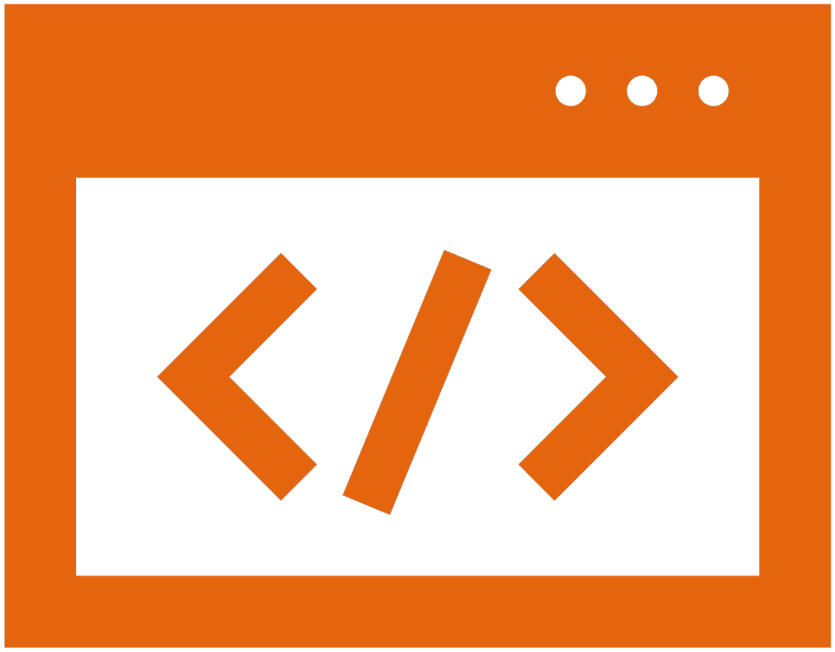
Relationships

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name
ChildOf		668	Exposure of Resource to Wrong Sphere

SUMMARY

Encapsulation mechanisms are convenient design tools of software engineering. On the other hand, it should be clear for developers that they are not, in general, security mechanisms.



WE NOW MOVE TO MEMORY SAFETY

THE WORLD OF MEMORY

local variables are allocated on the stack

- their lifetime is the lifetime of the function call

global variables are put in the **data segment**

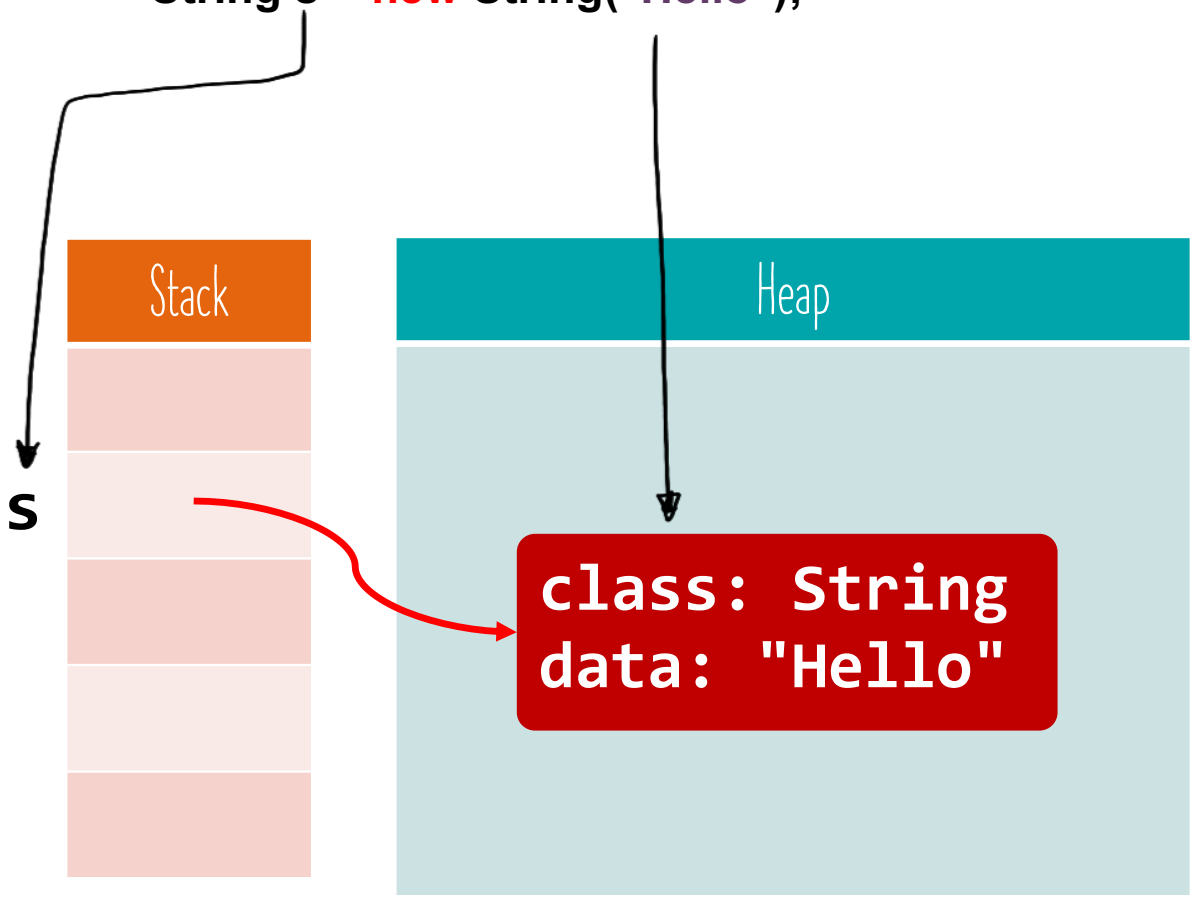
- this area is **automatically allocated** by the OS
- its **lifetime** lasts for your entire program
- there is also a **read-only** data segment for constants

