Systems 3

C Review and Summary

Marcel Waldvogel

(Handout)

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

Chapter Goals

- Putting it all together
- Use the knowledge for memory allocation
- Summary of the C programming part

Photo by David Edelstein on Unsplash

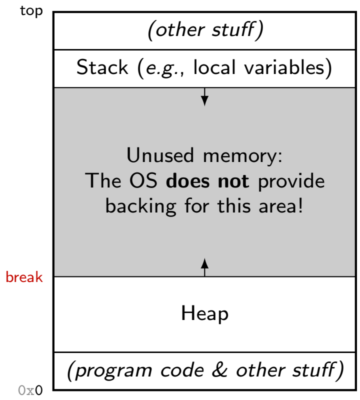
Excursus: Inside Malloc

This excursus demonstrates:

- How to impose a meaning on a region of memory.
- Heavy use of pointer arithmetics.
- Glimpse under the hood of memory allocation.

Process memory layout

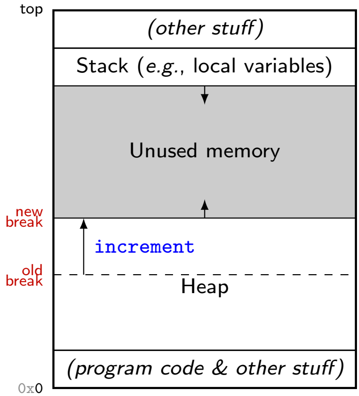
The program's view of its memory: **virtual RAM**.



(simplified picture)

- The stack contains the variables local to a function call. Grows downwards.
- The **program break** is the first location after the program's *data segment* (*cf.* later).
- Incrementing the break is **allocating heap memory**: Ask the OS to provide backing for the increased consumption of memory.

Allocating heap memory



- sbrk(2) can **move the break**[1].

```
void *sbrk(intptr_t increment);
```

- Returns address of old break, or `(void *)-1` on error.
- If the break was *increased*, then the returned value is a pointer to **newly allocated** memory, backed by the OS!
- (There are other system calls to get memory from the OS, *cf.* later)

Note Avoid using `brk()` and `sbrk()`: the `malloc(3)` memory allocation package is the portable and comfortable way of allocating memory.

[1] there's also brk(2), for the same purpose — we use sbrk(2) only.

Implementing a memory allocator

How to write your own `malloc`[3]

- We know that we can get **fresh memory** from the OS via `sbrk(2)`.
- "The real" `malloc(3)` uses this, and other techniques. There are many different, *very sophisticated* implementations of memory allocators.
- We implement a **very simple** allocator.[2] Most prominently, we ignore data **alignment**.

The Interface

```
void *kr_malloc(size_t b);
void kr_free(void *ap);
```

- `kr_malloc` allocates `b` bytes and returns a pointer to the allocated memory, or `NULL` on error.
- `kr_free` frees memory pointed to by `ap`, which must have been returned by a previous call to `kr_malloc`.

[2] adapted from: Kernighan, Ritchie. The C Programming Language. Prentice Hall Software Series. Section 8.7, *A Storage Allocator*.

[3] Just because you can — in general, this is not a smart idea.

The rough plan

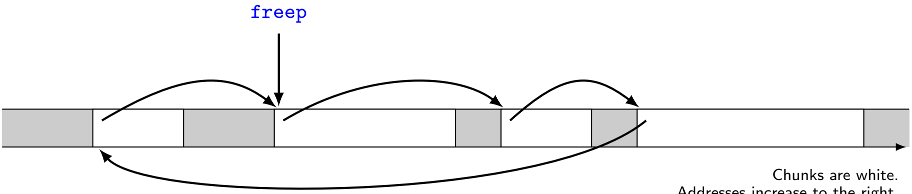
- System calls are **expensive**: Avoid using them often.
- So `kr_malloc` tries to get **a big chunk** of memory from the OS, and hands **smaller pieces** of that to the calling program.
- So we need to maintain a **list of free memory chunks**, that
  - have been allocated **from the OS** via `sbrk(2)`, but
  - have **not yet** been handed to the **program**,
  - or have been **returned from** the program.
- If the program frees memory, `kr_free` adds that to the list of free memory, but does not return it to the OS.
  - Obvious weakness: Memory consumption of the process never shrinks.

Note The functions `sbrk(2)`, `kr_malloc` and `kr_free` implement a concept of **transferring ownership** of memory between the OS, the allocator, and the program.

Chunks of free memory

Our allocator maintains a list of **free memory chunks**. These are not currently used by the program, *i.e.* they

- lie in memory allocated from the OS via `sbrk`,
- have not been given to the program by returning from `kr_malloc`, or have been given back to the allocator via `kr_free`.



Chunks are white.
Addresses increase to the right.

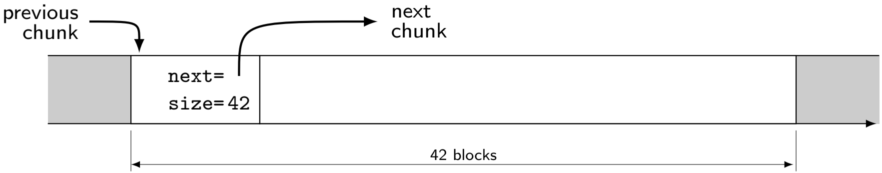
- **Circular list**: Every chunk points to the next chunk.
- List is **ordered by memory address**, with the obvious exception.
- A pointer `freep` is the **entry point** into the list. It may point to any chunk, and we will move it around quite a bit.

