TrapC: Memory Safe C Programming with No UB

Pr. Robin Rowe <Robin.Rowe@trasec.com>
Number: n3423 - TrapC
Date: 2025-01-07
For: ISO WG14 C Committee meeting at Graz University of Technology, Graz, Austria, February 24 to February 28, 2025

Introduction

The IEEE LangSec Committee considers the Internet insecurity epidemic a consequence of ad hoc input handling. IEEE describes LangSec (Language Security) as mission assurance for connected software and hardware exposed to attacks via malicious inputs mitigated through a practical data and code co-design methodology and filtering of legacy formats down to safe subsets.[1]

The NSA, the FBI, the White House, Five Eyes and other government agencies have declared that the lack of memory safety in C and C++, and resultant crashes and hacker exploits, is a nation security issue. The issue now so mainstream that memory safety is even covered by Consumer Reports.[2]

TrapC[3] is a programming language forked from C, with changes to make it LangSec and Memory Safe.[4] To accomplish that, TrapC seeks to eliminate all Undefined Behavior (UB)[5] in the C programming language. TrapC has about the same number of language keywords as C, is a much smaller programming language than C++. What's different about TrapC from other C code safety approaches, such as MISRA C,[6] is TrapC enforces code safety at the language level, not as a list of recommended best practices.

What is TrapC in a Nutshell?

- TrapC is a C language extension that improves code safety

[1] IEEE LangSec. https://langsec.org/

[2] Yael Grauer. Consumer Reports. (Jan 2023) Report: Future of Memory Safety. https://advocacy.consumerreports.org/wp-content/uploads/2023/01/Memory-Safety-Convening-Report-1-1.pdf

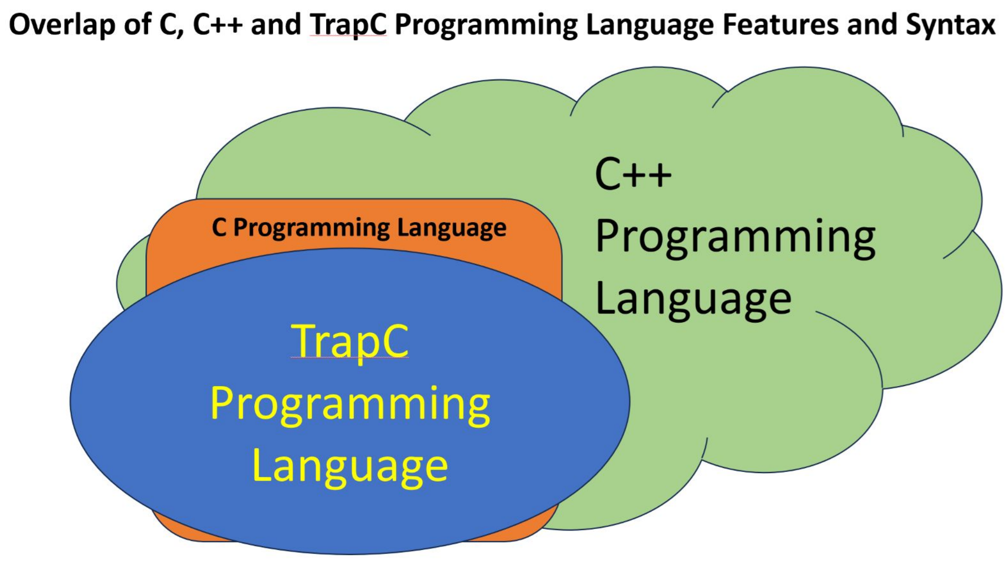
[3] TrapC. https://trapc.org/

[4] Memory Safety. https://en.wikipedia.org/wiki/Memory_safety

[5] Educational Undefined Behavior Technical Report n3308. https://www.open-std.org/JTC1/sc22/wg14/www/docs/n3308.pdf

[6] MISRA C. https://misra.org.uk/

- TrapC memory management is automatic, cannot memory leak, with pointers lifetime-managed not garbage collected
- TrapC pointers may look the same as in C/C++, but all pointers are diligently compiler-managed and memory safe
- TrapC pointers have Run-Time Type Information (RTTI), with typeof(), nameof() and other details accessible
- TrapC reuses a few code safety features from C++, notably member functions, constructors, destructors and 'new'
- TrapC adds 2 keywords unique to TrapC: 'trap' (error handling) and 'alias' (operator, function and data overloading)
- TrapC removes 2 keywords: 'goto' and 'union', as unsafe and having been widely deprecated from use
- TrapC uses castplates to make C containers typesafe, without the complexity of C++ templates
- TrapC printf() and scanf() are typesafe, overloadable, and have JSON and localization support built in
- TrapC has an integer-based 'decimal' fixed-point data type suitable for use in financial transactions
- TrapC is one-way ABI compatible with C, such that TrapC functions may call any C function as-is
- Passing a raw C pointer safely to a TrapC function requires extra steps, because TrapC pointers have hidden RTTI
- TrapC has API-compatible versions of C POSIX and C++ STL standard libraries, to not return raw pointers
- TrapC doesn't do more than C for thread safety to prevent race conditions, but may in the future