But ... what about pointers and dynamic data stractures

- new Object() w

- the **heap** is a part of the memory space independent of the stack and data segment.
- Example: in Java, the **new** operator creates a new heap object.

```
String s = new String("Hello");
```



new *allocates* memory on the heap and gives you a *pointer* to it.

new marks the beginning of the ***object's*** lifetime, which is **independent** of the variable(s) that point to it.

HEAP IN THE C-FAMILY

heap managed by the programmer

- to allocate memory, use **malloc:**

  All management is under the control of the programmer

  ```
  int* arr = malloc(sizeof(int) * 20);
  ```

  - this allocates a 20-item int array **on the heap**
  - **malloc** takes the number of **bytes** to allocate, makes a block of bytes *at least* that big, and returns a pointer to it
- when you're done with that block of memory, you use **free:**

  ```
  free(arr);
  ```

- since the programmers are the ones deciding when to deallocate the piece of memory… ***programmers* are the owners!**

PLS DO NOT DO THIS!!

```
while(1)
    malloc(1048576);
```

- The program will run out of memory
  - but you won't get an error
  - this will **loop forever** because…
  - if you run out of memory, **MALLOC RETURNS NULL**

EVEN WORSE

```
int* arr = malloc(sizeof(int) * 20);
free(arr);
arr[0] = 1000; // dangling reference
```

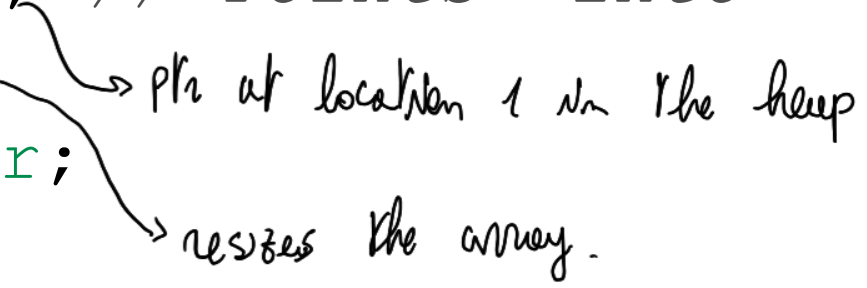
↳ value of array is null?

- when you free heap memory, **all pointers to it become invalid, and it's your responsibility to never use them again**

```
int* createInt() {
    int value = 42;
    int* ptr = &value;
    return ptr;
}
int main() {
    int* dangling_ptr = createInt();

    int result = *dangling_ptr;

    std::cout << "Dereferenced value: " << result << std::endl;

    return 0;
}
```