To maintain the list of free chunks, we install a header **at the start** of each chunk:

```
typedef struct header Header;
struct header {
    Header *next;
    size_t size;
};

#define BLOCKSIZE (sizeof(Header))
```

- `next` points to the next chunk in the circular list.
- `size` is the size of the **entire** chunk, given in the unit `BLOCKSIZE` bytes.



Questions If `Header *p` points to the header, then

- where does `p + 1` point to?
- where does `p + p->size` point to?

The `kr_malloc()` function

```c
Header *freep = NULL,   /* a global pointer to the free list */
       base;            /* and a dummy for the empty list */

void *kr_malloc(size_t bytes) {

    /* number of blocks required, including one more for the header */
    size_t reqd = 1 + (bytes + BLOCKSIZE - 1) / BLOCKSIZE;
    Header *prevp = freep; /* ptr to previous chunk */

    if (!freep) {                    /* make empty list if called for the first time */
        base.next = freep = prevp = &base;
        base.size = 0;
    }

    for (Header *p = prevp->next; ; prevp = p, p = p->next) {
```

Check the chunk `*p`. `return` a pointer if it is big enough. See the following slides.

```c
        if (p == freep) {       /* if we have unsuccessfully traversed the whole list... */
            p = morecore(reqd);          /* ...get more from the OS (cf. page 18)... */
            if (p == NULL) return NULL;                          /* ...or fail. */
        }
    }
}
```

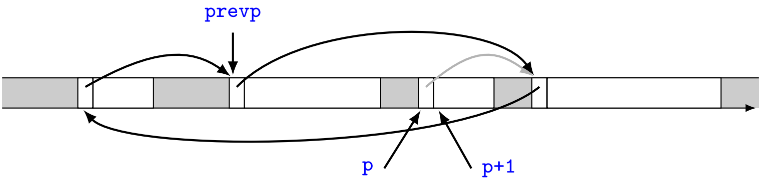
kr_malloc() — using a chunk that fits exactly

- Header *p points to current chunk, *prevp to the previous one.
- We need reqd blocks of free memory.

```
        if (p->size >= reqd) {  /* this chunk is large enough */
            if (p->size == reqd)  /* it fits exactly */
                prevp->next = p->next;  /* remove chunk from free list */
            else {
```

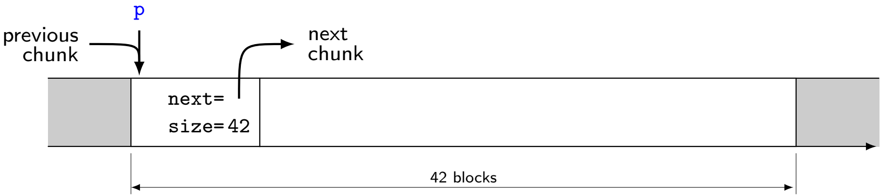
Split the chunk p points to. See the following slides.

```
            }
            freep = prevp;  /* next search continues from here */
            return p + 1;  /* memory address the program may write to */
        }
```

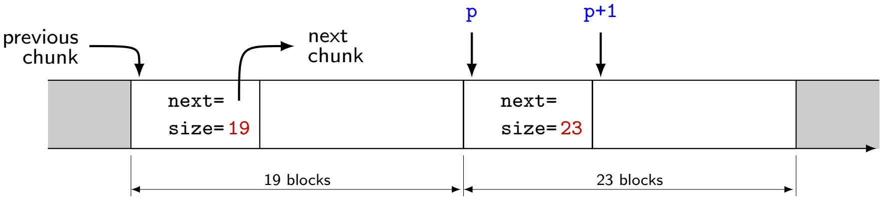


kr_malloc() — split a chunk that is too large

Assume we need reqd = 23 blocks, but the chunk has 42...




```
        p->size -= reqd;
        p += p->size;
        p->size = reqd;        /* Enjoy: We just make up a header here! */
```

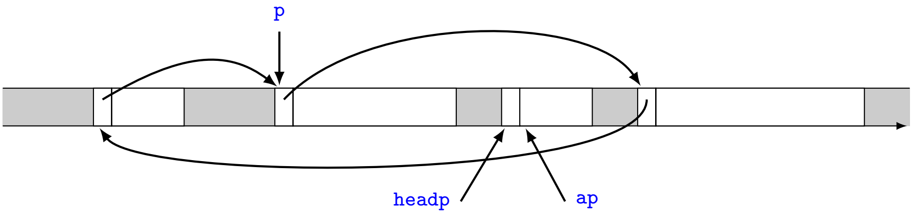


The kr_free() function

Where in the list should the freed chunk be linked?

```
void kr_free(void *ap) {
    Header *p,
      *headp = (Header *)ap - 1;  /* determine header of chunk *ap */

     /* Find p so, that headp belongs between p and p->next. */
    for (p = freep; !(p < headp && headp < p->next); p = p->next)
        if (p >= p->next && (p < headp || headp < p->next)) break;
```

