# D as Better C Compiler

by Walter Bright
dlang.org

# C

- Brilliantly conceived language
- Major force for 40 years
- Engine for major critical software
- Well known and understood
- “Man behind the curtain”

# All Is Not Roses

- 40 years of advancement in programming languages
- No memory safety

# Memory Safety

“being protected from various software bugs and security vulnerabilities when dealing with memory access, such as buffer overflows and dangling pointers.”

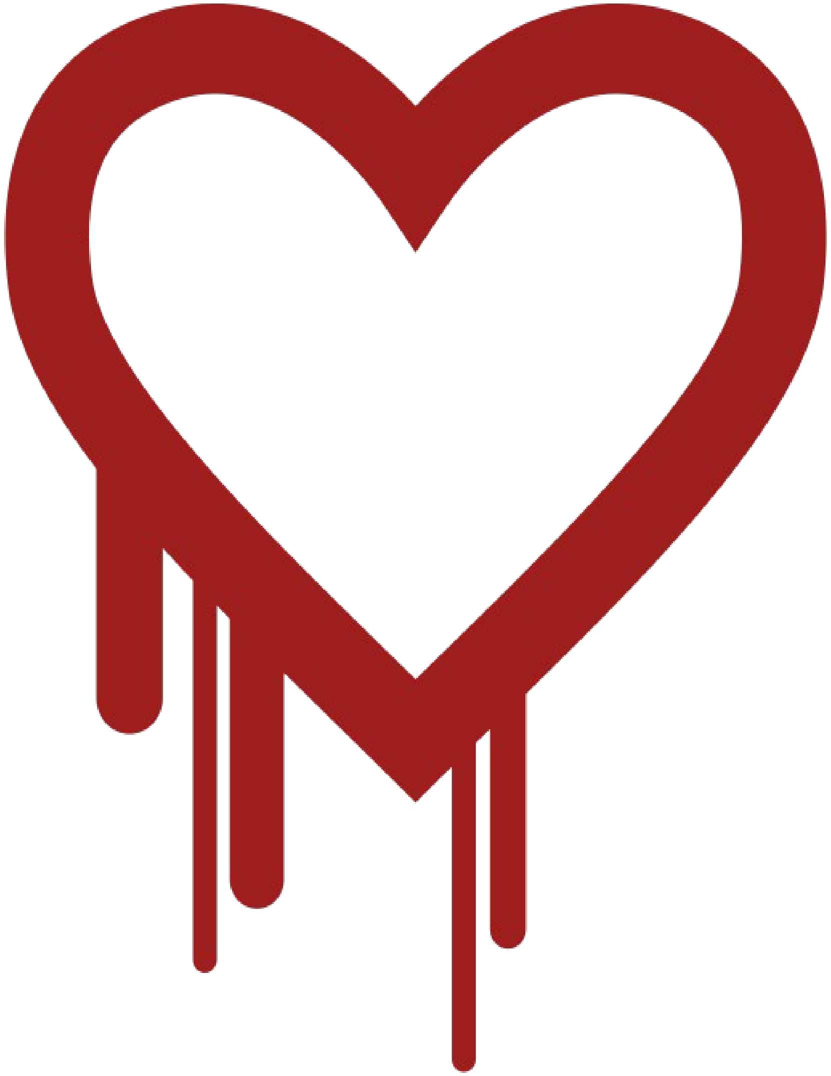
– wikipedia

# Costs of Memory Unsafety

- millions of dollars
- endless hours of effort to find and correct
- constant threat
- embarrassment

