Lecture 07 - Pointer arithmetic, arrays, and strings (last updated: 09/27/22)

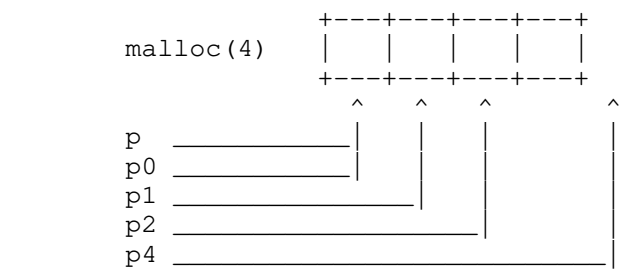
Pointers hold addresses, which are indices into memory, an array of bytes.
In this lecture, we will see how we can manipulate pointer values to access more than just the memory addresses obtained from the &-operator and malloc().

Pointer arithmetic

Since pointers are "just" numeric values, we can also perform arithmetic on them as if they were numbers. This is called "pointer arithmetic."

```
char *p = malloc(4);    // allocate 4 bytes of memory on the heap
char *p0 = p;           // points to exactly where p points
char *p1 = p + 1;       // points 1 byte past where p points
char *p2 = p + 2;       // points 2 bytes past where p points
char *p4 = p + 4;       // points 4 bytes past where p points,
                        // beyond the range we allocated
```

Although p is a pointer to a char (1 byte), we've allocated 4 bytes for it on the heap; we can use pointer arithmetic to access the other bytes we allocated. Here's what that looks like in memory (addresses increasing from left to right):

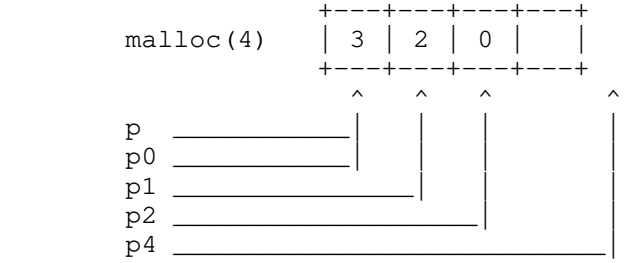


p4 points to memory we didn't allocate; dereferencing it is a memory error! When doing pointer arithmetic, always be mindful of what range of memory is valid to access.

We can dereference the other pointers to write into the heap:

```
*p = 4;
*p0 = 3;
*p1 = 2;
*p2 = 0;
```

Which gives us:

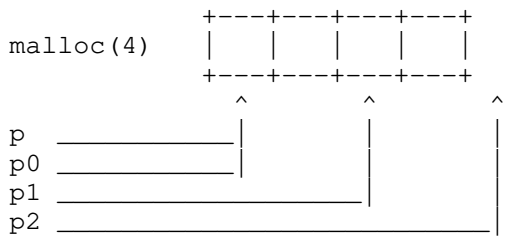


When you write something like p + i for pointer arithmetic, you don't always advance pointer p by i bytes; instead, you advance p by i * sizeof(*p) bytes.

So if p were declared as a pointer to shorts (2 bytes), i.e.,:

```
short *p = malloc(4);   // allocate 4 bytes of memory on the heap
short *p0 = p;          // points to exactly where p points
short *p1 = p + 1;      // points 2 bytes past where p points
short *p2 = p + 2;      // points 4 bytes past where p points
                        // beyond the range we allocated
```

The memory diagram would look like this instead:



When we write p + i, where p is a pointer, we sometimes call i the "offset". i cannot be another pointer, something like p + p is considered invalid pointer arithmetic (unless we cast p into an integer first).

We can, however, subtract pointers from one another, to obtain an offset. We can also subtract an offset from a pointer to obtain another pointer:

```
int i = p2 - p0;
assert(i == 2);

int *pp1 = p2 - 1;
assert(pp1 == p1);
assert(pp1 == p + 1);
```

We can compare pointers using any of the comparison operators:

```
assert(p == p0);
assert(p != p1);
assert(pp1 <= p1);
assert(p1 < p2);
assert(p2 > p);
assert(p2 >= pp1);
```

Pointers can also be incremented and decremented:

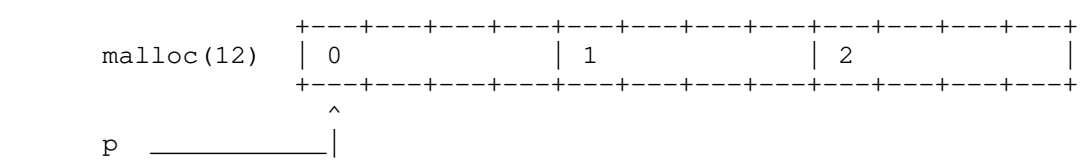
```
p++;                    // same as p = p + 1;
p += 2;                 // same as p = p + 2;
int i3 = *p--;          // p is decremented by 1, and
                        // i3 is assigned what p used to point to
```