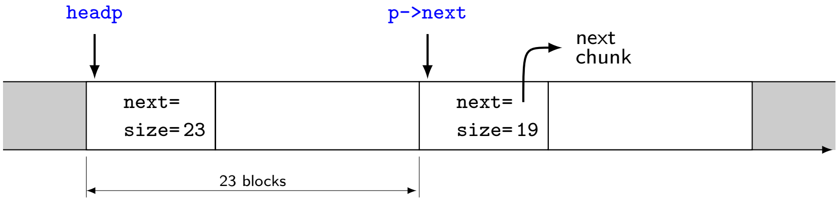


Question Now it would be easy to hook the freed chunk into the list:

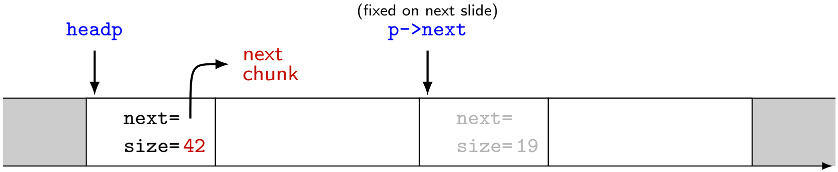
`headp->next = p->next;`   `p->next = headp;`

Why may this not be the smartest thing to do?

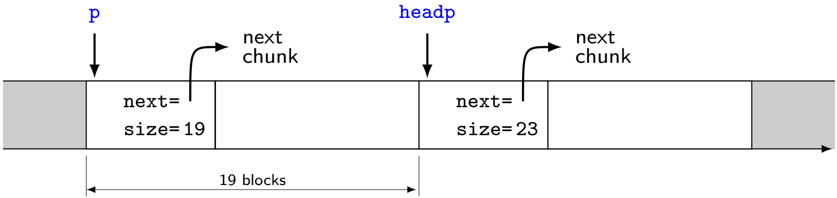
kr_free() — fuse/link with the following chunk



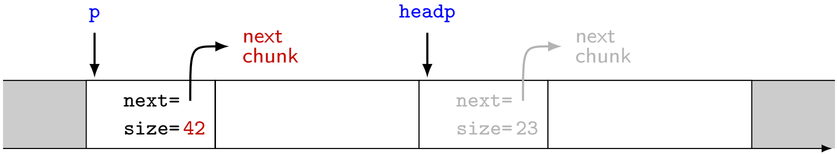

```
    /* If the chunk is adjacent to the following chunk, fuse the two into one... */
    if (headp + headp->size == p->next) {
        headp->size += p->next->size;
        headp->next = p->next->next;
    } else
        headp->next = p->next;  /* ...otherwise just link without fusing. */
```



kr_free() — fuse/link with the previous chunk




```
    /* If the chunk is adjacent to the previous chunk, fuse the two into one... */
    if (p + p->size == headp) {
        p->size += headp->size;
        p->next = headp->next;                    /* Exercise: Why is this required? */
    } else
        p->next = headp;  /* ...otherwise just link without fusing. */
```



kr_free() — after linking into the list

```
    /* We set 'freep' to point just before, or at the freed chunk.  Used by morecore. */
    freep = p;
}
```

The `morecore()` function

```
/* always get at least NALLOC blocks from the OS */
#define NALLOC 10240

Header *morecore(size_t reqd)
{
    if (reqd < NALLOC) reqd = NALLOC;

    /* Actually get memory from the OS. */
    Header *p = sbrk((intptr_t)(reqd * BLOCKSIZE));
    if (p == (void *)-1) return NULL;

    p->size = reqd;

    /* We simply call kr_free to do the linking. */
    kr_free(p + 1);

    /* kr_free makes freep point just before, or at the new chunk. */
    return freep;
}
```

Question Why do we call `kr_free` with `p+1` instead of `p`?

Photo by Joel Muniz on Unsplash

"Just because you can ..."

Weaknesses

- Not thread-safe!
- Not thread-aware, not thread-optimized
- Mixes sizes (and lifetimes) arbitrarily (fragmentation!)
- Does not deal with user bugs
- $\mathcal{O}(n)$

GNU `malloc`

- History and Overview
- Tricks to use
- Data structures and multiple regions
- Tracing
- Debugging tools/libraries

C Basics

- Differences between C and Java
- How registers, stack, and memory are used

- No overflow
- Little type checking
- No array bounds checks
- Manage memory yourself

