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

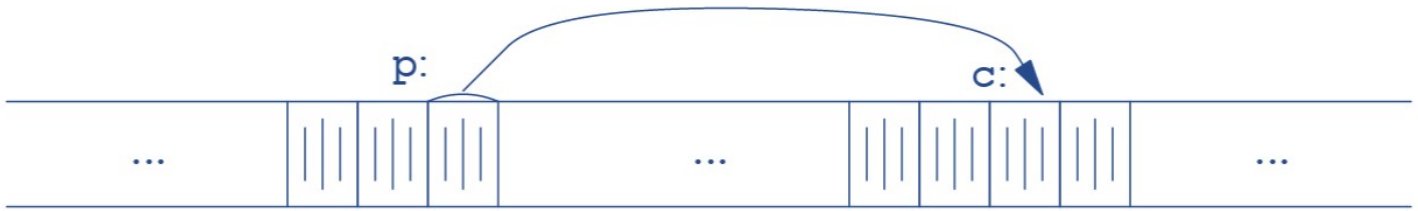
memory basics

the memory contains the entire state of your program: variables, constants, data, machine code

computer memory is a sequence of memory cells (bytes), essentially like a very large array

a memory address is an index in this array

pointers are *variables* that store the memory address of other variables



two useful operators

`&` (*address operator*) gives the address of an object, e.g.

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

assigns the address of `i` to the pointer `ip`

`*` (*dereferencing* operator) gives the value stored at the pointed address, e.g.

```
*ip = 2;
```

assignes the value 2 to `i`

pointers at work

after the assignment `ip= &i;` you can use `*ip` instead of `i` in any context, e.g. `*ip = *ip + 1;` adds 1 to `i`

pointers *of the same type* can be used directly, e.g. `iq = ip;` makes the pointer `iq` point to what `ip` points to

`void *` is the *generic pointer type* (normally used for pointer arguments) and any pointer can be cast to `void *` and back again without loss of information

pointers and data types

pointers point to the data type specified in their declaration, e.g. `int *ip;` says that `ip` is a pointer to integers, i.e. `*ip` is a `int`

the dereferencing operator can be used in a function declaration to specify that the function *accepts a pointer as a parameter*, e.g. `double f(char *c);` means that

- `f` returns a `double`
- the argument of `f` is a pointer to `char`

example

this program prints the memory address of an `int` and a `int *` variables

```
#include <stdio.h>
int main() {
  int i = 1;
  int *ip1, *ip2;  //declares two pointers to int
  int **iq;        //declares a pointer to a pointer to int
  ip1 = &i;        //assigns the address of i to ip1
  iq = &ip1;       //assigns the address of ip1 to iq
  *ip1 = 2;    //assigns the value 2 to i
  ip2 = ip1;       //makes ip1 and ip2 point to the same location
  printf("ip1=%p, iq=%p, ir=%p\n",
      (void *) ip1, (void *) ip2, (void *) iq);
  printf("*ip1=%d, *ip2=%d, *iq=%p\n",*ip1, *ip2, (void *) *iq);
}
```

except for `*ip1=2, *ip2=2`, the output *depends on the specific run* because memory is allocated randomly

declaring arrays

the declaration

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

allocates in memory 10 *consecutive* blocks of 4 bytes named `a[0]`, `a[1]`, `...`, `a[9]`

the declaration also specifies that the binary information stored in each block should be interpreted as an `int`

as the integers are stored in consecutive memory cells and the program only needs to know the address of the first element

after the declaration the memory is uninitialised

pointers and arrays

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

defines a pointer to the first element of `a`, i.e. the variable `pa` contains the address of `a[0]`

variables of type array

the value of `a` (without brackets) is the *address of its first element*:

- `pa = &a[0];` and `pa = a;` are equivalent statements
- `a[i]` and `*(a+i)` refer to the same object
- `&a[i]` and `a+i` are identical

inside a function, C converts `a[i]` to `*(a+i)` immediately but pointers are variables and array names are *not*, e.g.

- `pa = a` and `pa++` are legal
- `a = pa` and `a++` are illegal

example (1)

this program shows the equivalence between `a`, `&a[0]`, and `pa`

```
#include <stdio.h>
int main() {
  int i = 3;
  int a[10];
  for (int j=0;j<10;j++) a[j]= j + 1;
  int *pa, *pai;
  pa = a;
  pai = pa + i;
  printf("a=%p, pa=%p, pai=%p, &a[3]=%p\n",
      (void *) a, (void *) pa, (void *) pai, (void *) &a[3]);
  printf("*a=%d, *pa=%d, *pai=%d, a[3]=%d\n",
      *a, *pa, *pai, a[3]);
}
```

example (2)