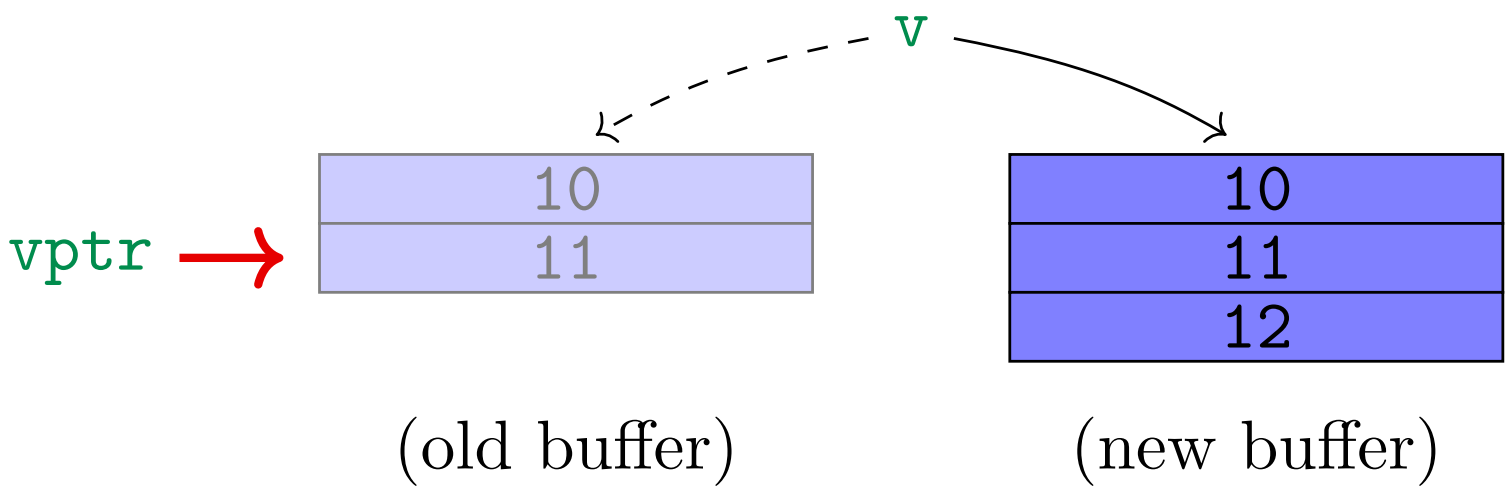
```
int* createInt() {
    int value = 42;
    int* ptr = &value; // ptr points to a local variable 'value'
    return ptr; // Returning a pointer to a local variable is problematic
}
int main() {
    int* dangling_ptr = createInt();

    // The 'createInt' function has returned, and 'value' has gone out of scope,
    // leaving 'dangling_ptr' pointing to invalid memory.

    // Attempting to dereference the dangling pointer is undefined behavior
    int result = *dangling_ptr;

    // Undefined behavior can lead to crashes, incorrect results, or other issues
    std::cout << "Dereferenced value: " << result << std::endl;

    return 0;
}
```

```
1 std::vector<int> v { 10, 11 };
2 int *vptr = &v[1]; // Points *into* 'v'.
3 v.push_back(12);
4 std::cout << *vptr;
```



```
1 std::vector<int> v { 10, 11 }; //a resizable array of integers
2 int *vptr = &v[1]; // Points *into* 'v'. A partial overlap
3 v.push_back(12); // Here no more space for an additional element
                  // a new buffer is allocated
                  // all the existing elements are moved over.
4 std::cout << *vptr;
```

Why is this case interesting?



MEMORY USAGE: AN INTERESATING CASE

- both pointers were aliasing: an action through a pointer (v) will in general also affect all its aliases (vptr)
- vptr becomes a dangling reference!!!

A GENERAL PATTERN

- *iterator invalidation:*
  - *a container data structure (collection)*
  - *an iterator over the collection*
  - *the iterator, accidentally, calls an operation that mutates the data structure.*
- In our example assume that the pointer vptr is maintained internally by the iterator.

Douglas P. Gregor and Sibylle Schupp. “Making the usage of STL safe”. In: *IFIP TC2/WG2.1 Working Conference on Generic Programming*. July 2002

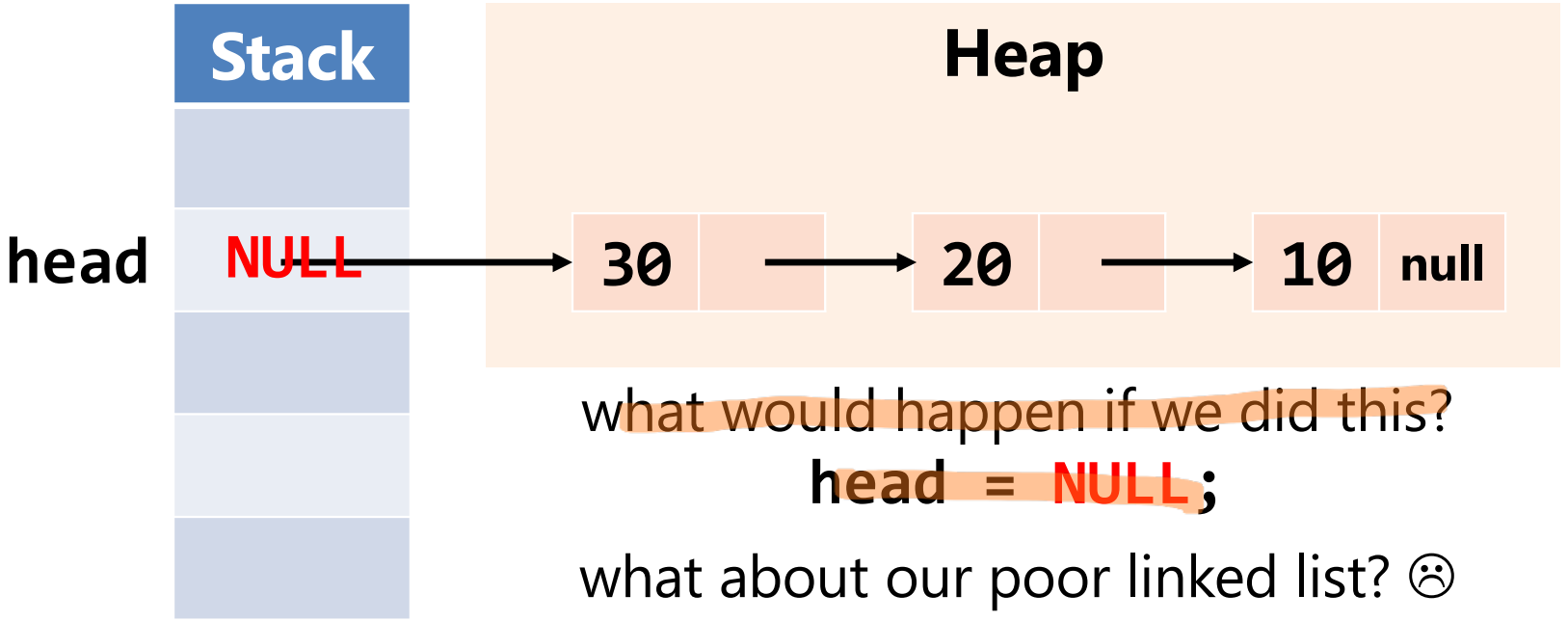
Yes! This slide is referring to **iterator invalidation**, a common issue in C++ when working with **iterators** and **modifiable container data structures** (like `std::vector`, `std::list`, or `std::map`). It happens when an iterator becomes **invalid** due to modifications in the underlying container, leading to **undefined behavior** if used afterward.