C popularity

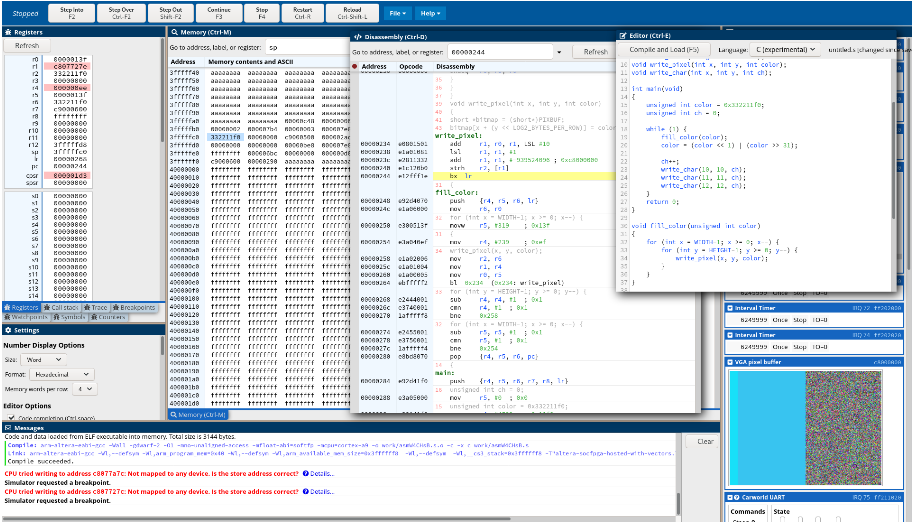
- Requirements that make C mandatory:
  - embedded systems (close to hardware, scarce resources)
  - extreme performance (better usage of resources)
  - the world is built on C and C++ (with C++ being a superset of C)
    — Herb Sutter. C++ and Beyond.[4]
  - C is simple & powerful
    — Damien Katz (CouchDB). The Unreasonable Effectiveness of C.[5]
- Programming Languages Rankings
  - 1st/2nd place in TIOBE[6] (1989—2019)
  - 8th/9th place in RedMonk[7], with C++ ranking 5th—7th (2012—2019)

[4] https://www.youtube.com/watch?v=xcwxGzbTyms
[5] http://damienkatz.net/2013/01/the_unreasonable_effectiveness_of_c.html
[6] https://www.tiobe.com/tiobe-index/
[7] https://redmonk.com/kfitzpatrick/2019/07/31/june-2019-redmonk-programming-language-rankings-monkchat/

C vs. Java

C	Java
∼1970, procedural, low(er)-level	1995, object-oriented, high-level
compiled to machine code	compiled to byte code
suitable for systems programming	—
explicit `free()`	garbage collection
explicit pointers (+arithmetic)	implicit pointers in object variables
—	native threading
type casting	type checking
preprocessor	method overloading
default public	default private
global variables	—
`goto` statement	—
`struct`, `union`, bitfields	object
varargs	—

Register, Stack, Memory



Function call

Calling conventions (CPU, ABI dependent)

1. Some registers are flushed (callee-modified ones)
2. Some parameters are passed in registers (especially the first ones), ...
3. ... some on the stack (especially varargs, see `printf(3)`)
4. Return values are typically in a register

5. Return address is placed on stack/in register
6. Caller typically cleans the stack (especially for varargs)

Function call (cont'd)

Callee job (CPU, ABI dependent)

1. Frame pointer (FP) is pushed onto stack
2. FP $\leftarrow$ SP (args above, locals below)
3. Some registers are saved (callee-preserved ones)
4. Space for local variables is reserved on the stack

5. Return value is put into designated space/register
6. SP $\leftarrow$ FP (free locals)
7. Pop old FP from stack
8. Return to return address (stack or register)

Photo by Stephan Henning on Unsplash

Pointers

- Duality array/pointer
- `sizeof` on them
- `const` (and string literals)
- `volatile`
- Function pointers

Data types and sizes

Sizes are machine-dependent

- Each compiler is free to choose **appropriate sizes** for its own hardware. ISO C defines compile-time limits:
  - `char` is *at least* 8 bits (`CHAR_BIT`)
  - `short` and `int` are *at least* 16 bits
  - `long` is *at least* 32 bits
  - `short` is no longer than `int`, `int` is no longer than `long`
- Can be obtained with the `sizeof` operator.
- Numerical limits[8] are documented in `<limits.h>` and `<float.h>`. Additional limits are specified in `<stdint.h>`[9]

On my machine

char	1
short int	2
int	4
long int	8
long long int	8
float	4
double	8
long double	16
void *	8

[8] ISO C99 : 7.10/5.2.4.2 : Numerical limits

[9] ISO C99 : 7.18 : Integer Types (see also https://en.wikipedia.org/wiki/C_data_types#Basic_types)

Arrays and Pointers

- An array variable never is an l-value (*i.e.*, one cannot assign to it).
- The value of an array variable is the **address of the first element**.

```
int a[2];
int *pi = a;  /* the same as &a[0] */
```

Exceptions If the array is…

- …operand of `sizeof`, the size of the array is returned,

```
printf("%zu\n", sizeof(a));    /* size in chars, not array cells */
```

- …operand of `&`, the address of the first element is returned, typed as "pointer to array"

```
int (*pi2)[] = &a;        /* we will come back to this later */
```

- …a literal string initializer for a character array, the array is initialised with the string.

```
char a[] = "Hello world";
```

`sizeof` and pointers, arrays

1. Operator, not function
2. Array-valued arguments are pointers(!) → use as pointers only!

```c
#include <stdio.h>
#define PS(x) printf("%s: sizeof %s=%zd", \
    __func__, #x, sizeof(x))

void arrsize(char *a0, char a1[100])
{
  char buf[100];
  PS(a0); PS(a1); PS(buf);
}

int main(int argc, char **argv)
{
  PS(argc);    PS(argv);
  PS(argv[0]); PS(argv[0][0]);
  arrsize(argv[0], argv[1]);
  return 0;
}
```

```
$ ./sizeof
main: sizeof argc=4
main: sizeof argv=8
main: sizeof argv[0]=8
main: sizeof argv[0][0]=1
arrsize: sizeof a0=8
arrsize: sizeof a1=8
arrsize: sizeof buf=100
```

For `someType arr[1000]`, `sizeof a[0]` is the size of an element and `sizeof a/sizeof a[0]` is the number of elements.

