1 Memory errors and memory safety in C, Java and Rust

Paul C. van Oorschot **version: 5 May 2023**

Abstract *These notes aim to provide an accessible, self-contained explanation of* ***memory errors*** *in software programs, and the related concept of* ***memory safety*** *in programming languages, along with brief discussion of* ***type safety****. The context is software security. We focus on C, Java, and Rust.*[1]

For major operating system and browser vendors who make heavy use of systems languages, 65–70% or more of reported software vulnerabilities in recent years have involved *memory errors* [18, 34]. For Microsoft, the share has remained near 70% for a dozen years, per a 2019 report [32]. In a broad study counting the number of exploits in the US National Vulnerability Database over 2013-2017, the top category was (memory) *buffer errors*, among 19 vulnerability categories [11]. Though these statistics lag by a few years, decades-old problems clearly remain with us today.

In these notes we aim to provide background and context, for students and security novices, to provide an understanding of memory safety and type safety. Definitions for these terms remain hard to find in older computer security textbooks, while discussion meaningful to practitioners is often absent from the research literature. To that end, we ground our discussion using C for simple examples of security vulnerabilities involving memory errors related to language design. While we note Java for some contrasting design choices, we give greater focus to the Rust programming language, explaining how a number of its major design features directly address memory errors.

In teaching software security, we believe much can be gained by giving greater focus to the impact of programming language design choices and features, over the historical focus of (mitigating) malware exploits on target executables. This may help steer development teams to better programming language choices for long-term projects. A language-centered discussion of software security and memory errors may also serve students well by motivating them to learn more about comparative aspects of programming languages and their designs.

1.1 C language features and security pitfalls

In contrast to high-level programming languages such as Java and modern scripting languages like Python, low-level systems languages like C prioritize efficiency and programmer access to memory addresses over built-in security protections. Programming of device drivers and hardware interfaces is supported by program access to explicit memory addresses (raw pointers) and easy drop-down into assembly language or machine code.

As is well known, this comes with a downside. C and closely related C++ have historically been accomplices in a lion's share of security vulnerabilities. To understand the relationship between programming language design and software security, we review language features that have historically been associated with programming errors underlying software vulnerabilities. To this end, and to set the background for understanding *memory errors*, we first review basic C language features and syntax that are highly relevant to security; C experts can skip over this. We later discuss how the design of recent programming languages aims to avoid such security problems.

[1]A shortened version of these notes appear across a two-part article in the March-April and May-June 2023 issues of *IEEE Security & Privacy* magazine, titled "Memory errors and memory safety".

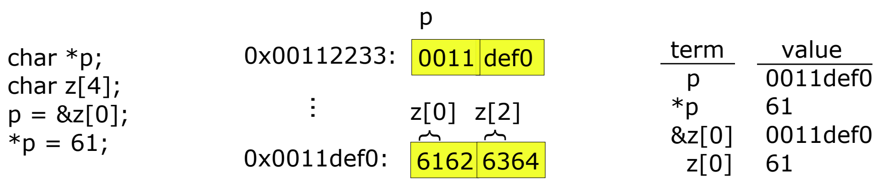

Figure 1: C pointers, and the operators `*` and `&`.

C pointers and casting. First recall some basic C syntax. If `p` is a variable denoting a value that is a pointer, then the value of `*p` is the value pointed to (i.e., the value stored in the memory location indicated by the pointer). Here `*` is the *dereference* operator. Its opposite is `&`, the *address* operator; `&z` denotes the address at which the value denoted by variable `z` is stored.

C values of type *pointer* are **raw pointers** (often 4 or 8 bytes, based on machine architecture), giving programming language access to memory addresses. The addressed memory may be any data, e.g., integers, floating point numbers, bytes of a string, or executable code in a function.

Like most languages, C uses *data types*, i.e., is a *typed* language. Pointer declarations define the *type* of data item that is pointed to. For example:

```
int *intptr;  // an equivalent declaration is:  int* intptr
```

defines a "pointer to int", or read as: `*intptr` is an integer. Synonyms can be created for existing data type names. For example:

```
typedef char *String;
```

makes the identifier `String` a synonym for `char *`, i.e., pointer to char.

In principle, *"every pointer points to a specific data type"* [27, p.94]. But there are exceptions: `void *` ("pointer to void") is *"used to hold any type of pointer but cannot itself be dereferenced"*. The intent here is that such a pointer will later be manually `cast` to (any specific) desired type.

Such *type-casting* in C allows programmers to convert one data type to another, including from integers to pointers. The **cast** operation is indicated by putting the desired type name in parenthesis before an expression. For example, integer 3 is converted to a floating point double in C by writing:

```
    (double) 3
```

C supports **pointer arithmetic**, e.g., you may add or subtract an integer from a pointer, in which case the integer is implicitly scaled by the size (in bytes) of the data type of the object pointed to.

C primitive types and expressions. Aside from *pointers* (above), basic data types in C include the following [27].

Integer (five widths): char (8 bits), short int (16), int (16 or 32), long int (32), long long (64).

Floating point (reals): float, double, long double (platform-dependent, e.g., 32, 64, 80-128 bits). Integers and float types can be given clarifying modifiers: `signed` or `unsigned`, `short` or `long`.

Enum (enumerated types): these associate integers with names. Example:

```
    enum colors {RED = 2, BLUE, GREEN}   /* BLUE = 3, GREEN = 4 */
```

Boolean: true/false. Traditionally, 0 is interpreted in C as `false`, and non-zero as `true`. This was formalized by the addition of boolean as a primitive data type in the C99 revision (1999).

Struct: a programmer-defined object (record) grouping multiple (typically different) data types. For a struct (or union) with name `t` and pointer `tptr = &t`, a field `fld` (or union member `memb`) is accessed using as syntax either a dot or arrow operator: `t.fld` or `(*tptr).fld` or `tptr->fld` (thus dot and dash-arrowhead are in effect dereference operators for a struct field or union member).

Union: this type allows one memory area to be used by two or more different data types at different times. Space is allocated to accommodate the largest variant. Readout from memory does not change the representation of the data (proper interpretation is the developer's responsibility).

Character strings are not supported as a basic C type, but by convention a string is an array of `char`, terminated by a NUL char `0x00`, and referenced using a pointer (address of string's first byte). Standard C library (libc) functions provide string utilities based on this convention—however, they are a notorious source of programming errors and security vulnerabilities, as discussed shortly.

Array: a contiguous sequence of *n* data values of one type. The memory address referred to by `b[i]` is `b+(i)` evaluated in pointer arithmetic. Here `b` serves as a base address, `i` is an index, and `(i)` signals that the offset implied by the index is scaled up by the element size of the referred object (as defined by its data type). In some contexts, an array's name `b` implies its address `&b`, which is also the address of its first element `b[0]`. An array can hold values of any type, including an array:

```
int myarray[4][10]  //4 items, each item being an array of 10 int
```

Scaling within array indexing is an easily-forgotten detail when done manually; this leads to programming errors, especially combined with *casting*. Critically, C does no bounds checking (on reads or writes) to ensure array accesses are within the memory allocated for the data structure; this is the programmer's responsibility. Out-of-bounds accesses result in "undefined behavior".

Evaluation of C expressions. To evaluate expressions per C's language specification, compilers may generate code that does implicit **coercion** (discussed also page 7), i.e., automated type conversions not requested by the programmer. This is part of type-checking system designs. For example, to ease programming of arithmetic expressions combining integer types of different width, some shorter integer data types are converted to a wider type compatible with longer integers; this is called *integer promotion*. C's specification calls for coercion not only in the case of arithmetic expressions that mix integer types of various widths and signedness with floats, but also in assignment statements, function return values, and function arguments whose types do not match declared formal arguments types; a syntax error is declared if no compatible conversion exists.

C objects and storage classes. In C, an **object** is "a named region of storage" [27, p.197]. Program variable names (including for pointers) are used in the usual way to access values stored in memory. In principle, a variable's type "determines the meaning of values found in the identified object" [27, p.195]. A variable's **storage class** determines its *lifetime*, i.e., the validity period of the association between the variable and the memory assigned to store its value. C variables may be:

- **static** (e.g., variables private to a function or file, or globals imported to a program's namespace by keyword `extern`). These retain value across block and function exit and re-entry.
- **automatic** (including `register`). These are local to a block and discarded on block exit.

1.1.1 C string issues and buffer overflows

Compounding C's lack of bounds-checking on array accesses, many utility functions in the standard C library (`libc`) fail to carry out error-checks that might be expected. This is left as a responsibility of human programmers, who of course make mistakes. Instances of these mistakes produce both "normal" errors (incorrect results, run-time exceptions, program crashes) and exploitable errors (*software vulnerabilities*). For example, one type of **buffer overflow** vulnerability enables overwriting a run-time stack return address, which can then alter execution control flow.[2]

Our point is not to repeat from your introductory systems programming course that C programs are tricky and error-prone, but to emphasize that programming errors can have severe consequences, including surprisingly often, enabling remote parties to run unauthorized code (*malware*) that can take control of host machines. You might not think this concerns you directly, but your view may change when somehow money is removed from your online bank account and your bank insists that it was your fault for not properly protecting your account password, or a *ransomware* attack results in all content of your laptop being rendered inaccessible, or malicious software results in disruption of the power supply at the hospital while your mother is on the operating room table.

One subset of libc utilities is notoriously problematic: functions supporting string operations. Strings are not a built-in (primitive) data type. Instead, C uses the efficient but error-prone convention that a string is a sequence of `char` (bytes) terminated by the zero byte `'\0'`, which we denote `NUL`. (Thus we say pointers may be `NULL`, strings are `NUL`-terminated; `NUL` and `NULL` differ in size.)