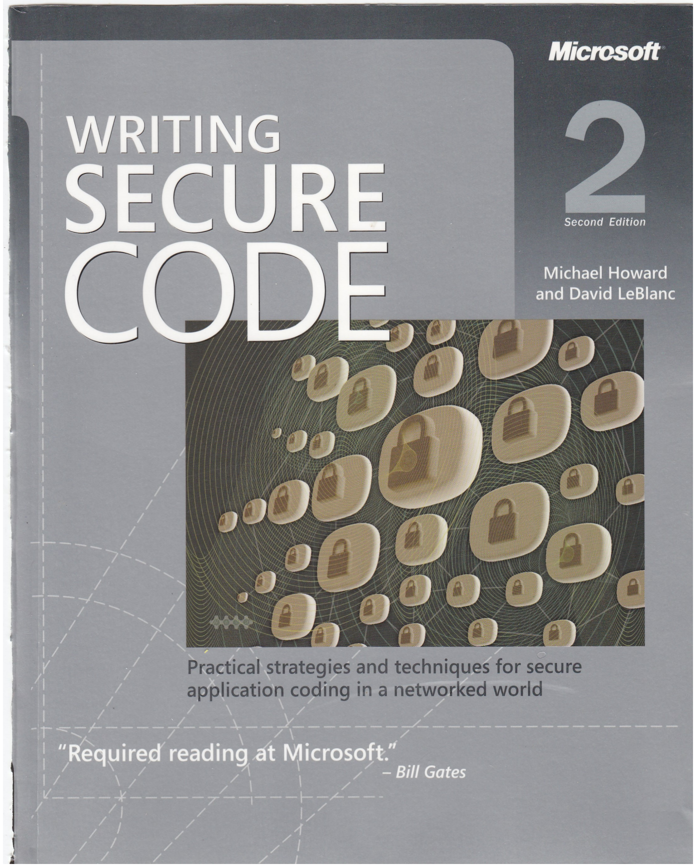
# Buffer Overflow

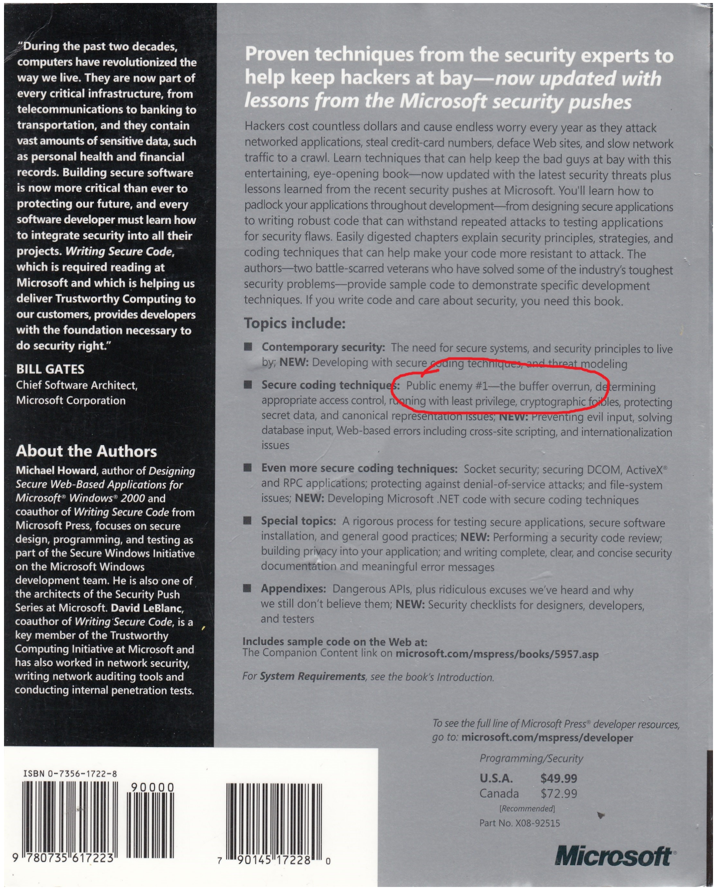
Heartbleed



https://en.wikipedia.org/wiki/Heartbleed

# Buffer Overflow


Microsoft
WRITING
SECURE
CODE
2
Second Edition
Michael Howard
and David LeBlanc
Practical strategies and techniques for secure application coding in a networked world
"Required reading at Microsoft."
– Bill Gates



"During the past two decades, computers have revolutionized the way we live. They are now part of every critical infrastructure, from telecommunications to banking to transportation, and they contain vast amounts of sensitive data, such as personal health and financial records. Building secure software is now more critical than ever to protecting our future, and every software developer must learn how to integrate security into all their projects. *Writing Secure Code*, which is required reading at Microsoft and which is helping us deliver Trustworthy Computing to our customers, provides developers with the foundation necessary to do security right."

**BILL GATES**
Chief Software Architect,
Microsoft Corporation

## About the Authors

**Michael Howard**, author of *Designing Secure Web-Based Applications for Microsoft® Windows® 2000* and coauthor of *Writing Secure Code* from Microsoft Press, focuses on secure design, programming, and testing as part of the Secure Windows Initiative on the Microsoft Windows development team. He is also one of the architects of the Security Push Series at Microsoft. **David LeBlanc**, coauthor of *Writing Secure Code*, is a key member of the Trustworthy Computing Initiative at Microsoft and has also worked in network security, writing network auditing tools and conducting internal penetration tests.