Valid Pointer Operations

Legal pointer operations summarized

- Assignment of pointers of the same type, or `void *`.
- Assigning or comparing to `NULL`.
- Adding or subtracting a pointer and an integer.
- Subtracting or comparing pointers to members of the **same array** (or same memory area as returned by e.g. `malloc`.

Illegal pointer operations

- Multiply, divide, shift, or mask pointers.
- Add `float` or `double` to pointers.
- Assign a pointer of one type to a pointer of another type without cast (exception is `void *`).
- Subtracting or comparing pointers to members of **different arrays**, or not pointing to arrays at all.

A function can **promise not to modify** a value passed by reference:

```c
1  #include <stdio.h>
2
3  int nice(int const * x)
4  {
5     /* *x = 3; */  /* causes error */
6     return *x + 2;
7  }
8
9  int sloppy(int * x)
10 {
11    return *x + 2;
12 }
```

```c
13 int main(void)
14 {
15    int i = 12;
16    int const j = 23;
17
18    nice(&i);
19    nice(&j);
20    sloppy(&j);  /* causes warning */
21
22    printf("%d\n", i);
23    return 0;
24 }
```

- Passing a reference to a constant object to a function that does not promise not to modify it, causes a **warning**! (line 20)
- A **string literal** in C is constant, and must not be written to!

```c
int stringlen(char * foo);
stringlen("hello");  /* warning */
```

```c
int stringlen(char const * foo);
stringlen("hello");  /* fine */
```

Cast away `const` — pun intended

- Review the warning issued by line 20 on the previous slide:

```
const2.c:28:2: warning: passing argument 1 of ‘sloppy’ discards ‘const’
qualifier from pointer target type [enabled by default]
sloppy(&j);
```

- If you
  - absolutely must use that function (it may come from a library),
  - and you absolutely know that it will not change the value
  - and you absolutely cannot create a copy and pass that instead,

  then you may cast the type into a non-`const` one:

```c
int sloppy(int * x)
{
   return *x + 2;
}
int modify(int * x)
{
   (*x)++;
   return *x+2;
}
```

```c
int main(void)
{
   int const j = 23;

   sloppy((int*)&j);  /* no warning */
   modify((int*)&j);  /* you're on your own */
   printf("%d\n", j);

   return 0;
}
```

Thread-shared variables

Optimizer

The compiler's optimizer tries to remove unnecessary instructions and memory accesses.

`for (int i=0; i<N; i++) x++;` → `x += N;`

Memory accesses can be forced with `volatile`.

When to `volatile` a variable?

- When a variable is shared between multiple threads
- ...with an interrupt handler
- ...with a signal handler
- ...with hardware

Fields, identifiers, tags

The names and tags you use in a C program, live in different **namespaces**.

- **Identifiers** of variables and types share one namespace.
  ⇒ You cannot name a variable like a type (*e.g.*, `int int;`)

  (There is an exception, but simply don't do it!)

- **Field names** are like variables, with a scope limited to that struct.

```
struct { int foo; } x;  x.foo = 3;
struct { char foo; } y;  y.foo = 'w';
double foo = 3.14;
```

- **Tags** have their own namespace, which is *shared* by unions, structs, and enumerations.

```
struct point { int x; int y; };
char point = '.';  /* valid */
enum point { infinity, closeby };  /* invalid redefinition of the tag point */
```

Tag names and identifiers are limited to the scope they are defined in.

Easily spotted mistakes

Some observations about **parentheses** in declarations (Note: ... is a meta-placeholder!):

- **Invalid types**, *i.e.*, in a type declaration, you will **never** see

  `foo(...)(...)` Functions cannot return functions.

  `foo(...)[...]` Functions cannot return arrays.

  `foo[...](...)` Arrays cannot contain functions.

- **Valid types**

  `int bar[...][...];` `bar` is an array of arrays.

  `int (*fun(...))(...);` Function `fun` returns a *pointer to* a function.

  `int (*foo(...))[];` Function `foo` returns a *pointer to* an array.

  `int (*arr[...])(...);` `arr` is an array of *pointers to* functions.


Photo by Drew Hays on Unsplash

Big Programs

- Scopes and lifetimes
- `static` in functions
- `#include` interface in implementation as well

Unscrambling C declarations

Precedence rules for reading C declarations.

1. Parentheses group parts of the declaration.
2. Read **type specifiers** as atomic tokens, *e.g.*,
   - `double`,
   - `struct foo`, or
   - `unsigned short int`.
3. The keyword `const`:
   - If *next to* a type specifier, it belongs to that, making the **value** constant.
   - Otherwise, it belongs to the asterisk to its *left*, making the **pointer** constant.
4. The **postfix** operators, being one of
   - **parentheses** `(...)` indicating a function, or
   - **brackets** `[...]` indicating an array.
5. The **prefix** operator **asterisk** `*` indicating a pointer.

