C Basics 10: Static and Dynamic Memory

1. Overview of C and C vs. Java
2. C Program Elements
3. The Preprocessor
4. Identifiers and Data Types
5. Pointers and Arrays
6. Strings
7. Library I/O
8. Errors and Error Handling
9. Functions and Parameter Passing

C Basics 10: Static and Dynamic Memory

10. **Static and Dynamic Memory**
    - **scope and linkage**
    - **storage durations and storage-class specifiers**
    - **static storage and static variables**
    - **static keyword**
    - **dynamic/heap memory**
    - **memory management functions**
    - **dynamic arrays**
    - **reasons to use, costs, and bugs**
11. Multidimensional and String Arrays
12. Structs (Structures)
13. Higher-Order Functions
14. Variadic Functions
15. Miscellaneous

Scope and Linkage

Before we discuss the main topics, we need to define two key programming languages concepts: **scope** and **linkage**.

Scope refers to the portion of a program where a particular **binding** (of a value to an identifier) is *visible* (and so accessible).

E.g., suppose one defines a variable `ivar` as: `int ivar = 10;`
The *scope* of `ivar` is the portion of the program where you can refer to `ivar` and retrieve the value 10.

The C99 standard says:
"For each different entity that an identifier designates, the identifier is visible (i.e., can be used) only within a region of program text called its scope...There are *four kinds of scopes:* **function**, **file**, **block**, and **function prototype**."

Scope and Linkage (contd.)

Linkage refers to whether an identifier can refer to the same object only in a single "**compilation unit**" or throughout a whole program.

E.g., normally a "**global variable**" is global only to functions within the file the global is defined in, but it is possible to make the variable global to an entire, *multi-file* program.

The C99 standard says:
"An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called linkage. There are *three kinds of linkage:* **external**, **internal**, and **none**."

Storage Durations

In C, **storage duration** refers to the **lifetime/extent** of an object: the portion of program execution during which storage is guaranteed to be reserved for the object.

Each data object in C has one of three *storage durations*:

- **automatic** – allocation/deallocation is *automatic* upon entry/return from the containing function or block
- **static** – allocated permanently in executable, accessible *throughout program execution*
- **allocated** – allocation/deallocation is done *manually* (via calls to *memory management functions*)

An object's storage duration is determined by the *declaration* of its *identifier* (containing *variable*) or by how the object is created.

Storage Durations (contd.)

Automatic duration is the *default* for most variables such as function parameters and local variables.

Static duration is the default for **global variables** (variables declared outside of any function body).

Allocated duration objects must be created with specific functions (e.g., `malloc()`).

Default durations can be overridden with the `auto`, `static`, and `extern` **storage-class specifiers** (more on this later).

Storage Durations (contd.)

The storage duration of an object determines *what part of the executable's address space* it is stored in:

- **automatic** – storage is within the **stack**, *deallocation is automatic* upon return from the containing **stack frame**
- **static** – storage is in **initialized/uninitialized data segments** of the executable (set up by compiler)
- **allocated** – storage is within the **heap**, based on explicit calls to *memory management functions*

Dynamic memory or **dynamic storage** are the terms more commonly used to refer to *allocated storage*.

Storage-Class Specifiers

The C standards use the term **storage-class specifier** to refer to keywords that affect the storage duration (and possibly other characteristics) of identifiers/variables.

It is fairly common to see material on C talk about the "**storage class**" of identifiers and objects, typically relating "storage class" to where objects are stored in memory.

The C standards, however, refer minimally and inconsistently to the *"storage class"* of identifiers (and even objects).

Storage-Class Specifiers (contd.)

The storage-class specifiers are:

- **auto** – object *storage duration* will be *automatic*
- **static** – object *storage duration* will be *static*
- **register** – try to make access to object as fast possible (e.g., store in CPU register if possible)
- **extern** – sets the **linkage** of an identifier to **external**, which results in *static storage duration*

Static Storage/Variables

Objects for identifiers/variables with **static storage duration** are maintained *throughout program execution*.

A variable has static storage duration if:

1. it is **global** (declared outside of any function definition), or
2. it is local but declared with the `static` *storage-class specifier:*

```
static int i = 1;
```

Static storage is allocated by the compiler, in the **data segment**.

If the variable declaration defines an initial value, that value is setup in the executable (**initialized data segment**).

Otherwise, the variable is setup with a default zero/`NULL` value (**uninitialized data segment**).

Static Storage/Variables (contd.)