Breaking It Down:

1. **A Container Data Structure**
   - This refers to a **data structure** that holds multiple elements, like `std::vector`, `std::list`, or `std::unordered_map`.
2. **An Iterator Over the Collection**
   - An **iterator** is an object used to traverse elements in the container (e.g., `std::vector<int>::iterator it`).
3. **The Iterator Accidentally Calls an Operation That Mutates the Data Structure**
   - If an operation **modifies the container** (e.g., inserting, deleting, or resizing elements), the **iterator may become invalid**.
   - Accessing an invalid iterator leads to **undefined behavior** (crashes, corrupted data, etc.).

What's Garbage Collection?

Operating over linked lists



what would happen if we did this?

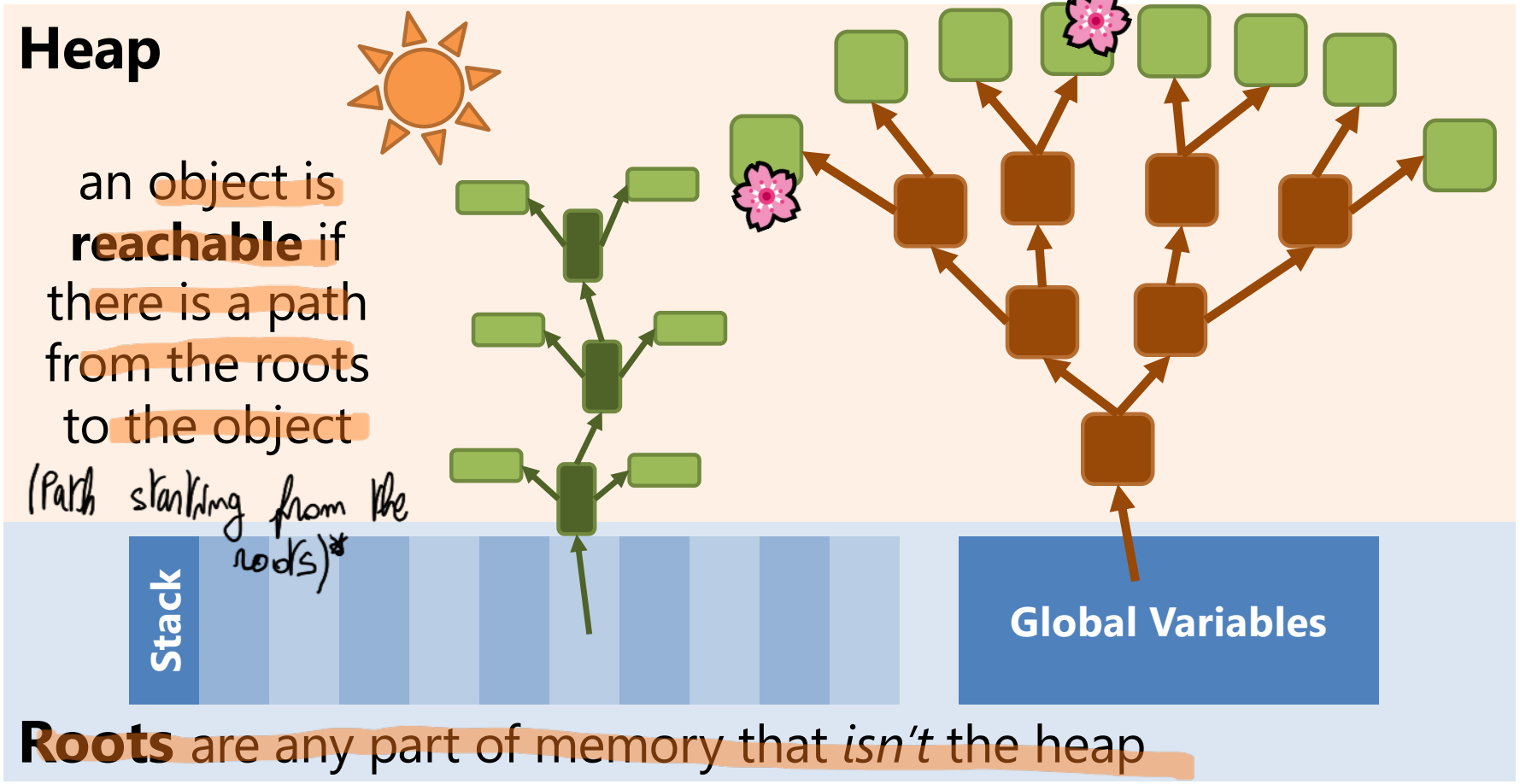
```
head = NULL;
```

what about our poor linked list? ☹

who will come along and clean up the mess you did?