Note Inside parenthesis, a declaration may contain *further* declarations of function arguments! These do *not necessarily* have a name.

An algorithm for reading declarations

1. Start at the leftmost identifier that is *not* a type specifier. That is being declared.
2. Do not leave parenthesis while:
   1. Handle the **postfix** operators, *i.e.*, optional `(...)` or `[...]` to the **right**, do so from left to right.
      - For a function, apply the whole algorithm to each parameter.
      - For an array, optionally note the size.
   2. Handle the **prefix** operators `*` to the **left**, do so from right to left.
3. If inside parenthesis, leave them, and restart with 2.
4. Read the **type specifier** on the left.

tl;dr — look right, look left.

Example `int *(*list[42])(void)`

- `int *(*list[42])(void)` `list` is...
- `int *(*list[42])(void)` ...an array of 42...
- `int *(*list[42])(void)` ...pointers to

Leaving parenthesis, we're done with them. Goto step 2 of algorithm:

- `int *(*list[42])(void)` ...function of ...
- `int *(*list[42])(void)` ...no arguments...
- `int *(*list[42])(void)` ...returning a pointer to...
- `int *(*list[42])(void)` ...an integer.

Example `int (*f)(const char *s)`

- `int (*f)(const char *s)` `f` is...
- `int (*f)(const char *s)` ...a pointer to...
- `int (*f)(const char *s)` ...a function of **(**...
- `int (*f)(const char *s)` ...`s`, which is...
- `int (*f)(const char *s)` ...a pointer to...
- `int (*f)(const char *s)` ...a `const`ant `char`acter **)**...
- `int (*f)(const char *s)` ...returning an integer.

Example `void f(char *x[])`

- `void f(char *x[])` `f` is a...
- `void f(char *x[])` ...function of **(**...
- `void f(char *x[])` ...`x`, which is...
- `void f(char *x[])` ...an array of unspecified size of...
- `void f(char *x[])` ...pointers to...
- `void f(char *x[])` ...`char`acter **)**...
- `void f(char *x[])` ...not returning anything.

The declaration `void f(char **x)` is equivalent, specifying array dimensions does not make any sense in this case (*cf.* page 31).

Example `void *f(char *(*p)[5])`

- `void *f(char *(*p)[5])` `f` is a...
- `void *f(char *(*p)[5])` ...function of **(**...
- `void *f(char *(*p)[5])` ...`p`, which is...
- `void *f(char *(*p)[5])` ...a pointer to...
- `void *f(char *(*p)[5])` ...an array of five...
- `void *f(char *(*p)[5])` ...pointers to...
- `void *f(char *(*p)[5])` ...character **)**...
- `void *f(char *(*p)[5])` ...returning a pointer to...
- `void *f(char *(*p)[5])` ...data of unspecified type.

In this case, specifying the array dimensions makes sense: In the body of `f`, `sizeof(*p)` will return 40 if the size of a pointer is 8. This also effects pointer arithmetics on `p`.

Note Function parameters need not be named in a **declaration**!

`double (*f)(double x)` ≡ `double (*f)(double)`

This makes it occasionally hard to find out what is being declared.

Example `int f(char *[])` (Example from page 44)

- `int f(char *[])` `f` is a…
- `int f(char *[])` …function of **(**…

No identifier: So *"it"* is to the right of all `*`, and to the left of all `(…)` and `[…]`.

- `int f(char *[])` …an array of…
- `int f(char *[])` …pointers to…
- `int f(char *[])` …`char`acter **)**…
- `int f(char *[])` …returning an `int`eger.

This is actually equivalent to `int f(char **)`.

Question What is this: `int f(char (*)[23])` ?

Easily spotted mistakes

Some observations about **parentheses** in declarations (Note: … is a meta-placeholder!):

- **Invalid types**, *i.e.*, in a type declaration, you will **never** see

  `foo(…)(…)` Functions cannot return functions.

  `foo(…)[…]` Functions cannot return arrays.

  `foo[…](…)` Arrays cannot contain functions.

- **Valid types**

  `int bar[…][…];` `bar` is an array of arrays.

  `int (*fun(…))(…);` Function `fun` returns a *pointer to* a function.

  `int (*foo(…))[];` Function `foo` returns a *pointer to* an array.

  `int (*arr[…])(…);` `arr` is an array of *pointers to* functions.

Lexical Scope

- An identifier (*e.g.*, a function name, a variable, a structure tag, ...) must be **in scope** to be used.

- The scope of an identifier which is...
  - ...declared inside a block $\{ \cdot \}$, extends from the end of the declaration to the end of that block. These are called **local**, or sometimes *internal* variables.
  - ...declared as parameter in a function definition, extends to the body of that function. These are also local variables.
  - ...declared at toplevel (*i.e.*, outside any function definition), extends from the end of the declaration to the end of the **compilation unit**[10]. These are called **global**, or sometimes *external* variables.

- Variables in (syntactically) inner scopes **shadow** variables of the same name in outer scopes.

