MEMORY MANAGEMENT IN C: AN INTRODUCTION

Paul R. Potts

WHAT I'LL COVER (1)

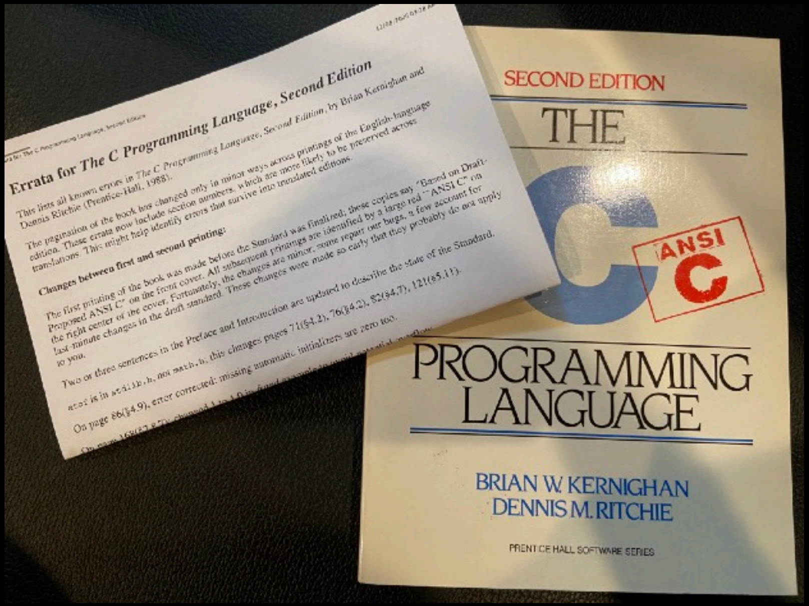
- 3 Object Lifetimes (2 are also C storage classes)
  - static
  - automatic
  - dynamic
- The static storage class (not to be confused with static declarations)
- Declarations and definitions
- External linkage
- Static initialization: implicit and explicit
- Don't say "global" when you're talking about C

WHAT I'LL COVER (2)

- Blocks
- Implementation of the automatic storage class, and implications
- The heap or "free store"
- `malloc`, `free`, and family
- Memory management and memory leaks

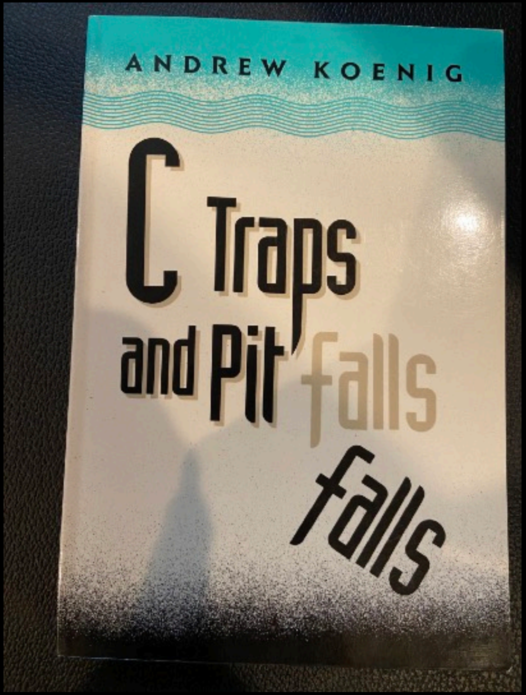
MY FAVORITE BOOKS ON C PROGRAMMING

- *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie
  - Indispensable, not comprehensive. Skips briefly over topics that really need to be explained in greater depth. Programmers who stop here will think C is far easier to get right than it really is.
  - Look up the errata too!


Errata for The C Programming Language, Second Edition
SECOND EDITION
THE C PROGRAMMING LANGUAGE
ANSI C
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

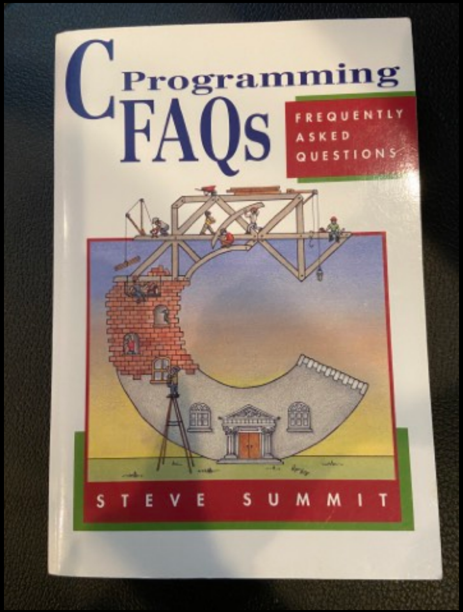
ASIDE: MY FAVORITE BOOKS ON C PROGRAMMING

- *C Traps and Pitfalls* by Andrew Koenig
  - Somewhat outdated, but a quick and fun read that makes you realize that you need to know more than what's in K&R.