Overlap of C, C++ and TrapC Programming Language Features and Syntax
C++ Programming Language
C Programming Language
TrapC Programming Language

Overlap of C, C++ and TrapC Programming Language Features and Syntax

The TRASEC trapc Compiler

The startup TRASEC and the non-profit Fountain Abode have a TrapC compiler in development called trapc. Mission is to enable recompiling legacy C code into executables that are safe by design and secure by default, without needing much code refactoring. TrapC the programming language and trapc the compiler are highly compatibile with C, and to some extent with C++. The TRASEC trapc cybersecurity compiler with AI code reasoning is expected to release as free open source software (FOSS) sometime in 2025.

While TrapC the programming language cannot make any performance claims, a goal of the trapc compiler is to produce excutables that are smaller and faster than from C compilers. It is widely believed that C builds faster and smaller executables than other platforms thanks in part to UB. The thinking is C programs cutting corners on safety have fewer things to check, therefore less computation, that the high goal of being both safer and faster cannot be achieved. The author anticipates that with TrapC programming language improvements, safety can be faster, that trapc will produce optimizations that cannot be done in C. That said, the rest of this paper describes the programming language TrapC, not the compiler trapc.

Unrelated C-to-Rust Translation Efforts

TrapC differs in approach from efforts to achieve memory safety by rewriting C code in Rust. Starting in March 2025, the DARPA TRACTOR “Translating all C to Rust” research project will attempt to use AI to translate C code wholesale into the equivalent Rust code.[7] And, there are

[7] DARPA TRACTOR. https://www.darpa.mil/research/programs/translating-all-c-to-rust

efforts to migrate C-to-Rust manually. Germany's Soverign Tech Fund (STF) is a sponsor of FreeBSD, the Rustls TLS library written in Rust, and the uutils rewrite of the Linux coreutils library in Rust.[8]

Hello World: C and C++ Code Compilation Compatibility

TrapC is compatible with most C code…

```
// hello.c
#include <stdio.h>

int main()
{   puts("hello world");
    return 0;
}
```

TrapC compatibility when compiling C code is limited in a small way by the removal of 'goto' and 'union'. Legacy system maintainers who like 'goto' and 'union' may prefer to keep such code in C. From TrapC that may be linked to as a C library, or by linking to a C++ library using extern 'C'. Linking TrapC applications to C/C++ libraries comes with the risk that code compiled as C/C++ is not memory safe. For memory safety, recompile legacy C code with a TrapC compiler.

TrapC is compatible with simple C++ code…

```
// hello.cpp
#include <iostream>

int main()
{   std::cout << "hello world" << std::endl;
    return 0;
}
```

TrapC compatibility when compiling C++ code is limited by TrapC being a much smaller programming language than C++. TrapC seeks compatibility with the restricted best practices code safety features of C++, most commonly encountered building safety-critical embedded systems. Note that the C++ example above is a bit misleading. The TrapC programming language doesn't support C++ namespaces. (The trapc compiler ignoring 'std::' when parsing a cpp file in C++ compatibility mode is a cheat.) Do not anticipate complicated C++ code to be so easily compiled as TrapC when migrating legacy code.

Memory Safety, No Buffer Overruns

Here is a classic example of dangerous C code that may buffer overrun.

[8] FreeBSD Gets €686,400 to Boost Security Features. https://www.darkreading.com/application-security/freebsd-gets-stf-funding-boost-security-features

```
// gets_input.h (CWE-242, CWE-120, CWE-77)
#include <stdio.h>

inline
void gets_input()
{   char buffer[24];
    printf("Please enter your name and press <Enter>\n");
    gets(buffer);// TrapC will terminate on overrun!
    printf("%s",buffer);
}
```

If the user inputs more data than the 24-byte buffer can hold, that is a typical C buffer overrun error. In C/C++, such an error is not implicitly caught, may crash or enable a hacker exploit. The action TrapC takes on overrun, or any other unanticipated error condition, such as divide-by-zero, is termination with a useful error message. That is, unless there is a programmer-supplied 'trap' error handler.

