

Safe System Programming

Luca Abeni

`luca.abeni@santannapisa.it`

November 10, 2025

Safe System Programming

- Two concepts: **system programming** and **safe program** / safety
- System programming: programming system software
 - Operating System (both kernel and user-space)
 - Important “system libraries”
 - Other software not traditionally considered part of the OS
 - Virtual Machine Monitors, ...
- Safety: not easy to define...
 - People often identify different kinds of safety
 - Different levels of safety...

System Software and its Importance

- Why is system software special?
 - Why “safe system programming” and not generically “safe programming”?
- All software needs to be “safe” and “trusted”...
 - All software is equal, but some software is more equal than others
 - Seriously, the safety of system software affects all the other software running in the system!
- For application software, we can use techniques that are not usable on system software

Not for System Software

- Crashing might be an option for “regular” applications...
 - ...But I do not want my OS to crash!!!
- Sometimes, safety is enforced by heavyweight runtimes...
 - ...That are not available in an OS kernel!!!
 - Example: Java avoids risks of “double free” by using a garbage collector...
 - ...Implemented in the JVM → cannot be used for programming a kernel!
- User-space programs can rely on kernel protection...

Requirements for System Software

- System software is **performance critical** and **safety critical**
 - Conflicting requirements
- Traditionally implemented focusing (mainly) on performance
 - Using low-level languages such as Assembly and C
 - Sometimes C++
 - All *unsafe* languages!!!
- Safety mainly considered imposing constraints on the coding/development style
 - Example: MISRA C

System Programming Languages

- Designed/used to write system software
 - Focus on performance, performance, performance!
 - Must allow to directly access hardware resources
 - Generic/unsafe pointers
 - For kernel development, must allow to build programs without relying on syscalls → non-hosted/bare-metal → no runtime!
- What about safety/security?
 - Generally overlooked
- So, what is **safety**?

Safety

- No unique or formal definition
 - Many different definitions in literature
- Informally: a program is considered “safe” if it is possible to formally prove that it behaves correctly
 - “Behaves correctly”?
 - Or, “it does not do anything dangerous”...
 - Different possible variations...
- What about “safe” programming languages?
 - Safe programming language → enforces safety
 - A well-formed program cannot do anything dangerous
 - Given a well-formed program it is possible to formally prove that it behaves correctly

Different Kinds of Safety

- Type safety: well-formed programs cannot exhibit bugs due to type errors
 - Applying the wrong operation on the wrong type,
...
- Memory safety: well-formed programs cannot exhibit bugs due to wrong memory accesses
- Thread safety: well-formed programs cannot exhibit race conditions, deadlocks, and synchronization errors
- Other kinds of safety...
 - For example, a well-formed program has a well-defined behaviour (no UBs in C, etc...)

Memory Safety

- No bugs due to wrong memory accesses...
 - Difficult to provide a generic definition
- Definition “by examples”... Wrong memory access:
 - Buffer overflow
 - NULL pointer dereference
 - Use after free
 - Use of uninitialized memory
 - Illegal free (of an already-freed pointer, or a non-malloced pointer)
- Things like “no accesses to uninitialized memory” do not properly catch buffer overflows, etc...

Memory Safety vs Type Safety

- Sometimes, there is no clear distinction between type safety and memory safety
- Clear buffer overflow (violation of memory safety):

```
int *v = malloc(sizeof(int) * 10);  
v[10] = 666;
```

- What about this:

```
int v[10];  
v[10] = 666;
```

- Is it a buffer overflow or a type error?
 - Defines an array of 10 elements, and accesses the 11th...
- OK, C arrays are pointers, but what about C++:

```
std::vector<int> v(10, 0);  
v[10] = 666;
```

Enforcing Safety

- Safety can be enforced at compile time
 - Unsafe programs — whatever this means — do not even build
- Or at execution time
 - Some kind of “trusted language runtime” ensures that nothing bad happens
- According to someone, a safe program is a program that can rely on a trusted runtime
- Languages like Java try a mix of the two
 - No free() → remove the possibility to have use after free, etc
 - The JVM also enforces consistency, etc...