The garbage collection story

- there are two useful concepts from GC: **roots** and **reachability**

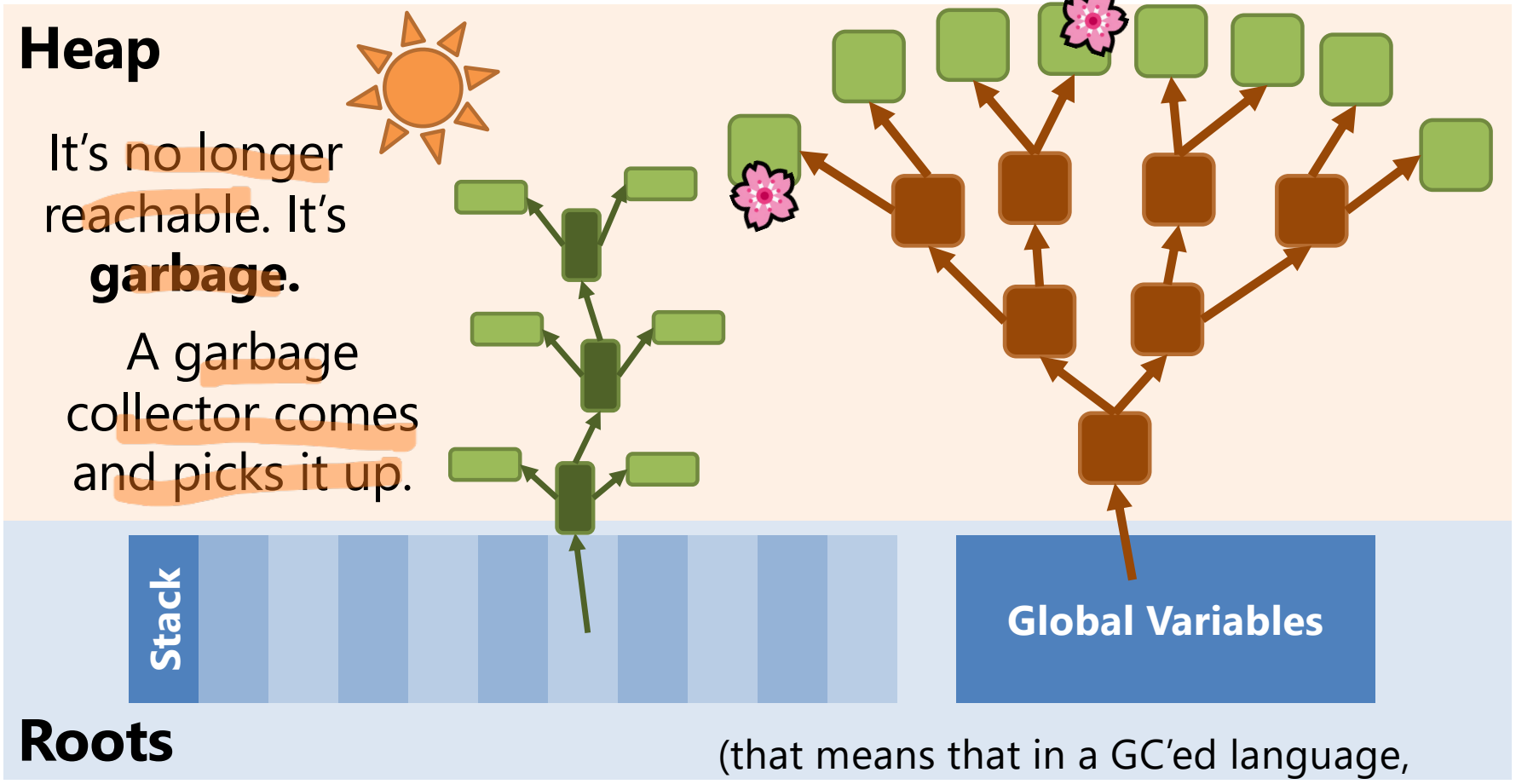


Variables in the stack or global variables pointing to the heap

* An object is reachable if it is reachable from the roots or if it is reachable from a reachable object.

Mark and Sweep

- If we remove the only link to an object…



(that means that in a GC'ed language, *who* owns all the heap memory?)

So we need a system to be able to know if an element is reachable for the GC. First el. of GC is called Marking (mark as unused). Then second part is picking it up.