[10] roughly: the current file; more exact: see later

Questions

- What identifiers are declared, and what is their scope?
- Why is it good to declare a variable as late as possible? Why is it bad?
- What is wrong in this example?

```c
int f(void) {
  return y++;
}

int y = 1, x = 2;

int g(void) {
  int c = f();
  return x + c;
}
```

Storage classes

- A **declaration** brings something into scope, describing its nature.
- But a **definition** reserves **storage** for it.
- All variable declarations we have seen so far were implicit definitions!

There are alternatives:

- The **storage class** of an object describes the **lifetime** and **visibility** of a variable. Further details, *e.g.*, initialization, depend on that.
- A declaration can be modified with a storage classes **specifier**:
  - `auto`,
  - `static`,
  - `extern`,
  - `register`, and
  - yeah, well, `typedef` — a rather odd one here! Defining a type, instead of doing anything with a variable.

Automatic variables

- **Storage** for automatic variables is reserved *automatically* for each call of the function, and is reserved only until the function returns.
- **Local** variables default to storage class `auto`.
- They will contain garbage if they are not initialized.

Example

```
1 int f(int x)            /* x is an automatic variable */
2 {
3   int y = 42;           /* y is an automatic variable */
4   auto int z = 23;      /* z is an automatic variable */
5   ...
```

- One may explicitly declare a variable as automatic, using the `auto` **keyword**, as in line 5.
- Rarely used, because this is the **default**. (backwards compatibility) (compare to, e.g. FORTRAN)

Static objects

- *If in scope*, **external** objects can be accessed by name by any function, **anywhere** in the program.
  By default, even from other **compilation units**.
- External variables can be used instead of argument lists to **communicate data** between functions. *(prone to errors)*
- External variables retain their values between function calls: Their **lifetime** spans the program's entire **runtime**.

$\Rightarrow$ They have **static storage**.

Local declaration of external variables

Sometimes, we know about the existence of an **external object**, but it is not yet in scope.

- An external object can be **brought into scope**, by *declaring* it with the keyword `extern`.
- A declaration of an external object **is not a definition**. It only states the type of the object, and brings it into scope.
- Such an object must be **defined elsewhere**, exactly once, outside a function. This then reserves storage for it.

Example

```
int f(void) {
  extern int y;   /* declare variable y that is defined elsewhere */
  return y++;
}

int y = 1,  /* declare, define and initialize variable y */
    x = 2;

int g(void) {
  int c = f();
  return x + c;
}
```

Note `extern` does not define an external variable — it requires one!

Note Use of externs is discouraged in the Linux kernel. To allow their use in the exercises, we have added the flag `--ignore AVOID_EXTERNS` when calling `checkpath.pl`.

Static local variables

- Sometimes, one wants variables that **retain their value** between function calls (*i.e.*, have static storage), but are **not accessible** from outside the function.
- A **local variable** declared with the keyword `static`, has the **lifetime** of an external variable, but the **scope** of a local variable.
  - You can have *different* static variables with the **same name** in *different* functions. (provides **encapsulation**, and stops **namespace pollution**.)
  - You may `return` **pointers** to static variables, and use them outside the function defining the static variable.
- Static variables are **initialized exactly once**, defaulting to zero if no other value is given.

Example

```
int f(void)  /* this function never returns the same value twice */
{
  static int y;      /* initialized to zero at program start */
  return y++;
}
```

Static global objects

- The **visibility** of *global* objects can be limited to the current compilation unit with the keyword `static`.

Confusion warning

Is `static` something else for local *vs.* global variables?

- External and static local variables are handled in a very similar way: Their storage is allocated for the entire lifetime of the program.
- The difference is their visibility, and accessibility.
- **Roughly**, `static` always means:
  - Lifetime until program ends (entirely correct).
  - Accessibility limited to scope if local, or to module if global (beware of pointers, though).

Register variables

A register variable is declared with the keyword `register`.

- **Hint** to the compiler that the variable in question will be heavily used. The idea is to place it in a **machine register**.
- Can only be used with **automatic** variables.
- Not possible to take the address of a register variable.

But

- This is not the place to start optimizing your code.
- Compilers are free to **ignore** the advice.
- Compilers are usually **very smart** about where to store variables.

⇒ This is rarely used.

Initialisation

Automatic variables

- May be initialized when they are defined, otherwise they contain **garbage**.
- When declared and initialized in a block they are initialized **each time** the block is entered.

External and static variables

- **Guaranteed to be initialized** to default values (zero if unspecified).
- Initializer must be a **constant expression**, *i.e.*, known at compile time.
- Initialization is done once, **before** the program begins execution.

Summary

Storage	Level	Visibility	Lifetime	Initialisation
`static`	file	file→	full	once
	block	block→	full	once
`extern`	file	file→	(inherited)	N/A
	block	block→	(inherited)	N/A
Other	file	file→	full	once
	block	block→	block	every time

- Function parameters behave as if defined inside the function block
- Loop definitions behave as if the loop was enclosed in another block:

```
for (int i = 0; i < 10; i++) { body; }
{int i = 0; for (; i < 10; i++) { body; } }
```

Photo by Zhipeng Ya on Unsplash

Macros

- #define for constants
- #include
- #if/#ifdef/#ifndef
- function-like definitions (max etc.)
  - Possible double/multiple evaluation
  - inline functions are as efficient (-finline-functions, -O3)
  - Exception: macros like assert(3) with string/symbol concatenation, stringification, access to __LINE__ etc.
  - Exception: for code generation (command list, function list), if it is worthwhile

Macros with arguments

- A directive of the form

```
#define name( identifier[,identifier] ) token...
```

where there is **no space** between the name and the '(', is a macro definition with parameters given by the identifier list.