`strcpy(char *dst, char *src)`, for example, is a string-copy utility. It copies whatever string is found at the `src` address to the address indicated by `dst` in the usual C way: byte-by-byte, ending once a `NUL` byte is found and copied. If fewer bytes were allocated for the destination buffer than present in this source string, that does not matter—the copying blindly continues until a `NUL` byte is found. This overwrites (corrupts) memory beyond the end of the `dst` buffer, continuing into the adjacent data structure or executable code at the next higher memory address, and the one after that and so on. Eventually a `NUL` byte is copied (a random byte has value `0x00` with chances 1 in 2^8). Not all such instances are exploitable; by this we mean, it might "only" trigger an access violation or segmentation fault (e.g., if a corrupted byte is later retrieved for use as part of an address for a memory access, and the corrupted address is not in the address space of the current process), or "only" an OS crash (e.g., if critical kernel data structures were corrupted). Security-wise, these are typically considered less serious outcomes than (true) "exploits"!

While such an overrun can clearly arise if the `dst` buffer is shorter than the `src` buffer, it can also occur even when the `dst` data structure (number of bytes allocated) is the same size or larger than that of `src`, in the case that the `src` buffer itself contains no ending `NUL` byte. Distinct from outcomes mentioned above, this can result in a problem at the `src` end: reading of non-`NUL` bytes beyond the end of the `src` buffer may result in an *information leak* exposing potentially sensitive data (e.g., secret keys) from memory beyond the intended `src` buffer.

`gets()` and `puts()`, the get-string and put-string functions, raise similar issues. They respecitvely read a string from standard input into a buffer, and write a string from a buffer to standard output. Another example is the string concatenation function: `strcat(char *dst, char *t)`. It concatenates the string denoted by `t` to the end of that denoted by `dst`, and "*assumes* [without checking] *there is enough space in* `dst` *to hold the combination*" [27, p.47].

`strncpy()` is a libc utility that a developer seeking a safe alternative to `strcpy()` might discover. It takes a third parameter *n*, specifying a maximum number of bytes to copy. Copying stops after either the first `NUL` byte or *n* bytes, whichever comes first. This enables a different error: the resulting destination string may end up not being terminated by a `NUL` char. If the programmer doesn't

[2]While widely available and easily understood, it is not our goal to explain the details of how buffer overflows are exploited to allow execution of malicious software. Among many other sources, see for example [36, §6.3-§6.5], available at: `https://www.scs.carleton.ca/~paulv/toolsjewels/TJrev1/ch6-rev1.pdf`.

check for this, then because later code will expect that this sequence of bytes is a conventional `NUL`-terminated string, a later memory error will almost surely occur.

`strncat()` is another utility with a third parameter n for length, here a concatenation alternative. Sadly, it also induces errors. In this case, n specifies the maximum number of non-`NUL` characters concatenated to the end of the `dst` string, but a terminating `NUL` byte is always inserted—extending the length of the `dst` string by possibly $n+1$ bytes. This semantic inconsistency versus how n is used in `strncpy()` induces so-called **off-by-one** memory errors, corrupting single bytes.

`strlen(s)` sets another off-by-one trap: this utility returns the number of bytes in its string argument, excluding the terminating `NUL` byte—so one byte more than the returned count is needed to store such a string. Thus for example, the following code may, depending on the sizes of the objects in play, be off-by-one:

```
        if (strlen(src) <= sizeof dst) strcpy(dst, src);
```

The problem is that while the compile-time unary operator `sizeof object` yields the storage size in bytes (as does `sizeof(typename)`), and `strlen` counts the number of non-`NUL` bytes, `strcpy` will copy also the terminating `NUL` byte from the `src` string. That's one byte more.

Documentation for such library utilities does give warnings, e.g., that for `strncat` the resulting behavior is "undefined" if the `dst` character array has insufficient space to store the combination of the original string plus the first n chars of the string being appended plus the ending `NUL` char; and also if the pointer to the original `dst` string is not itself originally `NUL`-terminated; and also if the string arguments overlap. But a warning in documentation does not stop errors from occurring.

What we need is library utilities that are resilient to human errors. Instead, C programmers are expected to be familiar with all these nuances, and to avoid all such string-related traps. The result: C programmers continue to fall into traps, and security vulnerabilities creep into code, silently awaiting exploitation at unknown later times. Meanwhile, various `libc` utilities are declared "unsafe" by warnings in many tools, but often remain available for backwards compatibility.

Homework. When C was first introduced, programmers had a sense of how code would map to machine instructions. This is no longer generally true, as compilers now carry out a wide variety of optimizations on specific platforms. To understand this, read Chisnall [10].

1.2 Manual memory management vs. garbage collection

The next piece of the memory error puzzle to discuss is **garbage collection**.

When a process is created, the OS assigns to it a *memory space* (set of virtual addresses) distinct from kernel memory and other processes. In C environments, this is often split into five regions: run-time call stack (for data related to function calls, e.g., arguments, return addresses, local variables), heap (for dynamically allocated data structures), text segment (program code), data segment (initialized global data), and the BSS area (uninitialized global data, sometimes called the block storage segment). Memory errors most frequently involve the run-time stack and heap.[3]

In many systems languages, heap memory is *manually allocated* by programmatically requesting blocks of the desired size (*chunks*). In C, calls to `malloc()` return a pointer `p` to a chunk of uninitialized heap memory, or a `NULL` pointer if the request cannot be filled; the related `calloc()`

[3]See Fig. 6.3 in [36, p.166] at `https://www.scs.carleton.ca/~paulv/toolsjewels/TJrev1/ch6-rev1.pdf`. Or for a memory layout with a few more details, see [4] at: `https://tiemoko.com/blog/blue-team-rust/`

returns a zero-initialized chunk. Memory no longer needed is released by calling `free(p)`; if not, the memory is effectively lost (called a *memory leak*), inducing out-of-memory errors over time.

Such manual memory management is avoided in most higher-level languages (e.g., Java) through a process called *garbage collection* (GC), whereby memory chunks no longer used or reachable (inaccessible) are automatically reclaimed and returned to a system **free pool** of available memory chunks. Any of various *storage allocator* approaches are used to manage this heap memory, often arranging the *free pool* in a double-linked list ordered by size, and occasionally reorganized.[4]

As advantages of GC, beyond unburdening programmers from the responsibility of allocating and deallocating memory, it eliminates *temporal memory errors* and can significantly reduce *memory leaks* (these terms are discussed shortly, in §1.4).

However, GC disadvantages include significant processor time costs, and unpredictable process delays when the GC routines engage to actually collect and consolidate (*compact*) free memory. The performance impact can be reduced by provisioning larger pools[5] of available memory, thereby trading off memory for processing time. Basic GC approaches are also unsuitable in time-critical applications required to meet strict timeliness constraints.

Exercise. Suppose a C pointer variable `p` is declared as a global variable in the `main()` entry point, and later a function `f()` assigns to `p` the address of a local (stack) variable of `f()`. Explain why this may result in unpredictable program behavior after the function returns.

1.3 Data types and type checking

As we build towards defining *memory errors* and *memory safety*, and their relationship to *type safety*, it helps to recall the main ideas of data types, type systems, and type checking.

Why do programming languages need **data types** and **typing**? We naturally associate data types with objects; functions and operators (e.g., unary and binary arithmetic and logical operators) also require inputs of expected types. Functions, expressions, and assignments of values to objects all involve values from specific domains with expected structures, and similarly produce outputs with expected characteristics; *data types* specify and convey these expectations.[6] Every programming language has a **type system** that defines built-in types, and dictates rules governing what is required and enforced related to these types; most also allow creation of user-defined types.

How to define *data type* in greater detail is harder. As important aspects, types convey details about an object's properties, structure, and semantics,[7] and help in identifying mismatches between operations and the objects that they expect as operands. Data types convey the operations or methods allowed on data values, and information on how values are represented or formatted in storage (this may be hidden from programmers in an attempt to simplify programming or hide platform-specific details). Type mismatches that may have security implications are of special interest to us.

Based on a type system's types and rules, **type checking** is the process of detecting *domain incompatibilities*—improper or unexpected uses of objects—at compile time (**static** type checking),

[4]Kernighan & Ritchie [27, §8.7] give an example implementation of a simple heap storage allocator, with diagram.

[5]Hertz and Berger [22] explored the effects of using 2–5 times as much memory.

[6]While the concepts discussed here generalize, we omit discussion of *overloaded* operators and *polymorphic* functions (also called *generic* functions) designed to work properly for inputs whose types may vary across a pre-defined range.

[7]Recall that *syntax* relates to form, while *semantics* relates to meaning or use.

run time (**dynamic** type checking), or both. Not all incompatibilities can be detected at compile time, as not all values are known. Out-of-bounds array accesses when element indexes are dynamic inputs, and division by zero, are examples of errors that are not caught at compile time, while examples that often can be are multiplication and exponentiation of string values and booleans.

Requiring a programmer to specify the data type of every single object and value is tedious, and often redundant. Thus to varying degrees, programming languages relax the requirement of explicit **type declarations**, some requiring few if any explicit declarations. The language processor then uses **type inference** to assign a data type (hopefully as intended by the programmer) based on context and the rules of the language. Of course, this might result in a mismatch with programmer's expectations, and a syntactically correct program having semantic errors neither flagged by the compiler nor caught by the run-time system. Thus there is a language design tradeoff: strictly requiring type declarations may be viewed as burdening programmers, while being lax risks undetected mismatches between programmer intent and a language processor's actions.

Coercion (implicit type conversion). When the operands of an operation (including assignment) or arguments of a function are incompatible by strict rules, a programming language could simply declare a compilation error or run-time exception. However, another option is to (automatically) generate code that converts one or more values to a value of a compatible type, in common cases where this makes sense, e.g., addition of integers and floating point numbers. This is called **coercion** (mentioned already in the context of evaluating C expressions, page 3).

