

A Recap of the C Programming Language

Computational Science II (CAAM 520)

Christopher Thiele

Rice University, Spring 2021

Overview

Before we start using C for our purposes, we will review

- variable initialization,
- pointers,
- segmentation faults,
- pointers and structures,
- pointers to pointers, etc.,
- void pointers and casts,
- function pointers,
- arrays, and
- dynamic memory allocation.

Variable initialization

Variables must be initialized ***explicitly!***

```
int i = 123;  
int k;  
  
printf("%d\n", i);  
printf("%d\n", k); // Undefined result!
```

Pointers

Pointers are variables that store the address of another variable.

```
int i;  
int *ptr = &i;  
  
i = 123;  
  
printf("%p\n", ptr); // Print the address of i.  
printf("%d\n", *ptr); // Print the value of i.
```

Pointers

Variables can be modified through pointers.

```
int i;  
int *ptr = &i;  
  
*ptr = 123;  
  
printf("%p\n", ptr); // Print the address of i.  
printf("%d\n", *ptr); // Print the value of i.
```

Pointers

Typical use case: output arguments

```
int foo()  
{  
    return 123;  
}
```

VS.

```
void foo(int *result)  
{  
    *result = 123;  
}
```

Pointers

Typical use case: output arguments

Particularly useful for multiple output arguments and when using error codes!

```
int divide(int x, int y,  
          int *quotient, int *remainder)  
{  
    if (y == 0) return -1;  
  
    *quotient = x/y;  
    *remainder = x%y;  
    return 0;  
}
```

Pointers

Like other variables, pointers must be initialized!

```
int *ptr;
```

```
// Undefined behavior, likely a segmentation fault!
```

```
*ptr = 123;
```


Pointers

We use NULL to indicate invalid pointers.

```
int *ptr = NULL;

// ...

// Check if pointer is valid.
if (ptr) { // Equivalent to ptr != NULL
    // ...
}

// Check if pointer is invalid.
if (!ptr) { // Equivalent to ptr == NULL
    // ...
}
```

Pointers

Pointers to structures allow more convenient notation.

```
typedef struct
{
    int i;
} my_struct_t;

void init(my_struct_t *s)
{
    (*s).i = 123;
    // Equivalent, but more convenient:
    s->i = 123;
}
```

Pointers

Pointers can point to variables of any type, including other pointers.

```
int i = 123, *ptr, **ptrptr;  
  
ptrptr = &ptr;  
*ptrptr = &i;  
  
// Print value of ptrptr/address of ptr.  
printf("%p\n", ptrptr);  
// Print value of ptr/address of i.  
printf("%p\n", *ptrptr);  
// Print value of i.  
printf("%d\n", **ptrptr);
```

Pointers

Why would we need pointers to pointers?

Pointers

Why would we need pointers to pointers?

E.g., for functions that need have pointers as output arguments.

```
void ptr_max(const int *i, const int *k, int **max)
{
    if (*i > *k) {
        *max = i;
    }
    else {
        *max = k;
    }
}
```

Pointers

Constant pointers and pointers to constant things:

```
int i;  
  
// What does each declaration do?  
int *ptr1 = &i;  
const int *ptr2 = &i;  
int *const ptr3 = &i;  
const int *const ptr4 = &i;
```

Pointers

Void pointers represent generic memory addresses. They can point to variables whose type is unknown.

```
int i;  
  
void *ptr = &i;  
  
// Error: Compiler does not know that ptr  
// points to an integer!  
*ptr = 123;  
  
// We must cast void pointers before  
// dereferencing them.  
*((int*) ptr) = 123;
```

Pointers

Why would we need void pointers?

Pointers

Why would we need void pointers?

E.g., for functions that operate on data of any type:

```
// Copy any type of data.  
void* memcpy(void *dst, const void *src, size_t n)  
  
// Allocate and deallocate memory.  
void* malloc(size_t size)  
void free(void *ptr)
```

Pointers

Caution: C and C++ handle void pointers slightly differently!

```
// The following works *only* in C:
```

```
int *ptr = malloc(sizeof(int));
```

```
// In C++ we must cast explicitly.
```

```
int *ptr = (int*) malloc(sizeof(int));
```

→ It might be a good idea to cast, as someone else could use our C code in their C++ project.

Function pointers

We can create pointers to functions as well:

```
int add(int i, int k) { return i + k; }  
  
// ...  
  
// Create a pointer to the add function.  
int (*add_fptr)(int, int) = add;  
  
// Call add through the pointer.  
add_fptr(3, 7);
```

→ Notice the parentheses in the function pointer declaration!

Arrays

C supports arrays of fixed size and any type:

```
int array[16];

// Indices start at zero!
for (int i = 0; i < 16; i++) {
    array[i] = i;
}

// What does this do?
array[16] = 123;
// How about this?
array[12345] = 123;
```

→ Be very careful with indices!

Arrays

Arrays of characters (strings) are particularly common and useful.

```
const char string[] = "hello , world!";  
  
printf("%d\n", strlen(string)); // 13  
printf("%d\n", sizeof(string)); // 14 – why?
```

Dynamic memory allocation

We can allocate "arrays" ***dynamically***.

→ Technically, arrays and pointers to allocated memory are ***not*** the same, but the differences are negligible for our purposes.

```
int *array = (int*) malloc(n*sizeof(int));  
  
if (!array) { // Equivalent to array == NULL  
    fprintf(stderr, "Error: Allocation failed!\n");  
    return -1;  
}  
  
// We can access the array as usual.  
array[n - 1] = 123;
```

Dynamic memory allocation

Other ways to (re)allocate memory dynamically:

```
// Allocates *uninitialized* memory:
```

```
void* malloc(size_t size)
```

```
// Allocates memory and sets it to zero:
```

```
void* calloc(size_t num, size_t size)
```

```
// Reallocates memory, i.e., if we need more:
```

```
void* realloc(void *ptr, size_t new_size)
```

Dynamic memory allocation

If memory is allocated, it must be deallocated with `free()`.

```
int *array = (int*) malloc(n*sizeof(int));  
  
// Deallocate memory.  
free(array);
```

If memory is not released when it is no longer used, we have a **memory leak**!

→ Why are leaks problematic?

Dynamic memory allocation

Example 1: Memory is allocated repeatedly, but never deallocated.

```
for (int i = 0; i < 16; i++) {  
    int *array = (int*) malloc(n*sizeof(int));  
  
    // Work with array in the loop.  
  
    // free() is missing.  
}
```

Dynamic memory allocation

Example 2: Memory is allocated, but the programmer is not aware of it.

```
char* get_message()  
{  
    char *msg = (char*) malloc(64);  
    strcpy(msg, "hello , world!");  
    return msg;  
}
```

Dynamic memory allocation

Rule of thumb: For each call to `malloc()`, `calloc()`, or `realloc()`, there must be a matching call to `free()`.

Note to C++ programmers:

- If memory was allocated with `new`, it **must** be deallocated with `delete`.
- If memory was allocated with `malloc()`, `calloc()`, or `realloc()`, it **must** be deallocated with `free()`.

Pointer arithmetic

C allows arithmetic with pointers:

```
int *array = (int*) malloc(16*sizeof(int));  
  
// Access 7th element.  
array[6] = 123;  
// Equivalently:  
*(array + 6) = 123;
```

→ For `int*`, pointer arithmetic works in increments of `sizeof(int)` etc.

Pointer arithmetic

C allows arithmetic with pointers:

```
int *array = (int*) malloc(16*sizeof(int));  
  
int *ptr = &array[3];  
ptr += 3;  
--ptr;  
  
// Which element does this set?  
*ptr = 123;
```