CSE 2421

C Pointers – Part 2

C Pointers – Part 2 – Overview

- Arrays and pointers
  - Statically allocated arrays
  - Dynamically allocated arrays
- Pointers to void (void *)
- Dynamic memory allocation and pointers
- Freeing (deallocating) dynamically allocated storage
- Pointer arithmetic
- Function parameters and pointers

Arrays in C

- Different from arrays in Java in a number of respects.
- Arrays are closely related to pointers in C; elements of arrays can be accessed using a pointer in C as well as by index.
- But, when arrays get complex, it’s almost always accessed via pointers
- C has two different types of arrays:
  - Statically allocated arrays: The compiler generates code to allocate the space for the array elements at compile time (and to initialize them, if they are initialized as part of the declaration). The space is allocated on the stack or the heap depending upon the declaration statement.
  - Dynamically allocated arrays: A C library function is called to request space for the array elements at runtime (that is, after the program begins running).

Declaration of Static Arrays

- Example:

```
int scores[6] = {19, 17, 18, 16, 15, 20};
int scores[] = {19, 17, 18, 16, 15, 20};
```

- C arrays declared as above are allocated storage at compile time, and have **a fixed size**.
- The expression enclosed in [ ] must be ***a constant***, the value of which is known at compilation time.
- If [ ] is empty, then the compiler will create an array with a number of elements equal to the number of values enclosed in braces.
- Thus, the two declarations above declare the same array.

```
int scores[10] = {19, 17, 18, 16, 15, 20};
int scores[] = {19, 17, 18, 16, 15, 20};
```

  These two declarations do not declare the same array. The size is different.
- Whether the array elements are of static storage class or automatic storage class, the compiler will generate code to allocate memory storage for the elements of the array, and to store their initial values on the heap or on the stack. This is why these arrays are referred to as **static arrays**, or **statically allocated arrays** (Here, static means *at compile time*).

Array Initialization

- Arrays of static storage class will be initialized to 0 (all elements) by *most* compilers if no explicit initialization is given for any element.
- If you provide fewer values than the number of elements in the array, the remaining values will be initialized to 0 (this works for both static and automatic storage class arrays):

```
int scores[10] = {19, 20};      /* last 8 elements set to 0 */
```

- To explicitly initialize all elements to 0 (for a static storage class or automatic storage class array):

```
int scores[10] = {0};
```

Array Size – In General

- There is no library function in C that will tell you the size of an array (for statically or dynamically allocated arrays).
- This is because no array termination marker is stored in the array (except for <u>strings</u> stored in char arrays – more later).
- Therefore, you must keep track of the size, and check indexes "manually" (in the code you write) to ensure that they are within range.
- If you try to access elements beyond the last element, this will produce a run-time error (typically a segmentation fault) OR will read or write a value that is not an element of the array (a harder bug to find).

Only in the function that defines the static array can we get the size:

```
void unrelated()
{
        int things[] = { 1, 2, 3, 4, 5};
        int count = sizeof(things) / sizeof(things[0]);

        printf("There are %d things. The array is %d bytes,
each one is %d bytes\n", count, sizeof(things),
sizeof(things[0]));


}
```

Output:

```
There are 5 things. The array is 20 bytes, each one is 4 bytes
```

Copying Arrays

- There is no library function for copying arrays in C (except for strings, which we'll see soon).
- If you want to copy an array in C, you must copy the elements one by one (with a **for** or **while** loop). For example, we can use something such as:

```
int scores[6] = {19, 17, 18, 16, 15, 20}; /* Array to be copied */
int copy[6];        /* Copy of original array */
for (i = 0; i < 6; i++) {
        copy[i] = scores[i];
}
```

Arrays and Pointers

- int scores[6] = {19, 17, 18, 16, 15, 20};
- In C, the name of the array by itself is ***a constant pointer*** **to the first element of the array**; that is, **scores** is the same as **&scores[0] (don’t use &scores, though!)**
- Because this pointer is a constant, ***it cannot be changed*** (for example, you cannot assign a different address to it).
- All of the elements of the array are stored in contiguous memory locations.

