

**ACCU
2021**
VIRTUAL EVENT

Bloomberg
Engineering

undo

 **mosaic**
CONSULTANTS TO FINANCIAL SERVICES

Modern C and What We Can Learn From It

Luca Sas



Modern C?

Modern C?

- what if there is more to C than meets the eye?



ODIN



Modern C?

- what if there is more to C than meets the eye?

Modern C?

- what if there is more to C than meets the eye?
- present a modern way in which C can be used and address misconceptions

Modern C?

- what if there is more to C than meets the eye?
- present a modern way in which C can be used and address misconceptions
- explore how ideas inspired from C can improve our C++ code

Modern C?

- what if there is more to C than meets the eye?
- present a modern way in which C can be used and address misconceptions
- explore how ideas inspired from C can improve our C++ code
- showcase how these ideas manifest in other languages, old and new

About me

Core Systems Engineer @ Creative Assembly

Game Development

Low Level Systems

Programming Language



BananyaDev#9587



@SasLuca



sas.luca.alex@gmail.com

A refresh on C

A refresh on C

- Developed in 1972 (49 years ago)

SECOND EDITION

THE



PROGRAMMING
LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

PRENTICE HALL SOFTWARE SERIES

```
1 square (x, n)
2 int x, n;
3 {
4     int i, p;
5
6     p = 1;
7     for (i = 1; i <= n; ++i) p = p * x;
8
9     return (p);
10 }
```

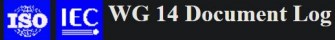
A refresh on C

- Developed in 1972 (49 years ago)
- Standardized by ANSI and later by ISO (ANSI C, C89, C99, C11, ...)

A refresh on C

- Developed in 1972 (49 years ago)
- Standardized by ANSI and later by ISO (ANSI C, C89, C99, C11, ...)
- C does still have an active ISO committee (WG14)

A refresh on C



Last Update: 2021/03/10

- [N2683](#) 2021/03/10 Svoboda, Towards Integer Safety (updates N 2681)
- [N2682](#) 2021/03/10 Ojeda, secure_clear (updates N 2631)
- [N2681](#) 2021/03/10 Svoboda, Towards Integer Safety (updates N 2669)
- [N2680](#) 2021/03/09 Seacord, Specific-width length modifier (updates N 2623)
- [N2678](#) 2021/03/04 Keaton, Version 3 Agenda for March, 2021
- [N2677](#) 2021/02/28 Keaton, Revised Agenda for March, 2021
- [N2675](#) 2021/03/07 Gustedt, simple lambdas v2
- [N2674](#) 2021/03/07 Gustedt, type inference for variable definitions and function returns v2
- [N2673](#) 2021/03/07 Ballman, __has_include for C
- [N2672](#) 2021/02/27 Thomas, C23 proposal - 5.2.4.2.2 cleanup
- [N2671](#) 2021/02/27 Thomas, C23 proposal - negative values
- [N2670](#) 2021/02/27 Thomas, C23 proposal - zeros compare equal
- [N2669](#) 2021/02/27 Svoboda, Towards Integer Safety
- [N2668](#) 2021/02/28 Uecker, Indeterminate Values and Trap Representations
- [N2667](#) 2021/02/27 Gustedt, Introduce the nullptr constant v.2
- [N2666](#) 2021/02/20 Keaton, Agenda for March, 2021
- [N2665](#) 2021/02/21 Seacord, Zero-size reallocations no longer obsolescent feature
- [N2664](#) 2021/02/21 Ballman, Feb 2021 C/C++ compat teleconference minutes
- [N2663](#) 2021/02/13 Uecker, life time, blocks, and labels
- [N2662](#) 2021/02/13 Uecker, maybe_unused attribute for labels
- [N2661](#) 2021/02/13 Uecker, nested functions
- [N2660](#) 2021/02/13 Uecker, improved bounds checking for array types
- [N2659](#) 2021/02/13 Ojeda, Safety attributes for C
- [N2657](#) 2021/02/13 Múgica, Outer
- [N2656](#) 2021/02/03 Ballman, C and C++ Compatibility Study Group Omnibus of WG21 Papers (Feb 2021)
- [N2655](#) 2021/01/30 Gustedt, Make false and true first-class language features v4
- [N2654](#) 2021/01/30 Gustedt, Revise spelling of keywords v5
- [N2652](#) 2021/01/30 Thomas, TS 18661-5 revision
- [N2651](#) 2021/01/30 Thomas, C2X proposal - fabs and copysign cleanup
- [N2650](#) 2021/01/30 Thomas, C2X proposal - signbit cleanup
- [N2649](#) 2021/01/30 Thomas, February 2021 CFP teleconference agenda
- [N2648](#) 2021/01/30 Thomas, January 2020 CFP teleconference minutes
- [N2647](#) 2021/01/30 Gustedt, Add new optional time bases v4
- [N2646](#) 2021/01/30 Ballman, Adding a Fundamental Type for N-bit integers (updates N2590)
- [N2645](#) 2021/01/30 Blower, Add support for preprocessing directives elifdef and elifndef
- [N2644](#) 2021/01/30 Gustedt, a common C/C++ core specification v. 3
- [N2643](#) 2021/01/30 Tydeman, Negative
- [N2642](#) 2021/01/30 Tydeman, Quantum exponent of NaN
- [N2641](#) 2021/01/30 Tydeman, Missing +(x) in table
- [N2640](#) 2021/01/30 Tydeman, Missing DEC_EVAL_METHOD

A refresh on C

- Developed in 1972 (49 years ago)
- Standardized by ANSI and later by ISO (ANSI C, C89, C99, C11, ...)
- C does still have an active ISO committee (WG14)
- What have we missed in the past 50 years and how is C different from C++?