ASIDE: MY FAVORITE BOOKS ON C PROGRAMMING

- *C Programming FAQs* by Steve Summit
  - Adapted from old Usenet discussions
  - Somewhat outdated, but still valuable as an on-ramp to the standards themselves

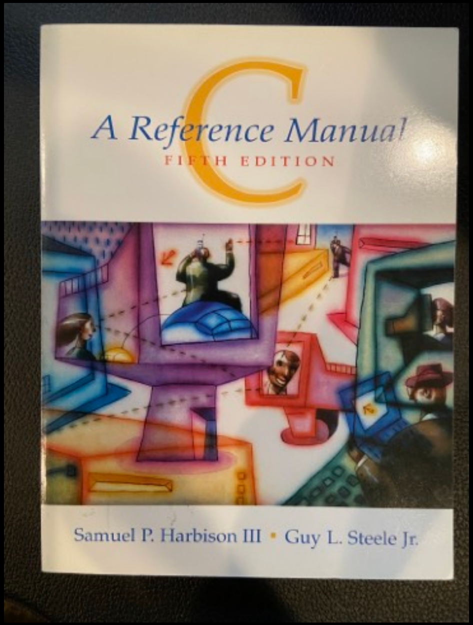


ASIDE: MY FAVORITE C PROGRAMMING BOOKS

- *Expert C Programming: Deep C Secrets* by Peter van der Linden
  - Indispensable!
  - This book explains the relationship between arrays and pointers in detail. You don't *really* understand them if you haven't studied this book.

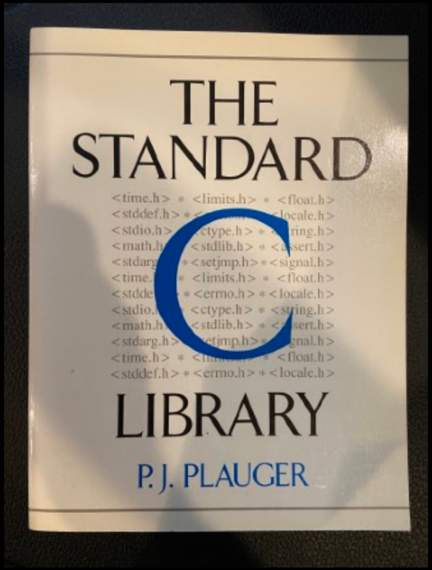
ASIDE: MY FAVORITE C PROGRAMMING BOOKS

- *C: A Reference Manual* by Samuel P. Harrison III and Guy L. Steele Jr., Fifth Edition
  - The best place to look up the details of standard library functions, although not entirely up to date.



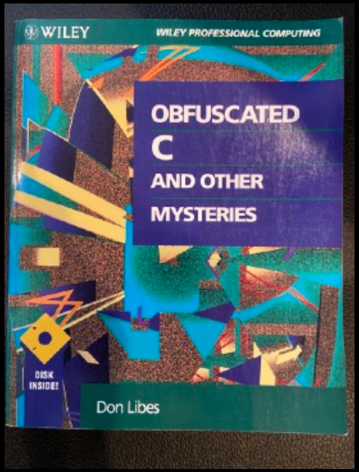
ASIDE: MY FAVORITE C PROGRAMMING BOOKS

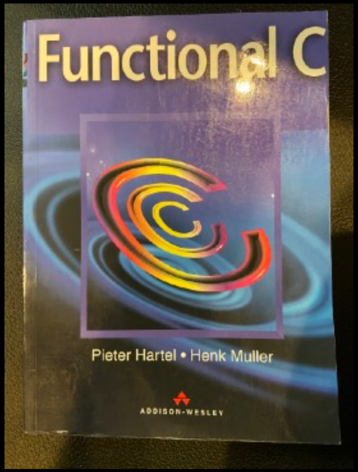
- *The Standard C Library* by P. J. Plauger
  - Contains a reference implementation of an outdated standard library. Very useful if you need to implement your own version of a standard function, or if you want to study how these functions are written. I learned a lot!



ASIDE: MY FAVORITE C PROGRAMMING BOOKS

- What if you want to compete in the International Obfuscated C Code Contest? https://www.ioccc.org/ (Or, just stare in horror and fascination at some of the weirdest C code very written?)
- What if you wanted to use C like a *functional* programming language? This book covers some interesting programming techniques.


WILEY
WILEY PROFESSIONAL COMPUTING
OBFUSCATED C AND OTHER MYSTERIES
DISK INSIDE!
Don Libes



ASIDE: RESOURCES FOR C PROGRAMMING INTERVIEWS

- "A C Test: The 0x10 Best Questions for Would-be Embedded Programmers"
  - https://rmbconsulting.us/publications/a-c-test-the-0x10-best-questions-for-would-be-embedded-programmers/
- A great talk on "Deep C" by Olve Maudal and Jon Jagger, with in-depth interview questions and answers (NOTE: this is a link to a PDF file)
  - http://www.pvv.org/~oma/DeepC_slides_oct2011.pdf

TERMINOLOGY

Many books on C and C++ use different terms
for the same concepts.

I have adopted terms used by Kernighan and Ritchie in
The C Programming Language, 2nd edition

This unfortunately leads to some “operator overloading.” For example, I use “object” to refer to variables that may be integers, structures, arrays, etc., and “class” to refer to C “storage classes” — these terms don’t have meanings from object-oriented programming!

THE THREE LIFETIMES

There are three kinds of object lifetimes in C programs.

Kernighan and Ritchie refer to the first two kinds as “storage classes.”

THE FIRST LIFETIME: THE STATIC STORAGE CLASS

If an object has the static storage class, the lifetime of an object is the same as the lifetime of your program. The objects are ready to use when `main` is called, and they remain usable until your program exits.

The static storage class always applies to all objects that are "declared outside all blocks" [K&R, p. 195]

A block means "a group of declarations and statements" [Ibid.] inside curly braces { }. This includes functions, but is not limited to functions.

Objects with this storage class aren't necessarily "global." They may not be accessible to code in other files, if they are declared with the `static` keyword.