Consider this "simple" arithmetic expression mixing an integer and floating point: `(2 + 3.0)`. How should it be evaluated? One could convert the integer to a float, and then do a floating point add. (Keep in mind: hardware representations differ by data type, e.g., the integer might be in 16-bit two's complement, and the float a 64-bit double per IEEE 754.) A different option is to generate code to convert the float to an integer. Note that these two options produce different results if the float is not `3.0` but `3.4` and conversion to an integer implies truncation. Here, a natural choice is to convert the integer "upward" to a float, using pre-defined *conversion hierarchies*.

Should it be allowed to add integers to strings? Here *add* could mean concatenation after converting the integer to a string—or arithmetic addition after converting the string to the integer its string characters may represent. For example, JavaScript interprets `"1"+2` to mean concatenation after implicit type conversion. Different design choices occur in different languages. (If a rule is not known, a compiler's output for a given example will show what error message or code is generated.)

We now make a few observations about type checking and coercion.

- Data typing and type checking are more complicated than they first appear, even for integers and floats—as we have noted, C has numerous widths and types for integers alone.
- Languages that aggressively undertake implicit type conversions may lead to exploitable programming errors (including integer-based vulnerabilities, page 14).
- Static type checking may result in compile-time errors, or in generating code for run-time coercion or run-time type checking. If a compile-time check can rule out a type incompatibility, then this single check both avoids a run-time test at each instance the relevant code line runs, and catches errors during development versus during dynamic testing or post-deployment.

Static type declarations themselves also often help developers understand programs, e.g., knowing whether a variable is used as a `boolean`, or an array of `int`.

Exercise. Summarize C's coercion rules for *integral types*, i.e., including `char`, boolean and enumerated types, but excluding floating point. (Hint: [27, §A.6] for C89, [45, Ch.4] for C99.)

1.3.1 Case study: C is weakly-typed

C is *statically typed* (variables are declared before use) and has *static type-checking* (expressions and function parameters are type-checked at compile time). Related to this, C supports or allows:

- a wide range of coercions (automatic type conversions);
- raw pointers, pointer arithmetic, and converting pointers to one type to pointers to another;
- various code sequences noted by the specification to result in *undefined behavior* (compiler-related tools may give warnings, but doing almost anything is compliant);
- no bounds-checking on memory accesses via pointers, although compiler-related tools may offer warnings (array accesses amount to dereferencing offsets from pointers);
- bypassing the type system by the `union` construct, and manual type conversion, including casting between pointer types and integers, and from a function pointer to another whose function signature differs (e.g., in return value type, argument types, or number of arguments).

We highlight also two statements from Kernighan and Ritchie's 1988 description of ANSI C.
A: Every non-void pointer points to an object of some type.
B: The programmer is responsible for tracking which type is currently stored in a union.
A is not guaranteed, if the intent is a *valid* object. B signals that C abdicates responsibility for checking union variant types. C similarly abdicates on enforcing the data type of a pointer's *referent* (types can be changed by pointer casting, pointer arithmetic, and array indexing). So, while C is a *typed* language, and requires that variables have type declarations before use, it is lax in ensuring object uses are compatible with their type, or memory referenced by a pointer matches its type.

Based on such observations, C is said to be **weakly-typed**. Its design and language processors do not reliably guarantee domain compatibility in the use of objects, and no run-time type-related support addresses compile-time deficiencies. We say a bit more on being *weakly-typed* in §1.5.

1.4 Memory errors (categories and outcomes), and memory safety

Here we consider several types of security problems enabled by language features, with focus on C.

A wide variety of errors—called *memory errors*—involve improper memory access (read or write), including due to dereferencing invalid pointers and failure to check that array accesses are within bounds [48], [50]. Some result in the CPU (hardware) raising errors conditions, which may be available for processes to *catch* (via exception handling code); exceptions such as access violations or segmentation faults may terminate a process, or *crash* the operating system itself. For exploitable code, a common adversary goal is to carry out a task (execute unauthorized code) before or without an operating system crash, perhaps without terminating the exploited process itself.

Some exploitable errors combine several language features, as in the following exercise.

Exercise. A *jump table*, or array of function pointers indexed by an integer, is often used to organize exception handlers. The C snippet below gives an example. Discuss how malicious program input might manipulate such an indexed function pointer table.

```
functiontable[4] = {fna, fnb, fnc, fnd}; // array of function ptrs
i = getexternalinput();
functiontable[i](); // calling a function through a jump-table
```

Returning to memory errors, consider first a few possible outcomes of *writing* to an unintended memory address, e.g., beyond the bounds of a referenced array, or an unexpected result from pointer arithmetic. Perhaps the value being written is controlled by a malicious input to the program.

a) The value overwritten may be a *code pointer*, e.g., a stack return address or function pointer. This will alter later execution paths when the code pointer is loaded into the CPU instruction pointer (IP register). As a severe case, the IP may then point to attacker-specified code. This is said to break the program's *control-flow integrity*.

b) The memory overwritten may hold program data excluding code pointers (above case). This includes data variables and pointers to them (*data pointers*). This breaks program *data integrity*. It may also indirectly alter execution paths that depend on the altered values (for example, consider a sensitive boolean value corresponding to a variable `isRoot` or `isTrusted`).

c) The memory overwritten will later be executed as code itself. This directly breaks program *code integrity*. While the details are beyond our scope here, this is a common tactic using so-called *shellcode*, where an attacker crafts special code as malicious program input, allowing, as a severe case, execution of such special code of their choosing on your machine.

Consider next two outcomes of errors associated with *read* access, which may lead to **information leaks** (disclosure of sensitive data).

d) Data is read from an *uninitialized object*, part of whose associated memory retains values left (uncleared) from when the same memory was used for an earlier object, possibly exposing that data externally. (C does not require initialization of automatic variables; static vars are set to 0 or NULL. Reading from an **uninitialized variable** is said to result in *undefined behavior*.)

e) Data is read from perhaps any address within the process' address space, the address being unintended or unanticipated by a benign programmer.

   Example. A severe case is the 2014 *Heartbleed* incident [9], where a function in the OpenSSL cryptographic library failed to check array bounds. This allowed memory blocks of 64 Kbytes at a time to be returned from vulnerable TLS servers to a malicious TLS client. This so-called **buffer overread** error was in a routine supporting an extension of the TLS protocol (used in HTTPS connections). In this way, the memory contents of at least hundreds of thousands of web servers were subject to exposure, at a rate of 64 Kbytes per attacker HTTPS connection; such server memory typically contains sensitive information including user passwords, TLS long-term private keys, and depending on the server, users' personal identification information, and perhaps banking or credit card information, medical records or tax records.

Distinct from outcomes (above), memory errors can be grouped into categories—we now consider three. The first is **spatial safety errors** (*spatial* refers to a memory address or location), including:

1(i) memory access (read or write) that involves a pointer to one object, but results in access to **memory outside of the range** allocated for that object. In the case of a write access, this corrupts a separate object. For example, the first object may be an array whose elements are accessed using a base pointer and offset (index); the error equates to a failure to do bounds-checking. As noted (page 4), in the well known case of a **buffer overflow**, bytes are written continuously past the end of an array or memory buffer, spilling into adjacent object(s) at higher memory addresses. In general, the erroneous access may be to as little as one byte, or to a lower address (if the offset to a base pointer is negative).

1(ii) dereferencing a *wild pointer*. We define a **wild pointer** as any pointer whose use would result in undefined behavior per the language specification (e.g., in C, uninitialized and NULL pointers, among others). NULL pointer dereferencing is known to be exploitable in some cases (vs. simply causing an access violation). NULL, often represented `0x0..0`, might map to kernel memory through the virtual address translation implemented by the hardware platform.

The second category is **temporal safety errors** (*temporal* refers to time). These include two cases of using a *dangling pointer* (a pointer to an object in memory that was already deallocated or freed):

2(i) **use-after-free** error. This error involves dereferencing a dangling pointer. In this case, the referent is now considered to be an *invalid object*, and using any reference to it is an error.
Example 1. The object referred to is in heap memory.
Example 2. Memory for a local variable is allocated on the stack call frame for a function invocation, and the memory address of that variable is referred to after the function has returned (i.e., after the memory allocated on the stack frame is automatically deallocated).

2(ii) **double-free** error. This involves freeing an already-freed object (chunk), i.e., a second time. Thus the deallocation function is passed a dangling pointer. As discussed (page 6), deallocation puts a memory chunk back into the *free pool*; doing so twice may corrupt the storage allocator's internal data structures (if a chunk in the free pool is freed a second time), or may create a separate dangling pointer (if the chunk was already re-allocated to a new object).

A third category of memory errors (also resulting in *undefined behavior* in C) involves:

3) reading from **uninitialized variables** (discussed under information leaks, page 9). Here we exclude dereferencing wild pointers, viewing that instead as a spatial error—but we include the case of an uninitialized pointer whose value is read (e.g., to assign to a second variable) but not dereferenced. (If and when the second variable is later dereferenced while holding a wild pointer, we then as noted, view that as a spatial error.)

Note that aside from the third category, memory errors involve dereferencing or using an *invalid pointer* (e.g., a dangling or wild pointer). This observation helps clarify how the lack of constraints on C pointers contributes to memory errors.

1.4.1 Memory safety (levels L1 to L4)

We can now define *memory safety*, or rather, four levels useful to differentiate classes of memory errors that a language's design and core support tools (compilers, run-times) may address.

L1: **fundamental memory safety**. Level 1 aims to eliminate *spatial safety errors* and *temporal safety errors*. This addresses the first two memory error categories, and the most serious security issues related to pointers. This might be supported by maintaining and checking (low, high) memory bounds for each object before access, and likewise for an object *validity* flag (i.e., whether its memory remains allocated *and* the object remains in scope).

L2: **clean memory safety**. Level 2 aims to eliminate both information leaks and undefined behavior related to uninitialized variables.