Address math != Pointer math

- We code C in terms of pointers and what they point to. If p is a pointer to an int:
  - p+1 points to the next int.
  - **We don't have to scale when we do pointer math**
- The compiler turns pointer math into address math by scaling:
  - The compiler needs type to get size to do the scaling
  - The programming model says memory is byte-addressable
  - The compiler needs to generate a byte address
- The next int (pointer math) is sizeof(int) bytes away in memory (address math)

Arrays and Pointers cont.

- The compiler generates instructions to compute the exact address for any chosen element of the array when using an array; it can do this because it knows all the bytes used to store the array are contiguous.
- Therefore, to access scores[3], for example, the compiler generates instructions to compute the address as:

```
        scores + (3 * (sizeof (int)))        /* what the compiler does */
```

- Recall that scores is a constant pointer to the 1st element, so the compiler can compute the address above as:

```
        &scores[0] + (3 * (sizeof (int)))   /* not C code we write */
```

The two formulas above are what the compiler does, ***not what we do in C !***

Pointers to void (void *)

- Declaration example: void *ptr;
- A pointer declared to point to a certain type should not be assigned the address of an object of a different type (without casting):

```
float *float_ptr;
int int_var = 5;
float_ptr = &int_var;   /* Dangerous – the compiler will
                           give a warning here, because
                           there is no explicit cast! */
```

Pointers to void (void *)

- The exception is **pointer to void (void *),** which is *assignment compatible* with pointers to other data types:
  - /*With declarations above – valid */
    void_ptr = float_ptr;
  - /*With declarations above – also valid*/
    int_ptr = void_ptr;

Pointers to void continued

- A pointer to void **cannot be *dereferenced* without casting** (because the compiler won’t know how to generate an appropriate instruction to interpret the data which is pointed to by the pointer):
- Another way to say this is that – without casting – the compiler won’t know how many bytes to read starting at the given address

```
void *void_ptr;
int var1 = 1;
void_ptr = &var1;
printf(“*void_ptr equals %d”, *void_ptr);       /* Invalid – dereference of
                                               void pointer without cast */
printf(“*void_ptr equals %d”, * (int *) void_ptr) );    /* Valid */
 /* NOTE: the dereference operator has higher precedence than the cast, or type conversion,
operator */
```

Pointers to void – purposes

- Why use pointers to void?
- One purpose for them is that certain C library functions which allocate memory dynamically (i.e., at run time) return a void *, or pointer to void (because otherwise, there would have to be a different version of the library function to return each pointer type), which can then be assigned to a pointer to any other type.
- We'll see how this works shortly.
- Another reason is that pointers to void can be used to pass "typeless" parameters to functions, and then the function can use a cast with the pointer to access data which is to be interpreted in a certain way (we will only see one example of this later)

Dynamic Allocation

- Dynamic memory allocation: C has library functions for requesting additional static storage class space in memory while the program is in execution (thus, this is called ***dynamic*** allocation, meaning *runtime* allocation).
- These functions must be used when you do not know while coding *how much storage will be needed*.
- They can also be used if you know how much storage will be needed, but in that case, execution time will be increased (slightly) while the additional storage is allocated, so the use of these functions is not a good choice.

C library functions – Dynamic allocation

- We will learn about two functions for dynamic allocation:

  malloc()

  calloc()
- These functions return a pointer to void (void *), and they are declared in stdlib.h.
- Assuming that the allocation succeeds, these functions return a pointer which points to the address of the first byte of the allocated memory space **on the heap**. If the allocation fails (out of memory error), they return a null pointer.
- If more than one byte is allocated, the bytes will have contiguous addresses.
- The C operator *sizeof* is used to pass parameters to these library functions, so that the function knows how many bytes to allocate.
- You should ALWAYS use sizeof, even if you know the number of bytes required to store the values on the system the code is being written for, to make the code portable.

Malloc(3) Manual Page

```
MALLOC(3)              Linux Programmer's Manual              MALLOC(3)

NAME
       calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *calloc(size_t nmemb, size_t size);
       void *malloc(size_t size);
       void free(void *ptr);
       void *realloc(void *ptr, size_t size);

DESCRIPTION
       calloc() allocates memory for an array of nmemb elements of size bytes each and returns a
       pointer to the allocated memory.  The memory is set to zero.  If nmemb or size is 0, then
       calloc()  returns  either  NULL, or a unique pointer value that can later be successfully
       passed to free().

       malloc() allocates size bytes and returns a pointer to the allocated memory.  The  memory
       is  not  cleared.   If  size is 0, then malloc() returns either NULL, or a unique pointer
       value that can later be successfully passed to free().

       free() frees the memory space pointed to by ptr, which must have been returned by a  pre-
       vious  call  to  malloc(), calloc() or realloc().  Otherwise, or if free(ptr) has already
       been called before, undefined behavior occurs.  If ptr is  NULL,  no  operation  is  per-
       formed.
```

malloc()