Comments

```
1 /*  
2 Original C comments  
3 */  
4  
5 // Added
```

Variables and structs

```
1 struct cat { ... };
2
3 void best_cats(void)
4 {
5     struct cat marshmallow, milo, bishop;
6     int i;
7     for (i = 0; i < 89; i++) ...
8 }
```

Variables and structs

```
1 typedef struct cat { ... } cat;
2
3 void best_cats(void)
4 {
5     for (int i = 0; i < 99; i++) ...
6     cat marshmallow = {0};
7     cat milo = {0};
8     cat bishop = {0};
9 }
```

Primitive types

```
1 unsigned char int short long float double
```

Primitive types

```
1 unsigned char int short long float double
2
3 #include <stdint.h>
4 uint8_t uint16_t uint32_t uint64_t
5 int8_t int16_t int32_t int64_t
```

Primitive types

```
1 unsigned char int short long float double
2
3 #include <stdint.h>
4 uint8_t uint16_t uint32_t uint64_t
5 int8_t int16_t int32_t int64_t
6
7 // minimum width int types
8 int_least8_t
9 uint_least8_t
10
11 // fastest minimum-width int types
12 int_fast8_t
13 uint_fast8_t
```

Functions

```
1 main(argc, args)
2 int argc;
3 const char** args;
4 {
5
6 }
7
8 int main(int argc, const char** args)
9 {
10
11 }
```


Functions

```
1 void foo();
```

```
2
```

```
3 void foo(void);
```

C++ is not C

- C++ is not fully compatible with C

C++ is not C

- C++ is not fully compatible with C
- They are compatible enough that C headers will mostly work in C++

C++ is not C

- C++ is not fully compatible with C
- They are compatible enough that C headers will mostly work in C++
- C++ did inherit a lot of baggage from this attempted compatibility with C

**What else is different
between C and C++?**

Struct initialization

Struct initialization

- Unlike C++, structs in C are just plain old bags of data (PODs)

Struct initialization

- Unlike C++, structs in C are just plain old bags of data (PODs)
- C99 introduces a new and powerful form of struct initialization

Struct initialization

```
1 typedef struct v2 { float x, y; } v2;  
2  
3 v2 v = { 1.0f, 1.0f };  
4
```

Struct initialization

```
1 typedef struct v2 { float x, y; } v2;  
2  
3 v2 v = { .x = 1.0f, .y = 1.0f };
```

Struct initialization

```
1 typedef struct v2 { float x, y; } v2;  
2  
3 v2 v = { .x = 1.0f }; // y is implicitly set to 0
```

Struct initialization

```
1 typedef struct v2 { float x, y; } v2;  
2  
3 v2 v = { .x = 1.0f }; // y is implicitly set to 0  
4  
5 v = { .y = 1.0f } // ERROR!!!
```

Struct initialization

```
1 typedef struct v2 { float x, y; } v2;  
2  
3 v2 v = { .x = 1.0f }; // y is implicitly set to 0  
4  
5 v = (v2){ .y = 1.0f } // Good!
```

Struct initialization

```
1 typedef struct v2 { float x, y; } v2;  
2  
3 // wont work in C++  
4 #define some_constant (v2) { 1, 2 }
```

Struct initialization

```
1 typedef struct v2 { float x, y; } v2;
2
3 #ifdef __cplusplus
4 #define literal(T) T
5 #else
6 #define literal(T) (T)
7 #endif
8
9 // Will work in C++
10 #define some_constant literal(v2) { 1, 2 }
```

Struct initialization

```
1 typedef struct bar { foo f; } bar;  
2  
3 bar b = {  
4     .f.some_member_of_foo = ...  
5 }
```


Struct initialization

```
1 typedef struct bar { foo f; } bar;  
2  
3 bar b = {  
4     .f = {  
5         ...  
6     }  
7 }
```

Struct initialization

```
1 typedef struct bar { foo f[COUNT]; } bar;
2
3 bar b = {
4     .f[0].some_member = ...
5     .f[1] = {
6         ...
7     }
8 }
```

Struct initialization

```
1 typedef struct bar { foo f[COUNT]; } bar;
2
3 bar b = {
4     .f = {
5         [0].some_member = ...
6         [1] = {
7             ...
8         }
9     }
10 }
```