Arrays

Now that we know pointer arithemtic, we have everything we need to work with arrays in C! For example, here's a 3-element array of heap-allocated integers:

```
int *p = malloc(sizeof(int) * 3);
*p       = 0;
*(p + 1) = 1;
*(p + 2) = 2;
```

And its memory diagram:



We write *(p + i) so often that C gives us special notation (syntactic sugar) for dereferencing arrays:

```
p[0] = 3;               // same as writing *(p + 0) = 3; or *p = 3;
p[1] = 4;               // same as writing *(p + 1) = 4;
int *p2 = &p[2];        // same as writing int *p2 = &*(p + 2);
                        //              or int *p2 = p + 2;
```

There is really nothing special about writing p[i] vs *(p + i). In fact, you can abuse the notation and write this (but please don't):

```
1[p] = 4;               // same as writing *(1 + p) = 4;
                        //   equivalent to *(p + 1) = 4;
                        //   equivalent to p[1] = 4;
```

The following two ways of 0-initializing an array are almost equivalent:

```
int *p = malloc(sizeof(int) * 10);

for (int i = 0; i < 10; i++)
    p[i] = 0;           // initialize the ith element to 0
```

versus:

```
int *p = malloc(sizeof(int) * 10);

for (int i = 0; i < 10; i++)
    *p++ = 0;           // write 0 to what p points to, then advance p

// p now points past the end of the range we allocated;
// reset it to point to the beginning of the heap array:

p -= 10;
```

Some facts about arrays:

- We refer to arrays using a pointer to its first element.
- Arrays are always contiguous in memory; that is, their elements are always laid out next to each other in memory, so that their addresses can be easily obtained using pointer arithmetic.
- Always make sure to initialize your array elements before reading from them! Uninitialized array contents are a common source of memory errors.
- Arrays don't usually carry information about how many elements they hold. If you aren't careful, you can easily make out-of-bounds array acccesses, which are another common source of memory errors.

Named arrays

Arrays can also exist as local or global variables. We define them like this:

```
int a[10];              // 10 integers in contiguous memory

for (int i = 0; i < 10; i++)
    a[i] = 0;           // initialize the ith element to 0
```

When the variable a appears in an expression, it "decays" into a pointer to the first element of the array allocated for a:

```
int i2 = a[2];
int i3 = *(a + 3);
int *p4 = &a[4];
int *p5 = a + 5;
```

Accessing memory beyond what is allocated for an array is still a memory error:

```
a[11] = 5;              // memory error
```

I like to call these local or global arrays "named arrays" because, unlike heap-allocated arrays, they have a variable name (namely, "a"); most people just call them "arrays."

Named arrays are different from pointers in two ways:

- Named arrays cannot be incremented or reassigned like regular pointers can; the following statements lead to compiler errors when a is a named array:

```
a++;
a = a + 6;
```

- The sizeof operator measures the size of the array allocated for a named array, rather than the size of a pointer.

```
int a[10];
int *p = a;

assert(sizeof(a) != sizeof(p));
assert(sizeof(p) == sizeof(int *));
assert(sizeof(a) == sizeof(int) * 10);
```

While we're discussing sizeof, something else to keep in mind: all pointers are the same size, no matter what type they point to:

```
assert(sizeof(int *) == sizeof(char *));
assert(sizeof(int *) == sizeof(void *));
assert(sizeof(int *) == sizeof(int **));
```

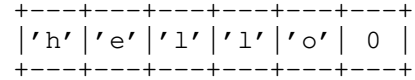
Strings and null terminator characters

A string is an array of characters; if you've already seen char *s being used as strings in C, that will hopefully make more sense now!

However, arrays don't usually carry any information about their size; after all, when we ask for an array, all we have is a pointer to the first element.

Strings work around this limitation by designating the value of zero to indicate the end of a string, we call this value the "null terminator."

For example, the string "hello" in C looks like this in memory:



Recall that character literals like 'h' and 'e' are just notation for the numbers 104 and 101. We usually write '\0' to convey that we are using the zero value as a null terminator for characters:

```
char *hello = malloc(6);
hello[0] = 'h';
hello[1] = 'e';
hello[2] = 'l';
hello[3] = 'l';
hello[4] = 'o';
hello[5] = '\0';
```

Note that we needed to allocate 1 extra byte for the null terminator character.