Cyber-security experts have long urged C programmers to replace using unchecked string-handling APIs with counterparts that specify string length, such as using strncpy() instead of strcpy(). That's unnecessary and even counterproductive with TrapC, where the compiler guards against buffer overruns.

The 'trap' in TrapC

```
// trap_test.tc
#include "gets_input.h"

int main()
{   gets_input();
    trap
    {   puts("ERROR: invalid input");
        return 1;
    }
    return 0;
}
```

The TrapC error-handling mechanism looks a bit like 'catch' from C++ exceptions, yet is quite different. With TrapC, quoting Yoda from Star Wars, "There is no try, only do or do not". Unlike C++ exceptions, TrapC has no alternate return type, nor an alternate return stack.

When an error is trapped in TrapC, the function returns immediately to the caller's 'trap' handler, if there is one. A normal return skips over the trap handler. TrapC has no alternate return stack like C++ exceptions. Faults must be trapped by the caller, do not get passed up the line like exceptions in C++. Aside from it just doesn't work that way, TrapC has no C++ exception stack to pass them up through. It is possible to rethrow in TrapC by using trap.return, somewhat like using 'throw' within a caught C++ exception.

What happens when a function gets trapped that must return a value? What does it return? It will return a block of memory that is sizeof the return type (and zeroed).

```
// c_or_trapc.c
int main()
{   bool ok = DoSomething();
#ifdef __trapc
    trap
#else
    if(!ok)
#endif
    {   puts("Didn't do somthing...");
        return 1;
    }
    return 0;
}
```

A key difference is what happens if the C programmer forgets to write the code that checks for ‘ok’ or the TrapC programmer forgets to ‘trap’ on DoSomething return.

```
// do_something.c
int main()
{   DoSomething();
    // Lacking trap handler, TrapC will terminate here on error, not return 0
    return 0;
}
```

If DoSomething() fails, perhaps due to attempting an overflow, TrapC will terminate the program printing an error message. The exit() code will be the errno of the error. The same code compiled in C will not report the error, will always return 0.

It is possible for a trap to be at a level higher, when it follows a compound function call…

```
#include <stdio.h>
#include <limits.h>

int increment(int x)
{   return ++x;
}

void print_x(int x)
{   puts(x);// same as printf("{}\n",x);
}   // see "TrapC Function Aliases" why this works

int main()
{   print_x(increment(INT_MAX));
    trap
    {   if(trap.errno == ERANGE)
```

```
        {   puts("Ignoring integer overflow, this time...");
        }
        else
        {   puts("Something strange happened!");
            trap.return;// Prints trap.msg, calls exit(errno)
    }   }
    return 0;
}
```

This trap is trapping both print and increment at the same time. The trap.return within a trap is conceptually like rethowing an exception in C++. Because the above trap.return is in main, it can only terminate. If called by some other function, the trap.return would return to that function’s trap handler. It is a compile-time error to have any other code between the semi-colon after a function call and its trap handler. The trap handler, if any, must immediately follow the function return being trapped.

There is no trap handler for main().

TrapC Function Aliases

```
int x = 10;
puts(x);// same as printf("{}\n",x);
```

How does puts() take an int? C code cannot accept an int where a string is expected, and to do the above in C++ would require an implicit conversion. The int x is somehow changing itself from an int into a string for printing. To do that in C++, one might reimplement puts() as an overloaded C++ function, like so…

```
void puts(const char* s);
void puts(int x);
```

C++ has name-mangling to decorate the two puts() function names, keep them separate. The linker will not see two functions with the same name. However, TrapC has no name-mangling. So how does TrapC make its puts() overload work? Using ‘alias’…

```
void puts(void* x) alias printf("{}\n",x);
```

A TrapC function alias works like a #define substitution. TrapC ignores free(). Has it’s own automatic memory management. An alias may define-away a function…

```
void *malloc(size_t size) alias void* new void[size/sizeof(void)];
void free(void* ptr) alias;// no-op, do nothing
```

Given this…

```
int* x = malloc(sizeof(int));
free(x);
```

With alias, the code the compiler sees is just…

```
int* x = new int[1];
```

In TrapC, unlike C++, malloc() and 'new' have the same memory pool.

