CS107 Handout 08
Autumn 2000 October 2, 2000

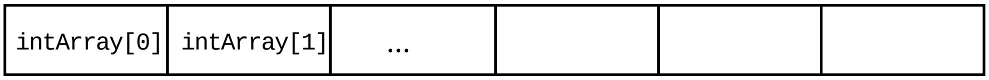
The Ins and Outs of C Arrays

This handout was written by Nick Parlante and Julie Zelenski.

C Arrays

As you recall, an array in C is formed by laying out all the elements contiguously in memory from low to high. The array as a whole is referred to by the address of the first element. For example, the variable `intArray` below is synonymous with the address of the first element and can be used in expressions like an `int *`.

```
   int intArray[6];
```

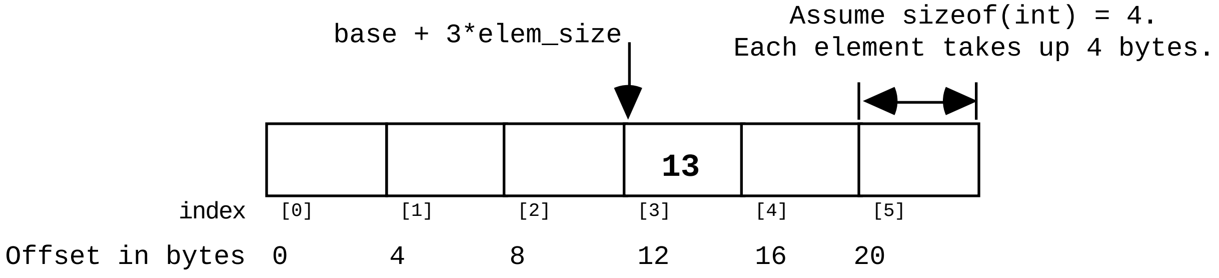


The programmer can refer to elements in the array with the simple [] syntax such as `intArray[1]`. This scheme works by combing the base address of the array with the index to compute the base address of the desired element in the array, using some simple arithmetic. Each element takes up a fixed number of bytes which is known at compile-time. So address of the `n`th element in the array (`0`-based indexing) will be at an offset of (`n * element_size`) bytes from the base address of the whole array.

```
 address of nth element = address_of_0th_element + (n * element_size_in_bytes)
```

The square bracket syntax [] deals with this address arithmetic for you, but it's useful to know what it's doing. The [] multiplies the integer index by the element size, adds the resulting offset to the array base address, and finally dereferences the resulting pointer to get to the desired element.

```
   intArray[3] = 13;
```



'+' Syntax

In a closely related piece of syntax, adding an integer to a pointer does the same offset computation, but leaves the result as a pointer. The square bracket syntax dereferences that pointer to access the nth element while the + syntax just computes the pointer to the nth element.

So the expression `(intArray + 3)` is a pointer to the integer `intArray[3]`. `(intArray + 3)` is of type `(int*)` while `intArray[3]` is of type `int`. The two expression only differ by whether the pointer is dereferenced or not. So the expression `(intArray + 3)` is exactly equivalent to the expression `(&(intArray[3]))`. In fact those two probably compile to exactly the same code. They both represent a pointer to the element at index 3.

Any `[]` expression can be written with the + syntax instead. We just need to add in the pointer dereference. So `intArray[3]` is exactly equivalent to `*(intArray + 3)`. For most purposes, it's easiest and most readable to use the `[]` syntax. Every once in a while the `+` is convenient if you needed a pointer to the element instead of the element itself.

Pointer++

If p is a pointer to an element in an array, then `(p+1)` points to the next element in the array. Code can exploit this using the construct `p++` to step a pointer over the elements in an array. It doesn't help readability any, so I can't recommend the technique, but you may see it in code written by others.