two consecutive runs (on the same machine) produced the following outputs

```
cim-ts-node-02$ ./a.out
a=0x7fff4b53e5d0, pa=0x7fff4b53e5d0, pai=0x7fff4b53e5dc, &a[3]=0x7fff4b53e5dc
*a=1, *pa=1, *pai=4, a[3]=4
cim-ts-node-02$ ./a.out
a=0x7fffe9773e80, pa=0x7fffe9773e80, pai=0x7fffe9773e8c, &a[3]=0x7fffe9773e8c
*a=1, *pa=1, *pai=4, a[3]=4
```

for each run, try to predict the output if you add the following extra lines to the program

```
printf("*(a + 2)=%d\n", *(a + 2));
printf("(a + 2)=%p\n", (void *) (a + 2));
```

arrays of `char`

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

the null-termination lets the compiler find the end and it is possible to compute their length *at runtime*

you can use both the following statements to *declare* and *initialise* a string

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

but `s` and `as` are not equivalent: `s` is a *pointer* (a variable) and `as` is an *array*, which is associated with a (fixed) amount of allocated memeory

strings vs arrays

as for arrays, strings are *accessed* by pointing to their first element

as for arrays, there are *no C operators for processing an entire string as a unit*

differently from arrays, you can use `printf("s=%s\n", s);` to print `s`, idem for `as`

string processing

portions of `s` (or `as`) can be accessed by specifying the address of a single characters within them

the following lines all print the substring `"constant"` (with `s` and `as` defined above)

```
printf("%s", &s[6]);
printf("%s", &as[6]);
printf("%s", s + 6);
printf("%s", as + 6);
```

example (1)

this program exploits the fact that strings are null-terminated to compute the length of **"hello world"**

```
#include <stdio.h>
int length(char *s) {
  int i = 0;
  while (*(s + i) != '\0')
    i++;
  return i;
}
int main() {
  char *s = "hello world";
  char as[] = "hello world";
  printf("s=%s, as=%s\n", s, as);
  printf("length(s)=%d, length(as)=%d\n", length(s), length(as));
  printf("s+6=%s, as+6=%s\n", s+6, as+6);
  printf("length(s+6)=%d, length(as+6)=%d\n", length(s+6), length(as+6));
}
```

example (2)

the output on the terminal is

```
s=hello world, as=hello world
length(s)=11, length(as)=11
s+6=world, as+6=world
length(s+6)=5, length(as+6)=5
```

how could you print only the first part of the string, e.g. the substring **hello**? add the following lines to the program above and observe what happens

```
*(as + 5) = '\0';
printf("as=%s\n", as);
```

can you do the same with **s**?

example (3)

this program attempts to remove the null-termination of `as`

```
#include <stdio.h>
int main() {
        char *s = "hello world";
        char as[] = "hello world";
        printf("s[1]=%c, as[1]=%c\n", s[1], as[1]);
        printf("s[11]=%c, as[11]=%c\n", s[11], as[11]);
        as[11] = 'x';
        as[12] = 'x';
        printf("s=%s, as=%s\n", s, as);
}
```

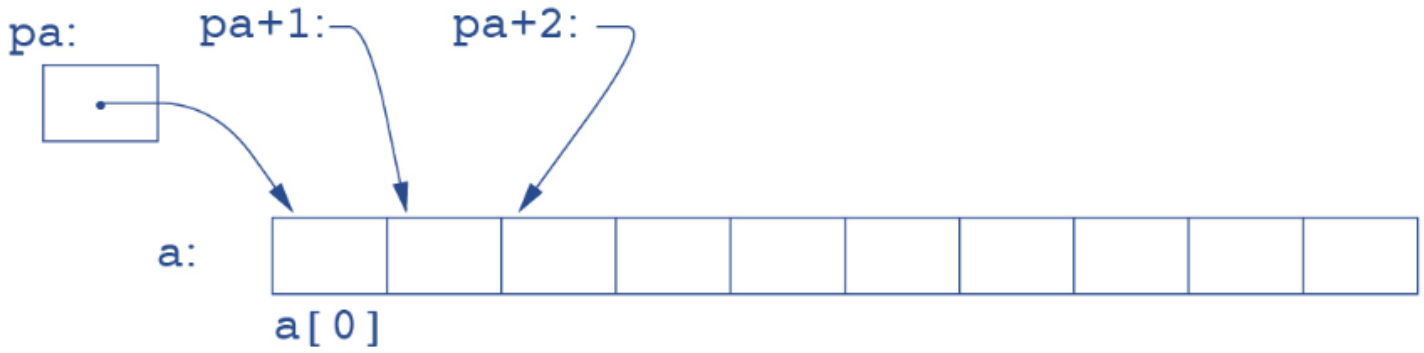
try to understand why the executable crashes and you obtain the following output

```
s[1]=e, as[1]=e
s[11]=, as[11]=
s=hello world, as=hello worldxxtR%2*oI
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

address arithmetics

pointers are variables and you can perform arithmetic operations with them

let `a` be an array of 10 `int` and `pa` a pointer to `a[0]`, then



e.g. `pa + 1` points to `a[1]` and `pa + 4` points to `a[4]`

pointer arithmetics is only allowed with arrays, i.e. if `int i = 5;` and `int *ip = &i;`, `*(ip + 1)` is undefined

a powerful tool

pointer arithmetics is useful to *access* single elements of an array (as above) and

- works regardless of the type of `a`, e.g. `a + 1` is the second element of `a` even when `a` is defined as `long a[10];`
- works regardless of the size of `a`, e.g. `pa + 100` points to some unreserved memory slot (and may cause unexpected behaviours)

go back to the codes in the previous slides to see a few examples of how to *implement* this in your code

pointers and functions

in C, arguments are passed to functions by value

there is *no direct way* for the called function to modify a variable in the calling function

to affect the calling program you need to pass a pointer to the variable i.e. to make a function accept pointer arguments

this is specified in the definition of the function, e.g.