Handling a C++ Exception vs. as a TrapC trap

```
// exception.cpp
#include <iostream>
#include <stdexcept>
using namespace std;

int divide_ints(int numerator,int denominator)
{   if(denominator == 0)
    {   throw runtime_error("Division by zero");
    }
    return numerator / denominator;
}

int main()
{   try
    {   int d = divide_ints(10,0);
        cout << "d = " << d << endl;
    }
    catch(const exception& e)
    {   cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

output:

```
Exception: Division by zero
```

Seme in TrapC…

```
# trap.tc
#include <stdio.h>

int divide_ints(int numerator,int denominator)
{   return numerator / denominator;
}

int main()
{   int d = divide_ints(10,0);
    trap
    {   puts(trap);
        // If we don't choose to continue...
        // return 1;
    }
    printf("d = {}\n",d);
    return 0;
}
```

output:

```
trap: msg = Divide by zero, errno = 1 (EPERM), function = divide_ints
d = 0
```

To allow ‘d’ be treated as 0 on divide-by-zero may, or may not be, a good design choice. After trapping a fault, it is up to the programmer to decide what’s next. Whatever data is returned by a trapped function will be zeroed. In the above, is the int zero.

Data returned via trap, e.g., trap.msg, is thread local.

TrapC ‘alias’ and Castplates

A danger in C is that containers, such as Glib GArray,[9] are not typesafe. Here is a growable TrapC array like GArray, but is typesafe and memory safe…

```
// array.tc
#include <string.h>
#include <type_info>

struct array
{   void* data alias array;// alias lets array[0] be the same as data[0]
    void* append(void x) alias += // let array+= be the same as array.append
    {   void* temp = data;// void acts like 'typename T' of a C++ template
        data = new void[countof(temp)+1];// countof elements, from type_info
        *data = *temp;// copy contents of temp, except for last element still
null
        *lastof(data) = x;// copy x into last element
        return array;// like return 'this' in C++, TrapC has no 'this'
}   }

int main()
{   array<int> a = new void[2];
    // Or, same as above using malloc:
    // array<int> a = malloc(2*sizeof(int));
    a[0] = 1;// like in C
    a[1] = 2;// like in C
    a += 3;// really a.append(3);
    printf("{}\n",a);// implicit for-each loop iterates over array
```

[9] Glib Data Structures. https://docs.gtk.org/glib/data-structures.html

```
    return 0;
}
```

output:

```
    1, 2, 3
```

Castplates make C containers typesafe. TrapC castplates are implicitly created. Any C struct that contains void* becomes a TrapC castplate. There is no ‘castplate’ keyword.

When instantiating a castplate in TrapC, the programmer must define what type it holds. This is a breaking change from C containers. TrapC requires specifying what void* really means, so the compiler will connect the right RTTI to it. TrapC will enforce that void* continues to be that type for the lifetime of the pointer.

As with ‘trap’, TrapC castplates are new syntax, not anything from C/C++. Using a castplate may mimic the syntax of a C++ template, but TrapC castplate containers are implemented quite differently from C++ template containers. To have castplates with the same C++ syntax, requires rewriting the templates. Like C++ provides the STL, TrapC provides the SCL, castplates that are workalikes for the C++ STL library, with iostream, string and container functionality. The SCL is mimimalist, not intended to do everything the STL can do. For example, the SCL is UTF-8, has no wide char type.

TrapC ‘alias’ overloading can do operator overloading or data overloading, or both at the same time. The alias keyword feels like a #define, in that it is a simple substitution, syntactic sugar, yet unlike #define is typesafe. In the code above, ‘data’ is overloaded. This allows the object ‘array’ to be operated upon the same as the array ‘data’. It would take much more code to do the same overloading in C++, to create a template that implements operator[] and operator+=().

Implementing a TrapC Castplate ‘vector’ to Mimic C++ Template ‘std::vector’

Castplates provide typesafe containers with an STL-like API without C++ templates. Using a TrapC castplate may look the same as using a template, but defining a castplate is quite different. Castplates cast pointers. Are much simpler than C++ templates. Because castplates do not replace types enthusiastically like C++ templates may, castplates may avoid the complexity and code bloat that often results with C++ templates.[10]

```
// vector.tc
#include <stdio.h>

struct vector
{   void* data alias vector;// let data stand in for vector, have the same
syntax
    vector(size_t size)
```

[10] On C++ Code Bloat https://dirtyhandscoding.github.io/posts/on-cpp-code-bloat.html

```
    {   data = new void[size];
    //  may use malloc instead: data = malloc(sizeof(void)*size);
    }
    size_t size() const
    {   return countof(data);
}   };

// vector.tc
int main()
{   vector<int> v(10);
    for(int i = 0;i < v.size();i++)
    {   v[i] = i;
    }
    printf("{} ",v);// implicit for-each loop over vector
    return 0;
}
```

output:

```
0 1 2 3 4 5 6 7 8 9
```

A limitation of castplates, and potential issue when porting code, is that castplates only support one type at a time. It's not easy to create a castplate workalike for C++ template std::map, that uses two types. Or, to deal with a C struct that contains a bunch of void* pointers that were created as different types.

Lifetime Safety

For C compatibility, it is ok to call free() in TrapC, but it is ignored.