The null terminator allows us to compute the length of a string using only a pointer. For example, this is how strlen() is implemented:

```
int strlen(char *s) {
    int i = 0;
    while (*s++)
        i++;
    return i;
}
```

When we write *s++, this should be read as *(s++), i.e., post-fix increment the pointer s, but also dereference the value s currently points to.

It is an excellent exercise to implement various string operations on your own, such as strcat(), strncat(), strcpy(), strncpy(), strchr(), and strrchr(); consult their man pages for their prototype and intended uses.

There are multiple ways to implement each; for example, here are three different ways to implement strcpy():

```
while ((s[i] = t[i]) != 0) i++;

while ((*s = *t) != 0) { s++; t++; }

while ((*s++ = *t++) != 0) ;
```

Make sure you understand how each of these implementations works.

We can also truncate strings using the null terminator; for example, we can truncate the hello string from before like this:

```
hello[4] = '\0';        // now hello points to "hell"
hello[2] = '\0';        // now hello points to "he"
hello[0] = '\0';        // now hello points to ""
```

Even if there is other data beyond the null terminator, string-manipulating functions will not read past the '\0' character.

String literals

When a string literal appears in an expression, their value is a pointer to the first element of a char array:

```
    char *p = "abc";        // p points to the 1st element of a 4-char array

    assert("abc"[1] == 'b');
    assert(*"abc"   == 'a');
    assert("abc"[3] == '\0');
```

String literals are not stored in the stack, heap, or the data section of memory; instead, they are stored in the code or static data section, depending on the compiler and OS, and should not be modified at runtime:

```
    *p = 'A';               // result undefined (probably segmentation fault)
```

However, we can still read from them without issue:

```
    int s[10];
    strcpy(s, "hello");     // copy "hello" to s
```

Note that string literal syntax can also be used to initialize named arrays:

```
    char a[] = "abc";       // short-hand for: char a[4] = {'a','b','c','\0'};
                            // the array of characters resides on the stack
```

But this should not be confused with string literals when they appear as expressions, and evaluate to a pointer value to the string data:

```
    char *p = "abc";        // p points to the 1st element of a 4-char array,
                            // which resides in the code/static data section
```

Because the named array a lives on the stack, we can safely modify its contents without fear of a segfault:

```
    *a = 'A';               // this is ok
```

Pointers to pointers

Say we wanted an array of strings. What would that look like?

Here is one possibility:

```
    char *a[] = { "hello", "wold" };

    printf("%s %s\n", a[0], a[1]);
```

Here is another, which looks a little different in memory (see K&R2 p114 for an illuminating picture):

```
    char a[][10] = { "hello", "world" };

    printf("%s %s\n", a[0], a[1]);
```

Command line arguments are passed to main() as an array of char pointers.
When we define main() as:

```
int main(int argc, char **argv) { ... }
```

instead of:

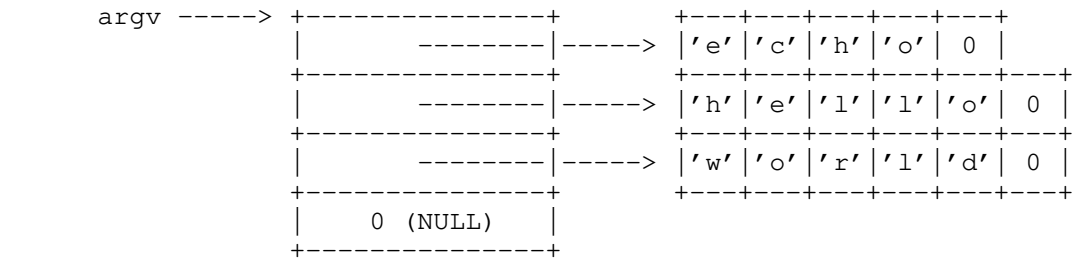
```
int main(void) { ... }
```

we can access the command line arguments through argc ("argument count") and argv ("argument vector").

For example, when you run:

```
echo hello world
```

the following data structure is passed to the main() of the echo program:



argc is set to 3, and argv[argc] is set to NULL (as illustrated).
Note that for the array of pointers, argv, we use a NULL pointer as a null terminator for pointer array; for the array of chars, i.e., each of the argument strings, we use a null character '\0' as a null terminator for each string.

Here's three different ways to implement the 'echo' program (K&R2, p115):

```
for (i = 1; i < argc; i++)
    printf("%s\n", argv[i]);

while (--argc > 0)
    printf("%s\n", *++argv;);

argv++;
while (*argv)
    printf("%s\n", *argv++);
```

Summary

Don't confuse these things:

- *p++ vs (*p)++
- pointers versus named arrays
- sizeof vs strlen()
- NULL pointers versus null terminator
- '0' versus 0 or '\0'
- string literals versus string-initialized named arrays