## Proven techniques from the security experts to help keep hackers at bay—*now updated with lessons from the Microsoft security pushes*

Hackers cost countless dollars and cause endless worry every year as they attack networked applications, steal credit-card numbers, deface Web sites, and slow network traffic to a crawl. Learn techniques that can help keep the bad guys at bay with this entertaining, eye-opening book—now updated with the latest security threats plus lessons learned from the recent security pushes at Microsoft. You'll learn how to padlock your applications throughout development—from designing secure applications to writing robust code that can withstand repeated attacks to testing applications for security flaws. Easily digested chapters explain security principles, strategies, and coding techniques that can help make your code more resistant to attack. The authors—two battle-scarred veterans who have solved some of the industry's toughest security problems—provide sample code to demonstrate specific development techniques. If you write code and care about security, you need this book.

### Topics include:

- **Contemporary security:** The need for secure systems, and security principles to live by; **NEW:** Developing with secure coding techniques, and threat modeling
- **Secure coding techniques:** Public enemy #1—the buffer overrun, determining appropriate access control, running with least privilege, cryptographic foibles, protecting secret data, and canonical representation issues; **NEW:** Preventing evil input, solving database input, Web-based errors including cross-site scripting, and internationalization issues
- **Even more secure coding techniques:** Socket security; securing DCOM, ActiveX® and RPC applications; protecting against denial-of-service attacks; and file-system issues; **NEW:** Developing Microsoft .NET code with secure coding techniques
- **Special topics:** A rigorous process for testing secure applications, secure software installation, and general good practices; **NEW:** Performing a security code review; building privacy into your application; and writing complete, clear, and concise security documentation and meaningful error messages
- **Appendixes:** Dangerous APIs, plus ridiculous excuses we've heard and why we still don't believe them; **NEW:** Security checklists for designers, developers, and testers

**Includes sample code on the Web at:**
The Companion Content link on **microsoft.com/mspress/books/5957.asp**

*For **System Requirements**, see the book's Introduction.*

*To see the full line of Microsoft Press® developer resources, go to:* **microsoft.com/mspress/developer**

ISBN 0-7356-1722-8
90000
9 780735 617223
7 90145 17228 0

*Programming/Security*

| | |
|---|---|
| **U.S.A.** | **$49.99** |
| Canada | $72.99 |

*[Recommended]*

Part No. X08-92515

Microsoft

# Public Enemy #1

# Microsoft's Writing Secure Code 2

# Buffer Overflow

Morris Worm

First internet worm relied on buffer overflow C bug

# Buffer Overflow

#3 on list of top 25 security vulnerabilities

http://cwe.mitre.org/top25/index.html#CWE-120

# Not a Buffer Overflow

# Good News!

It's not your fault

- Has persisted for 40 years
- Not a tractable problem
- Not fixable by fixing the programmer

# It's a Tooling Problem

```
int sum(int *array, size_t length) {
    int sum = 0;
    for (size_t i = 0; i <= length; ++i)
        sum += array[i];
    return sum;
}
```

– Wizard of Oz

# D Arrays are Phat Pointers

```
struct Array {
    size_t length;
    int* ptr;
}
```

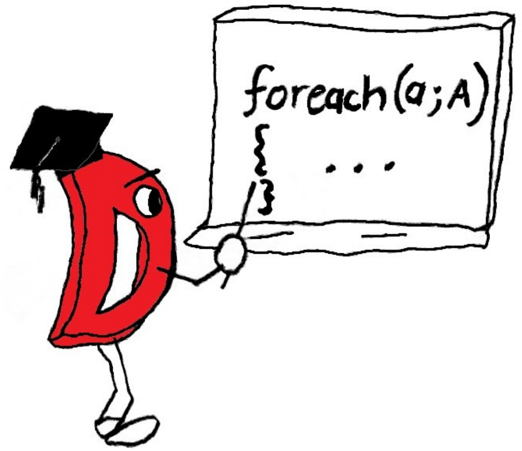
https://dlang.org/spec/arrays.html#dynamic-arrays