```
// use_after_free.c

int main()
{   int* p = malloc(sizeof(int));
    free(p);//TrapC ignores free, p not freed yet
    *p = 10;//UB in C, no problem in TrapC
    return 0;// Leaving scope, TrapC now frees p automatically
}
```

It isn't possible to use-after-free in TrapC. Because the TrapC compiler determines when it frees memory, the free() call shown above does nothing. That free() and 'delete' are a no-op is for C/C++ compatibility, to enable legacy code to compile in TrapC.

Unlike C/C++, returning a pointer to a local variable is not an error in TrapC, whether heap or stack.

```
// local_return.c
```

```
int* foo(bool ok)
{   int* p = nullptr;
    int i = 0;
    if(ok)
    {   p = &i;
    }
    *p = 42; // TrapC will trap if p is null
    return p;// TrapC will safely return p, not rug-pull the memory of i
}
```

Inherent to the memory safety of TrapC is there is never garbage in a TrapC pointer. Because pointers are managed differently in TrapC than C, it is possible to have code that is wrong in C, yet right in TrapC, as above. This violates a longstanding industry practice of reproaching sloppy and lazy programmers, of having the compiler or 'lint' reject code to force the programmer to fix it.

How a compiler handles TrapC local memory corrections like the above is an implementation detail. Do not assume TrapC uses reference counting like in C++.

Typesafe printf() with "{}"

Using printf() in TrapC is typesafe. When using the TrapC "{}" specifier, typechecking is automatic. If using old school C printf type specifiers, such as "%d", the compiler checks it is right.

C:

```
printf("%d",var);// ok only if var is an int, programmer be careful
```

Rust:

```
print!("{}",var);
```

C++23:

```
std::print("{}",var);
```

TrapC:

```
printf("{}",var);
```

With "{}", TrapC chooses the right format specifier based on the type passed. That means that when compiling code that uses printf, whether the OS is 32-bit or 64-bit, there's no need to use the C PRI macros hack from inttypes.h. In TrapC, the C printf with "%d" as above is valid, provided that 'var' is an int. TrapC will reject printf calls that mismatch type. When possible to detect at compile-time, printf format specifier errors will not compile.

TrapC Pointer Intelligence: get_type_info() and the XXXof() RTTI API

TrapC pointers look just like C/C++ pointers, but behind-the-scenes TrapC manages pointers diligently. TrapC tracks Runtime Type Information (RTTI) for pointers created in TrapC, whether heap, stack or static.

```
// trapc_rtti.tc
#include <type_info>

int main()
{   int* p = new int[10];
    assert(p+9 == lastof(p));// last member of any array
    const type_info* t = get_type_info(p);
    printf("{}\n",t);// implicitly iterates over struct elements
    assert(testof(p));//confirms is TrapC pointer, not raw C pointer
    return 0;
}
```

output:

```
nameof = p, typeof = int*, sizeof = 80, countof = 10, offsetof = 0, lastof =
9, addressof = 00000235FB3A66C0
```

Except for sizeof() from C, none of the XXXof functions are TrapC language keywords. Are part of the TrapC type_info library API.

As with C++, a TrapC struct is implicitly a typename, doesn't require a typedef like in C. How the struct type_info is defined internally is a compiler implementation detail, conceptually like this…

```
struct type_info
{   const char* name;
    const char* type;
    size_t size;
    size_t count;
    size_t offset;
    size_t address;
    bool test;
};
```

Programmers may use the get_type pointer to print the whole thing, as above, or call any of the type_info helper functions…

```
const char* nameof(void* p);
const char* typeof(void* p);
size_t sizeof(void p);
size_t countof(void* p);
size_t offsetof(void* p);
size_t lastof(void* p);
size_t addressof(void* p);
bool testof(void* p);// returns false if the pointer failed test
```

In TrapC, attempting to manipulate a mystery pointer, that is, a pointer with no RTTI, will trap. Except for testof(), any attempt to use the pointer will also zero it.

Memory Management in TrapC vs. ANSI C vs. Proposed C ‘defer’

A worrying example that C programmers may easily get wrong is freeing the pointer that was manipulated by strsep() after it has been advanced to the next token. It takes due care to get this right in C.