Breaking Memory Safety

- Features that might break memory safety:
 - No array bounds checks (or, is this type safety???)
 - Pointer arithmetic
 - NULL pointers (someone says, only if they cause UB)
 - Low-level memory management
- Low-level memory management:
 - Explicit C-style malloc()/free() (some say "use new" and "do not free()")
 - Explicit assignment of arbitrary values to pointers

Possible Solutions/Mitigations

- Some “coding standards” generically forbid dynamic memory allocation
 - This is crazy: they ban the usage of functions!
- Some others (MISRA C) forbid dynamic allocation from the heap (malloc()/free())
 - Still, a partial solution.
- Alternative: using a garbage collector
 - Coming from functional programming languages; then used by Java
- Pointers in general are dangerous (some languages try to avoid them)

Static vs Dynamic Checks

- Consider the code

```
int v[10];  
v[10] = 666;
```

- Should it fail to compile, or should it generate an exception at runtime?
- Static type checking: build failure
 - Early notification of (potential) bugs
 - Not always possible: what about `v[i] = 666;`?
- Dynamic type checking: exception/crash
 - Still safe (???)... Someone says “to make C safe, change all the UBs into crashes”...
 - Less useful for developers... But more for users?
 - Need for runtime support

Static or Dynamic?

- Dynamic checks are more permissive... Consider

```
int StrangeFunction(bool v)
{
    if (v) {
        x = 10;
    } else {
        x = "WTH???";
    }

    if (v) {
        return x * 2;
    }

    return len(x);
}
```

- But, is this really useful?
- If my program has potential bugs, I want it to fail to build!

Static Typing and Static Checks

- Static typing: programs with (even potential) type errors fail to build
- Dynamic typing: programs with type errors crash/generate exceptions
 - Still safe, but I prefer early notification
- Static typing requires a strong type system
 - Example: avoid the C's “automatic type promotion”
- We will see that this can help with memory safety too

The Dream

- Goal: “problematic code” (code that can have potential issues) does not even build
 - Eliminate an entire class of vulnerabilities before they ever happen
 - Cost: some valid code is considered invalid
- Need for some support at the language level!
 - Type theory can help, here!
 - Not a new idea: functional programming languages have already been there (for example)!
- Avoid heavyweight runtimes
 - Garbage collection, etc...

Tools for Safety — 1

- Static code analysis tools: search for possible issues in the code (without executing it)
- Taint analysis: check how “corrupted data” can affect the system
 - Performed as static analysis on source code or binary code
- Tools like valgrind, Address Sanitizer (asan) or other sanitizers, etc...
 - Maybe associated with fuzz testing
 - Still, this is testing, does not prevent dangerous code to build

Tools for Safety — 2

- Lots of **warnings** from compilers...
 - Warnings tend to change from compiler to compiler and from version to version
 - Only considered as “suggestions”
- Adopting “safe” development practices
 - Again, coding rules... Can be checked with some tools, at least
- Manual code review

Summing Up

- Lots of **external** tools for code analysis
 - Not really integrated with the language
- Mechanisms to detect memory errors, concurrency errors, and similar **at runtime**
 - Useful for testing
 - Prevent UBs
 - Need some runtime support (kasan does exist, but needs support in the Linux kernel!)
- Type/Memory safe languages exist
 - Java, C#, Haskell, Go, pick your name
 - All need a “not so lightweight” runtime
 - Still, safety is sometimes intended as “exception at runtime”...

Type/Memory Safe Languages

- Impossible to build programs that result in memory errors at runtime
- Again, various definitions of “memory error”...
- Example: Java
 - Null pointers do exist!
 - ...And null pointer dereference can happen even if you do not explicitly use null pointers!
 - But Java is safe because null pointer dereferences result in exceptions!
- Safety is often checked only dynamically
 - Sometimes, there are no other options!
- What about **safe system languages**?