Here's a sequence of versions of strcpy written in order: from most verbose to most cryptic. In the first one, the straightforward for loop needs a little follow-up to ensure that the terminating null character is copied over. The second removes that opportunity for error by rearranging into a while loop and moving the assignment into the test. The last two are **cute** (and they demonstrate using ++ on pointers), but not really the sort of code you want to maintain. Among the four, I think the second is the best stylistically. With a smart compiler, all four will compile to basically the same code with the same efficiency.

```
void strcpy1(char dest[], const char source[])
{
   int i;

   for (i = 0; source[i] != '\0'; i++)
      dest[i] = source[i];
   dest[i] = '\0';   // don't forget to null-terminate!
}
```

```
// Move the assignment into the test
void strcpy2(char dest[], const char source[])
{
   int i = 0;

   while ((dest[i] = source[i]) != '\0')
      i++;
}

// Get rid of i and just move the pointers.
// Relies on the precedence of * and ++.
void strcpy3(char dest[], const char source[])
{
  while ((*dest++ = *source++) != '\0') ;
}


// Rely on the fact that '\0' is equivalent to FALSE
void strcpy4(char dest[], const char source[])
{
  while (*dest++ = *source++) ;
}
```

Pointer Type Effects

Both `[]` and `+` implicitly use the compile time type of the pointer to compute the element_size which effects the offset arithmetic. When looking at code, it's easy to assume that everything is in the units of bytes.

```
int *p;

p = p + 12;    // at run-time, what does this add to p? 12?
```

The above code does not add the number 12 to the address in p— that would increment p by 12 *bytes*. The code above increments p by 12 *ints*. Each int takes 4 bytes, so at run time the code will effectively increment the address in p by 48. The compiler figures all this out based on the type of the pointer.

You can manipulate this using casts. For example, the following code really does just add 12 to the address in the pointer p. It works by telling the compiler that the pointer points to `char` instead of `int`. The size of `char` is defined to be exactly 1 byte (or whatever the smallest addressable unit is on the computer). In other words, `sizeof(char)` is always 1. We then cast the resulting `(char*)` back to an `(int*)`. You can use casting like this to change the code the compiler generates. The compiler just blindly follows your orders.

```
p = (int*) ((char*)p + 12);
```

Arithmetic on a void pointer

For a `(void*)` pointer, array subscripting and pointer arithmetic don't quite make sense. These manipulations include implicit multiplication by the size of the element type, and what is sizeof(void)? Unknown! Some compilers assume that it should treat it like a (char*), but if you were to depend on this you would be creating non-portable code. To be precise and correct, you should cast the (void*) to (char*) before doing any math to make clear that all arithmetic is done in one-byte increments and any necessary multiplication will be done explicitly.

```
void *ptr;

p = (char*)ptr + 4;  // increments ptr by exactly 4, no extra multiplication
```

Note that you do not need to cast the result back to (void*), a (void*) is the "universal recipient" of pointer types and can be freely assigned any type of pointer. It is best to use casts only when you absolutely must.

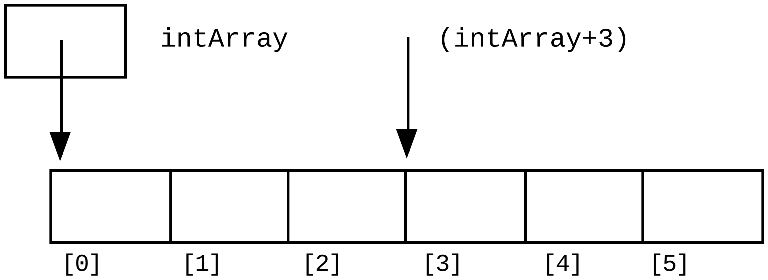
Arrays and Pointers

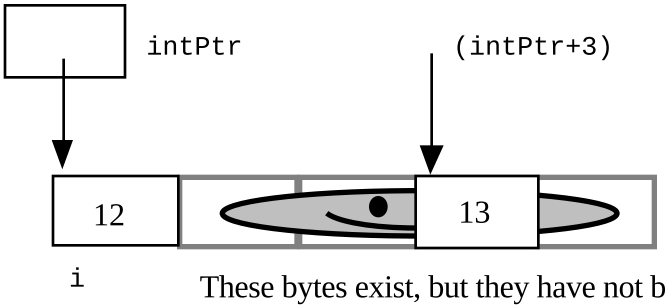
One effect of the C array scheme is that the compiler does not meaningfully distinguish between arrays and pointers— they both just look like pointers. In the following example, the value of intArray is a pointer to the first element in the array so it's an (int*). The value of the variable intPtr is also (int*) and it is set to point to a single integer i. So what's the difference between intArray and intPtr? Not much as far as the compiler is concerned. They are both just (int*) pointers, and the compiler is perfectly happy to apply the [] or + syntax to either. It's the programmer's responsibility to ensure that the elements referred to by a [] or + operation really are there. Really its just the same old rule that C doesn't do any bounds checking. C thinks of the single integer i as just a sort of degenerate array of size 1.

```
{
   int intArray[6];
   int *intPtr;
   int i;

   intPtr = &i;

   intArray[3] = 13;    // ok
   intPtr[0] = 12;      // odd, but ok. Changes i.
   intPtr[3] = 13;      // BAD! There is no integer reserved here!
}
```