INITIALIZATION OF OBJECTS WITH STATIC STORAGE CLASS

Objects with static storage class are always initialized for you, even if you do not specify an initial value.

This is important to know. Later we will contrast this with objects with automatic storage class.

It's worth repeating: *objects with static storage class are always initialized for you, even if you do not specify an initial value!*

I will come back to this point when we contrast it with initialization of objects with automatic storage class.

EXAMPLE:
THE STATIC STORAGE CLASS

```
int a;
static int b;

/*
   The variables a and b both have
   static storage class. The keyword
   static in this context does not
   determine the storage class! It
   gives b “internal linkage.”
*/
```

DIGRESSION: DECLARATIONS AND DEFINITIONS

In the previous example, we defined our variables, in the same lines of code where we declared them. However, this isn't required.

There can be `extern` declarations that are not definitions, but which specify the type and name of an object.

A declaration is like a *customs declaration* – it describes what is in the package. It isn't the same as the package.

DIGRESSION: EXTERNAL LINKAGE

We write `extern` declarations to allow code in one file to access objects in another.

The objects must have *external linkage* (that is, they must *not* be declared `static`).

If they aren't, we get a linker error. We convinced the compiler that there is a variable there, but the linker can't can't find it, because it is considered private to the file where it is defined.

DIGRESSION: DECLARATIONS AND DEFINITIONS

I'll repeat this point: a declaration is like a *customs declaration* – a piece of paper that describes what is in the package.

The *definition* actually *creates the package*.

There can be many declarations for the same object.

DIGRESSION: DECLARATIONS AND DEFINITIONS

There must be only one* *definition* of an object.

(Technically, multiple identical definitions of the same object outside all blocks are allowed within the same file, but the compiler will consider them all to comprise a single definition. This is very poor style, though.)

DIGRESSION: DECLARATIONS AND DEFINITIONS

Defining objects of the same name, with external linkage, in multiple files, should produce a linker warning, when the linker attempts to hook up multiple compiled files into a single program.

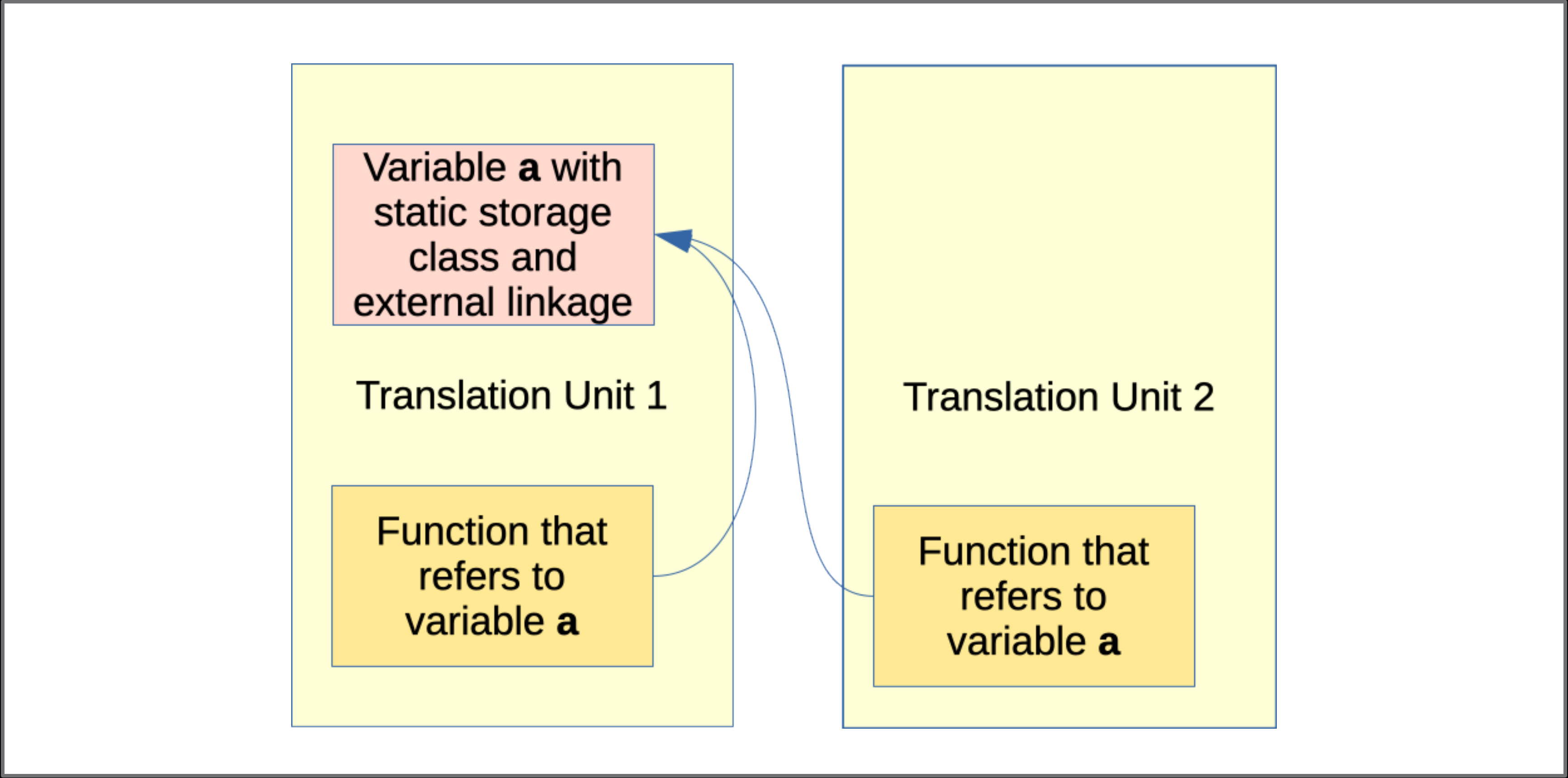
DIGRESSION:
DECLARATIONS AND DEFINITIONS

I once spent a week debugging a program with a strange bug.