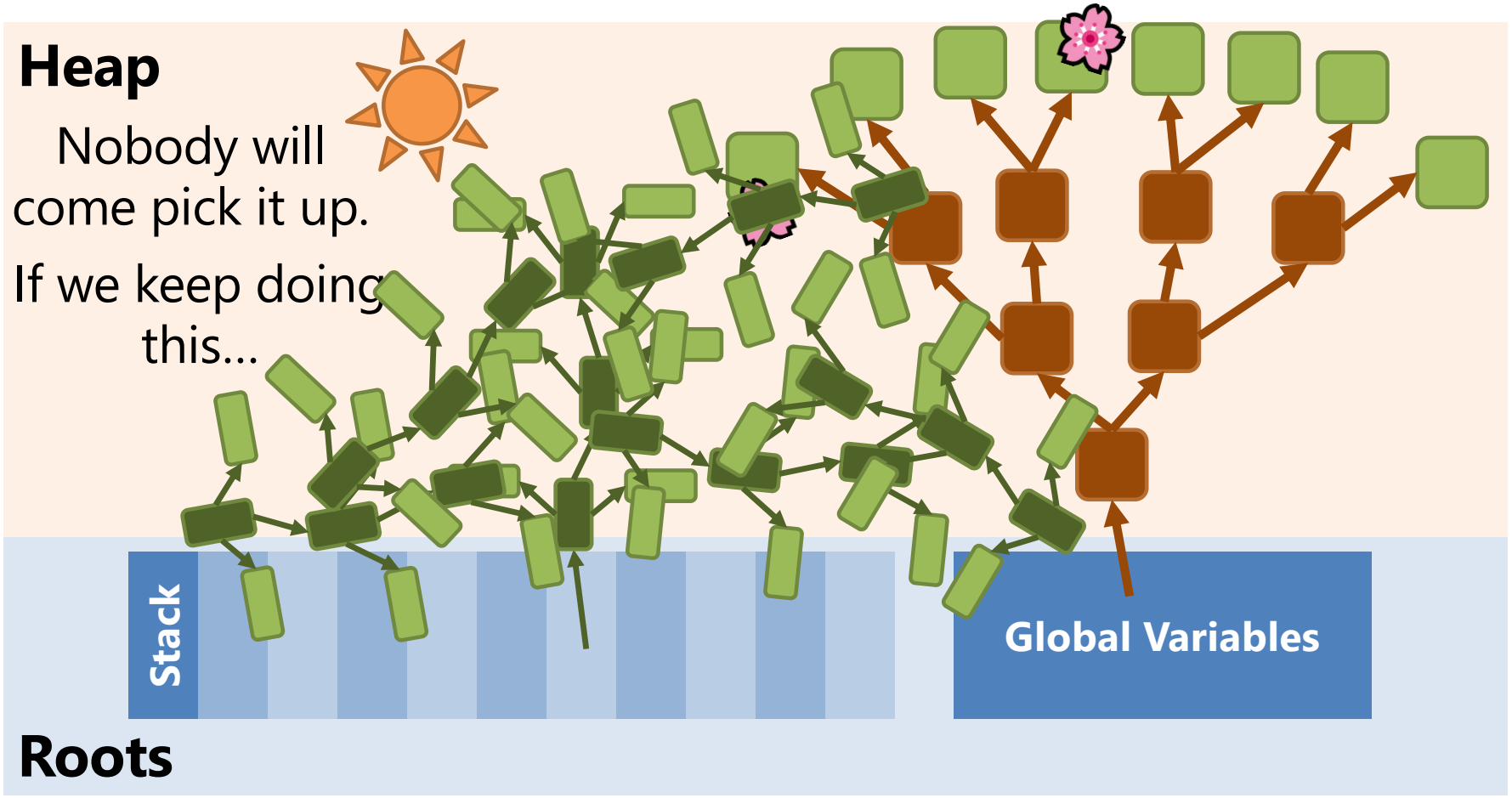
In garbage collection, **marking** is the first phase of the **mark-and-sweep** algorithm. During this process, the garbage collector identifies all objects that are still reachable and marks them as **alive**.

How It Works:

1. **Start from Root References** – The collector begins with a set of "roots," which typically include global variables, static fields, and local variables in active function calls.
2. **Traverse the Object Graph** – It follows all references from these roots, recursively visiting objects that are still in use.
3. **Marking** – Every visited object is marked as alive, meaning it should not be collected.
4. **Unmarked Objects Are Garbage** – Any object that is **not** marked is considered unreachable and will be removed in the next phase (sweeping).

This marking process ensures that only objects that are no longer in use are eventually collected, helping prevent accidental deletion of active data.

What about C ….


Heap
Nobody will come pick it up.
If we keep doing this…
Stack
Global Variables
Roots

MEMORY LEAKS

C language: if heap memory becomes unreachable, it's a memory leak, the only way to deallocate it is to exit the program.

Research have gone into making GC useful!. But when programming you never take it for granted!

MEMORY SAFETY

C is not memory safe.
Java is so because of GC.

Rust is a multi-paradigm, general-purpose programming language that enforces memory safety (meaning that all references point to valid memory)
without a garbage collector.

