# Destroy All Memory Corruption

by Walter Bright

Dconf 2020

# Memory Corruption

- A pernicious and expensive problem
- Impractical to manually review code for it
- Corruption can easily be introduced by unwary changes
  - (again with the review problems)

#1 Memory corruption problem is buffer overflows

# Ending Buffer Overflows

- Array overflow protection
- Attractive to use dynamic arrays rather than raw pointers
- Use of `ref` rather than raw pointers
- Use of `const` and `immutable`

# Ending Stack Corruption (pointers into expired stack frames)

- ref
- return
- scope

# Ending Aliasing Problems

- Casting non-pointers to pointers
- Unions overlaying pointers with other types

# Ending Allocation Bugs

- Use the Garbage Collector

# But I Don't Want To Use the GC!

- Explicit malloc/free
  - Including writing your own allocator
- RAII
  - i.e. destructors
- Reference counting

# Explicit malloc/free

- malloc without free (memory leaks)
- Use after free
- free more than once
- free without malloc

# RAII

- A natural fit for scoped objects
- Not so good for other patterns

# Reference Counting

```
struct S { // ref counting machinery omitted for brevity
    int* p;
}

void fun(ref S s, ref S t) {
    s = S();   // previous contents of s destroyed
    *t.p = 1;   // boom!
}

void main() {
    S s;
    int i;
    s.p = &i;
    fun(s, s);
}
```

Two pointers to the same object, one or both of which is mutable.

# DIP 1021
# Argument Ownership and Function Calls

Disallow more than one reference to the same memory object being passed to a function's parameters, if any of them are mutable.

https://github.com/dlang/DIPs/blob/master/DIPs/accepted/DIP1021.md

# Generalizing...

Disallow more than one reference to the same memory object if any of those references are mutable.

# Clarifying

- Allowed
  - One mutable reference
  - Many const references
- Not Allowed
  - More than one mutable reference
  - Mixed mutable and const references

# Ownership

A single mutable reference to a memory object is said to “own” the object.

```
int* f();
int* p = f(); // p now owns the object returned by f()
```

# Moving (Transferring) Ownership

Moving a mutable reference transfers ownership.
The previous reference becomes invalid.

```
int* p = f(); // p is now the owner
int* q = p;  // q is now the owner, p is invalid
*q = 3;       // ok, as q owns it
*p = 4;       // error, p is invalid
```

# Copying (Borrowing) a Reference

Copying a mutable reference borrows ownership. When the borrow is done, ownership is returned. Borrowing is indicated with `scope`.

```
int* p = f();           // p is now the Owner
scope int* b = p; // b borrows from p
*b = 3;                 // ok, as b temporarily owns it
*p = 4;                 // ok, ownership is returned to p
*b = 5;                 // error, b is invalid
```

# I Know What You're Thinking!

Wait? Whaaaaaat? When, how does
the borrowed reference q become invalid?

# A Borrowed reference ends when one of the following holds:

- The last use of the borrowed reference
- The borrowed reference goes out of scope
- The owner is used again

From the borrowing to one of those three is called the lifetime of the borrow. (Also known as “non-lexical” scoping.)

This is determined using...

# Data Flow Analysis

- Decompose a function's structure into a collection of blocks of code connected by edges that represent paths from one block of code to another
- Construct Data Flow Equation for each block in the form: Output = Transformation(Input)
- Solve the N equations for N unknowns.

In this case, the Input and the Output are the states of each of the variables being tracked.

# Pointer Creation

A pointer is created when a function
is called that returns a pointer.

```
int* f();       // function returns an owning pointer
int* p = f(); // which is moved to p
```

# Pointer Destruction

A pointer is destroyed when it is moved to a function.

```
void g(int*);
g(p);     // p gives up its ownership
*p = 3;  // error, p is invalid
```

# Dangling Pointer

```
@live void sun()
{
    int* p = f();
} // error, p is live on exit
```

# Functions Taking Ownership

```
@live void g(int* p)
{
}  // Error, p is dangling

@live void h(int* p)
{
    g(p);  // transfer to g()
} // ok, p is g()'s problem
```