The engineer who wrote the code originally included definitions of objects, both with the same name, with external linkage in different translation units (C source files).

He did not understand why the linker was giving him “multiple definition” warnings. The warnings had annoyed him, so he disabled them in the project settings.

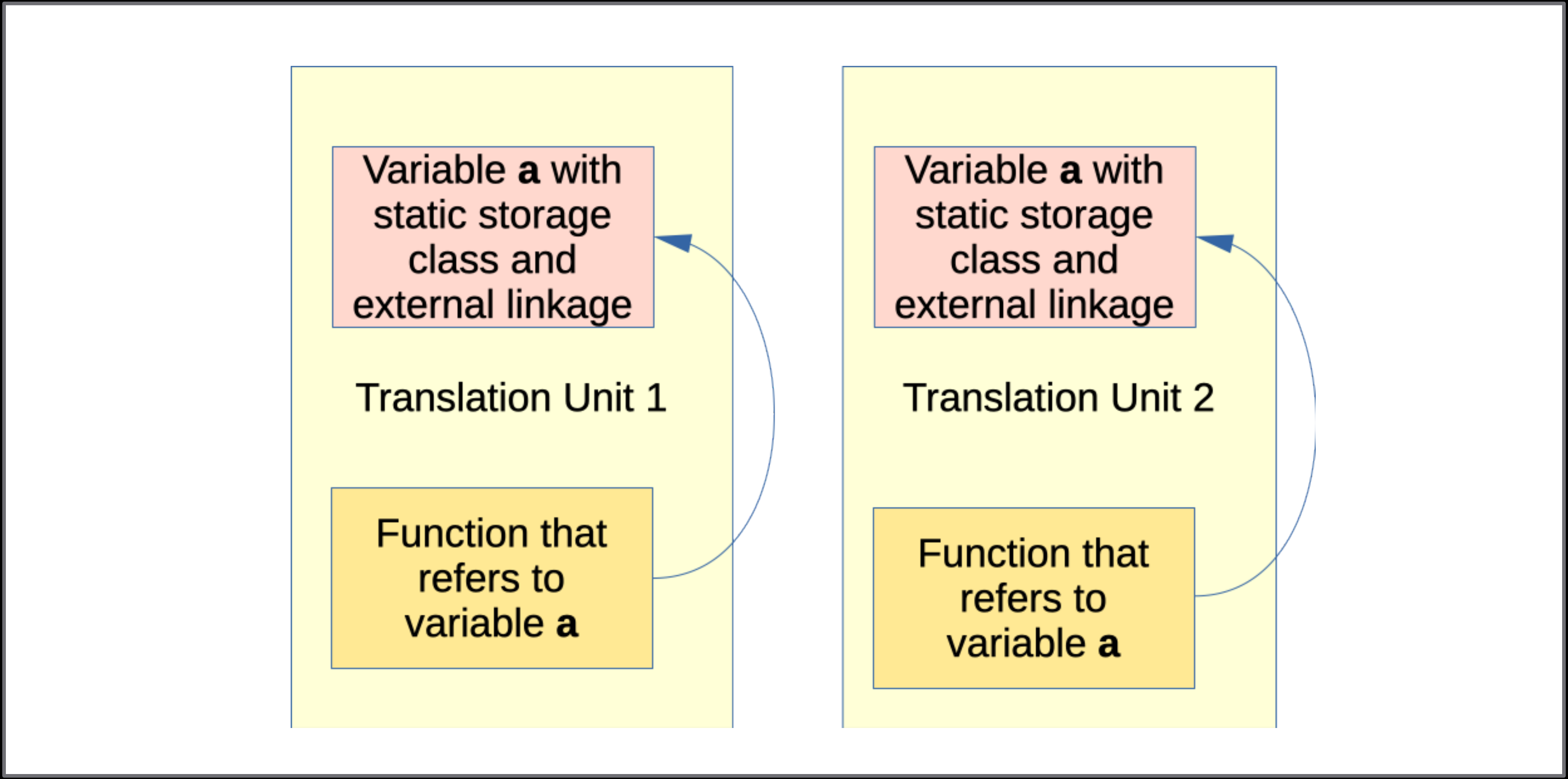
WHAT I THOUGHT WAS HAPPENING


Variable **a** with static storage class and external linkage
Translation Unit 1
Function that refers to variable **a**
Translation Unit 2
Function that refers to variable **a**

DIGRESSION: DECLARATIONS AND DEFINITIONS

As a result of the bug, which was hidden because of the disabled warning, functions compiled in two different files appeared to be accessing the same object, but they were really accessing different objects with the same name.