```
// get_token_ansic.c
char* get_token()
{   char* p = strdup("123;...");//p pointer we will need to free...
    char* tokenp = strsep(&p,";");//strsep advances p, tokenp points to where 
p was
    assert(p != tokenp);//yes, p has changed!
    char* token = strdup(topkenp);//duplicate token
    //DO NOT DO THIS! free(p);
    free(tokenp);// correct!
    return token;
}
```

There’s a proposal to add ‘defer’ to C for automating clean-up code, sort of like ‘finally’ in Java. The ‘defer’ code is automatically executed at scope end, like a C++ destructor. However, the same thing that breaks careless programmers will break the same using ‘defer’.[11]

```
// get_token_defer.c
char* get_token_defer()
{   char* p = strdup("123;...");
    defer free(p);
    char* tokenp = strsep(&p,";");
    return strdup(tokenp);//CRASH! defer will free p, not tokenp
}
```

TrapC gets it right by freeing the original pointer, not the changed pointer.

```
// get_token_trapc.tc
char* get_token_trapc()
{   char* p = strdup("123;...");
    char* tokenp = strsep(&p,";");
    assert(p-offsetof(p) == tokenp);
```

[11] Alejandro Colomar (8 Nov 2024) ‘defer’ (n3199) concerns. https://inbox.sourceware.org/gcc/hag6ajqczmlhbl4le3yqepts7k37xmbffawv3f4prpcmf6ecm5@qzhzj3jcdjb6/T/

```
    return strdup(tokenp);//All good, TrapC will free tokenp, not p
}
```

TrapC Nulls Buffer Overrun Pointers

TrapC pointers always point to valid memory. Cannot overrun or contain garbage.

```
// trapc_ptr_nulled.c

int main()
{   const char* p = "Hello World";
    while(p)
    {   printf("%c",*p); // print one char at a time
        p++;  // When p steps off the end, TrapC nulls p
    }
    assert(p == 0);// true in TrapC, not true in C/C++
    return 0;
}
```

In C/C++, the above code is treacherous. Will keep going and segfault. Will not overrun in TrapC.

TrapC Z-strings Are also P-strings

```
// z_pstring.tc
#include <stdio.h>

int main()
{   char s[] = "Hello World";
    size_t before_len = strlen(s);
    s[before_len] = 's';// Overwriting z-string null terminator, not an
overflow
    puts(s);// Unterminated string ok, TrapC knows buffer size, UB if in C
    size_t after_len = strlen(s);// unterminated still ok, uses p-string size
    assert(after_len == before_len+1);
    return 0;
}
```

output:

```
    Hello Worlds
```

This example is not a recommendation to overwrite Z-string null terminators, merely pointing out it does no harm in TrapC. On a technicality, overwriting the terminator is not a buffer overrun. While overwriting s[before_len] does no harm in TrapC, overwriting s[before_len+1] would ‘trap’, because that would be trying to touch invalid memory past the null terminator. It’s feasible a TrapC compiler might warn or even trap on Z-buffer terminator overwrite, but maybe not useful.

TrapC strcat vs. C strcat

```
// strcat.c
char* strcat (char *restrict to, const char *restrict from)
{   strcpy (to + strlen(to), from);// DANGER! Could overrun and segfault
    return to;
}

// strcat.tc
#include <string>

char* strcat(char* to,const char* from)
{   string s = to;
    s += from; // realloc with copy-append if needed, no segfault
    return s; // returning local ok in TrapC, no segfault, no leak
}
```

If the realloc() is undesirable, due to performance, one solution is to make the string size longer in the first place. The string won't copy-append where append would do. Unlike C, it won't crash if append-without-expand would buffer overrun. That TrapC strings expand is a library design choice. As with C++, if you want strings to truncate, not expand, you can create your own fixed-length string object that works that way.

TrapC Object-Oriented printf and scanf, and the TrapC C++ Workalike Libraries

```
// printf_scanf.tc
#include <stdio.h>
#include <string>
#include <sstream>

struct Point
{  int x;
   int y;
   string printf()
   {  string s;
      s.append(x.printf());//TrapC built-in types have printf and scanf
overloads
      s.append(",");
      s << y;// same as s.append(y.printf());
      return s;
   }
   void scanf(string s)
   {  stringstream ss(s);//from <sstream>
      ss >> x;
      ss >> y;
}  };
```

```
int main()
{   Point p;
    string* s = p.printf();
    scanf("{}",&p);// same as p.scanf(s);
    printf("Point = ({}), was ({})\n",&p,s);
    return 0;
}
```

output:

```
Point = (10,42), was (0,0)
```

The scanf and other POSIX library calls in TrapC are trap-aware wrappers to the C functions. The string and sstream libraries are TrapC workalikes, not the C++ libraries of the same name.

No to C++ References

It's been reported that a compelling reason to add references to C++ was to make operator overloading syntax work. In C++, if returning string from operator+=, that would create a copy, not modify the original. Returning a C++ string pointer as 'this' would mess up the operator syntax. So, in C++ a reference *this is returned, having object syntax but pointer behavior.

C++11 introduced the rvalue reference, such as string&&, to enable move semantics, to improve performance when working with objects that manage dynamically allocated memory. That can be confusing. TrapC takes a different approach, that operator overloads and move semantics work witout lvalue or rvalue reference syntax.

Decimal Data Type and JSON

TrapC adds decimal fixed-point money type and support for JSON. The decimal format is implemented as an integer type, not a float type with a coefficient and exponent. JSON support is part of printf().

```
// decimal_json.tc
#include <stdio.h>
#include <stdlib.h>
#include <decimal>

int main()
{   FILE *fp;
    fp = fopen("example.txt", "rw");// TrapC may terminate or call handler
    decimal x = 1.5;// TrapC fixed-point decimal, not C/C++ float
    Point point1 = {50,100};//Create a point with x,y = 50,100
    fprintf(fp,"%j",x,&point1);//Output point1 to file in JSON format
    Point point2 = {0,0};// Create a point at the origin
    fseek(fp,0,SEEK_SET);// Rewind file
    fscanf(fp,"%j",x,&point2);//Read from file into point2
    assert(point2 == point1);
    printf("{}\n%j",x,&point2);//print decimal x and then point2 as JSON
    fclose(fp);// unnecessary, in TrapC fclose is RAII
}
```

output:

```
{ "type": "decimal",
  "x": 1.5
}
{ "type": "struct",
  "Point": "point2",
  { "x": ["int","50"],
    "y": ["int","100"]
} }
```

The fopen() in TrapC has the same API as the POSIX Linux API, but is trap-aware and RAII-safe. RAII design uses constructors and destructors to make data be consistently sane and safe. Being RAII-safe means the code was written such that it can handle having its RAII destructor called twice. That will happen in the above, due to the explicit call to fclose().

When FILE* is implicitly freed by TrapC, that is like ‘delete’ in C++. Will call the object’s destructor. Calling fclose() a second time does no harm here because the FILE destructor is RAII-safe. Inside our library, the FILE destructor calls fclose() a second time, but the handle was invalidated after being closed the first time, so skips it. Programmers are expected to make handles RAII-safe and leak-safe, typically by setting the handle to a known-bad state such as 0 or -1. RAII protection of programmer-defined handles is not automatic in TrapC, maybe in the future.

Localization

How to localize in C has often been implemented using gettext().[12]

```
// gettext.c
#include <stdio.h>
#include <stdlib.h>
#include <libintl.h>
#include <locale.h>
#define _(STRING) gettext(STRING)

int main()
{   /* Setting the i18n environment */
    setlocale (LC_ALL,"");
    bindtextdomain("hello","/usr/share/locale/");
```

[12] Emmanuel Fleury. A Quick Gettext Tutorial. https://www.labri.fr/perso/fleury/posts/programming/a-quick-gettext-tutorial.html

```
    textdomain("hello");
    printf(_("Hello World\n"));
    return 0;
}
```

Same in TrapC…

```
// gettext.tc
#include <stdio.h>

int main()
{   printf("{}\n","Hello World");// _() is implicit in TrapC 'printf'
    //for C compatibility, same with printf("%s\n",_("Hello World"));
    return 0;
}
```

output:

```
$> ./hello
Hello World
$> LANG=fr_FR ./hello
Bonjour le monde
```

Changing the locale to zh_CN causes string s to be translated to Chinese.

```
// local_set.tc
#include <stdio.h>
#include <locale.h>

int main()
{   printf("{}\n","Hello World");// Hello World
    setlocale(LANG,"zh_CN");// Switch to Chinese
    printf("{}\n","Hello World");// implicit translation
    string s = "Goodbye";// implicit translation
    puts(s);
    return 0;
}
```

output:

```
你好世界
再见
```

TrapC translation adds little or no overhead to strings that are not translated. TrapC localization may look like gettext, but how TrapC translation is implemented is a compiler implementation detail. Translations might be from a local dictionary, be captured ad hoc from user-provided input or be AI-generated across the Internet.

Code Example in Rust, C++ and TrapC

Just for comparison, finding the max element of an array.

In Rust…