# First D Version

```
int sum(int[ ] array)
{
    int sum = 0;
    for (size_t i = 0; i < array.length; ++i)
        sum += array[i];
    return sum;
}
```

# Using foreach

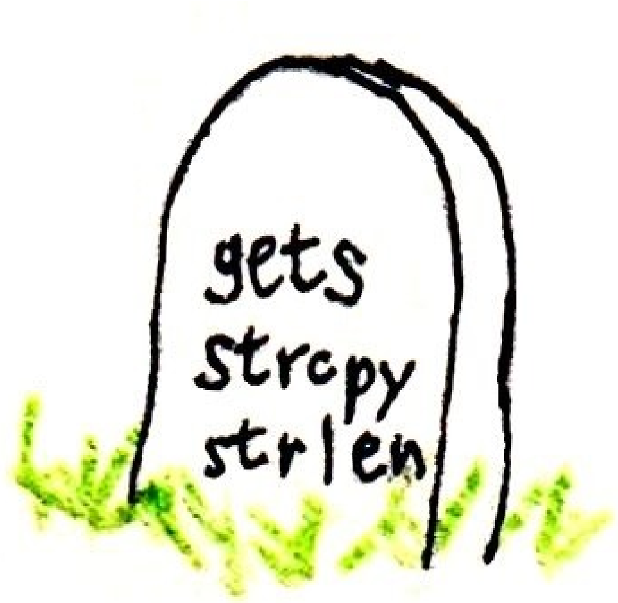
```
int sum(int[ ] array)
{
    int sum = 0;
    foreach (i; 0 .. array.length)
        sum += array[i];
    return sum;
}
```



# More Advanced foreach

```
int sum(int[ ] array)
{
    int sum = 0;
    foreach (value; array)
        sum += value;
    return sum;
}
```

# Same problem with 0 terminated strings



D strings are just arrays of characters

# And Much More Safety

- no uninitialized pointers
- no pointers to expired stack frames
- no aliasing pointers with other types
- no implicit narrowing conversions
- pointer lifetime tracking
- etc...

# Calling C Code from D

```
extern (C) void* malloc(size_t);

T[ ] allocArray(T)(size_t numElements)
{
    auto p = malloc(T.sizeof * numElements);
    assert(p);
    return p[0 .. numElements];
}
```

# Problem

D code needs to link with the D runtime library. But an existing C project does not link with the D runtime library. Linking it in produces issues with the size of the D runtime library, and how/when it is initialized compared to that of the existing C project.

# Solution - BetterC

Create a subset of D that does not require the D runtime library.

# Features Altered

- assert() failures now go to C standard library
- RAII no longer unwinds exceptions
- No dynamic type info
- No exception handling
- No automatic memory management

What's this going to cost?

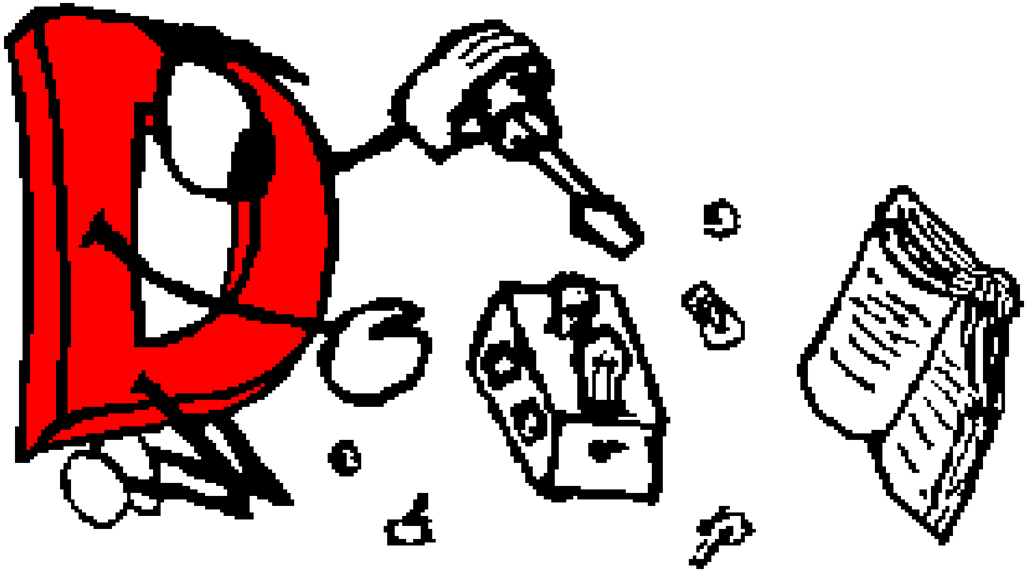
# C

```
#include <stdio.h>

int main(size_t argc, char** argv)
{
    for (size_t i = 0; i < argc; ++i)
        printf("arg[%zd] = %s\n", i, argv[i]);
    return 0;
}
```