Further levels are associated with mitigating two other common classes of memory errors:

L3: **memory leaks**. Level 3 aims to eliminate memory leaks. These can crash programs or the OS, and can be viewed as security-related in that they may enable denial-of-service attacks.

L4: **data races**. Level 4 aims to eliminate security issues and unpredictable outcomes due to *data races*, which can arise in the case of concurrent reads and writes to shared memory, e.g., if one execution thread changes a data value while another is using it. This and related concurrency issues fall in the broader context of *thread safety*.[8]

Other categories of memory-related errors exist. For example, two that overlap L1 and L2 involve exploitable *format strings* (e.g., user-defined or user-controllable formats in C's `printf` family of formatted-output functions); and *variadic functions*, taking a variable number of arguments [48]. Another level beyond L1/L2 might avoid cleartext secret keys or passwords in memory (some systems offer support to store these encrypted while in memory, aside from at instants of actual use).

We can use memory safety levels L1–L4 to assess different languages, as in §1.6 and §1.7. From our discussion, it is easy to see that C has deficiencies in all four. To be more precise, we should distinguish between C's specification and implementations. For example, the C spec declares various instances of memory errors to result in "undefined behavior", without taking responsibility for ensuring that all memory accesses are proper (for this reason, C may be called *memory-unsafe*); the degree to which these memory errors result in security vulnerabilities may vary greatly across compilers and runtime environments. Related to this, memory errors may be viewed as violations of the expectations set by a specification about how programs should interact with memory [43].

Homework. Read the CyBOK chapter on Software Security [43]. Summarize its discussion of memory errors, and compare to the discussion herein.

[8]For more on *concurrency* issues in Java and related *non-atomic check-and-use races*, see the *Homework* on page 17. Features to help address some concurrency issues in C were added in the 2011 revision (C11).

1.5 Type safety and type confusion

We now discuss *type confusion* and *type safety* (type-safe languages).

Type confusion occurs during program execution when the program has a reference of one type to an object in memory with a different type. This could be due to an implementation error in a tool, or a semantic error (confused programmer). Type confusion often leads to unexpected results.

Recall that we said that C is **weakly-typed** (page 8). This term commonly appears in two contexts. (1) For a *typed* language such as C, *weakly-typed* may suggest that the language is not fully *type-checked* to rule out *undefined behavior* [8]. (2) For an *untyped* language such as assembly languages and interpreted or script languages (e.g., JavaScript), *weakly-typed* may refer to the language being "loose" with types, e.g., objects of one inferred type being eagerly coerced to or combined with objects of different types (recall page 7). Note that despite being weakly-typed in the second sense, JavaScript is nonetheless **memory-safe** in the L1 sense.

Weakly-typed languages in both senses can lead to security problems. Security-wise, we prefer a language whose design, processors, and run-time support strive to ensure that variables and expressions have types consistent with operations they are used in, and generally prevent memory assigned to values of one type being used as another type, except through explicit conversions.

A type system's design should also use distinct types for objects with distinct semantic properties. Thus pointers to data (*data pointers*) should have different types than function pointers, and integers should be distinct from pointers—e.g., because a data pointer plays an addressing role (locating and accessing data), whereas regular integers are used in arithmetic expressions. As noted earlier, data types help ensure that runtime operations involve operands of the expected structure.

In *weakly-typed languages*, instances of type confusion are typically unintended (and if syntactically allowed, may go undetected). In contrast, when a programmer intentionally uses the same region of memory at different times for objects of different types, this is called **type punning**. (A *pun* is a play on words, where a word has two meanings.) For example, C `union` types (untagged unions) enable type punning; union types effectively bypass C's type system (cf. page 8). As the ISO C standard states (ISO/IEC 9899:2018, p.59):[9]

> *If the member used to read the contents of a union object is not the same as the member last used to store a value in the object, the appropriate part of the object representation of the value is reinterpreted as an object representation in the new type.*

This may result in misinterpreting the value, or reading out more bytes from memory than stored into a shorter union member.

Next, we discuss the term **type-safe language**, sometimes shortened to *safe language*. Characterizations of this term by experts include the following [8, 42].[10]

- A **safe language** *guarantees the integrity of the abstractions that it provides.* This sets the expectation that abstractions defined through the language are enforced at run time (e.g., boundaries of objects, allowed methods to access array elements).

[9] Kernighan [27, pp.147–148] notes C unions are similar to Pascal *variant records*. Cf. COMP2401 notes, p90.ch2.

[10] The term *strongly-typed* is used interchangeably with *type-safe* by some but not all experts; we thus avoid it here.

- *A safe language has a specification that fully defines program behavior, ruling out undefined behavior.* The language takes responsibility for ensuring that programs behave predictably (vs. making programmers responsible).
- *A safe language precludes untrapped errors at run time.* The language promises that any error not detected statically will be trapped immediately upon occurrence at run time, and that no undetected run-time errors can occur (e.g., which might cause unexpected behavior later).
- Informally, type safety promises that: *runtime operations don't silently go wrong.*

While the above are characteristics only, precise technical definitions of *type safety* are given in the literature on type systems (e.g., [42]), and typically model or support memory safety properties desired in practice. Type safety in running systems often relies on both compile-time and run-time checks to find errors specifically targeted by the language and the goals of its type system.

Finally, a *type-safe language* is harder to design than a **type-safe program**, as a single program may simply avoid programming pitfalls and problematic features in an unsafe language.

Casting vs. coercion. In C, a *cast* is a directive to change the compile-time type (type tag) that a compiler associates with an object; this generates no code, nor changes value representations. In contrast, *coercions* typically result in code generation to convert value representations. (Consider what is needed to coerce a 16-bit integer to 32-bit integer, or to double float.) To use analogy, the `cast` operation might be viewed as having a barn and then one day starting to call it a house, and later a hotel, but without ever doing any renovations; type coercion involves internal renovations. In this sense, a `cast` is a superficial labelling change, while coercion changes representation and preferably stores the result (often of a different size) in a different location. Explicit function calls that create new objects from old are distinct from superficially changing an object type.

Example. C is not type-safe for many reasons: unions, lack of bounds-checking, a `cast` operation allowing direct conversion of integers to pointers and across pointers to different types of objects.

Example [48]. Vulnerability due to overflow that can be abused to overwrite security-related data.

```
struct S {char name[15]; bool isRoot; };
struct S svar; char buf[16];
memcpy(svar.name, buf, sizeof(buf));  // form:  memcpy(dst, src, nbytes)
```

Example [48]. Information leak via part of an object remaining uninitialized.

```
struct S {int data[2]; };
struct S *sptr = (struct S *)malloc(sizeof(struct S));
sptr->data[0] = 0; // part of the struct remains uninitialized
send_to_hostx(sptr, sizeof(struct S)); // hostx may be external/untrusted
```

Example. Languages with type safety properties: `https://en.wikipedia.org/wiki/Type_safety`

Exercise. Explain how each of the three categories of memory error violates common typing goals.

Exercise. Consider the statement: coercion is more closely related to type checking than to type safety. Do you agree? Discuss and explain.

Exercise. (a) Explain C++ `static_cast` and `dynamic_cast`. (b) Discuss C++ `reinterpret_cast`.

1.6 Relating language choice and software security

Garbage collection is used by most languages that offer memory safety guarantees [42], as a common means to avoid errors related to manual memory management, including memory leaks, spatial errors and temporal errors. As should now be clear, the language used to write a program has a huge impact on the resulting software security—as some languages eliminate entire categories of errors.

However, language choice alone is not a full solution to avoiding memory errors related to systems languages like C++ or C. These are often also called compiled or **native languages**, in contrast to high-level languages that run in *managed environments* (e.g., virtual machines, interpreters) [49]. The challenge is that "memory-safe" high-level languages may themselves rely on supporting components or run-time libraries written in native languages—e.g., to interface to hardware, or for performance or historical reasons (legacy code, backwards compatibility). On the positive side: some categories of simple memory vulnerabilities are becoming harder to exploit in newer products, due to not only development tools that help avoid them, but also effective run-time defenses.[11]

Languages themselves also cannot eliminate all security problems related to software. As one example, definitions of memory safety and type safety are silent on language-related categories of errors such as *integer overflows* and *underflows* (next paragraph). As another, neither memory safety nor type safety address security issues related to lack of *input validation*—e.g., preventing a `char` string `<script>` tag, embedded in user input to a web discussion forum, from being executed as JavaScript.[12] Language-based software security is also orthogonal to *social engineering* ("download and install this cool software [please ignore the malware hidden within it]").

Integer overflows. *Integer overflows* and *underflows* are known to indirectly enable a variety of exploits (next paragraph). In C, for a 16-bit `unsigned int x` holding value $2^{16}-1 = 65535$, `x+1` yields 0. We commonly call this an integer overflow, but C officially states that integers declared `unsigned` "*obey the laws of arithmetic modulo* 2^n *where n is the number of bits in the representation, and thus arithmetic on unsigned quantities can never overflow*" (i.e., no error is flagged, and the mod 2^n result is considered correct). On the other hand, for a `signed char` (8 bits) with value 127, adding 1 yields -128 on most implementations (oh my!), but officially the behavior is undefined. Many C programs do indeed rely on the standardized mod 2^n behavior for unsigned integers.

Integer overflow/underflow, and consequences of coercions (e.g., integer width conversions that result in truncations and extensions), often lead to a broader category of errors called **integer-based vulnerabilities**. These include a range of issues that while not themselves memory errors, can indirectly contribute to them via integer-based errors that impact, e.g., branching conditions, loop counters, or array access indexes. Surprises often result from mixing signed and unsigned integers in C—thus the general advice: avoid mixing signed and unsigned data types.