- Returns a pointer to void (i.e., void *), which points to the address of the first byte of the allocated memory space on the heap.
- This function takes one parameter – which may be an expression – which specifies the number of bytes that are being requested (Use sizeof() for portability!).
- For example, to request enough bytes for 4 integer values, we could use:

```
int *p;
p = malloc ( 4 * sizeof(int) );        /* malloc (4 * 4) is not portable! */
```

- If we only needed the number of bytes for one int, we could use:

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

- The memory returned by malloc is *uninitialized* (contains garbage values), so be sure to initialize it before you use it!

calloc()

- Returns a pointer to void (i.e., void *), which points to the address of the first byte of the allocated memory space on the heap.
- This function takes two parameters – which may be expressions – which specify the number of elements for which storage is being requested, and the size of each element in bytes (Use sizeof() for portability!).
- So, to request memory space for 4 integer values, we could use:

```
int *p;
p = calloc ( 4, sizeof(int) );    /* calloc (4, 4) is not portable! */
```

- If we only needed space for one int, we could use:

```
int *p;
p = calloc ( 1, sizeof(int) );
```

- The memory returned by calloc is *initialized* to 0, so if you do not plan to initialize the values before using them, use calloc, and not malloc!
  - This means that calloc() will take more CPU time to execute than will malloc(). It may or may not be an issue, but worth thinking about depending upon how much space you are requesting.

What if allocation fails?

- If the requested memory cannot be allocated, both malloc() and calloc() return **the null pointer (defined in stdlib.h)**, which has a value of 0.
- Therefore, before using the pointer to access any of the allocated memory, you should check to make sure that the pointer returned was not null. For example:

```
int *p;
p = calloc (10, sizeof(int) );
if (p != 0) {  /*Also (if p != NULL), NULL is #defined in stdlib.h */
    . . . .
    . . . .       /* OK to access values in allocated storage */
}
else {  . . . .   /* Some code to handle the allocation failure*/
}
```

Freeing allocated storage

- If your program uses storage which has been allocated dynamically, then you should *free* it (return it to the operating system) once it is no longer being used.
- The C library function *free()* is used for this; it returns void, and has a single parameter, which is a pointer to the first byte of the allocated storage to be freed, and this pointer MUST be pointing to the 1st byte of some dynamically allocated storage!
- free() is also declared in stdlib.h.

Freeing allocated storage

- Here’s an example of how to free dynamically allocated storage:

```
int *p;
p = calloc (10, sizeof (int) );
……
free (p);  /* releases storage to which p points back to the OS */
p = NULL;
```

- The pointer which was passed to free should also be set to NULL or 0 after the call to free(), to ensure that you do not attempt to access it inadvertently. To do so can cause a segmentation fault.

Malloc() frequent errors…(from Wiki)

The improper use of dynamic memory allocation can frequently be a source of bugs. These can include security bugs or program crashes, most often due to segmentation faults. The most common errors are as follows:

- **Not checking for allocation failures**
  - Memory allocation is not guaranteed to succeed, and may instead return a null pointer. Using the returned value, without checking if the allocation is successful, invokes undefined behavior. This usually leads to a crash (due to the resulting segmentation fault on the null pointer dereference), but there is no guarantee that a crash will happen so relying on that can also lead to problems.
- **Memory leaks**
  - Failure to deallocate memory using free() leads to buildup of non-reusable memory, which is no longer used by the program. This wastes memory resources and can lead to allocation failures when these resources are exhausted.
- **Logical errors**
  - All allocations must follow the same pattern: allocation using malloc(), usage to store data, deallocation using free. Failures to adhere to this pattern, such as memory usage after a call to free (dangling pointer) or before a call to malloc() (wild pointer), calling free twice ("double free"), etc., usually causes a segmentation fault and results in a crash of the program. These errors can be transient and hard to debug – for example, freed memory is usually not immediately reclaimed by the OS, and thus dangling pointers may persist for a while and appear to work.