# D as Better C

```
import core.stdc.stdio;

extern (C)
int main(size_t argc, char** argv)
{
    foreach (i, s; argv[0 .. argc])
        printf("arg[%zd] = %s\n", i, s);
    return 0;
}
```

# Some Assembly Required



```
    sub ESP,01Ch
    mov 014h[ESP],ESI
    mov ESI,020h[ESP]
    mov 018h[ESP],EDI
    mov EDI,024h[ESP]
    mov 010h[ESP],EBX
    xor EBX,EBX
    test ESI,ESI
    je  L1
L2: mov ECX,[EBX*4][EDI]
    mov 8[ESP],ECX
    mov 4[ESP],EBX
    mov [ESP],offset _DATA
    call _printf
    inc EBX
    cmp EBX,ESI
    jb  L2
L1: mov EBX,010h[ESP]
    mov ESI,014h[ESP]
    mov EDI,018h[ESP]
    add ESP,01Ch
    xor EAX,EAX
    ret
```

```
    sub ESP,01Ch
    mov 014h[ESP],ESI
    mov ESI,020h[ESP]
    mov 018h[ESP],EDI
    mov EDI,024h[ESP]
    mov 010h[ESP],EBX
    xor EBX,EBX
    test ESI,ESI
    je  L1
L2: mov ECX,[EBX*4][EDI]
    mov 8[ESP],ECX
    mov 4[ESP],EBX
    mov [ESP],offset CONST
    call _printf
    inc EBX
    cmp EBX,ESI
    jb  L2
L1: mov EBX,010h[ESP]
    mov ESI,014h[ESP]
    mov EDI,018h[ESP]
    add ESP,01Ch
    xor EAX,EAX
    ret
```

# No Compromise

The same code is generated!

# How to compile and link in a BetterC function into a C project...

# The Original C Code

```
#include <stdio.h>
int sum(int *array, size_t length) {
  int sum = 0;
  for (size_t i = 0; i < length; ++i)
        sum += array[i];
  return sum;
}
int main(size_t argc, char** argv) {
    int array[3] = { 1, 37, 28 };
    int s = sum(array,sizeof(array)/sizeof(array[0]));
    printf("sum = %d\n", s);
    return 0;
}
```

# main.c

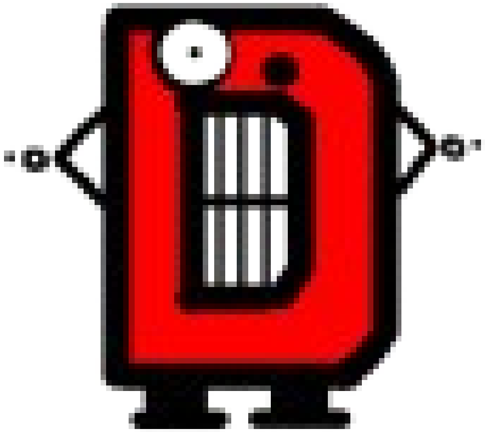
```
#include <stdio.h>

int sum(int *array, size_t length);

int main(size_t argc, char** argv) {
    int array[3] = { 1, 37, 28 };
    int s = sum(array,sizeof(array)/sizeof(array[0]));
    printf("sum = %d\n", s);
    return 0;
}
```

# sum.d

```
extern ( C )
int sum(int *array, size_t length) {
    return sum(array[0 .. length]);
}

@safe int sum(int[ ] array) {
    int sum = 0;
    foreach (value; array)
        sum += value;
    return sum;
}
```

# Compiling

```
dmd -c sum.d
dmc main.c sum.obj
```



# Converting a C file to D

- Remove preprocessor metaprogramming
  - (should get rid of it anyway)
- Copy the code to a .d file
- Translate one function at a time
  - run the test suite after each