Example. Comparing signed and unsigned `int` values results in **promotion** (a coercion, page 3) of the signed `int` to an unsigned of same width; this converts negative values to positive, often unexpectedly. Compilers may or may not warn about this. For concreteness, consider the C code:

[11]As of 2019, Microsoft reported that stack buffer overflows had almost disappeared among their vulnerabilities [32].

[12]This type of *input validation* may be viewed as a higher-level type checking issue above programming languages. Cifuentes and Bierman [11] discuss the failure of mainstream programming languages to provide features that protect against this and related *injection errors* and other major categories of software vulnerabilities.

```
    if(namelen < 64)        // if namelen is int, this is a signed comparison
        strncpy(dstbuf, srcbuf, namelen);
```

Here the intent is that `srcbuf` holds a filename of length `namelen`. Suppose both are external inputs under attacker control, and a negative `namelen` is somehow provided. The `if` test will pass, and `strncpy` will be called. But because it takes an `unsigned` int as third parameter, the compiler implicitly coerces `namelen` to `unsigned`, making it a large positive. A buffer overflow results.

Language continuum. If we drew a software security continuum comparing popular languages by number of language features that support (or undermine) security, C and C++ would be on the (insecure) left end, positioned only slightly right of assembly languages and Unix shell scripting. On the (secure) right end would be Python, Java, and Rust among other modern languages.

Exercise. (a) Draw a continuum as noted above, placing the following languages on it, in a linear order that you believe is justifiable: assembly language, C, C++, Java and shell scripting. Then give your justification. (b) Extend part (a) by adding two additional languages of your choice.

Exercise. Discuss how performance overhead and cognitive load on developers impact the practical adoption of programming language abstractions that aim to improve security. (Hint: [11, §3].)

Exercise. (a) Write your own C program to find the lowest (or most negative) and highest values representable by each of the 5 integer types noted on page 2, each in three cases: no modifier, `signed` modifier, `unsigned modifier`. Summarize in a table. (b) Find and specify the hardware platform that your program executed on (the answers vary across platforms). (c) For each of the 15 cases, extend your program (and table) to show results on adding the constant `1` to the largest representable value, and on subtracting `1` from the lowest representable value.

Project. Most of our discussion has implicitly assumed compiled languages, but languages such as Python are *interpreted* (often called *scripting languages*). (a) What is the difference between a compiler and an *interpreter*? (b) Explain what it means to be a **dynamically-typed** programming language. (c) Summarize Python's memory safety and type safety properties, and the major classes of security-related errors that its features rule out.

Project. This question asks for a summary comparison of security-related features of programming languages, using a table as a road map and summary. (a) In a 2x2 table, label three rows as: C, C++, Java. Then label four columns as our four levels of memory safety: L1 to L4. In each of the 12 cells, include a notation (e.g., empty, half, full) giving a qualitative rating on whether the language delivers the associated level or protection, and briefly explain each rating. (b) Do the same for 3 further languages selected from: JavaScript, Python, Java, C#, Go, Swift, Rust.

1.6.1 A closer look at Java

We now look at Java as a specific language example, with focus on security-related aspects including data typing and memory safety. Although beyond our scope here, we note that Java is distinguished in that its programs are compiled to *bytecode*, which is then loaded and run on a JVM (page 17).

Java was designed to allow substantial type checking at compile time, but some checks must be done dynamically, e.g., for objects whose types are not statically known. Thus while being statically *typed* (in the sense that variables must be declared before use, similar to C), Java is not entirely a statically-*checked* language.

Java's approach of using *references* (next paragraph) rather than raw pointers shares properties with many other high-level languages that use *garbage collection* (page 5), as it does.

References in Java. A Java variable for a non-primitive object does not directly store a value. Instead, the storage location bound to the variable contains information that enables access to an object value on the heap. For example, `rec = new Record()` allows access via `rec.firstfield`, but no memory address can be derived from variable `rec`. This rules out programmatic manipulation of addresses; the abstraction hides address details from programmers. The internal implementation of a Java reference to objects (e.g., growable strings or vectors) nonetheless involves a pointer to the value (as well as metadata that enables validity checks). But Java's design eliminates C's explicit pointer dereference (`*`) and address (`&`) operators.

Java allows setting a reference to empty (`rec = null`), implying no object present. The value bound is then a semantic value denoting `null`. In this case, trying to access `rec.firstfield` throws an exception on the attempt to dereference a null reference—called `NullPointerException` for historical reasons. Thus Java's implementation of a **reference** does not simply try to follow a raw pointer as would be done in C, but instead involves semantic checks (including bounds-checking).

If `a` and `b` are variables for objects of the same type in Java, then `b = a` does not result in a copy of object `a` being created for `b`, but rather, `b` becomes a reference to the same object as `a`. Now `a` and `b` are *aliases* for the same object, and modifying the value of one changes both. (We will see later how Rust avoids this issue.)

Notes relative to C, C++, Rust. While C *raw pointers* provide a concrete means to access a value, and in that sense *refer* to a value, they are quite distinct from Java *references*. So-called **smart pointers** in C++ support some features of Java references (e.g., metadata allowing run-time bounds-checking on array accesses), but are perhaps better viewed as *wrapped pointers* more akin to Rust's smart pointers (§1.7.1). Features associated with C++ and Rust smart pointers also support memory management, i.e., freeing memory (recall that neither language is garbage-collected).

Type-casting. In Java, casting creates a reference to an object having a type compatible with the original (the original object is unchanged). Here to be *compatible* requires an existing relationship in Java's object inheritance hierarchy. *Valid* casts include explicit *downcasts* from a parent *superclass* type to a child *subclass*,[13] and *upcasts* from a child class to its parent class (*supertype*). Primitives can be downcast (*narrowcast*) to narrower types, e.g., `int` to `short`, or `float` to `int`; the expression `(int) doublefloatx` could be assigned to an `int` variable, narrowing to a value of 10 if variable `doublefloatx` had value 10.8. The opposite, a *widecast*, is done implicitly (automatically) by the compiler when needed; however narrowing requires an explicit cast (to avoid error messages).

Overall, type conversions for Java non-primitives (objects) have tight constraints, but mirror those for primitives (implicit/coercion, explicit/casting). Recall the class (type) hierarchy for Java's numeric (arithmetic) primitive data types, from top down: `double`, `float`, `long`, `int`, `short`, `byte`.

[13]Recall that `extend` is used to define a subclass (child) from a base class (parent), to provide specialized properties.

While some casting errors can be statically detected, compatibility checks are done at run time if a type is unknown at compile time. Run-time cast errors result in a `ClassCastException` being thrown. *Run-time type information* (below) is thus needed for type-checking dynamic casts—and for expressions involving the binary operator `instanceof`, which is often used to avoid this exception. For example, `(obj instanceof class)` returns a boolean, allowing control-flow decisions based on a run-time test of an object's type against a statically known type `(class)`.

Java run-time type information [55]. Recall from first year courses that defining a Java *class* creates a new Java *type*, which variables can be declared to have, thereby granting access to methods within the class. On compilation, the resulting *class file* contains information loaded later by the Java Virtual Machine (JVM). From this the JVM builds a dictionary of *run-time type information*, and creates an object of class `java.lang.Class` for each type loaded (providing class metadata and *methods*). To use this type information in support of runtime type-checking (such as for dynamic casts), object instances in the JVM are associated with the dictionary type information for the object's class. This might be done by a pointer from the object instance to the dictionary type information. In contrast, note that traditional C maintains no type information at run time.

Memory safety in Java. Java is generally said to be *memory safe*. Our L1 is largely[14] delivered by Java's design of references (no program-based access to explicit addresses), type checking and casting rules, avoiding temporal errors by garbage collection and spatial errors by checking object bounds through compile-time and (when not ruled out as safe) run-time bounds-checks. L2 is achieved by flagging use of uninitialized local variables as a compile-time error, and assigning default values to uninitialized class variables (and fields therein): 0 or `false` for primitive types and `NULL` for objects (non-primitives including `String`). For L3, Java's use of garbage collection can substantially reduce memory leaks. While Java remains susceptible to programming errors related to **integer overflow** (page 14), its language specification does define that the resulting behavior must be as expected from two's complement arithmetic—whereas C officially only states that program behavior from overflow of anything other than `unsigned` integral types is undefined.

Homework. Read further details on how Java type-checking works, and an example of type confusion (which typically relies on implementation errors in supporting infrastructure), from McGraw and Felten's 1999 book [31, §2.10 and §5.7] at `http://www.securingjava.com/`.

Homework. This explores Java support to prevent *data races* (L4), towards *thread-safety*. Read Chapter 12 (*Threads and Multiprocessing*) in Eck's openly accessible book [13]. (a) Define *race condition*. (b) Describe how exclusive access to an object (through *mutual exclusion* as provided by a *lock*) can avoid **data races**. (c) Describe the syntax and use of Java `synchronized` statements and `synchronized` methods. (d) Explain why synchronized access, if not done carefully, can result in another problem called **deadlock** (begin by defining this term).

Exercise. As noted above, Java's design aims to avoid information leaks by requiring that local variables be initialized, and assigning default values to class variables that are otherwise uninitialized. However, a separate question is: do Java garbage collectors typically zeroize free pool memory? For

[14]L1 memory errors remain possible in Java (and other languages) through a general exposure to code used in run-time libraries (e.g., libc) and the supporting environment, possibly including in implementation of standard Java classes.

example, if a password or secret key object was in heap memory, and is freed (possibly compacted during garbage collection), will the secret still reside in memory? Explore, explain and discuss.

1.7 Rust introduction and motivation

Awareness of C's memory safety failures, and the security implications, continues to grow. The Rust programming language is the strongest serious contender as a systems-level alternative to C, i.e., a lower-level language suitable for operating systems and browsers and interfacing to hardware. Here we briefly introduce a selection of Rust's security-related design features, and encourage pursuit of further details in a separate course.