```
void f(int *a, int *b) {...}
```

how do you call `f`? what is the *value of the arguments* you need to pass?

example

this program calls two functions (a good and a bad one) for swapping two integers

```
#include <stdio.h>
void swap(int *i, int *j) {
  int temp = *i;
  *i = *j;
  *j = temp;
}
void swapWrong(int i, int j) {
  int temp = i;
  i = j;
  j = temp;
}
int main() {
  int i = 1, j = 2;
  swap(&i, &j);
  printf("i=%d, j=%d\n", i, j);
  swapWrong(i, j);
  printf("i=%d, j=%d\n", i, j);
}
```

example (2)

```
#include <stdio.h>
void editString(char *s) {
  *(s+2) = 'x';
}
void editArray(int *a) {
  *(a+2) = -1;
}
int main() {
  int a[10];
  char as[] = "hello world";
  for (int i=0;i<10;i++) a[i] = i+1;
  editString(as);
  editArray(a);
  printf("as=%s\n", as);
  for (int i=0;i<10;i++)
    printf("a[%d]=%d\n", i, a[i]);
}
```

why does the program above work even if no address operator appears in `main`? and what happens if you replace `char as[] = "hello world";` with `char *as = "hello world";`?

pointers to pointers

pointers are variables and can be stored in arrays

for example, an array of pointers to `int` and an array of pointers to strings can be declared as

```
int *pa[10];
char *sa[10];
```

each entry of the array is a pointer and can be later *assigned* to a variable of the specified type

example (1)

this program declares and initialises an *array of integers*, `a`, an *array of strings*, `sa`, and an *array of pointers to integers*, `pa`

```
#include <stdio.h>
int main() {
  char *sa[3];
  sa[0] = "one";
  sa[1] = "two";
  sa[2] = "three";
  int *pa[3];
  int a[3];
  for (int i=0;i<3;i++) a[i] = i + 1;
  for (int i=0;i<3;i++) pa[i] = &a[i];
  for (int i=0;i<3;i++) {
    printf("sa[%d]=%s\n", i, *(sa +i));
    printf("a[%d]=%d\n", i, *(a +i));
    printf("pa[%d]=%p\n", i, (void *) *(pa +i));
    printf("*pa[%d]=%d\n", i, **(pa + i));
  }
}
```

example (2)

the output of the program is

```
sa[0]=one
a[0]=1
pa[0]=0x7fff13580c10
*pa[0]=1
sa[1]=two
a[1]=2
pa[1]=0x7fff13580c14
*pa[1]=2
sa[2]=three
a[2]=3
pa[2]=0x7fff13580c18
*pa[2]=3
```

what happens if you i) try to re-assign a pointer, e.g. you add `sa[0] = "minus one";` or `*pa[0] = -1;` before printing and ii) try to re-assign a specific `char` of the string constant `"one"`?

command-line arguments

you can make a program accept command-line arguments by defining your `main` as

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

where:

- `int argc` is the number of optional arguments
- `char *argv[]` is a string array
- the entries of `argv` point to the start of each optional argument (arguments are treated as strings)
- `argv[i]` is a pointer to the `i`th argument
- argc ≥ 1 because `argv[0]` is the name of your program
- `argv[argc] = NULL` is a *null pointer*

example

this program prints an *arbitrary number of* command-line arguments and exits

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  for (int i = 1; i<argc; i++)
    printf("argv[%d]=%s\n", i, argv[i]);
}
```

the output depends on how the executable is run, e.g.

```
>./a.out
>./a.out one
argv[1]=one
>./a.out one two 3
argv[1]=one
argv[2]=two
argv[3]=3
```