Struct initialization

- Unlike C++, structs in C are just plain old bags of data (PODs)
- C99 introduces a new and powerful form of struct initialization
- We can easily initialize complex data structures with nested initializers

Struct initialization

- Unlike C++, structs in C are just plain old bags of data (PODs)
- C99 introduces a new and powerful form of struct initialization
- We can easily initialize complex data structures with nested initializers
- Everything we don't explicitly initialize gets set to 0 (ZII)

Struct initialization

- Unlike C++, structs in C are just plain old bags of data (PODs)
- C99 introduces a new and powerful form of struct initialization
- We can easily initialize complex data structures with nested initializers
- Everything we don't explicitly initialize gets set to 0 (ZII)
- This is one of the features that lead to Modern C

Struct initialization: sokol gfx

```
1 sg_pipeline_desc pip_desc = {
2     .layout = {
3         .buffers[0].stride = 28,
4         .attrs = {
5             [ATTR_vs_position].format = SG_VERTEXFORMAT_FLOAT3,
6             [ATTR_vs_color0].format  = SG_VERTEXFORMAT_FLOAT4
7         }
8     },
9     .shader = shd,
10    .index_type = SG_INDEXTYPE_UINT16,
11    .depth_stencil = {
12        .depth_compare_func = SG_COMPAREFUNC_LESS_EQUAL,
13        .depth_write_enabled = true,
14    },
15    .rasterizer.cull_mode = SG_CULLMODE_BACK,
16    .rasterizer.sample_count = SAMPLE_COUNT,
17    .label = "cube-pipeline"
18 };
```

Modern C

- Challenging the way APIs have historically been structured in C
- Using newer features of C in order to improve the ease of use and safety of the API
- Decoupling the data manipulations from hosted services (allocators, io)
- Centralizing resource management (custom allocators, system-wide resource managers)
- Let's explore some other features and then more examples

C11 Additions

- The latest major version

C11 Additions

- The latest major version
- As of recent it is now supported by all major compilers including MSVC

C11 Additions: static assert

```
1  #include <assert.h>
2  int main(void)
3  {
4      // Test if math works.
5      static_assert(2 + 2 == 4, "Whoa dude!"); // or _Static_assert(...
6
7      // This will produce an error at compile time.
8      _Static_assert(sizeof(int) < sizeof(char),
9                      "this program requires that int is less than char");
10 }
```

C11 Additions: _Generic and overloading

```
1 float minf(float, float);
2 int mini(int, int);
3
4 #define min(a, b) _Generic((a), float: minf(a, b), int: mini(a, b))
```

C11 Additions

- The latest major version
- As of recent it is now supported by all major compilers including MSVC
- `static_assert`
- overloading
- atomics (available in older versions with libraries like `c89atomics`)
- `thread_local` (available via compiler extensions in C99)

Awesome Macros

Awesome Macros

- Macros CAN be evil!

Awesome Macros

- Macros CAN be evil!
- There are some cases in which they can make our life much nicer.

Awesome Macros: defer

```
4 begin();  
5  
6 ...  
7  
8 end();
```

Awesome Macros: defer

```
1  #define macro_var(name) concat(name, __LINE__)
2  #define defer(start, end) for (      \
3      int macro_var(_i_) = (start, 0); \
4      !macro_var(_i_);                 \
5      (macro_var(_i_) += 1), end)      \
6
7  defer(begin(), end())
8  {
9      ...
10 }
```

Awesome Macros: defer

```
1  #define profile defer(profile_begin(), profile_end())
2  profile
3  {
4  |    ...
5  }
```

Awesome Macros: defer

```
1  #define profile defer(profile_begin(), profile_end())
2  profile
3  {
4  |    ...
5  }
```

```
1  #define gui defer(gui_begin(), gui_end())
2  gui
3  {
4  |    ...
5  }
```

Awesome Macros: defer

```
1 #define profile defer(profile_begin(), profile_end())
2 profile
3 {
4     ...
5 }
```

```
1 #define gui defer(gui_begin(), gui_end())
2 gui
3 {
4     ...
5 }
```

```
2 file_handle_t file = file_open(filename, file_mode_read);
3 scope(file_close(file))
4 {
5     ...
6 }
```

Awesome Macros: defer

```
2  file_handle_t file = file_open(filename, file_mode_read);  
3  scope(file_close(file))  
4  {  
5  |    ...  
6  }
```



API Design

API Design: Math

```
1 struct vec2 { float x, y; };
2
3 void vec2_add(const struct vec2* a, const struct vec2* b, struct vec2* out)
4 {
5     out->x = a->x + b->x;
6     out->y = a->y + b->y;
7 }
```


API Design: Math

```
1 struct vec2 { float x, y; };
2
3 void vec2_add(const struct vec2* a, const struct vec2* b, struct vec2* out)
4 {
5     out->x = a->x + b->x;
6     out->y = a->y + b->y;
7 }
```