More in Rust later

TYPES IN PROGRAMMING LANGUAGE

Types are an abstraction made available by language

TYPES IN PROGRAMMING LANGUAGES

- The notion of type provides a data abstraction mechanisms
  - It defines the set of **values** an expression can take at run-time
  - It defines the set of **operations** that can be applied to an identifier
  - It defines the **signature** of these operations [what types to be expected as input and output]
  - It defines how variables should be **declared**, **initialized**, etc.
  - (formal) type systems are used to drive the implemention of type-checking algorithms

TYPES IN PROGRAMMING LANGUAGES

- The notion of type provides a data abstraction mechanisms
- Type system allows to (safely) reject some meaningless syntactically correct programs
  - type-checking is a must for secure developments, preferably both static and strong to ensure early detection of ill-formed constructs.
- Types in programming languages:
  - **typed** vs **untyped** languages
  - type **checking** and/or type **inference*** (1: you have a program and statically check (one component checks if your working respects the rules of type system).
  - **static** and/or **dynamic** type checking/inference

* Used in OCaml

- **Type Checking** is the process of verifying that values and expressions in a program conform to expected types. This can be done **statically** (at compile time, as in C, Java, or Rust) or **dynamically** (at runtime, as in Python or JavaScript). Type checking ensures that a function expecting an integer does not receive a string, for example, reducing potential runtime errors.
- **Type Inference** is when the compiler or interpreter automatically determines the type of an expression without explicit type annotations from the programmer. For example, in languages like Haskell or Rust, writing `let x = 42;` allows the compiler to infer that `x` is an integer without requiring `let x: i32 = 42;`.

Essentially, type checking ensures correctness, while type inference reduces the need for explicit type declarations, making code more concise. Some languages, like TypeScript, do both: they infer types where possible but still perform strict type checking.

Robin Milner

Turing Award 1991 For

- LCF – Automated Theorem Proving
- ML – Programming Language (Meta language for defining proof for theorem prover)
  - With Polymorphic Type Inference
  - Type-Safe Exception Handling
- CCS – A General Theory of Concurrency

"Well-typed programs never go wrong . . . " [Robin Milner]

TYPES AS A SECURITY SAFEGUARD

- Type safety: **NO** meaningless well-typed programs
  - "out of semantic" programs are not executed,
  - no **untrapped** run-time errors,
  - no undefined behaviors, . . .
- **According to this definition:**
- C, C++ are **not** type safe
- OCAML, Java, C# are type safe

- "**No meaningless well-typed programs**" → If a program is well-typed (meaning it follows the type rules of the language), then it should always make sense semantically. In other words, you won't get bizarre results just because of a type issue.
- "**Out of semantic programs are not executed**" → If something doesn't make sense in the language's type system, the program won't run. This prevents errors from propagating.
- "**No untrapped run-time errors**" → Errors related to type mismatches should be caught early, either at compile time or by built-in safety mechanisms at runtime.
- "**No undefined behaviors**" → The program should never enter an unpredictable state due to type-related errors.

According to this definition, **C and C++ are not type safe** because they allow operations that can cause undefined behavior, like accessing memory in unintended ways (e.g., dereferencing wild pointers, casting between incompatible types, buffer overflows). Meanwhile, **Java and C# are considered type safe** because they enforce strict type rules and runtime checks to prevent such issues, reducing the likelihood of crashes or unpredictable results.

ABOUT TYPE SAFE LANGUAGES

- Well-typedness is **preserved** at execution
- (bit strings) values are processed according to their (programming level) types
- (meaningless) ill-typed programs are rejected either at **compile time** or at **execution time**
- But type systems are usually incomplete
  - They may also reject **meaningful** programs

TYPES AS A SECURITY SAFEGUARD

Weakly typed languages:

- implicit type cast/conversions
  - integer ~ float, string ~ integer, etc.
- operator overloading
  - + for addition between integers and/or floats
  - + for string concatenation
  - + for pointer arithmetic
- Overall: weaken type checking may confuse the programmer . . .
  (runtime type may not match with the intended operation performed)

WEAK TYPE SYSTEMS IN PRACTICE

Many of the type systems you use are weak type systems

- It happens in many widely used programming languages . . . (C, C++, PHP, JavaScript, etc.)
- It may depend on compiler options / decisions
- often exacerbated by a lack of clear and un-ambiguous documentation

use same symbol (ex: +) to represent an operation over two different domains (ex: int and floats)

CASRTING AND OVERALOADING

- To ease developer's work programming language toolchains provide automated mechanisms allowing for overloading (using the same name for different operations) and automated casts or coercions.
- Casts and overloading do not provide a comfort zone but often a disguise.
  - We briefly discuss the consistency of design choices.

EXAMPLE: C

```
int x=3;
int y=4;
float z=x/y;
```

Is it correct, what's the value of z ?

EXAMPLE: C

```
int x=3;
int y=4;
float z=x/y;
```

Is it correct, what's the value of z ?