Local variables are *automatic duration* by default, which raises two issues in their use:

1. pointers to their values *cannot be returned* from functions (because the storage will be automatically deallocated)
2. their values are not *retained between function calls*

Issue #1 can be addressed in two ways:

1. by returning *static storage* objects
2. by returning *allocated storage* objects (i.e., dynamic memory—discussed shortly)

Issue #2 can be solved only through the use of *static storage*: Declaring local variables to be `static`, will cause their values to be maintained throughout program execution, and thus between function calls.

Example: Static Local Variable in Function

Consider a function that needs to keep track of how many times it has been called:

```
void check_count()
{
  //Initialize num_calls:
  //(done once at program initialization)
  static int num_calls = 0;

  //check_count called again, so increment maintained count:
  num_calls++;

  if (num_calls > 10)  //E.g., do something if check_count
    ...                //has been called more than 10 times

  return;
}
```

static Keyword

The keyword `static` is a bit confusing, as it has two distinct uses:

- to change the **storage duration** of a variable
- to *limit* the **scope** of functions and global variables

By default, *functions* have **global scope**:
they can be called from any function in a program, even from functions that are defined in separate *files*.

(In C, *files* are the **compilation modules** or **translation units**.)

(Technically, C functions have **file scope** and **external linkage**, which results in what is generally referred to as *global scope*.)

static Keyword (contd.)

Adding the `static` modifier to the definition of a function or global variable reduces its scope to *the remainder of the file containing the definition*:

```
static void check_count() {...}
```

This causes the function to have what is called **file scope**.

(Technically, it now has *file scope* plus **internal linkage**.)

File scope is a type of "**information hiding**" mechanism.

It can prevent functions and globals from being accessed from other program modules.

This also helps limit *name conflicts*, since C has a *single* **namespace** for most identifiers.

Dynamic Memory

Dynamic memory is the most commonly used term for storage space allocated on the program's **heap**.

In C, dynamic memory means *allocated storage duration* objects.

This storage is called *dynamic* because it can be allocated or deallocated at any point during program execution.

It has **indefinite extent**: it remains until explicitly deallocated (it is not automatically deallocated when a function/block ends).

Allocated storage is created by calling **memory management functions**.

Memory Management Functions

Dynamic memory management involves a set of four functions:

- **malloc** – allocates a number of bytes and returns a pointer to the first byte of the block
- **calloc** – like `malloc`, but allocates space for a number of objects of specified bytes each, plus *zeros memory*
- **realloc** – changes the size of a memory block previously allocated by `malloc`, etc.
- **free** – reclaims a previously allocated memory block

Memory Management Functions (contd.)

Syntax of memory management functions:

```
void *malloc(size_t size)
void *calloc(size_t num, size_t objsize)
void *realloc(void *ptr, size_t size)
void free(void *ptr)
```

- `size` is the number of *bytes* to be allocated
- `num` is the number of objects
- `objsize` is the size in bytes of each object
- `ptr` must be a pointer (address) previously returned by one of these calls
- `void*` return is either a pointer to an appropriate size memory block, else NULL on error

`malloc(num * objsize)` is equivalent to `calloc(num, objsize)` except that `calloc` also zeros out the memory block.

Example: Dynamic Memory

Using dynamic memory to return a new string that is
a concatenation of two argument strings:

```
char *concat_strings(const char *str1, const char *str2)
{
  //Allocate space for final string:
  char *concat = malloc(strlen(str1) + strlen(str2) + 1);

  strcpy(concat,str1);  //Copy str1 into concat
  strcat(concat,str2);  //Concat str2 onto end of str1

  return concat;
}
```

Dynamic Arrays

Dynamic memory is often used to provide "**dynamic arrays**."

Recall that an array is a contiguous block of memory, effectively represented by a pointer to the first byte of the array.

This is exactly what `malloc()` provides, but using heap memory.

Thus, we can use *array notation* with dynamic memory:

```
int *iarr = malloc(sizeof(int)*num_ints);
for (int i = 0; i < num_ints; i++)
  iarr[i] = 100 + i;
```

Not only are the sizes of "dynamic arrays" determined at runtime, but they can be *resized* (using `realloc()`).

Example: Resizing Dynamic Array

Resizing an int array:

```
int *iarr = malloc(sizeof(int)*num_ints);
for(int i = 0; i < num_ints; i++)
  iarr[i] = 100 + i;
...
numints += 10;
iarr = realloc(iarr,sizeof(int)*num_ints);
for(int i = numints - 10; i < num_ints; i++)
  iarr[i] = 100 + i;
```

Whether the array is resized *in-place* or the data has to be copied to a new block is transparent to the user as long as `iarr` is updated with the return value from `realloc()`.