WHAT WAS ACTUALLY HAPPENING

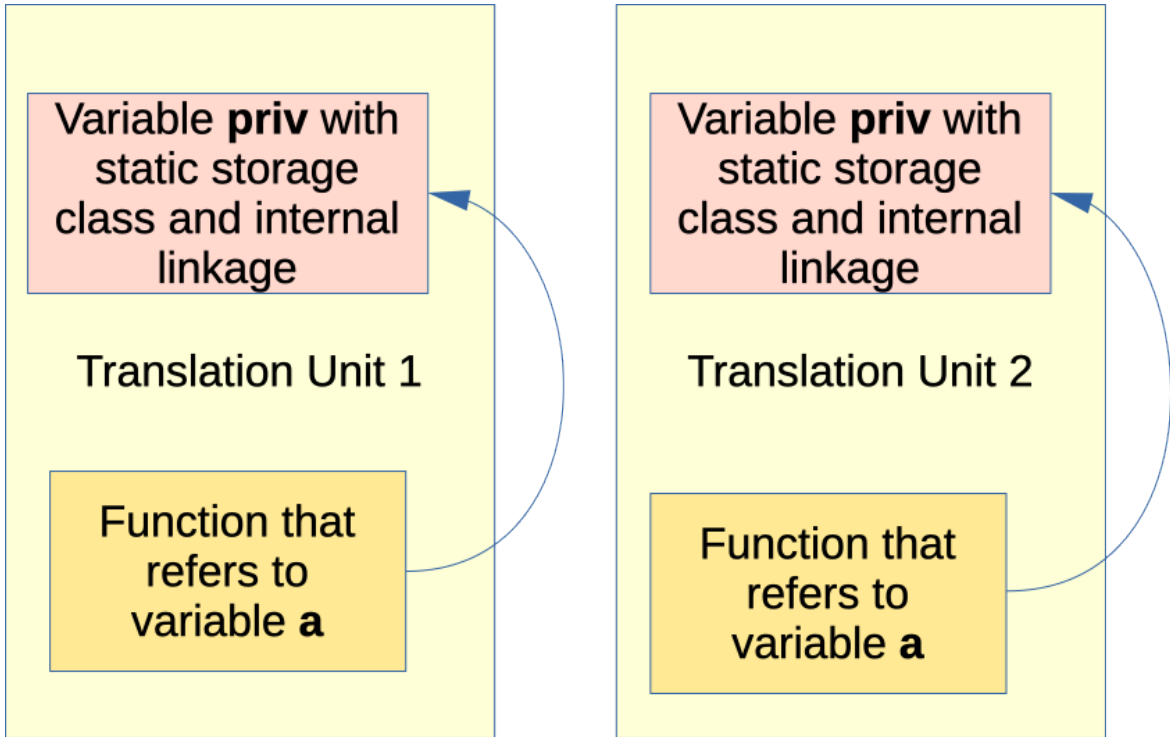

Variable **a** with static storage class and external linkage
Translation Unit 1
Function that refers to variable **a**
Variable **a** with static storage class and external linkage
Translation Unit 2
Function that refers to variable **a**

DIGRESSION: DECLARATIONS AND DEFINITIONS

This can happen if you accidentally include a definition, not a declaration, in a header file, which is then included in multiple source files.

Note that defining objects of the same name, with *internal* linkage, in multiple files, is allowed; this is an important mechanism to minimize clashes between objects with the same name that aren't meant to be shared between source files.

IT'S FINE TO DO THIS


Variable **priv** with static storage class and internal linkage
Translation Unit 1
Function that refers to variable **a**
Variable **priv** with static storage class and internal linkage
Translation Unit 2
Function that refers to variable **a**

EXAMPLE:
USING EXTERN

```
/* In file one.c */
int a;
static int b;

/* In file one.h */
extern int a;

/* In file two.c */
#include "one.h"

int main()
{
   a = 1; /* The object a, defined in one.c, is modified */
   return 0;
}
```

DIGRESSION: "GLOBAL"

You can see now why I avoid the term "global variable" when talking about C.

It does not adequately describe what is happening, when you see an object declaration outside all blocks.

A "global variable" in C is an object declared outside all blocks, with *external linkage*. This makes it potentially accessible to code in other files. However, to access it, that code needs to include an `extern` declaration for the object.

Since a "global variable" in C is not a single thing in the code, the things that together make a variable "global" should not necessarily have a single name.

DIGRESSION: “GLOBAL”

When writing C, give as few of your variables and functions external linkage as possible. C does not have nested lexical scopes, user-defined namespaces, a module system, or other features to help limit conflicts between names.

“With the benefit of practical experience, default global visibility has been conclusively and repeatedly demonstrated to be a mistake.” [Peter van der Linden, *Expert C Programming: Deep C Secrets*, p. 42]

THE STATIC STORAGE CLASS, CONTINUED

There's another important way to use the static storage class.

The static storage class may also apply to objects within *blocks*. Remember that blocks include function bodies.

That is, *lexically local* variables (but K&R does not use this term).

EXAMPLE: A STATIC OBJECT WITHIN A BLOCK

```
/*
   This function returns the number of times
   it has been called since the program started.
*/
int callcount(void)
{
   static int sum;
   /*
      This use of static gives sum
      static storage class
   */
   return ++sum;
}
```

THE STATIC STORAGE CLASS, CONTINUED

Because a block-scoped static object has static storage class, it is *perfectly legal and safe* (although unusual and in poor style) to return its address, and use the address to access it from elsewhere in your program.

EXAMPLE: RETURNING THE ADDRESS OF A STATIC OBJECT

```
int * get_count_p(void)
{
    static int count = 0;
    return &count;
}

int main()
{
    int * count_p = get_count_p();
    (*count_p)++;
    (*count_p)++;
    (*count_p)++; /* What is the value of count now? */
}
```

EXAMPLE: RETURNING THE ADDRESS OF A STATIC OBJECT

```
int * get_count_p(void)
{
    static int count = 0;
    return &count;
}

int main()
{
    int * count_a_p = get_count_p();
    int * count_b_p = get_count_p();
    (*count_a_p)++;
    (*count_b_p)++; /* What is the value of count now? */
}
```

DIGRESSION: STATIC INITIALIZATION

C offers a feature that helps you safely use variables with the static storage class. They are *always initialized* before `main` is called. This applies to variables with the static storage class outside all blocks *and* inside blocks.