0.000000

Correct from pov of types, but unexpected behaviour

EXAMPLE: JAVA

```
short x = Short.MAX_VALUE;
System.out.println(x+1);
```

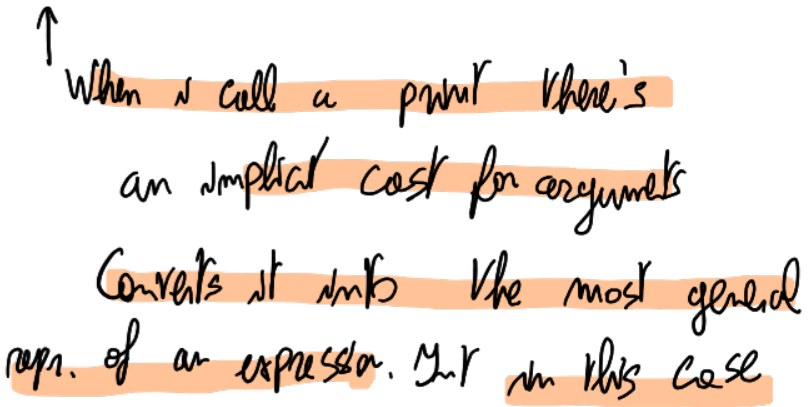
Is it correct, what is the printed value ?

EXAMPLE: JAVA

```
short x = Short.MAX_VALUE;
System.out.println(x+1);
```

Is it correct, what is the printed value ?

```
jshell> short x = Short.MAX_VALUE;
   ...>
x ==> 32767


jshell> System.out.println(x+1);
32768
```



Why?

Java promotes smaller integer types (`byte`, `short`, `char`) to `int` when used in arithmetic operations.

Here's what happens step by step:

1. `x + 1` involves a `short` (`x`) and an `int` (`1`), so `x` is **implicitly promoted** to `int` before performing the addition.
2. The calculation is performed using `int`, so `32767 + 1` results in `32768`, which **fits inside an int**.
3. `System.out.println(int_value);` prints `32768` correctly because `int` supports it.

Yes, Java **automatically promotes** `short` values to `int` before performing arithmetic operations, even if both operands are `short`.

So, for this code:

Java Copy code

```java
short x = 1;
short y = 2;
short z = x + y; // Compilation error
```

You will get a **compilation error**:

Copy code

```
incompatible types: possible lossy conversion from int to short
```

Why does this happen?

1. Both `x` and `y` are `short`, but **Java promotes them to `int`** before doing `x + y` because arithmetic operations on `byte`, `short`, or `char` **always** result in an `int`.

EXAMPLE: JAVA

```
short x = Short.MAX_VALUE;
short z = x+1;
```

Is it correct, what is the value of z ?

EXAMPLE: JAVA

```
short x = Short.MAX_VALUE;
short z = x+1;
```

Is it correct, what is the value of z ?

```
jshell> short z = x+1
|  Error:
|  incompatible types: possible lossy conversion from int to short
|  short z = x+1;
|            ^_^
```

↳ in order to perform this operation you should have an integer there but you have a short

```
PIN_CODE=1234
echo -n "4-digits pin code for autentication: " read -s INPUT_CODE;
echo
if [ "$PIN_CODE" -ne "$INPUT_CODE" ];
        then
                echo "Invalid Pin code"; exit 1
        else
                echo "Authentication OK"; exit 0
 fi
```

```
PIN_CODE=1234
echo -n "4-digits pin code for autentication: " read -s INPUT_CODE;
echo
if [ "$PIN_CODE" -ne "$INPUT_CODE" ];
      then
            echo "Invalid Pin code"; exit 1
      else
            echo "Authentication OK"; exit 0
 fi
```

This script checks whether a PIN code is valid or not for authentication.

If the provided code is 1234, authentication will succeed, whereas any other value is rejected.

BASH

```
PIN_CODE=1234
echo -n "4-digits pin code for autentication: " read -s INPUT_CODE;
echo
if [ "$PIN_CODE" -ne "$INPUT_CODE" ];
      then
            echo "Invalid Pin code"; exit 1
      else
            echo "Authentication OK";
exit 0
 fi
```

This script checks whether a PIN code is valid or not for authentication.

If the provided code is 1234, authentication will succeed, whereas any other value is rejected.

The script relies on a shell command which is expecting numerical values.

Assume to provide a non-numerical value such as blah, this test would return an error code interpreted as being not true by the if statement, resulting in undue authentication.

STRONGLY TYPED AND TYPE SAFE LANGUAGES

Java, C#, OCAML, F#, ... are strongly typed languages

- strong and consistent type annotations programmer provided and/or automatically inferred or checked
- Semantic preserving type-checking algorithm

Well typed programs can never go wrong for the type ops

Does these features ensure safe and secure code?

Security problems may arise as well!!!!!

- beware of *unsafe* constructions of these languages (often used for "performance" or "compatibility" reasons)
- beware of code integration from other languages . . .

(TEMPORARY) CONCLUSIONS

- Some programming language features lead to unsecure code . . .
- How do you choose a programming language ?
  - Usually a mix from performance, efficiency, knowledge, existing code, etc.
- What about security?
  - no “perfect language” yet . . . but some languages are improperly used !
  - Skill and knowledge on programming languages is welcome!!

READINGS

- Aleph one: Smashing The Stack For Fun And Profit
- E. Jaeger, O. Levillain, P. Chifflier, Mind your Language(s), In Proc. Security & Privacy 2012
- X. Wang, H. Chen, A. Cheung, Z. Jia, M. Frans Kaashoek Undefined Behavior: What Happened to My Code?, APSys 2012
- From John Regehr's Embedded in Academia blog: A Guide to Undefined Behavior in C and C++ (https://blog.regehr.org/archives/213)