While well known to security experts for over 35 years—even before the 1988 Internet worm incident—the importance of security in programming languages is now reaching wider audiences. Recent efforts to deliver this message include Gaynor's short 2019 piece encouraging VPs of Engineering to choose languages other than C/C++ [18], a high-level overview of memory safety from the US National Security Agency in November 2022 [34], and a January 2023 report [21] summarizing a discussion sponsored by Consumer Reports on how to encourage adoption of memory-safe languages. Along the same lines, arguments were made in 2021 for using Rust in the Linux kernel [53],[15] as the *Rust for Linux* project gained momentum.[16] Arguments for favoring Rust and abandoning C in introductory OS courses date back to 2013 [14].

1.7.1 Rust overview: ownership, smart pointers, and memory safety

Here we provide a selective tour of Rust design features that support security, and related syntax (C-like at core). Almost from the start, Rust developers must be aware of the location of memory associated with an object (heap, stack, or a data segment). Heap memory is of special concern, as heap objects internally involve raw pointers pointing to values. As we will see, Rust's design avoids a single object (at any one time) being pointed to by more than one pointer having the capability to modify the object's value. Such write-capable *aliases* are historically a root cause of major security vulnerabilities in C, and aliases in general complicate static analysis (e.g., by compilers) to detect a variety of programming errors and dangerous practices. Rust's design enables such detailed static analysis, and provides significant features for handling errors at run time—together addressing many of the memory errors that we have discussed.

Mutability and moving. By default, the value of a Rust variable cannot be changed once assigned. To do so requires an explicit declaration of the variable as *mutable* (keyword: `mut`):

```
1: let mut i = 7;     // we say this binds the integer value 7 to variable i
2: i = i + 1;         // update allowed, because the variable is mutable
3: let j = i;         // type inference used (type declaration optional here in Rust)
```

After line 1, `i` is said to *own*[17] this instance of the value `7`. At line 3, a copy of value `8` is made and bound to variable `j`. Both `i` and `j` have value `8`, stored as separate copies on the run-time stack.

In contrast to `i`, `j`, consider Rust variables `u`, `v` of type `String`. If `u` had string value `"STRing1"`, then `let v = u` would *not* result in a copy of the string being made—because dynamic strings are

[15]See also Filho [15] and Anderson [3].

[16]`https://en.wikipedia.org/wiki/Rust_for_Linux`

[17]*Ownership* is a design concept used in other languages including C++.

stored in heap memory, and by default Rust does not copy heap data (which is called *deep copying*). Instead, value `"STRing1"` is said to be *moved* from `u` to `v`, transferring ownership of the value to `v`. The value is no longer accessible through `u` (attempted access yields a compiler error).

Thus from the beginning, we see that different rules apply for heap-allocated objects (such as dynamic strings) than for simple objects of known size at compile time (e.g., an integer), which are easily and efficiently copied and typically use stack memory, which can be freed by stack pops.

The rules on *owning* and *moving* values apply to all assignments—not only explicit assignment statements, but also function calls, both when variables as instances of actual arguments are converted to formal parameters, and in passing return values back to calling functions. These, and evaluation of expressions, are all areas to watch closely, as instances where values are bound to variables. The importance of rules around assignment of values should not surprise us, as that is where the action is, e.g., type conversions, opportunities for type confusion, and creation of aliases.

Passing a variable of type `String` to a function *moves* the string value (and its ownership) to the scope of the function. However, passing a primitive-type variable (such as `i:u32`, unsigned 32-bit) to a function does not move ownership of its value; instead, a copy is made, based on the design decision that a `u32` (and other fixed-size data types) can be *shallow-copied* using a known, constant amount of stack memory.

Ownership rule #1. Each value has a single *owner* object at any one time. The owner is the only entity with authority to alter the value (i.e., a single owner has *write* privileges).

Scope rule #1. When an object goes *out of scope*, associated dynamic memory is freed. This is done automatically by the compiler inserting code to deallocate memory (via a *destructor* function).

As a baseline, the scope of a non-global object is the function or block it is in; ownership can be transferred out of a block through a return value. Rust tightens an object's scope (*lifetime*) to its minimum span of actual use within a block (e.g., rather than the end of a block or function); in some cases this avoids otherwise violating *Ownership rule #2*, further below. Beyond an object's lifetime, neither the object nor its value is *valid* for access; compiler errors (on attempts to access invalid objects) clarify valid spans.

A variable's scope may be refined by blocks delimited by braces ({ }) and by explicit annotation (using optional scope labels in variable declarations). While the scoping rules at first appear to add time and complexity burdens on programmers, they enable detailed compile time type checking that helps prevent memory errors and related security vulnerabilities. The win is long-term, in the form of fewer bugs, as a result of the safety guarantees that the compiler can deliver.

Shadowing. The `let` keyword declares a new variable. A variable identifier can be reused in the same scope (possibly with different type and mutability) by again using `let`; the keyword reminds us that it is a new declaration. The old instance of the variable name and its value become invalid; they are no longer accessible, and are said to be *shadowed* by the new instance and value.

References and borrowing. The syntax `&i` denotes a *reference* to variable `i` (not its value). The reference can be used, e.g., to assign to another variable or as a *pass-by-reference* argument to a function (vs. passing a copy of the value, *pass-by-value*). As an example:

```
        let b: &i32 = &a;        // &i32 denotes b's type as: reference to a 32-bit integer
```

Unlike C's raw pointers, Rust aims to guarantee that throughout the lifetime of a reference, it always refers (points) to valid values of a given type. References to non-primitive types are not raw pointers, but *fat* or *smart pointers* (these and dealing with NULL pointers are discussed shortly).

Like other variables, references are immutable unless declared `mut`. To illustrate, consider a variable `playerdata` whose type is a `struct` with an integer field `birthyear`:

```
0:  struct Year {birthyear:  u16,}       // struct definition, one field only
...
1:  let playerdata = Year {birthyear:  2000};    // initialized field
2:  let p = &playerdata;
3:  p.birthyear = 2001;      // causes compile error
```

The error is fixed by declaring `p` to be a *mutable reference* (`&mut`), and also `playerdata` to be mutable—making explicit, to both compiler and developer, the intention to modify data:

```
1a:  let mut playerdata = Year {birthyear:  2000};
2a:  let p = &mut playerdata;
```

An `&` reference allows reading but not writing (analogy: it is improper to write in a borrowed book). In Rust, `&` is called the *borrow* operator (for shared read access), and an `&` reference is said to *borrow* a value; in contrast, a mutable borrow, `&mut`, also allows write access (but no shared reads), and *moves* a value's ownership. A borrow's read access privilege ends (is returned) when the variable's lifetime (scope) ends. This is tracked by a component of the compiler called the *borrow checker*, and supports enforcement of our second ownership rule:

Ownership rule #2. Rust allows either one single mutable reference to a value (and no immutable references in this case), or multiple immutable references (with read-only access).

Invalid accesses. Some Rust data types are fixed-length once declared (e.g., *array* and *tuple* types). Others are variable-length, e.g., `String`, and *vector* type `Vec<T>` for growable collections of a fixed basic type `T` (say, `u32`, for 32-bit unsigned ints). Accessing an invalid collection element, e.g., the ninth element of a vector that has only elements 0..7, is caught as an error. For example, a vector element access may be attempted by: `&myvec[8]`, or `myvec.get(8)`. The first would trigger a run-time *panic* error (program crash). The second would return a semantic value `None`, enabling programmatic error-handling. Notably, in neither case do (L1 spatial) memory errors occur.

`Option` types and NULL pointers. To help eliminate NULL pointers in regular code, Rust supports a generic type, `Option<T>`, where `T` is a type parameter. For references to objects `x` of type `T`, `Option<T>` takes values:

- `Some(x)`, for the non-NULL case, or
- `None`, for the NULL case (the absence of a value).

Proper code explicitly handles both cases, replacing *ad hoc* (or missing) NULL pointer checks in C code. In the case of `Some(x)`, the value `x` (said to be *wrapped*) can itself be accessed using `.unwrap()`. C programmers and novice *Rustaceans* may initially view this as tedious; the tradeoff is a short-term cost to avoid later costs of fixing hidden bugs and their consequences.

The `Option` type also appears in one of the built-in method categories available for custom handling of integer overflow. (Arithmetic overflow results in defined behavior in Rust, but this does not eliminate related programming errors.) By default, integer overflow triggers a run-time panic in debug builds, and *wrap-around* (arithmetic modulo 2^n) in release builds [6]. If this is not as desired, the defaults can be overridden by using built-in integer operation methods: (1) *checked overflow* returns `Some(x)` if a result `x` is representable within the type's range, else `None`; (2) *saturation* returns a result `x` pinned to the maximum or minimum range value closest to the correct result; and (3) *wrap-around overflow* is as per the default, or a variation can furthermore indicate TRUE if wrapping actually occurred (a result tuple is comprised of the value, and a boolean flag).

Homework. For a better understanding of the use of the `Option<T>` construct in Rust, read the explanation and code examples in (§32: Option and Result) of David MacLeod's online book [29]: `https://dhghomon.github.io/easy_rust/Chapter_31.html`

String. The `String` type noted above is an example of a Rust *smart pointer* (as is `Vec<T>`). `String` is implemented as a built-in (C-like) `struct` (see Fig.2). One field is a protected[18] raw pointer to a value in heap memory, a second supports bounds-checking (`len`, the size currently in use), and a third helps manage dynamic growth of a string (`capacity`, the maximum size per memory currently allocated to this object). Rust `String` objects are well-supported by this combination of pointer (to string value data) plus metadata used to deliver guarantees; in contrast, C strings are programmer-built using a raw pointer to a first `char` and a hope that the following sequence of bytes is terminated by a NULL char before the end of the memory allocated for the string.