- Out params are harder to spot at the call site

API Design: Math

```
1 struct vec2 { float x, y; };  
2  
3 void vec2_add(const struct vec2* a, const struct vec2* b, struct vec2* out)  
4 {  
5     out->x = a->x + b->x;  
6     out->y = a->y + b->y;  
7 }
```

- Out params are harder to spot at the call site
- There are indirection and aliasing issues caused by the pointers which can harm performance

API Design: Math in Modern C

```
1  typedef struct vec2 { float x, y; } vec2;
2
3  vec2 vec2_add(vec2 a, vec2 b)
4  {
5      vec2 result = { a.x + b.x, a.y + b.y };
6      return result;
7  }
8
9  vec2 v = vec2_add(a, (vec2){ ... })
```

- value oriented design

API Design: Math in Modern C

```
1  typedef struct vec2 { float x, y; } vec2;
2
3  vec2 vec2_add(vec2 a, vec2 b)
4  {
5      vec2 result = { a.x + b.x, a.y + b.y };
6      return result;
7  }
8
9  vec2 v = vec2_add(a, (vec2){ ... })
```

- value oriented design
- we can use literals in the function call and readability is improved

API Design: Math in Modern C

```
1  typedef struct vec2 { float x, y; } vec2;
2
3  vec2 vec2_add(vec2 a, vec2 b)
4  {
5      vec2 result = { a.x + b.x, a.y + b.y };
6      return result;
7  }
8
9  vec2 v = vec2_add(a, (vec2){ ... })
```

- value oriented design
- we can use literals in the function call and readability is improved
- performance is actually better

API Design: Math in Modern C

```
1 typedef union hmm_vec2
2 {
3     struct { float X, Y; };
4     struct { float U, V; };
5     struct { float Left, Right; };
6     struct { float Width, Height; };
7     float Elements[2];
8 } hmm_vec2;
```

API Design: Error handling

API Design: Error handling

```
1 error_code_t do_something(some_arg_t args, out_t* out);
```


API Design: Error handling

```
3 void do_stuff()  
4 {  
5     error_code err;  
6  
7     err = do_task1(...)  
8     if (err)  
9     {  
10        ...  
11    }  
12  
13    err = do_task2(...)  
14    if (err)  
15    {  
16        ...  
17    }  
18  
19    err = do_task3(...)  
20    if (err)  
21    {  
22        ...  
23    }  
24 }
```

API Design: Error handling

```
3 void do_stuff()  
4 {  
5     error_code_t err;  
6  
7     err = do_task1(...)  
8     if (err) goto error;  
9  
10    err = do_task2(...)  
11    if (err) goto error;  
12  
13    err = do_task3(...)  
14    if (err) goto error;  
15  
16    return;  
17  
18    error:  
19    ...  
20 }
```

API Design: Error handling in Modern C

```
1 typedef struct file_contents_t
2 {
3     char* data;
4     isize_t size;
5     valid_t valid;
6 } file_contents_t;
7
8 file_contents_t read_file_contents(const char*);
```

Error handling in Modern C

```
1 file_contents_t fc = read_file_contents("milo.cat");
2 if (fc.valid)
3 {
4     ...
5 }
```

Error handling in Modern C

```
1  file_contents_t fc = read_file_contents("milo.cat");
2  image_t img = load_image_from_file_contents(fc);
3  texture_t texture = load_texture_from_image(img);
4
5  if (texture.valid)
6  {
7      ...
8  }
9
```

Error handling in Modern C

```
1 typedef struct file_contents_t
2 {
3     char* data;
4     isize_t size;
5     valid_t valid;
6 } file_contents_t;
7
8 file_contents_t read_file_contents(const char*);
```

```
8 std::optional<file_contents> fc;
```

Error handling in C++ with optional

```
std::optional<image> get_cute_cat (const image& img) {  
    return crop_to_cat(img)  
        .and_then(add_bow_tie)  
        .and_then(make_eyes_sparkle)  
        .map(make_smaller)  
        .map(add_rainbow);  
}
```

Error handling in Modern C

```
8  image_t get_cute_cat(image_t img)
9  {
10     img = crop_to_cat(img);
11     img = add_bow_tie(img);
12     img = make_eyes_sparkle(img);
13     img = make_smaller(img);
14     img = add_rainbow(img);
15     return img;
16 }
17
18 image_t img = get_cute_cat(some_cat_image);
19 if (img.valid)
20 {
21     ...
22 }
```


Error handling in Modern C

```
1 typedef struct file_contents_t
2 {
3     char* data;
4     isize_t size;
5     error_code_t error_code;
6 } file_contents_t;
```

Error handling in Modern C

```
1 typedef struct file_contents_t
2 {
3     char* data;
4     isize_t size;
5     error_code_t error_code;
6 } file_contents_t;
```

Error handling: std::expected

```
8 std::expected<file_contents_t, error_code> read_file_contents(const char*);
```

Error handling in Rust

```
use std::fs::File;
use std::io;
use std::io::Read;

fn read_username_from_file() -> Result<String, io::Error> {
    let f = File::open("hello.txt");

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(e) => Err(e),
    }
}
```



Error handling in Zig

```
1 fn doAThing(str: []u8) !void {
2     const number = parseU64(str, 10) catch |err| {
3         //...
4     };
5
6     // ...
7 }
8
9 fn doAThing(str: []u8) !void {
10    const number = try parseU64(str, 10);
11    // ...
12 }
```



Error handling in C++



KEYNOTE: De-fragmenting C++: Making exceptions more affordable and usable - Herb Sutter [ACCU 2019]

14K views • 1 year ago



ACCU Conference

CPP #ACCUConf #exceptions Error handling has fractured the C++ community into incompatible dialects, because of ...