If you provide initial values, the compiler will use them. Otherwise it will set the variables to zero (sort of)*.

*Integral types like `char`, `int`, and `long` will have all their bits set to zero. Floating-point types will hold `0.0`. Pointer types will be initialized to the *null pointer constant*. These may not actually have all their bits set to zero! This depends on your hardware platform.

DIGRESSION: STATIC INITIALIZATION

C provides initialization of objects with static storage class by default, because it increases safety without incurring significant runtime cost.

Avoiding hidden runtime costs was one of the primary design goals of C.

The specified initialization values may be loaded directly from your program's executable file at startup, or set once by startup code generated by the compiler before `main` is called. The details vary, depending on your platform, but the outcome is the same.

DIGRESSION: STATIC INITIALIZATION

Objects with static storage, including objects in blocks (within your functions) are only initialized *once*… and *not* each time the block is executed!

Again, this happens if an explicit initialization value is provided, or not.

DIGRESSION: STATIC INITIALIZATION

It is good style to *always provide an explicit initialization value*, even if it is the same as the default initialization value the compiler would have provided for you.

Doing this does not incur any additional runtime cost at all, and helps to clarify what the author of the code was thinking.

It is *especially* important when using pointers.

EXAMPLE:
NON-EXPLICIT INITIALIZATION

```
/*
   Catastrophically wrong code!
   On most platforms you don’t actually want
   to write to memory location zero.
*/
int * p;

int main()
{
   *p = 1;
   return 0;
}
```

EXAMPLE: EXPLICIT INITIALIZATION

```
/*
   Still catastrophically wrong, but
   at least more visibly so.
*/
int * p = NULL;

int main()
{
   *p = 1;
   return 0;
}
```

THE AUTOMATIC STORAGE CLASS

If an object has automatic storage class, it means that its storage space is allocated as it comes into scope, and this storage space is discarded (and may be recycled) when it goes out of scope.

This scope corresponds to the C “block” where the object is declared and defined.

They are often allocated on the *run-time stack*, a reserved area of memory, but this is not technically required by the C standard.

THE AUTOMATIC STORAGE CLASS

In early C, the `auto` keyword was used to specify automatic storage class.

```
void f(void)
{
   auto int a;
   static int b;
}
```

Don’t use `auto` in new code. Objects declared in blocks are `auto` by default. Also, C++ now uses this keyword to mean something different: *automatic types*.

THE AUTOMATIC STORAGE CLASS

Objects with automatic storage class are *not initialized by default* and may initially contain *any pattern of bits*.

Using uninitialized objects makes your program's behavior *undefined*. An immediate crash is actually the *best-case* scenario, because you'll know something is wrong. Subtle data corruption and incorrect results are more likely.

Objects with the automatic storage class

ARE NOT INITIALIZED BY DEFAULT!

THE IMPLEMENTATION OF THE AUTOMATIC STORAGE CLASS

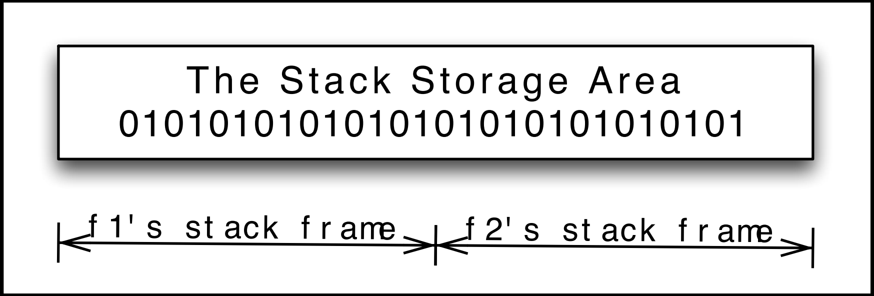
Objects with automatic storage class are traditionally allocated on the *call stack*, sometimes called the *execution* or *run-time* stack.

The stack contains *stack frames*, sometimes called *activation records*. These frames are managed automatically when functions are called (hence the name).

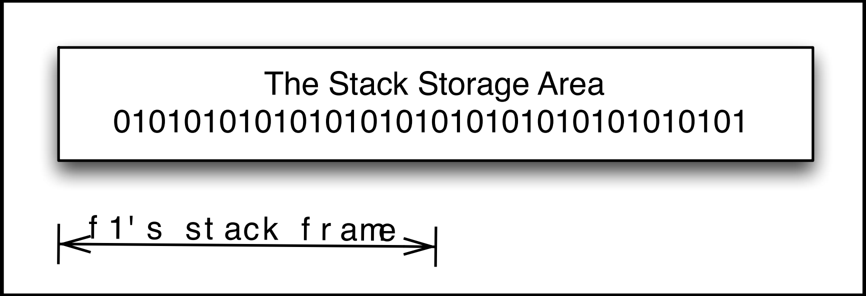
A stack frame traditionally also holds storage for function parameters and the return address (to return to the calling function).

THE IMPLEMENTATION OF THE AUTOMATIC STORAGE CLASS

If a function `f1` calls another function `f2`, then while `f2` is executing, the stack will contain frames for `f1` and `f2`.