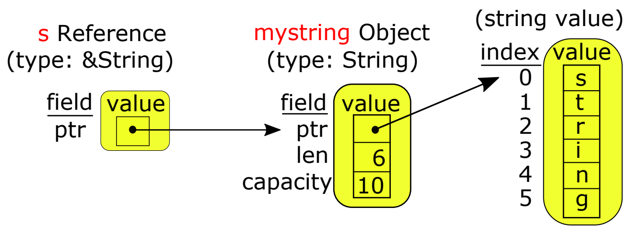

Figure 2: Rust smart pointer for `String` object `mystring` with value "`string`". The figure shows the result of a call `fn_on_str(&mystring)`, for a function with signature `fn_on_str(s: &String)`. A reference to the object of type `String` is passed, and populates the formal parameter `s`.

String traits. The `String` type has two so-called *traits* relevant to our interests. Traits provide functionality available for a given type (functions operating on values of that type). The `Deref` trait allows dereferencing the smart pointer (using $\star$). The `Drop` trait enables a function specifying customized clean-up actions when an object goes *out of scope*; the compiler automatically inserts code to call this function, and default code to deallocate (free up) heap memory. This lightens the programmer's burden and avoids programmer errors leading to memory leaks, or releasing memory twice—double-frees. The `Drop` trait thus supports implementation of *Scope rule #1*.

Rust's combination of compile time static analysis and automated insertion of code to reclaim heap memory amounts to a **third approach to dynamic memory management**, distinct from traditional garbage collection, and manual programmer allocation plus deallocation of heap memory.

&str (string slice). Rust's *string slice*, denoted `&str`, is another smart pointer data type. It is used to reference a substring (possibly the entirety) of either a dynamic string (`String`) or a constant string (constant strings are immutable and have type `str`, but no inherent metadata). Objects of

[18]By *protected* we mean not subject to alteration (e.g., by pointer arithmetic) in (`safe`) Rust code. However, blocks denoted by the `unsafe` keyword open up a special compiler mode that relaxes some constraints, allowing developers to carry out operations not normally permitted—including potentially dangerous ones such as dereferencing raw pointers. The cost is the loss of some safety guarantees, such as pointers always pointing to valid objects and being non-NULL.

type `&str` have a pointer (to the first character of the substring value) and a `len` field, but not the `capacity` field found in the `String` struct of Fig.2.

Box pointers. The `Box<T>` pointer type is the simplest Rust smart pointer. It results in a fixed-size object allocated on the run-time stack, holding simply a pointer to a value of type `T`, allocated from heap memory (rather than stack). A common use is to avoid inefficient *deep copying* of large data structures, when the value is to be moved to a new owner (the new owner object can simply be given a pointer to the heap value). See Fig.3.

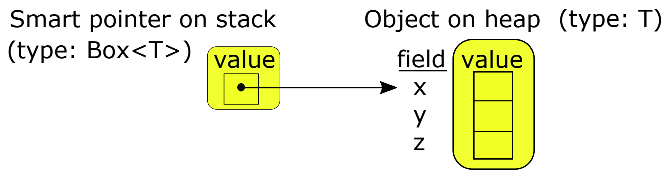

Figure 3: `Box<T>` pointer. As with other smart pointers in Rust, it has `Drop` and `Deref` traits.

Reference-counting pointers. The `Rc<T>` type is a further example of smart pointers. It allows a data value to have multiple *owners*, but all are read-only, to avoid **data races** (page 11). A count is kept of how many references point to the data value; if the count is zero, the value becomes invalid and is cleaned up.[19] The `Rc<T>` type also has a `Drop` trait, of use to update reference counts.

Memory safety and `unsafe` Rust. Code blocks denoted by the `unsafe` keyword open up a special compiler mode that relaxes some constraints, allowing developers to carry out operations not normally permitted in Rust—including potentially dangerous pointer arithmetic, and dereferencing of raw pointers. The price is thus the loss of some safety guarantees, such as pointers always being non-NULL and pointing to valid objects. Rust's hope (and culture) is that programmers strictly scrutinize and minimize use of `unsafe` blocks. This localizes and significantly reduces the volume of code at higher risk of memory vulnerabilities, but we should remember that the difficulty of finding security flaws by 'scrutinizing' is what motivated using Rust versus C in the first place.

Excluding `unsafe` blocks, most security issues involving invalid pointers are eliminated by Rust's features, such as ownership and borrowing rules enforced by the *borrow checker* and static type checking system; smart pointers; and `Drop` traits supporting automated memory deallocation. These and scope/lifetime rules allow tracking of object validity and verification that all references (pointers) are valid. In particular, dangling references are flagged as compile time errors.

Regarding our memory safety levels L1-L4, spatial and temporal memory errors (L1) are largely eliminated by the above features. For L2, reading from uninitialized variables is flagged as a compile-time error, concerns about wild pointers and invalid heap objects are addressed by Rust's guarantees that all references are validated before use, and the `Option` data type mitigates NULL pointer errors. On L3, Rust's approach to dynamic memory management, including support of `Drop` traits and reference-counting pointers, helps reduce *memory leaks* by facilitating compiler-supported memory deallocation. Rust does not fully eliminate memory errors, due to `unsafe` code, including run-time libraries that may use `unsafe` blocks, and components not written in Rust.

Finally, data races (L4) are eliminated in `safe` Rust by its enforcement of *Ownership rule #2*, combined with features supporting multi-threaded programs such as syntax allowing the movement and tracking of object ownership across threads. Unsurprisingly, other race conditions related to

[19]Those familiar with Unix file system `inodes` will recognize this strategy.

thread synchronization remain (e.g., *deadlock*[20]). Overall, Rust (excluding blocks marked `unsafe`) is viewed as type-safe and memory-safe.

While Rust is a big step forward in memory safety for systems languages, security issues remain. One category stems from the reality that while most binaries rely on libraries from a variety of source languages and tool chains, security guarantees are not composable across such components. Rust delivers guarantees that rely on compile-time analysis, as well as optimizations via eliminating apparently redundant runtime checks—but this combination has been shown to enable use of safe Rust code to exploit non-Rust code in *mixed binaries*, e.g., combining output from C/C++, and Rust compiler tool chains that have differing approaches to delivering memory safety [33], [37].

Homework. Read "Introduction to Memory Unsafety for VPs of Engineering" [18]. Summarize its main technical points, similarities to, and differences with our memory safety discussion herein.

Homework. To better understand ownership, references, borrowing, and how strings are represented in Rust, read Chapter 4 of Klabnik and Nicols [28] at `https://doc.rust-lang.org/book/` and work through its examples. (You need not install Rust locally to work through the examples.)

Homework. (a) Using the *Rust Playground* at `https://play.rust-lang.org/`, get a feel for Rust by selecting and modifying a few examples from the online book *Easy Rust* [29]. (b) Explore the resources at *Rust By Example*: `https://doc.rust-lang.org/rust-by-example/`

Homework. Review the following page discussing undefined behavior in `unsafe` Rust blocks: `https://doc.rust-lang.org/reference/behavior-considered-undefined.html`

Exercise. With the aid of a summary table, compare and briefly explain the differences in major security-related language features of Rust and C (e.g., type systems, approach to dynamic memory management, syntax for casting and pointer arithmetic, features supporting memory safety).

Conclusion. Longterm, it remains to be seen whether Rust will displace C and its cousins, or ongoing hardening of C via tools and runtime support (software and hardware) will eventually mitigate enough classes of vulnerabilities to weaken the motivation to broadly adopt Rust.

Rust is known to have a steep learning curve (relative to C), especially for first-time programmers [30], [17]. On the other hand, there is a big difference between learning enough C syntax to write simple programs, and gaining sufficient experience to avoid writing code with hidden security vulnerabilities. This observation has led to suggestions that while it takes time to become a productive Rust developer, it takes longer to learn C well enough to avoid writing dangerous programs.

[20]This, discussion of concurrency, and **thread safety** are beyond our scope. See [28, §16: Fearless Concurrency].

Bibliographic notes. Tennent [51] gives a concise introduction to data typing; for type systems specifically, see Cardelli [8] for insightful exposition, and Pierce [42] for authoritative book-length treatment. Zorn [60] and Hertz [22] discuss garbage collection. Eck [13] gives a clear introduction to Java including safety properties. McGraw and Felten [31] give an overview of Java security circa 1999, including type confusion and type checking, the JVM and bytecode verifier; see Holzinger et al. [24] for a long-term look at Java security vulnerabilities. Song et al. [48] give a taxonomy of memory errors in C, and survey dynamic tools that find C security vulnerabilities. Fluet [16] offers insights on teaching Rust. For open access introductory Rust books see Klabnik and Nichols [28], or MacLeod [29] (in simple English with examples using Rust Playground [46]); Blandy et al. [6] provide a more advanced treatment in the insightful style of Kernighan and Ritchie's C book [27]. For C, see also Prinz and Crawford [45]. For recent work on memory errors related to stack objects, see Huang [25]. For background on data races, see Savage et al. [47].

The references below include additional resources relied on or related to these notes.

References

[1] P. Akritidis. *Practical memory safety for C*. PhD thesis. UCAM-CL-TR-798, Cambridge (UK) Computer Lab. 2011.

[2] A.A. de Amorim, C. Hritcu, B.C. Pierce. The meaning of memory safety. POST (Principles of Security and Trust), pp.79–105, 2018.

[3] Tim Anderson. Rusty Linux kernel draws closer with new patch adding support for Rust as second language. The Register. Dec 7 2021. `https://www.theregister.com/2021/12/07/rusty_linux_kernel_draws_closer/`

[4] T. Ballo. What is 'memory safety', really? Blue Team Rust blog, July 29, 2020. `https://tiemoko.com/blog/blue-team-rust/`. An overview of memory-related safety features designed into the Rust PL.