Generic APIs in C

- C lacks generic programming (eg: templates)

Generic APIs in C

- C lacks generic programming (eg: templates).
- C can also force us to think outside the box and not rely excessively on templates.

Generic APIs in C

- C lacks generic programming (eg: templates).
- C can also force us to think outside the box and not rely excessively on templates.
- There do exist some solutions for some very common use cases.

Generic APIs in C: Dynamic Arrays

```
1 // C++ dynamic array
2 std::vector<int> v;
3 v.push_back(1);
```

Generic APIs in C: Dynamic Arrays

```
1 // C++ dynamic array
2 std::vector<int> v;
3 v.push_back(1);
```

```
2 // Modern C dynamic array
3 #define dynarray(T) T*
```

Generic APIs in C: Dynamic Arrays

```
1 // C++ dynamic array
2 std::vector<int> v;
3 v.push_back(1);
```

```
2 // Modern C dynamic array
3 #define dynarray(T) T*
```

```
5 typedef struct dynarray_info
6 {
7     isize_t size;
8     isize_t capacity;
9     isize_t element_size;
10 } dynarray_info;
```

Generic APIs in C: Dynamic Arrays

```
2 // Modern C dynamic array
3 #define dynarray(T) T*
```

```
5 typedef struct dynarray_info
6 {
7     isize_t size;
8     isize_t capacity;
9     isize_t element_size;
10 } dynarray_info;
```

```
12 #define dynarray_add(arr, ...) dynarray_ensure_capacity(arr); (*arr)[dynarray_size(*arr)] = __VA_ARGS__
13
14 dynarray(int) arr = dynarray_init(int, 10);
15 dynarray_add(&arr, 99);
```

Generic APIs in C: Map

```
2 typedef struct kv
3 {
4     const char* key;
5     value_t value;
6 } kv;
7
8 #define hash_map(KV) KV*
```

https://github.com/nothings/stb/blob/master/stb_ds.h

Libraries in C

- Historically messy due to the multitude of build systems and no standard.

Libraries in C

- Historically messy due to the multitude of build systems and no standard.
- Single header libraries arise as a convenient way to solve this problem.

Libraries in C

- Historically messy due to the multitude of build systems and no standard.
- Single header libraries arise as a convenient way to solve this problem.

```
1 #define LIB_IMPLEMENTATION
2 #include "lib.h"
```

Libraries in C

- Historically messy due to the multitude of build systems and no standard.
- Single header libraries arise as a convenient way to solve this problem.

```
1 #define LIB_IMPLEMENTATION
2 #include "lib.h"
```

- Sane CMake is another good option. Single headers can optionally be auto-generated.

Great talk on Cmake

cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON

About this talk

- Modular design
- Build systems (CMake in particular)
- ... and how to combine that to improve your codebase



Modern CMake
for modular design

CppCon.org

CppCon 2017: Mathieu Ropert "Using Modern CMake Patterns to Enforce a Good Modular Design"

53,495 views • Oct 13, 2017

819 25 SHARE SAVE ...

All

From your search

C++

Angular

Rel >

CppCon 2019: Chandler Carruth
"There Are No Zero-Point"

Libraries in newer languages

- Newer langs improve significantly on the library situation.
- Rust has a very well received easy to use build system and centralized database for libraries.
- Zig takes a unique approach by having the build system be part of the language.



When writing libraries

- Avoid allocations if possible, request allocators or buffers from the user.

When writing libraries

- Avoid allocations if possible, request allocators or buffers from the user.
- Try and make your library freehosted.

When writing libraries

- Avoid allocations if possible, request allocators or buffers from the user.
- Try and make your library freehosted.

```
2 typedef struct allocator_t
3 {
4     void* user_data;
5     void* (*proc)(allocator_t* this_allocator, isize_t amount_to_alloc, void* ptr_to_free);
6 } allocator_t;
```

Memory management

- Consider centralizing allocations.
- Differentiate between temporary and long lived allocations.
- Use buffers with maximum sizes where possible.
- Consider handles instead of pointers (eg: ECS)

Temporary allocators

```
1
2  allocator_t temp_allocator = make_allocator(arena);
3
4  while (game_is_running)
5  {
6      ...
7
8      dynarr(string) strings = get_strings(temp_allocator);
9
10     ...
11     free_temp_allocator(temp_allocator);
12 }
```