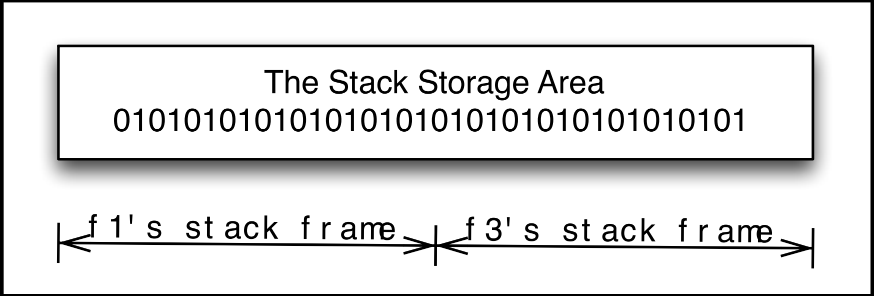


After `f2` returns to `f1`, the program no longer needs `f2`’s frame.



THE IMPLEMENTATION OF THE AUTOMATIC STORAGE CLASS

If the function `f1` then calls a third function `f3`, the runtime is free to re-use the stack space that previously held the frame for `f2`.



This has important implications for the programmer!

THE IMPLEMENTATION OF THE AUTOMATIC STORAGE CLASS

Never store a pointer to an object with automatic storage class, outside of the scope in which it is declared.

Why not?

EXAMPLE: WHAT HAPPENS WHEN YOU USE A POINTER TO AN OBJECT WITH AUTOMATIC STORAGE CLASS

```
void f1(void)
{
    int a = 1;
}

int * f2(void)
{
    int b = 2;
    return &b;
}
```

```
void f3(void)
{
    int c = 3;
}

int main()
{
    int * p = NULL;
    f1();
    p = f2();
    f3();
    printf( “%d\n”, *p );
    return 0;
}
```

GETTING THE VALUE OF AN OBJECT WITH AUTOMATIC STORAGE CLASS

In the previous example, `f2` returns a pointer which the caller then uses. This usage is not valid, but there are safe ways to get the value of an object with automatic storage class.

A function might *return* the value. Return values are always *copied*, so the destruction of the stack frame does not affect this copy.

A function might pass a pointer as a parameter, and let the called function `f2` set the value in the context of the calling function `f1`.

EXAMPLE: SETTING STORAGE OWNED BY THE CALLING FUNCTION USING A POINTER

```
void f1(int * p)
{
   *p = 99;
}

int main()
{
   int b = 0;
   f1(&b);
   printf( "%d\n", b );
   return 0;
}
```

THE IMPLEMENTATION OF THE AUTOMATIC STORAGE CLASS

Modern compilers and run-times do not always use traditional stack frames in memory to implement objects with the automatic storage class.

Compilers often store these objects in CPU registers for efficiency. On many modern platforms, parameters and return addresses are also placed in registers, depending on the platform's ABI (Application Binary Interface).

Wherever they are stored, the storage for automatic objects must not be accessed outside the block in which they are declared.

This is a fundamental part of the runtime behavior of C programs, whether the objects are stored in memory or in registers.

INITIALIZATION OF OBJECTS WITH THE AUTOMATIC STORAGE CLASS

C does not initialize objects with the automatic storage class by default.

We have seen that initialization of objects with the static storage class adds little to runtime, and that time is spent only at the start of the program.

If C supported default initialization of objects with the *automatic* storage class, this would imply that each time a function is called, the runtime must initialize those objects.

The C programming language was designed to avoids extra run-time cost, unless the programmer clearly requests it.

INITIALIZATION OF OBJECTS WITH THE AUTOMATIC STORAGE CLASS

The lack of default initialization of objects with the automatic storage class is a *major source of bugs*.

Modern compilers should warn you if you ever use an object with automatic storage class, without initializing it first.

You should get in the habit of always specifying an initial value for all objects with automatic storage class.

As I mentioned, it's not a bad idea to do this for objects with static storage class, too, for the sake of clarifying what initial value you expect.

MORE ABOUT BLOCKS

In traditional C one would typically declare the variables used inside a function only at the start of the function.

Modern C supports declaring variables closer to where you use them. This can help make your code safer and clearer.

You can actually introduce variables in any block, whether it is the block that defines a function, or a nested block.

MORE ABOUT BLOCKS

Let's say you need a variable to perform a swap, but it is only ever used inside the body of an `if` statement. You can just declare it where you use it.

```
void f1(void)
{
   /* …some code… */
   if ( a > b )
   {
      int temp = a;
      a = b;
      b = temp;
   }
   /* …some more code… */
}
```

MORE ABOUT BLOCKS

The variable `temp` goes out of scope at the end of the body of the if statement. It is treated like any other object with automatic storage class; its storage is invalid after it goes out of scope.

If you use a lot of local variables, this technique can help minimize the total amount of stack space needed by your function.

MORE ABOUT BLOCKS

Objects with automatic storage class were often used to index `for` loops, so C99 and C++ allow you to declare a variable in the initialization of the loop.

```
for (int i = 0; i < count; i++ )
{
    /* Loop body */
}
```

With this feature, we can reuse a loop variable name for multiple loops in the same function. The initialization is clear and obvious, and accidentally using the loop variable outside of the loop is impossible.

DYNAMIC LIFETIME

The third lifetime is not tied to an object storage class.

It is known as dynamic *lifetime* and uses memory allocated by `malloc`, `calloc`, or `realloc`. The memory comes from a pool called the *heap* or *free store*.

The pointers that access this dynamically-allocated memory can be objects with either static or automatic storage class.