These bytes exist, but they have not been explicitly reserved. They are the bytes which happen to be adjacent to the memory for i. They are probably being used to store something already, such as a smashed looking smiley face. The 13 just gets blindly written over the smiley face. This error will only be apparent later when the program tries to read the smiley face data.

Array Names Are Const

One subtle distinction between an array and a pointer, is that the pointer which represents the base address of an array cannot be changed in the code. Technically, the array base address is a `const` pointer. The constraint applies to the name of the array where it is declared in the code— the variable `intArray` in the example below.

```
{
   int intArray[100]
   int *intPtr;
   int i;

   intArray = NULL;           // no, cannot change the base addr ptr
   intArray = &i;             // no
   intArray = intArray + 1;   // no
   intArray++;                // no

   intPtr = intArray; // ok, intPtr is a regular pointer which can be changed
                      // here it is now pointing to same array intArray is

   intPtr++;      // ok, intPtr can still be changed (and intArray cannot)
   intPtr = NULL; // ok
   intPtr = &i;   // ok

   foo(intArray); // ok (possible foo definitions are below)
   foo(intPtr);   // ditto
}
```

Array parameters are passed as pointers. The arguments to the two definitions of `foo` look different, but to the compiler they mean exactly the same thing. It's preferable to use whichever syntax is more accurate for readability. If the pointer coming in is going to be treated as the base address of a whole array, then use `[]`, if it is merely a pointer to one integer, the `*` notation is more appropriate.

```
void foo1(int arrayParam[])
{
   arrayParam = NULL;   // Silly but valid. Just changes the local pointer
}

void foo2(int *arrayParam)
{
   arrayParam = NULL;   // ditto
}
```

Note that either `intArray` or `intPtr` (from above) could be passed to either version of `foo`. In each case, the address of an integer is passed— for `intArray` it is a copy of base address of the array, for `intPtr`, it is a copy of its current value. Either way, once in the function foo, either pointer can be changed to point elsewhere without affecting the value of the pointers in the calling function.

Dynamic Arrays

Since arrays are just contiguous areas of bytes, you can allocate your own arrays in the heap using `malloc`. The following code allocates two arrays of 1000 ints— one in the stack the usual way, and one in the heap using `malloc`. Other than the different allocations, the two are syntactically similar in use.

```
{
   int a[1000];
   int *b;

   b = malloc(sizeof(int) * 1000);

   a[123] = 13;   // Just use good ol' [] to access elements
   b[123] = 13;   // in both arrays.

   free(b);
}
```

There are some key differences:

Advantages of being in the heap

- Size (in this case 1000) can be decided at run time. Not so for an array like `a`.
- The array will exist until it is explicitly deallocated with a call to `free`. This means a pointer to the array can safely be returned from the function— not so with a local stack variable that is deallocated on function exit.
- You can change the size of the array at will at run time using `realloc`. The following changes the size of the array to 2000. The `realloc` function will potentially just stretch the original region to the new size, but if necessary, it will move the region to a new location, copy over the old elements, and free the previous region.

```
b = realloc(b, sizeof(int) * 2000);
```

Disadvantages of being in the heap

- You have to remember to allocate the array, and you have to get it right.
- You have to remember to deallocate it exactly once when you are done with it, and you have to get that right.
- The above two disadvantages have the same basic profile: if you get them wrong, your code still *looks* right. It compiles fine. It even runs for small cases, but for some input cases it just crashes unexpectedly because random memory is getting overwritten somewhere like the smiley face on page 5. This sort of "random memory smasher" bug can be a real ordeal to track down. Thank god for Purify.