API Design: Modern C

- More declarative and functional, less stateful
- More value oriented and less pointers
- Zero initialization is used heavily (ZII)
- Uses C99 and C11 features and macros to modernize the code
- Centralized resource management
- Allocator aware

String handling in C

- Strings in C have been a historical disaster

String handling in C

- Strings in C have been a historical disaster
- Terrible standard API (strstr, strtok, strpbrk)

Public service announcement: avoid libc

- Very old
- Slow
- Terrible API design
- Very few parts are actually useful (stdint.h, memmove/memcpy/memset, math.h)

Replacing libc functionality: printf

```
1  typedef struct cat { ... } cat;
2  void print_cat(cat*);
3
4  logger_register_printer("cat", print_cat);
5
6  cat c = ...;
7  log("Cat: {cat}", c);
```

String handling in C

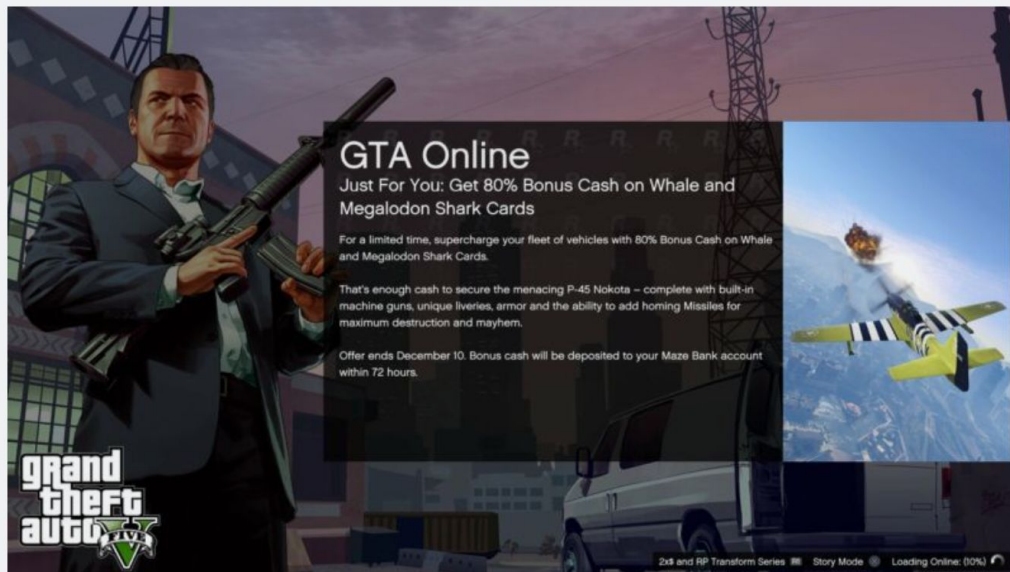
- Strings in C have been a historical disaster
- Terrible standard API (strstr, strtok, strpbrk)
- Null terminated strings are very slow

LOAD IT UP—

Hacker reduces *GTA Online* load times by roughly 70 percent

Homebrewed DLL solves inefficient parsing of in-game shop files.

KYLE ORLAND - 3/1/2021, 6:32 PM



[Enlarge](#) / You could spend less time looking at loading screens like this with a new DLL fix.

A hacker going by the handle T0st says he has [figured out a core issue](#) that caused longer-than-necessary load times in *Grand*

Saving time with disassembly

To get to the bottom of the problem, T0st **writes** that they started by profiling their own CPU to try to figure out why the game was maxing out a single CPU thread for over four minutes during loading. After using a tool to dump the **process stack** and disassembling the *GTA* code as it was running in memory, T0st noticed a set of (somewhat obfuscated) functions that seemed to be parsing a 10MB JSON file with over 63,000 total entries.

The JSON file in question appeared to be the "net shop catalog" that describes every single item *GTA Online* players can purchase with in-game currency. Parsing a 10MB file shouldn't be *too* much of a problem for a modern computer, but a few obscure problems in the specific implementation seem to lead to massive slowdowns.

For one, the specific function used to parse the JSON string (seemingly `sscanf`, in this case) was apparently running a time-intensive `strlen` checking function repeatedly after the read for *every single piece of data*. Simply caching that string length value to speed up those checks resulted in an over 50 percent reduction in load times on its own, T0st writes.

After parsing all this JSON data, *GTA Online* seems to load it into an array in an extremely inefficient way, checking the entire array for duplicates from scratch as it grows. Replacing that process with a **hash table** that can quickly check for duplicates led to a roughly 25 percent load time reduction on its own, T0st writes.

It's not just null terminated strings

Case study: std::string

```
void read_string(const std::string&)
```

Intent: reading an array of bytes representing text

Case study: std::string

```
read_string("Test");
```

Case study: std::string

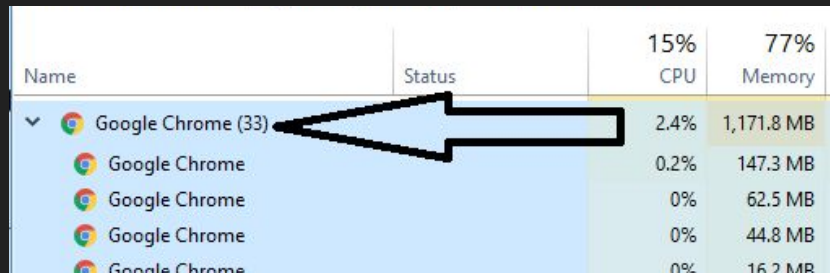
```
read_string("Test");
```

This causes a memory allocation.

