5.5 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`?

6
Variables, Declarations and Scope

6.1 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**[26]. These are called **global**, or sometimes *external* variables.
- Variables in (syntactically) inner scopes **shadow** variables of the same name in outer scopes.

[26] 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?

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

int y = 1, x = 2;

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

6.2 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.

6.3 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
5     auto int z = 23;       /* z is an automatic variable */
6     ...
```

- 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)

6.4 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**.

⇒ 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 */
    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).

6.5 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.

6.6 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.