Example: Resizing String

Reading in an arbitrary length line with `fgets()`:

```
char *readinline(FILE *fptr)
{
  int size = 51, start = 0;
  char *line = malloc(size), *result;

  while ((result = fgets(line+start,51,fptr)) != NULL) {
    if (line[strlen(line)-1] == '\n' || feof(fptr))
      break;  //End of line/file found so quit reading.
    else {
      start = size - 1;
      size += 50;
      line = realloc(line,size); }
  }

  //If read error, return NULL:
  if (result == NULL)
    return NULL;

  return line;
}
```

Correct Usage of realloc()

Since it is possible for the dynamic memory management calls to fail, they must be error checked as usual.

Doing this properly for `realloc()` can be somewhat subtle.

Consider a simple example:

```
char *line = malloc(50);  //No error checking here for simplicity
...
if ((line = realloc(line,100)) == NULL)
  free(line);  //Runtime error since line will be NULL!
...
```

The problem with the above `realloc()` call is that if it should fail, `line` will get set to `NULL`, and you will lose the pointer to memory block allocated with `malloc()`.

Correct Usage of realloc() (contd.)

If your program is going to terminate on such an error then this is fine.

However, if it is going to continue running, as with a server, then you have the makings for a **memory leak** (discussed later).

The correct approach uses a temporary variable:

```
char *line = malloc(50);  //No error checking here for simplicity
...
char *temp;
if ((temp = realloc(line,100)) != NULL)
  line = temp;
else
  deal appropriately with realloc call having failed
...
```

Reasons to Use Dynamic Memory

There are three reasons for using dynamic memory:

1. to allow a data object to exist beyond the **scope** in which it is created (e.g., outside a function)
2. to create *arrays* whose *size is determined at runtime*
3. to provide the ability to *resize an array* (including strings)

Because dynamic memory remains accessible until it is explicitly deallocated (`free`'d), it can be used to provide storage that exists beyond the span of a single function/block.

A key example of when this is required is when a *function returns a pointer type*, such as a string (`char*`).

The only other pointers (memory addresses) that are valid to return from functions are those for *static storage duration* objects.

Reasons to Use Dynamic Memory (contd.)

Originally, the size of a C array had to be declared with a *constant expression* only, meaning the array size had to be *known at compile time*.

C99 added **variable-length arrays**: the array size expression could involve variables/parameters, and so could be *determined at runtime*.

This meant that some uses of dynamic arrays could be replaced with variable-length arrays.

However, variable-length arrays *do not have indefinite extent*: storage for variable-length arrays is allocated *at the point of declaration* and *deallocated when the containing block is exited*.

Reasons to Use Dynamic Memory (contd.)

Furthermore, while the size of a variable-length array can be determined at runtime, once the storage has been allocated, the *array size cannot be changed*.

Dynamic arrays, on the other hand, can be *resized* as needed, using `realloc()`.

It is important to be aware, however, that resizing a dynamic array may result in the array contents having to be *copied to a new memory block* (by `realloc()`).

Since doing this frequently could waste significant CPU cycles, it may be more efficient to use a standard array that is large enough for all possibilities.

Costs of Dynamic Memory

When deciding whether to use dynamic memory it is important to understand that the memory management functions all add *runtime overhead* to a program.

Heap memory is organized so as to avoid **fragmentation** when repeatedly allocating and deallocating different sized blocks.

Thus a call to `malloc()` will require finding the best fit block size and updating the list of used/free blocks.

A call to `free()` will require verifying the memory pointer is valid and updating the appropriate list of used/free blocks.

Because there is *runtime overhead* to using dynamic memory, one should *use an array instead of dynamic memory if practical*.

Manual Memory Management

In C, *management of dynamic memory* is completely **manual**.

That is, allocating and reclaiming dynamic memory must be done via *explicit function calls*.

In Java, allocation is done manually with the `new` operator, but memory is reclaimed automatically via **garbage collection**.

Having automatic reclamation of dynamic memory avoids serious *memory management errors!*

Memory Management Bugs

Memory management errors can result in two key bugs:

- **memory leaks**
- **dangling pointers**

A **memory leak** occurs when `malloc()` (or related functions) is repeatedly called without calling `free()`.

This causes heap memory and so the overall program address space to grow over time, potentially *exhausting virtual memory*.

A **dangling pointer** occurs when a memory block gets `free`'d while a pointer to the block continues to be used.

These bugs are common in complex C programs because it can be very difficult to understand when it is correct to free dynamic memory if pointers are passed among functions.