What's the problem with a few couple of temporary strings here and there?

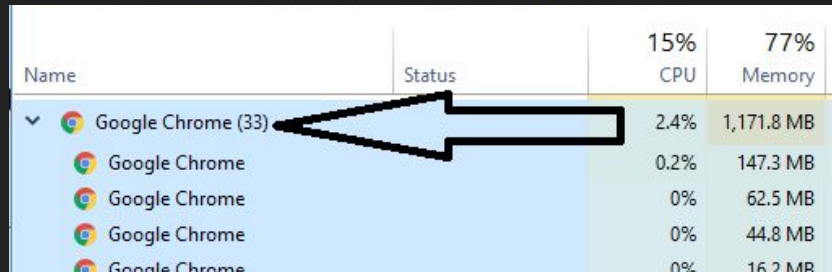
Real life examples: Chrome

- Half of all allocations are `std::string`.



| Name | Status | 15% CPU | 77% Memory |
|--------------------|--------|------------|---------------|
| Google Chrome (33) | | 2.4% | 1,171.8 MB |
| Google Chrome | | 0.2% | 147.3 MB |
| Google Chrome | | 0% | 62.5 MB |
| Google Chrome | | 0% | 44.8 MB |
| Google Chrome | | 0% | 16.2 MB |

Real life examples: Chrome



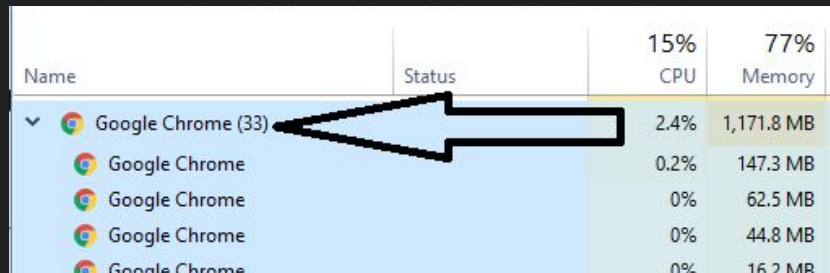
| Name | Status | 15% CPU | 77% Memory |
|--------------------|--------|------------|---------------|
| Google Chrome (33) | | 2.4% | 1,171.8 MB |
| Google Chrome | | 0.2% | 147.3 MB |
| Google Chrome | | 0% | 62.5 MB |
| Google Chrome | | 0% | 44.8 MB |
| Google Chrome | | 0% | 16.2 MB |

- Half of all allocations are `std::string`.
- Typing one character in the Omnibox used to result in over 25000 allocations.



www.google.com

Real life examples: Chrome



| Name | Status | 15% CPU | 77% Memory |
|--------------------|--------|------------|---------------|
| Google Chrome (33) | | 2.4% | 1,171.8 MB |
| Google Chrome | | 0.2% | 147.3 MB |
| Google Chrome | | 0% | 62.5 MB |
| Google Chrome | | 0% | 44.8 MB |
| Google Chrome | | 0% | 16.2 MB |

- Half of all allocations are `std::string`.
- Typing one character in the Omnibox used to result in over 25000 allocations.
- Performance issues resulted from lot's of copies and temporary objects.

String handling in Modern C

- No implicit conversions and constructors.

String handling in Modern C

- No implicit conversions and constructors.
- Stronger distinction between owning and non-owning strings.

String handling in Modern C

```
1  typedef struct str
2  {
3      char*    data;
4      isize_t size;
5  } str;
6
7  typedef struct str_buf
8  {
9      char* data;
10     isize_t size;
11     isize_t capacity;
12     allocator_cb allocator;
13 } str_buf;
```

String handling in Modern C

```
15 str_buf str_buf_make(isize_t size, allocator_cb allocator);
16 void str_buf_append(str_buf*, str);
17 void str_buf_insert(str_buf*, str, isize_t);
18 void str_buf_remove(str_buf*, isize_t, isize_t);
19 str str_buf_str(str_buf);
```

String handling in Modern C

```
15 str_buf str_buf_make(isize_t size, allocator_cb allocator);
16 void str_buf_append(str_buf*, str);
17 void str_buf_insert(str_buf*, str, isize_t);
18 void str_buf_remove(str_buf*, isize_t, isize_t);
19 str str_buf_str(str_buf);
```

```
21 bool str_valid(str);
22 bool str_match(str a, str b);
23 bool str_contains(str haystack, str needle);
24 str str_sub(str src, isize_t begin, isize_t end);
25 str str_find_first(str haystack, str needle);
26 str str_find_last(str haystack, str needle);
27 str str_remove_prefix(str src, str prefix);
28 str str_remove_suffix(str src, str suffix);
```