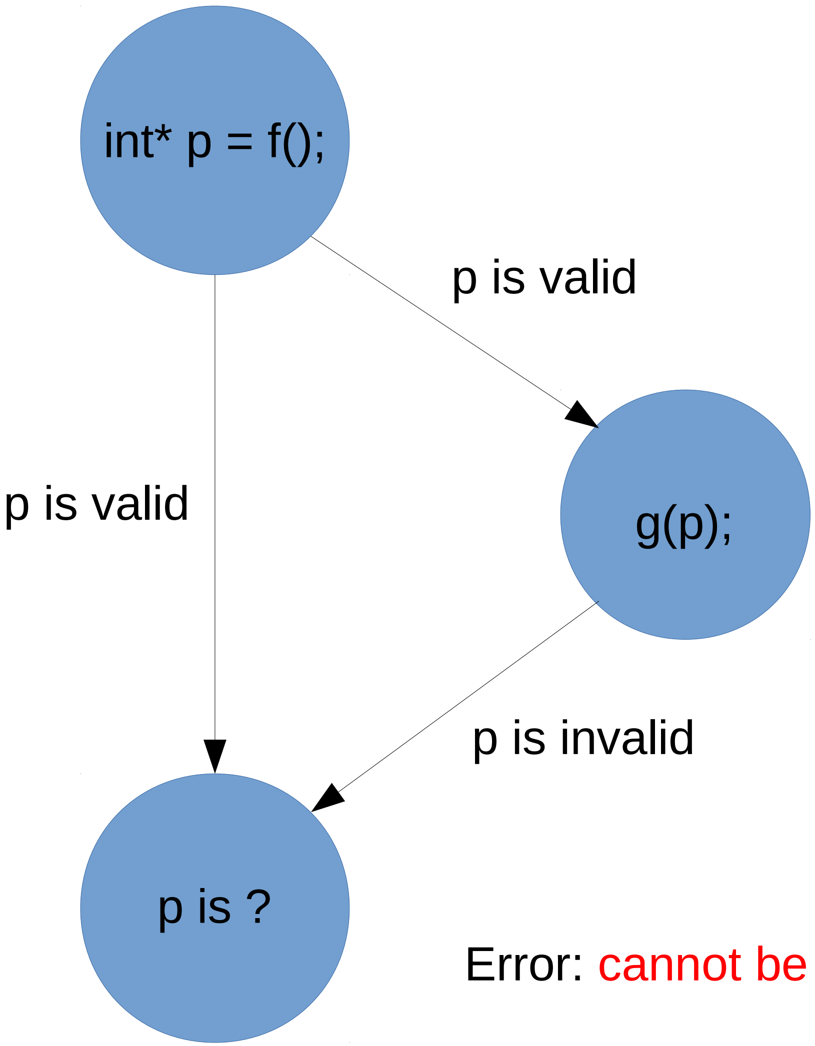
# Functions Borrowing Pointers

```
@live void m(scope int* b)
{
} // no error

@live void n(scope int* b)
{
    m(b); // ok
    g(b); // error, borrowed pointer escapes
}

void g(int* p);
```

# Control Flow




Error: cannot be both valid and invalid

# In Terms Of malloc() and free()

```
int* malloc();
void free(int*);
```

Note that these functions cannot be @live

# Memory Leak #1

```
void star()
{
    int* p = malloc();
} // error, p is live on exit
```

Note: malloc() and free() are *NOT* special to the language, meaning custom allocators can be written as first class citizens.

# Memory Leak #2

```
int* p = malloc();
p = malloc(); // error, overwrite of live pointer
```

# Double Free

```
int* p = malloc();
free(p);
free(p); // error, p has undefined value
```

# Use After Free

```
int* p = malloc();
*p = 3;   // ok
free(p);  // destroys p
*p = 4;   // error, p has invalid value
```

# Destroying Borrowed Pointer

```
@live void mars(int* p)
{
    scope int* b = p; // b borrows from p
    free(b);  // error, cannot turn borrowed pointer into owner
} // error, p is left dangling
```

# Constant Pointers

```
@live void pluto(int* p)
{
    scope const(int)* c1 = p;   // borrow a const reference
    scope const(int)* c2 = c1; // another const reference
    int i = *c1;   // c1 is live
    int j = *c2;   // c2 is live
    j = *c1;        // c1 is still live
    *p = 3;         // use p, invalidate c1 and c2
    i = *c1;        // error, c1 is invalid
    j = *c2;        // error, c2 is invalid
    free(p);        // dispose of p
}
```

# Calling Functions

```
void bar1(scope const int*, scope const int*);
void bar2(scope       int*, scope const int*);

@live void neptune(int* p)
{
    bar1(p, p); // compiles
    bar2(p, p); // does not compile
}
```

# Recall the Ref Counting Problem?

```
struct S { // ref counting machinery omitted for brevity
    int* p;
}

void fun(ref S s, ref S t) {
    s = S();   // previous contents of s destroyed
    *t.p = 1;   // boom!
}

@live void main() {
    S s;
    int i;
    s.p = &i;
    fun(s, s); // @live gives error here
}
```

# Global Variables

@live functions cannot access global variables.
They have to come in through the front door,
i.e. the parameter list.

# Other Pointer Types

- ref
- out
- Classes
- Implicit this
- Wrapped pointers
- Dynamic arrays
- Delegates
- Associative arrays

# GC Allocated Pointers

Handled just like any other pointer.
No distinction is made, or can be made.

# @live and Other Functions

- @live relies on non @live functions it interfaces with respecting the @live interface
- @system, @trusted, @safe can all interface with @live functions
- Hence @live functions can be added incrementally

# Exceptions

- Cause many complex edges between blocks
- Most data flow optimizers give up when encountering exception control flow
- @live relies on data flow analysis and can't just give up
  - Therefore, @live functions are nothrow
  - Part of why I've pushed for nothrow being the default

# Implementation

@live functions are now available in prototype form in the latest D compilers.

# Conclusion

- Builds on existing successful safety mechanisms in D
- Large step forward in achieving mechanically guaranteed memory safety
- Does not break any existing code
  - Can be added incrementally

# References

- https://dlang.org
- https://github.com/dlang/DIPs/blob/master/DIPs/accepted/DIP1021.md
- https://dlang.org/blog/2019/07/15/ownership-and-borrowing-in-d

music by Max Bright