```
// max_element.rs
fn main
{   let v = vec![1,2,3,4,5];
    let max = v.iter().max();
    match max
    {   Some(&max) => println!("Max element = {}",max),
        None => println!("The vector is empty"),
}   }
```

In C++…

```
// max_element.cpp
int main()
{   std::vector<int> v = {1,2,3,4,5};
    auto max = std::max_element(v.begin(),v.end());// DANGER! UB if v empty
    std::print("Max element = {}\n",max);
    return 0;
}
```

In TrapC…

```
// max_element.tc
int main()
{   vector<int> v = {1,2,3,4,5};
    int* max = max_element(v);// Will trap if vector empty
    trap
    {   puts(trap.msg);
        return 1;
    }
    printf("Max element = {}\n",*max);
    return 0;
}
```

Cmaker Build Manager vs. Rust Cargo

C/C++ does not have a standard build toolchain that is part of the language itself. Widely used build systems for C/C++ projects include CMake, UNIX ‘make’, UNIX autotools, Ninja, Meson and Microsoft MSBuild. Lazy and newbie programmers often embrace the easy to use, but proprietary, graphical build systems included with Microsoft Visual Studio and Apple Xcode. This causes trouble later when porting code across operating systems. Among portable toolchains, cmake is the most popular, but difficult to master. Cmake code looks nothing like C.

Rust takes a much more sane approach, a toolchain called Cargo.[13]

Rust Cargo:

```
$ cargo new hello_cargo
$ cd hello_cargo

$ cargo build
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs

$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

Rust generates a build file, named Cargo.toml.

It is vital to TrapC to have a portable, easy to use, toolchain like Rust, so programmers may quickly and easily create build systems that work across operating systems.

Cmaker[14] is a toolchain that generates cmake build files. When creating a C++ build system, Cmaker generates code for the cmake files and boilerplate code for classes, programs and unit tests. The author originally created Cmaker to generate the cmake build system for CinePaint, an open source graphics software project that he leads.[15] Cmaker can create cmake builds for new projects or retroactively for existing projects that may lack a build system. Like Cargo, Cmaker uses a simple command line syntax.

To generate cmake build system for legacy C/C++ project:

```
$ cd directory_of_existing_project
$ cmaker retro NameOfProject
$ mkdir build;cd build
$ cmake ..
```

For a new C++ project:

```
$ mkdir new_project
$ cd new_project
$ echo "Open Source MIT" > LICENSE
$ export AUTHOR="My Name"
$ cmaker project HelloWorld
$ cmaker class Hello World
```

[13] Why Cargo Exists. https://doc.rust-lang.org/cargo/guide/why-cargo-exists.html

[14] Cmaker. https://gitlab.com/robinrowe/cmaker

[15] CinePaint. https://gitlab.com/robinrowe/cinepaint

```
$ cmaker program hello_world
$ mkdir build;cd build
$ cmake ..
$ cmaker run
Hello World!
```

Cmaker supports C++ currently. Support for C and TrapC is coming.

C ABI Compatibility

Passing pointers from TrapC into a C function is fine. Go ahead and call the C library, but if that C library crashes or gets exploited that's not something TrapC can prevent. Dropping into C, TrapC cannot track it. Calling into C is like using 'unsafe' in Rust, or maybe even less secure than that. All the caveats of using C apply. If you cannot forego unions or goto, this is where you may go.

Passing raw C pointers to TrapC functions is not ok. There are no "mystery pointers" in TrapC. A TrapC pointer may be passed as-is to a C API, because that transition ignores hidden TrapC pointer RTTI. Attempting the reverse would have TrapC missing necessary pointer RTTI.

A solution is to pass TrapC pointers into C functions:

1. Create a TrapC wrapper function with the same name as the C function you wish to hide
2. In your TrapC wrapper, create a suitable pointer
3. Call the extern "C" function passing in the TrapC pointer
4. If something goes wrong, such as the C function returns false or errno, trap.return
5. Upon success, return TrapC pointer
6. Hide the C function so only the TrapC version can be called by TrapC application code

Although the syntax looks the same as creating a pointer in C, the pointer created in TrapC will have the hidden RTTI that TrapC needs.

Reddit r/trapc

- https://www.reddit.com/r/trapc/

Article in The Register

Note that November interview in The Register shared design ideas at that time. TrapC is still evolving. Based on reader feedback, malloc support since added.

- Thomas Claburn (12 Nov 2024). To kill memory safety bugs in C code, try the TrapC fork. The Register. https://www.theregister.com/2024/11/12/trapc_memory_safe_fork/