DYNAMIC LIFETIME

The `malloc` system call reserves memory and returns a pointer (or the null pointer constant, upon failure).

The `calloc` system call behaves like `malloc`, except that the memory block is cleared (all the bytes are set to zero).

The `realloc` system call can be used to change the size of a memory block. It may copy the contents to a new location and return a different pointer, or return the null pointer constant on failure.

The `free` system call gives the memory back to the system for recycling.

DYNAMIC LIFETIME

The pointer returned by `malloc` and `calloc` should always be checked against the null pointer constant before attempting to dereference it.

Remember that the zero-filled memory returned by `calloc`, may not contain a safe default value for all types, specifically pointer types. This depends on the platform!

DYNAMIC LIFETIME

Remember that `realloc` can fail, leaving your original memory allocation intact. Therefore, you should not overwrite the original pointer with the value returned by `realloc` unless it is not null*. This makes it somewhat tricky to actually use `realloc` safely.

**If you do so, your reallocation has failed and you have lost the pointer to your original block of memory, immediately creating a memory leak.*

In C, allocation system calls return an object with the *void pointer* type (`void *`). The typing rules of C allow this object to be assigned to a pointer object of any type, without a cast. This circumvents the C type system, so be cautious!

RULES FOR DYNAMIC LIFETIME

There are a lot of rules for working with allocated memory.
Breaking any one of them can crash your program,
or silently corrupt your data.

Sometimes a crash may happen long after you broke the rule.
If you damage a system data structure,
the problem might not show up immediately!

RULES FOR DYNAMIC LIFETIME

- Your program must not assume that the allocated memory has been set to zero, unless you called `calloc`.
- If the allocation fails, your program must not dereference the returned `null` pointer.
- After the memory is deallocated using `free`, your program must not access it.
- After calling `free`, your program must not call `free` again on the same pointer.
- Your program should `free` every block of memory it successfully allocates. *(This may be controversial. At a minimum, it should deallocate memory allocated on the fly, as opposed to only at startup).*

MEMORY LEAKS

If your program loses track of a pointer to allocated memory before deallocating it, the memory is *leaked* and your program cannot get it back.

Deallocated memory is not returned to the operating system immediately. The heap can only *grow* during the lifetime of your program.

When your program terminates, all its memory (including any leaked memory) will be returned to the operating system.

A program that leaks memory may impair not only its own performance, but also the performance of other programs running on the same computer.

DYNAMIC LIFETIME

This lifetime does not apply to the objects *in your program*. The variables in your program have the same two storage classes they always had.

This lifetime refers to the run-time state of the *heap*, where memory is made available temporarily for your program's use.

Your program's objects may *point to* this memory and use it.

This "disconnect" between the C *storage class* lifetime of your pointer object, and the dynamic lifetime of the memory you point to, is the fundamental cause of *many* run-time errors in C programs.

EXAMPLE: DYNAMIC LIFETIME

```
void f(void)
{
    int * a = malloc(100 * sizeof(int));
    if ( NULL != a )
    {
        /* Do something with this memory */
        free(a);
    }
    else
    {
       /* Report an allocation error here */
    }
}
```

DYNAMIC LIFETIME

In the previous example, the pointer has automatic storage class, but the memory pointed to does *not*. C does not have support for guaranteeing that the pointer cannot be used when in an invalid state. Your code has to avoid this "manually."

This mismatch in lifetimes creates an opportunity for several critical errors. If the pointer is dereferenced before it is set to the return value of a successful allocation, the program will access invalid memory.

If the pointer's block ends before `free` is called, the pointer goes out of scope and its lifetime ends, but the allocated memory has been leaked.

DYNAMIC LIFETIME

In the previous example, the allocation is checked to make sure it succeeds, and `free` is called only after successful allocation. The code is simple enough that obvious errors are clearly avoided.

But in real programs, dynamically allocated memory is rarely used like this. Generally, we use an object with static storage class to point to it, so that it can be accessed from different functions at different times.

The rules for safely using objects of dynamic lifetime must be obeyed *at every point* in the program's lifetime. This can require a considerable amount of care and bookkeeping.

DYNAMIC LIFETIME

"Whenever you write `malloc`, write the corresponding `free`. If you don't know where to put the `free` that corresponds to your `malloc`, then you've probably created a memory leak!"
[Peter van der Linden, *Expert C Programming: Deep C Secrets*, p. 193]

EXAMPLE: FREEING THE ELEMENTS OF A LINKED LIST (INCORRECTLY)

```
node * p = head_p;
while (p != NULL)
{
   free(p);
   p = p->next_p;
}
```

EXAMPLE: FREEING THE ELEMENTS OF A LINKED LIST (CORRECTLY)

```
node * p = head_p;
while (p != NULL)
{
   node * next_p = p->next_p;
   free(p);
   p = next_p;
}
```

DYNAMIC LIFETIME

"The reason people make these mistakes is typically
not maliciousness and often not even simple sloppiness;
it is genuinely hard to consistently deallocate every allocated object
in a large program (once and at the right time in a computation)."
[Bjarne Stroustrup, *The C++ Programming Language, 4th Ed.*, p. 279]

WHAT I COVERED (1)

- 3 Object Lifetimes (2 are also C storage classes)
  - static
  - automatic
  - dynamic
- Declarations and definitions
- External linkage
- Static initialization: implicit and explicit
- Don’t say “global” when you’re talking about C

WHAT I COVERED (2)

- Blocks
- Implementation of the automatic storage class, and implications
- The heap or “free store”
- `malloc`, `free`, and family
- Memory management and memory leaks

QUESTIONS?