CS2850 Operating System Lab

Week 3: Memory basics, Pointers, Arrays

nicolo colombo

`nicolo.colombo@rhul.ac.uk`

Office Bedford 2-21

Outline

Memory, pointers, arrays, and strings

Address arithmetic

Arrays and functions

Pointers to pointers and command-line arguments

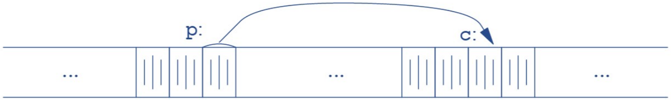
Memory basics

The memory is a *large array* of memory cells (bytes).

The array contains the entire state of your programs: variables, constants, data, and machine code.

Data are stored at specific memory addresses.

Pointers are *variables* for storing the memory address of other variables.



Two useful operators

```
int i = 1;
int *ip = &i;
```

The *address operator*, `&`, returns the *address of* `i`.

The *dereferencing* operator, `*`, returns the value stored at a given address.

`ip` is a pointer variable storing the address of `i`.

Pointers at work

```
int i = 1;
int *ip, *iq;
ip = &i;
*ip = *ip + 1;
iq = ip;
```

You can use `*ip` instead of `i` in any context, e.g. `*ip = *ip + 1;` adds 1 to `i`.

You can *initialize* or *redirect* pointers by copying the content of another pointer of the same type to it, e.g. `iq = ip;`.

Types and pointers

Pointers store the address of specific data type, e.g. `int *ip;` says that `ip` is a *pointer to integers*, i.e. `*ip` is a `int`.

All pointers have the same size, 8 bytes.

The *generic pointer type*, `void *`, can be cast to `void *` and back.

Example

This program returns the content of a `void*` address that is *casted* to a pointer to `int`.

```
int main() {
        int i = 1;
        int *ip = &i;
        void *iq;
        iq = (void *)ip;
        return *(int *)iq;
}
```

To see the return value on the terminal, run

```
gcc -Wall -Werror -Wpedantic pointers.c
./a.out
echo $?
```

Declaring arrays

The following declaration allocates 10 *consecutive* blocks of 4 bytes named `a[0]`, `a[1]`, . . . , `a[9]`,

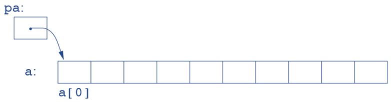
```
int a[10];
```

As `a[0]`, `a[1]`, . . . are *all* integers the program only needs to know the address of the first element, i.e. the *pointer and type to the first element*.

Pointers and arrays

The following defines a pointer to the first element of a,

```
int a[10];
int *pa = &a[0];
```



Pointers and arrays are *closely related*: the value of a (without brackets) is the *address of its first element*.

Strings

```
char *s = "hello world";
```

Strings are *null-terminated* arrays of `char`, i.e. their last `char` is `'\0'`.

The null-termination lets the program find the *end of the string*.

There are *no C operators for processing strings as units*. But you can use `printf("s=%s\n", s);` to print `s` or as because they are null-terminated.

String constants and character arrays are different

```
char *s = "string constant";
char as[20] = "character array";
```

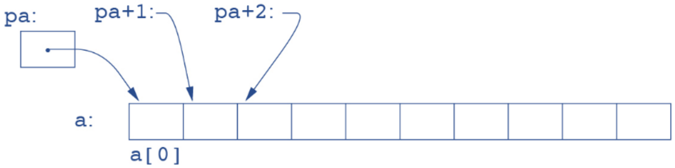
`s` is a pointer *to a constant* (un-modifiable) piece of memory.

`as` is the address of a 20-byte *character array* (you can write on it).

Address arithmetics

Let a be an array of 10 int and pa a pointer to int.

After writing pa = a;, pa + 1 points to a[1] and pa + 4 points to a[4][1]



a[i] and *(a+i) refer to the same object (the content of a[i]).

[1]The value of a (without brackets) is the *address of its first element*

Note

Portions of `s` (or `as`) can be accessed by specifying the address of a single character within them.

The following lines print the substrings `"constant"` and `"array"`,

```
char *s = "string constant";
char as[20] = "character array";
printf("%s", &s[7]);
printf("%s", s + 7);
printf("%s", &as[10]);
printf("%s", as + 10);
```

Pointers and functions

Arguments are passed to functions by value.

Functions cannot modify a variable *defined in the calling function*.

To *save the changes* you need to

- define a function with pointer arguments, e.g.

  ```
  void f(int *a) {*a = 5;}
  ```

- pass a pointer when you call `f`.

  ```
  int a = 3;
  f(&a);
  printf("a=%d\n", a);
  ```

Pointers to pointers

Pointers and can store the address of other pointers.

You can use an array of pointers to `char` to store a *list of strings*,

```
char *sa[10];
sa[0] = "hello";
sa[1] = ", ";
sa[2] = "world";
sa[3] = "!";
sa[4] = NULL;
int i = 0;
while (*(sa + i)) {
    printf("sa[%d]=%s\n", i, sa[i]);
    i++;
}
```

Command-line arguments

C programs accept command-line arguments through a strings array called `argv`.

```
int main(int argc, char **argv) {
    int i = 1;
    while (i < argc) {
            printf("argv[%d]=%s\n", i, argv[i]);
            i++;
    }
}
```

The output is as before,

```
cim-ts-node-02$ ./a.out hello ,  world !
argv[1]=hello
argv[2]=,
argv[3]=world
argv[4]=!
```