System Languages and Safety

- Bad news: system languages **have to** be unsafe...
 - Why? Think about I/O...
 - To access an I/O device, raw (and unchecked) memory access is needed...
- Similitude with pure functional languages
 - A pure functional language allows no side effects...
 - ...But side effects are needed! (again, I/O...)
 - Solution: isolate side-effect in a runtime/abstract machine/well-defined software component
- Maybe, it is possible to precisely isolate unsafe sections of code?
 - Of course, this risks to open cans of worms...

Source of some Problems, again

- Buffer overflow
 - Can be statically checked only in some cases
- Issues with pointers
 - NULL pointer dereference
 - Can we **really** avoid NULL pointers???
 - Issues with memory de-allocation (use after free, illegal free)
 - Can we avoid C-style free()...
 - ...Without relying on garbage collectors?
 - Use of uninitialized memory
- Can we **avoid pointers**???

Programs: Code and Data

- Von Neumann architecture: programs == sequences of instructions that operate on data
 - Instructions and data are stored in memory
 - Long sequences of 0 and 1...
- Programming in machine language is not simple (reading/writing long sequences of bits!)
 - Assembly helps a little bit, introducing mnemonics for the machine instructions, and symbolic names for memory locations
- High-level languages introduce **variables**, **types**, and **values**

Variables and Values

- Variable \leftarrow high-level programming languages
 - Used to abstract programs from the usage of physical/virtual memory
 - “Box” (set of memory locations) that can contain a value
 - Referenced by using a symbolic name
- Value: sequence of bits encoding some high-level concept (number, character, string, ...)
 - The encoding depends on the *type* of the variable
- Data type: defines the semantics of the variable
 - Set of possible values the variable can contain
 - Operations such values
 - ...

Immutable Variables

- Variables can be mutable or immutable
- Immutable variable: **binding** between a symbolic name and a value
 - Environment: set of bindings (name \rightarrow value)
 - Function mapping names into values
- Variable declarations modify the environment
- There is **no way** to modify the value bound to a variable name
 - No assignments! Only initializations...
 - The only thing we can do is to define a new binding that *shadows* the old one

Mutable Variables

- The environment maps names into “boxes” (variables), not directly into values
- Additional function (memory) mapping variables into their contained values
 - Assignments modify the memory function, changing the value assigned to a variable in a variable (R-Value in C/C++)
- Aliasing: the same variable can have multiple names

Pointers

- Pointer type: special type, expressing references to variables
 - Possible values: memory addresses (of variables)...
 - ... + one special value, representing invalid pointers
 - The **NULL** value!!!
- Dereference operator: accesses the value contained in the pointed variable
 - Dereferencing the NULL value results in a **runtime** error!
- NULL is a value like the others; NULL dereferences cannot result in build errors

More on Data Types

- Every programming language has a set of *primitive types*
 - And many languages allow to define new types
- Simple way to define new types: apply sum or product operations to existing types
 - Product $\mathcal{T}_1 \times \mathcal{T}_2$: type with possible values given by **couples** of values from \mathcal{T}_1 and \mathcal{T}_2
 - Sum $\mathcal{T}_1 + \mathcal{T}_2$: type with possible values given by values from \mathcal{T}_1 **or** values from \mathcal{T}_2
- Sum == **disjoint** union; Product == cartesian product
- If $|\mathcal{T}|$ is the number of values of type \mathcal{T} , then
$$|\mathcal{T}_1 \times \mathcal{T}_2| = |\mathcal{T}_1| \cdot |\mathcal{T}_2| \text{ and } |\mathcal{T}_1 + \mathcal{T}_2| = |\mathcal{T}_1| + |\mathcal{T}_2|$$

Algebraic Data Types

- A set (the set of the language's data types), a sum operation and a product operation... It's an algebra!
 - Algebra of the data types; types are called Algebraic Data Types!
- Issue: the sum is a **disjoint union**...
 - Easy to do “float + bool” (type with possible values integers or booleans)...
 - ...But what about “int + int” (or similar)?
 - The types have to be tagged somehow...