[5] T. Ballo, M. Ballo, A. James. *High Assurance Rust: Developing Secure and Robust Software*, 2022. `https://highassurance.rs`. Open online site and in-progress book, targetting a course that combines learning the Rust PL, software security, systems programming, and data structures, using production-grade tools.

[6] J. Blandy, J. Orendorff, L. Tindall. *Programming Rust: Fast, Safe Systems Development* (2nd edition). 735 pages. OReilly Media, 2021.

[7] L. Cardelli, P. Wegner. On understanding types, data abstractions, and polymorphism. *ACM Computing Surveys* 17(4):471-522, Dec 1985.

[8] L. Cardelli. Type Systems, pp.2208–2236 in: *The Computer Science and Engineering Handbook*, CRC Press, 1997. Revised version in 2nd Edition (2004), `http://lucacardelli.name/papers/typesystems.pdf`

[9] M.M. Carvalho, J. DeMott, R. Ford, D.A. Wheeler. Heartbleed 101. *IEEE Security & Privacy* 12(4):63-67, 2014 (July-Aug).

[10] D. Chisnall. C is not a low-level language. *Commun. ACM* 61(7):44-48, 2018.

[11] C. Cifuentes, G. Bierman. What is a secure programming language? Summit on Advances in Programming Languages (SNAPL), 2019.

[12] D. Dhurjati, S. Kowshik, V. Adve, C. Lattner. Memory safety without run-time checks or garbage collection. Languages, Compilers, and Tools for Embedded Systems (ACM LCTES), 2003.

[13] David J. Eck. *Introduction to Programming Using Java*. Version 9.0, May 2022. Hobart and William Smith Colleges. `https://math.hws.edu/javanotes/`

[14] D. Evans. Using Rust for an undergraduate OS course, 2013. `http://rust-class.org/0/pages/using-rust-for-an-undergraduate-os-course.html` A clear argument, predating Rust's current popularity, of why C should no longer be the language of choice in OS (systems programming) courses.

[15] W.A. Filho and Android Team. Rust in the Linux kernel. Google Security Blog, 14 April 2021. `https://security.googleblog.com/2021/04/rust-in-linux-kernel.html`

[16] M. Fluet. Experience Report: Two Semesters Teaching Rust. pp.45–58 in [30], August 2022. Appendix includes a set of Rust programming exercises.

[17] K Fulton, A Chan, D Votipka, M Hicks, M Mazurek. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. SOUPS 2021, pp.597–616. `https://www.usenix.org/conference/soups2021/presentation/fulton`

[18] A. Gaynor. Introduction to memory unsafety for VPs of engineering. Aug 12, 2019. Online: `https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/`.

[19] S. Ghorshi, Lachlan J. Gunn, H. Liljestrand, N. Asokan. Towards cryptographically-authenticated in-memory data structures. arXiv preprint, 20 Oct 2022. `https://arxiv.org/pdf/2210.11340.pdf`

[20] L. Gong. Java security architecture revisited. *Commun. ACM* 54(11):48-52, Nov 2011.

[21] Y. Grauer. Future of memory safety: Challenges and recommendations. January 2023. `https://advocacy.consumerreports.org/wp-content/uploads/2023/01/Memory-Safety-Convening-Report-1-1.pdf`

[22] M. Hertz, E.D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. Proc. ACM OOPSLA 2005, p.313-326. `https://people.cs.umass.edu/~emery/pubs/gcvsmalloc.pdf`

[23] Michael Hicks. What is memory safety? Blog article (4 pages), The Programming Languages Enthusiast, 21 July 2014. An insightful discussion from a co-author of the Cyclone work [26].

[24] P. Holzinger, S. Triller, A. Bartel, E. Bodden. An in-depth study of more than ten years of Java exploitation. ACM CCS, 2016. Cf. Paul and Evans [39].

[25] K. Huang, Y. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, T. Jaeger. The taming of the stack: Isolating stack data from memory errors. NDSS 2022.

[26] T. Jim, J.G. Morrisett, D. Grossman, M.W. Hicks, J. Cheney, Y. Wang. Cyclone: A safe dialect of C. USENIX Annual Technical Conference, pp.275-288, 2002.

[27] Brian Kernighan, Dennis Ritchie. *The C Programming Language (Second Edition)*, 1988. Prentice Hall. This covers the original "Standard C", also called C89/ANSI C.

[28] Steve Klabnik, Carol Nichols. *The Rust Programming Language* (covers Rust 2018). August 2019, 526 pages. No Starch Press. Online version: `https://doc.rust-lang.org/book/`

[29] David MacLeod (dhghomon). *Easy Rust*. `https://dhghomon.github.io/easy_rust/Chapter_1.html` (online, open access). Manning (to appear 2023, in print).

[30] B. Massey. Rust-Edu Workshop proceedings, `https://rust-edu.org/workshop/proceedings.pdf`, Aug 2022.

[31] G. McGraw, E.W. Felten. Securing Java: Getting Down to Business with Mobile Code. Wiley, 1999.

[32] Matt Miller (Microsoft). Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat Israel Conference, 7 February 2019. Slides 1–24. `https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf`

[33] S. Mergendahl, N. Burow, H. Okhravi. Cross-language attacks. NDSS 2022.

[34] National Security Agency. Software Memory Safety. Cybersecurity information sheet, Version 1.0, Nov. 2022. `https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF`

[35] P.C. van Oorschot. Toward Unseating the Unsafe C Programming Language. *IEEE Security & Privacy* 19(2):4-6 Mar-Apr 2021.

[36] P.C. van Oorschot. *Computer Security and the Internet: Tools and Jewels from Malware to Bitcoin*. Springer, 2021.

[37] M Papaevripides, E Athanasopoulos. Exploiting mixed binaries. *ACM Trans. Priv. Secur.* 24(2) 7:1-7:29, 2021.

[38] D.L. Parnas, J.E. Shore, D. Weiss. Abstract types defined as classes of variables. ACM SIGPLAN Conference on Data: Abstraction, Definition and Structure. March 1976, pp.149–154.

[39] N. Paul, D. Evans. Comparing Java and .NET security: Lessons learned and missed. *Computers & Security* 25 (2006) 338–350. Updates preliminary version from ACSAC 2004.

[40] M. Payer. How memory safety violations enable exploitation of programs. Chapter 1 (pp.1–23) in: *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, P. Larsen and A.-R. Sadeghi (eds.), ACM Books, 2018.

[41] M. Payer. *Software Security: Principles, Policies, and Protection*. Book notes used for EPFL course (v0.37 July 2021, 128 pages). Updated online at `https://nebelwelt.net/SS3P/softsec.pdf`

[42] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[43] F. Piessens. Software Security Knowledge Area (CyBOK, Version 1.0.1), July 2021. `https://www.cybok.org/media/downloads/Software_Security_v1.0.1.pdf`

[44] Erik Poll. *Lecture Notes on Language-Based Security*. Sept 2019. 51 pages. Radboud University, The Netherlands, `http://www.cs.ru.nl/~erikpoll/papers/language_based_security.pdf`. See especially Chapter 3 (Safe programming languages), pp.12–25.

[45] Peter Prinz, Tony Crawford. *C in a Nutshell* (first edition). O'Reilly, 2005. This covers C1999.

[46] rust-lang.org. Rust Playground (online). An open, browser-based environment for compiling and running Rust programs, as an alternative to locally installing a Rust compiler. `https://play.rust-lang.org/`

[47] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T.E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Trans. Comput. Syst.* 15(4):391-411, 1997.

[48] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, M. Franz. SoK: Sanitizing for Security. IEEE Symp. Security and Privacy, pp.1275-1295, 2019.

[49] D. Stuttard, M. Pinto. *The Web Application Hacker's Handbook*. Wiley, 2007.

[50] L. Szekeres, M. Payer, T. Wei, D. Song. SoK: Eternal war in memory. IEEE Symp. Security and Privacy, pp.48-62, 2013.

[51] R.D. Tennent. *Principles of Programming Languages*. Prentice-Hall International, 1981.

[52] J.-N. Tille. Unsafe languages, inadequate defense mechanisms and our dangerous addiction to legacy code. `https://blog.mi.hdm-stuttgart.de/index.php/2021/08/14/unsafe-programming-languages/`. 14 Aug 2021.

[53] S.J. Vaughan-Nichols. Keep calm and learn Rust: We'll be seeing a lot more of the language in Linux very soon. Nov 10 2021. The Register. `https://www.theregister.com/2021/11/10/where_rust_fits_into_linux/`

[54] V. van der Veen, N. dutt-Sharma, L. Cavallaro, H. Bos. Memory errors: The past, the present, and the future. RAID, 2012.

[55] Bill Venners. Design with runtime class information. JavaWorld. Feb 1, 1999. `https://www.infoworld.com/article/2076355/design-with-runtime-class-information.html`

[56] D.A. Wagner, J.S. Foster, E.A. Brewer, A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. NDSS, 2000.

[57] D.A. Wheeler, *Secure Programming HOWTO*. Sept 19, 2015 (v3.72). `https://dwheeler.com/secure-programs/Secure-Programs-HOWTO.pdf`. (distributable under the GNU Free Documentation License). For background information on this book see `https://dwheeler.com/secure-programs/`

[58] T. Würthinger, C. Wimmer, H. Mössenböck. Array bounds check elimination for the Java HotSpot client compiler. Symp. on Principles and Practice of Programmining in Java (PPPJ), pp123-133, 2007.

[59] Y. Younan, W. Joosen, F. Piessens. Code injection in C and C++: A survey of vulnerabilities and countermeasures. Katholieke Universiteit Leuven (Belgium), Dept of Computer Science, Report CW 386 (81 pages), July 2004. `https://www.cs.kuleuven.be/publicaties/rapporten/cw/CW386.pdf`

[60] B. Zorn, The measured cost of conservative garbage collection. *Software: Practice and Experience* 23(7):733-756, July 1993.