String handling in Modern C

```
30 str str_pop_first_split(str* src, str split_by);
```


String handling in Modern C

```
30 str str_pop_first_split(str* src, str split_by);
```

```
32 str date = cstr("2021/03/12");
```

```
33 str year = str_pop_first_split(&date, cstr("/"));
```

```
34 str month = str_pop_first_split(&date, cstr("/"));
```

```
35 str day = str_pop_first_split(&date, cstr("/"));
```

String handling in Old C

```
1 char* date = "1981/04/01";
2 char* year, month, day;
3
4 year = strtok(date, "/");
5 month = strtok(NULL, "/");
6 day = strtok(NULL, "/");
```

String handling in Old C

```
1 char date[] = "1981/04/01";
2 char* year, month, day;
3
4 year = strtok(date, "/");
5 month = strtok(NULL, "/");
6 day = strtok(NULL, "/");
7
```

String handling in Modern C

```
30 str str_pop_first_split(str* src, str split_by);
```

```
32 str date = cstr("2021/03/12");
```

```
33 str year = str_pop_first_split(&date, cstr("/"));
```

```
34 str month = str_pop_first_split(&date, cstr("/"));
```

```
35 str day = str_pop_first_split(&date, cstr("/"));
```

String handling in Modern C: overloading

```
37 str str_pop_first_split_impl(str* src, str split_by);
38
39 #define str_pop_first_split(src, split_by) \
40     _Generic(split_by, \
41     const char*: str_pop_first_split_impl(src, cstr(split_by)), \
42     default: str_pop_first_split_impl(src, split_by))
```

String handling in Modern C: overloading

```
37 str str_pop_first_split_impl(str* src, str split_by);
38
39 #define str_pop_first_split(src, split_by) \
40     _Generic(split_by, \
41     const char*: str_pop_first_split_impl(src, cstr(split_by)), \
42     default: str_pop_first_split_impl(src, split_by))
```

```
32 str date = cstr("2021/03/12");
33 str year = str_pop_first_split(&date, "/");
34 str month = str_pop_first_split(&date, "/");
35 str day = str_pop_first_split(&date, "/");
```

String handling in Modern C: another awesome macro

```
44 #define for_str_split(iter, src, split_by) \
45     for ( \
46         str macro_var(src_) = src, \
47         iter = str_pop_first_split(&macro_var(src_), split_by), \
48         macro_var(split_by_) = split_by; \
49         \
50         str_valid(macro_var(src_)); \
51         \
52         iter = str_pop_first_split(&macro_var(src_), macro_var(split_by_)) \
53     )
```

```
1 str date = cstr("2021/03/12");
2 for_str_split(it, date, "/")
3 {
4     print("{str}", it);
5 }
```

First optimization at CA

- First ever optimization I did at Creative Assembly was related to string operations.

First optimization at CA

- First ever optimization I did at Creative Assembly was related to string operations.
- A seemingly cheap function was splitting some string.

First optimization at CA

- First ever optimization I did at Creative Assembly was related to string operations.
- A seemingly cheap function was splitting some string.
- The function didn't seem to take much time unless you analyzed its cost cumulatively.

First optimization at CA

- First ever optimization I did at Creative Assembly was related to string operations.
- A seemingly cheap function was splitting some string.
- The function didn't seem to take much time unless you analyzed its cost cumulatively.
- The solution was to move to a non-allocating, lazily evaluated string splitter using `std::string_view`.

String handling in other languages

- Rust has a built-in `str` type which is different from the owning `String` class.
- Zig and Odin have built-in array-view types and string-view manipulation functions.
- Java and C# also follow a similar model.
- Consider starting to use `std::string_view` instead of `const std::string&` in C++.
- C++17 added `std::string_view`.
- C++20 ranges will make lazy non-allocating algorithms more easy to use.



In conclusion

- I hope that you gained a better appreciation for what can be done in a limited language like C.

In conclusion

- I hope that you gained a better appreciation for what can be done in a limited language like C.
- Checkout the latest developments in C++ and other systems languages.

In conclusion

- I hope that you gained a better appreciation for what can be done in a limited language like C.
- Checkout the latest developments in C, C++ and also other systems languages.
- Re-examine some of the patterns and preconceptions that you may hold.

In conclusion

- I hope that you gained a better appreciation for what can be done in a limited language like C.
- Checkout the latest developments in C, C++ and also other systems languages.
- Re-examine some of the patterns and preconceptions that you may hold.
- By challenging our current understanding and not taking things for granted we can discover new and better ways of doing things.

Shoutouts

Modern C for C++ Peeps: <https://floooh.github.io/2019/09/27/modern-c-for-cpp-peeps.html>

Odin: <https://odin-lang.org/>

Zig: <https://ziglang.org/>

Handmade Hero: <https://handmadehero.org/>

Handmade Network: <https://handmade.network/>