# Don't Refactor/Improve/Fix

- Wait until it is all translated
  - and passes all tests!
  - resist overwhelming temptation
    - or you'll be sorry!
- Just rote translate it

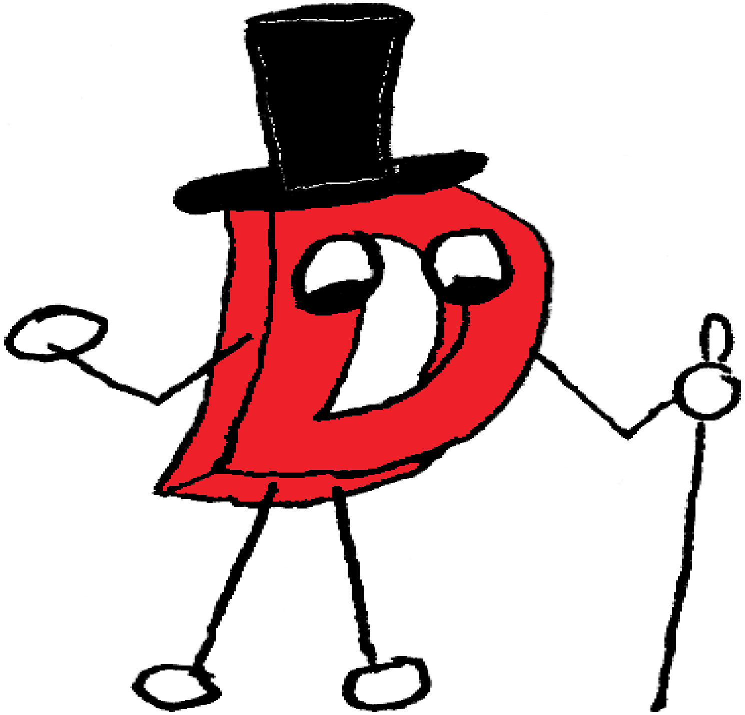
# Proof That This Works

The Digital Mars C/C++ Compiler Front End

https://github.com/DigitalMars/Compiler/tree/master/dm/src/dmc

# Now That It's Working in D

- What can be done to benefit?
  - (Besides using proper arrays!)



```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

bool see(void* buf, int maybe);

bool isPossible(int maybe) {
  void *buf = malloc(100);
  if (see(buf, maybe)) {
    free(buf);
    return true;
  }
  free(buf);
  return false;
}
```

```
import core.stdc.stdlib;  // modules!

bool see(void* buf, int maybe);

bool isPossible(int maybe) {
  void *buf = malloc(100);
  scope(exit) free(buf);  // scope guard
  if (see(buf, maybe))
    return true;
  return false;
}
```

# RAII

```
import core.stdc.stdlib;

bool see(void* buf, int maybe);

struct S { void* buf; ~this() { free(buf); } }

bool isPossible(int maybe) {
  S s;
  s.buf = malloc(100);
  if (see(s.buf, maybe))
    return true;
  return false;
}
```

```
int compute(int x, int y) {
    if (x == y)
        return -1; // error
    if (x == 5)
        return 2;
    if (x == y + 3)
        return 7;
    return -1; // error
}
```

```
int compute(int x, int y) {
    const bool log = true;
    if (x == y)
    { if (log) printf("fail\n"); return -1; }
    if (x == 5)
    { if (log) printf("success 2\n"); return 2; }
    if (x == y + 3)
    { if (log) printf("success 7\n"); return 7; }
    if (log) printf("fail\n");
    return -1;
}
```

# Nested Functions

```
int compute(int x, int y) {
    enum log = true;
    int fail() {
        if (log) printf("fail\n"); return -1;
    }
    int success(int i) {
        if (log) printf("success %d\n", i); return i;
    }

    if (x == y)
        return fail();
    if (x == 5)
        return success(2);
    if (x == y + 3)
        return success(7);
    return fail();
}
```

# Some Things D Can't Do

```
#define BEGIN {
#define END   }

int sum(int *array, size_t length)
  BEGIN
    int sum = 0;
    size_t i;
    for (i = 0; i < length; ++i)
      BEGIN
        sum += array[i];
      END
    return sum;
  END
```

# Get Started Today!

- Incrementally use D functions in large C project
- Stop buffer overflows and other safety bugs
- Use D to improve code

dlang.org