Algebraic Data Types and Constructors

- Solution adopted by many programming languages: do not sum types directly, but first apply a *tagging function* to them
 - Constructor: function generating the values of the type to be summed
 - Summing types generated by different constructors, the issue is solved!
- Variant: set of values generated by a constructor
 - Different constructors generate disjoint variants
 - Hence, instead of “int + int” we can use “Left(int) + Right(int)”

Examples

- C unions are a special case of tagged sum
- “test = i(int) + f(float)” is

```
union example {  
    int i;  
    float f;  
};
```

- Of course, algebraic data types are more generic (0-arguments or multi-argument constructors, etc...)
- All constructors with 0 arguments: enum type
- Haskell, ML and others fully support ADT

```
datatype test = i of int | f of real;
```

```
data Test = I Int | F Float
```


Example: Option Type

- Type containing a value or nothing
 - Two constructors: “Nothing” (without arguments) and “Just” (with one argument of the desired type)
- Example: integer or nothing \rightarrow `Option_int = Nothing + Just(int)`
- Idea: instead of using a null pointer...
- ...Use an option type: `Pointer_to_int = Nothing + Just(int *)`
 - Advantage: only the “Just” variant can be dereferenced...
 - NULL pointer dereferences do not even compile!

Generic Data Types

- The definition of a new type might depend on a “type variable”
 - Parametric type, depending on another type “ T ”, denoted by a variable
 - Type variables, generally indicated as greek letters
- Example: generic option type
 - Not “integer or nothing”, but “type α or nothing”
 - α : type variable
- In Haskell, something like

```
data Option a = Nothing | Just a
```
- Used for many other things too (lists, **Monads**, ...)

Recursive Data Types

- To define a data type, we must (also) define all its possible values
- Set of possible values \rightarrow can be defined by induction...
- Can induction/recursion be used to define a new data type?
 - How? We need **induction base** and **induction step**
 - **Induction base**: one (or more) constructor(s) having 0 parameters (or, no parameters of the data type we are defining)
 - **Induction step**: constructor having a parameter of the type we are defining
- Looks... Confusing??? Let's look at some examples!

Recursive Data Types: Example

- Let's define the “**natural numbers**” data type (set of values: \mathcal{N})
 - $0 \in \mathcal{N}$: constructor `zero` (with no parameters)
 - $n \in \mathcal{N} \Rightarrow n + 1 \in \mathcal{N}$: constructor `succ`, having as an argument a natural number

```
datatype nat = zero | succ of nat;
```

```
data Nat = Zero | Succ Nat
```

- How to use this funny definition?
 - Combination of *pattern matching* and *recursion*
 - Familiar to people knowing functional programming

More Interesting Example: Lists

- Lists can also be defined by induction/recursion (simple example: list of integers)
 - **Inductive base**: an empty list is a list
 - **Inductive step**: A non-empty list is an integer followed by a list
- Recursive Data Type: a non-empty list is defined based on the list data type (constructor receiving a list as a parameter)
- Two constructors
 - Empty list constructor
 - Constructor for non-empty lists

Lists as RDTs — 1

- Two constructors
 - Empty list constructor (no parameters)
 - Constructor for non-empty lists (two parameters: an integer and a list)
- Other operations
 - `car`: returns the first element of a non-empty list (head)
 - `cdr`: given a non-empty list, returns the “rest of the list”

Lists as RDTs — 2

- How are lists generally implemented?
- Functional languages (Haskell, ML Lisp & friends, ...)
 - Recursive data type!!!
 - “cons” constructor: parameter of type `int * list` (or, a parameter of type `int`, but returns a function `list -> list`)
- Imperative languages: pointers!
 - Structure with 2 fields (types “`int`” and “`list*`”)
 - Second field: **pointer to next element**
 - Cannot be of type “`list`”, → use “pointer to `list`”!

RDTs vs Pointers

- See? Imperative languages use pointers and explicit memory allocation...
 - Adding an element to list implies doing some `malloc()/new` for a node structure, setting some “next” pointers, etc...
- ...In functional languages, RDTs avoid the need for pointers, and memory allocation/deallocation is hidden...
 - Adding an element in front of a list “`l`” is as simple as “`let l1 = cons(e, l)`” or similar!
 - The implementation of the language abstract machine will take care of allocating memory, etc...