Example

```
#define isupper(c) ((c) >= 'A' && (c) <='Z')
```

- Why are there so many parenthesis?
- Why is there no ; at the end?

Example

Avoid the overhead of a function call ⇒ faster?

```
#define square(x) ((x) * (x))
double y = square(read_num_from(stdin));
```

- What do you think?

Concatenation

- Normally, CPP operates at the **granularity** of C tokens. (That's why the input should be lexically valid C code)
- The ## operator allows to **concatenate** two tokens, when used in a macro body.

Example

```c
struct command {
  char *name;
  void (*function) (void);
};

struct command commands[] = {
  { "quit", quit_command },
  { "help", help_command },
  { "calc", calc_command },
  /* ... (hundreds more) */
};
```

```c
struct command {
  char *name;
  void (*function) (void);
};

#define COMMAND(NAME) \
  { #NAME, NAME ## _command }

struct command commands[] = {
  COMMAND(quit),
  COMMAND(help),
  COMMAND(calc),
  /* ... */
};
```

The take-home message for programming

Today, there is no need for `#define`ing "optimized" function-like macros (e.g., `max(a,b)`) with their multiple-evaluation, precedence and semicolon problems.
(They are still useful if you need access to compile-time macros such as `__LINE__` or `__FILE__` (see `assert()`) or when symbol concatenation or stringification is needed (e.g. for variable/code generation).

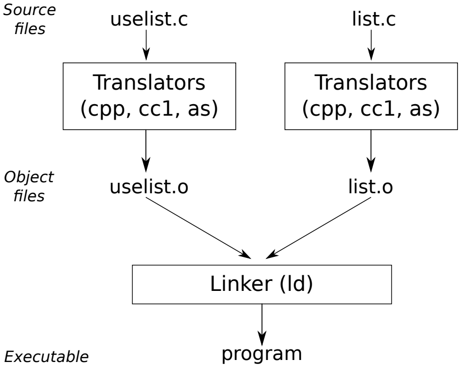
Photo by Shao Jie on Unsplash

Linking

- Minimal error checking (→ `#include "myself.h"`)
  1. Define variables, functions once (`.c`)
  2. Define macros once (`.h`)
  3. `#include .h` also in implementation
- Symbol resolution is one-way street → do create library hierarchy (tree, DAG)
- Remember shared libraries with their size, update advantages and plugin possibilities

GNU Compiler Collection

- We have already seen how separate compilation works (Lecture on 'Big Programs').
- The compiler driver gcc(1) employs a bunch of different tools for this task:
  - preprocessor cpp(1) — removes comments, applies macros.
  - compiler cc1 — compiles into assembler code.
  - assembler as(1) — translates into binary object file.
  - linker ld(1) — **links together the compiled object files**.



Linker Symbols

Relocatable object files come with a **symbol table**, that lists all the symbols an object file exposes.

- **Global** symbols are defined in the object file, and may be referenced from other object files (no modifier).
- **External** symbols are referenced by the object file, but not defined. *I.e.*, the definition must be provided in another object file (`extern`).
- **Local** symbols are defined and referenced only from within the object file (`static` or compiler-generated).

Note *Local symbols* have nothing to do with function-local variables in a C-program. Unless `static`, they are never visible in the symbol table. (Compare debugger symbols.)

Symbol resolution

- For each **local symbol**, the compiler guarantees exactly one definition. The name is modified to be unique (*e.g.* `count` above).
- If *the compiler* **finds no definition**, it expects it to come from another module, and leaves it to the linker, (*e.g.* `buf` above).
- When **the linker** resolves *global* symbols, several conditions can occur:
  - **No definition** is found in the symbol table of any input object file.
  - **Multiple definitions** are found in different object files, choose one.

Example No `main` function, and `buf` undefined.

```
$ gcc swap.o #without -c, try to build an executable
.../lib/crt1.o: In function ‘_start':
(.text+0x20): undefined reference to ‘main'
swap.o: In function ‘swap':
.../swap.c:12: undefined reference to ‘buf'
swap.o:(.data+0x0): undefined reference to ‘buf'
collect2: error: ld returned 1 exit status
```

- The linker tries to link with `crt1.o`, wich refers to the `main` function.

What else?

- After resolving symbols, the linker knows which definition belongs to each symbol.
- The linker does not know about the type, only about the size.

Recall

- Machine code does not use variable names any more.
- The compiler produced code that accesses variables and functions only by their **memory addresses**.

$\Rightarrow$ How does this go together with separate compilation and symbol resolution?

Shared Libraries

- Safe space when used by multiple programs (disk+RAM).
  - Shared libraries **increase code sharing** (and page sharing) more than static libraries.
  - Static library code **cannot be shared between different programs**, only between different instances of the same program.
- Can be updated independently of the application (especially security updates; e.g. OpenSSL).
- Shared libraries come with a **runtime overhead** for accessing any external symbols.