CS2850 Operating System Lab

Week 10: Review of pointers, strings, and process control

nicolo colombo

nicolo.colombo@rhul.ac.uk

Office Bedford 2-21

outline

Pointers

Strings

Pointers and functions

`fork`

Pointers (1)

Pointers are variables that store the memory address of other variables.

Address operator: `&i` is the address of `i`.

Dereferencing operator: `*ip` is the value of the variable stored at the address `ip`.

You can print on `stdout` the *unpredictable value* of `ip`, i.e. the address stored in `ip`, with

```
printf("%p",( void *) ip)
```

Pointers (2)

The type of a pointer is the type of variable stored at the pointed address.

The data type of a variable specifies how to interpret the data stored in the variable.

A pointer of a given type cannot store the address of variables of a different type, e.g.

```
int i = 1;
char *pc = &i
```

is illegal and produces compilation errors.

Example: int, char, int*, and char* (1)

```
#include <stdio.h>
int main() {
  int i = 10;
  int *pi = &i;
  char c = 'a';
  char *pc = &c;
  printf("sizeof(pi)=%lu\n", sizeof(pi));
  printf("sizeof(pc)=%lu\n", sizeof(pc));
  printf("pi=%p and  &i = %p\n", (void *) pi, (void *) &
      i);
  printf("pc=%p and &c = %p\n", (void *) pc, (void *) &c
      );
  printf("*pi=%d and i = %d\n", *pi, i);
  printf("*pc=%c and c = %d\n", *pc, c);
}
```

Output

```
./a.out
sizeof(pi)=8
sizeof(pc)=8
pi=0x7ffd17e6b974 and  &i = 0x7ffd17e6b974
pc=0x7ffd17e6b973 and &c = 0x7ffd17e6b973
*pi=10 and i = 10
*pc=a and c = 97
```

Strings (1)

String constants are *null-terminated* arrays of `char`, e.g.

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

`s` is a pointer to `char` that stores the address of the first element of the string `"hello world"`.

The last `char` of `s` is the null character `'\0'`.

Strings (2)

String *constants* and character arrays, e.g. `s` and `as` defined by

```
char *s = "hello world";
char as[12] = "hello world";
```

are *not equivalent*.

`s` is a pointer (a variable) and can be *assigned to a different address*.

`as` is an array and always refers to *fixed allocated* storage.

Strings (3)

`s[7]` and `as[7]` both return `'w'` but

```
s[7] = 'x';
```

causes a `Segmentation Fault`, while

```
as[7] = 'x';
```

is legal and sets the 8th character of `as` to `'x'`, i.e. `'o'` becomes `'x'`.

Strings (4)

You can use `printf("%s", s)` and `printf("%s", as)` to print the strings `s` and `as` up to their null-termination character.

In C, there are *no built-in operators* for processing an entire string as a unit, e.g.

```
char as[12];
as = 'hi world';
```

is *illegal*, but you can *re-assign* `s` to point to another string constant, e.g.

```
char *s;
s = 'hi world';
```

Strings (6)

You can print a portion of `s` or `as` by specifying the address of a single character, e.g.

```
s = "hi world";
printf("(s + 3) = %s", (s + 3));
```

prints `(s + 3) = world`.

You can cut a character array by setting one element to `'\0'`, e.g.

```
char as[10] = "hi world";
*(as + 2) = '\0';
printf("as = %s and (as + 3) = %s\n", s, (s + 3));
```

prints `as = hi and (as + 3) = world`.

Example: char* and char[13]

```
#include <stdio.h>
int main() {
  char *s = "hello world";
  char sa[13] = "hello world";
  //s[7] = 'x'; is illegal
  sa[7] = 'x';//is legal
  printf("s[7] = %c\n",s[7]);
  printf("sa[7] = %c\n",sa[7]);
  printf("(s + 1) = %s\n",s + 1) ;
  printf("(sa + 1) = %s\n",sa + 1);
  s = "hi world";
  printf("s = %s\n",s);
  //sa = s; is illegal
  s = &sa[6];//is legal and equivalent to s = sa + 6;
  printf("s = &sa[6] => s=%s\n",s);
  printf("(sa + 6) = %p and s = %p\n",(void *) (sa + 6),
       (void *)s);
}
```

Output

```
./a.out
s[7] = o
sa[7] = x
(s + 1) = ello world
(sa + 1) = ello wxrld
s = hi world
s = &sa[6] => s=wxrld
(sa + 6) = 0x7ffd23e608c1 and s = 0x7ffd23e608c1
```

Example: pointer arithmetic

```
#include <stdio.h>
int main() {
  char as[100] = "one two three four five";
  char vs[100]= "one two three four five";
  char *s = "one two three four five";
  for (int i = 0; i < 13; i++) s++;
  as[13] = '\0';
  printf("as = %s\n", as);
  printf("vs = %s\n", vs);
  printf("s = %s\n", s);
}
```

Output:

```
as = one two three
vs = one two three four five
s =  four five
```

Pointers and functions (1)

The de-referencing operator can be used in function declarations, e.g.

```
double f(char *c);
```

says that `f` returns a `double` and the argument of `f` is a pointer to `char`.

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 change the value of a variable in the calling function, pass the address of that variable.

Pointers and functions (2)

Arrays are represented by *the location of the initial element*.

When you pass an array to a function, you will see the changes in `main`, e.g.

```
#include <stdio.h>
void initialise(int *a) {
  int i;
  for (i = 0; i < 10; i++) *(a + i) = i;
}
int main() {
  int i, a[10];
  initialise(a);
  for (i = 0; i < 10; i++) printf("%d ", *(a  + i));
}
```

prints 0 1 2 3 4 5 6 7 8 9.

Example: swapping `int`

```
#include <stdio.h>
void swap(int *pi, int *pj);
void swapWrong(int i, int j);
int main() {
  int x = 1, y = 2, *px = &x,  *py = &y;
  printf("x = %d, y = %d, px = %p, and py = %p\n", x, y, (void *) px,  (void *) py);
  printf("calling swapWrong ... \n");
  swapWrong(x, y);
  printf("x = %d, y = %d, px = %p, and py = %p\n", x, y, (void *) px,  (void *) py);
  printf("calling swap ... \n");
  swap(&x, &y);
  printf("x = %d, y = %d, px = %p, and py = %p\n", x, y, (void *) px,  (void *) py);
}
```

Output:

```
./a.out
x = 1, y = 2, px = 0x7fff73496720, and py = 0x7fff73496724
calling swapWrong ...
x = 1, y = 2, px = 0x7fff73496720, and py = 0x7fff73496724
calling swap ...
x = 2, y = 1, px = 0x7fff73496720, and py = 0x7fff73496724
```

Auxiliary functions

A function to swap the *content of two addresses*.

```
void swap(int *pi, int *pj) {
  int temp = *pi;
  *pi = *pj;
  *pj = temp;
}
```

A function to swap two *stack variables*.[1]

```
void swapWrong(int i, int j) {
  int temp = i;
  i = j;
  j = temp;
}
```

[1] All changes are discarded when the function returns.

Process control (1)

`unistd.h` and `sys/wait.h` contains the functions you need to control a process from a C program.

`int getpid()` returns the process identifier (PID) of the running process.

`int fork()` creates a child process that is a copy of the running process.

`void exit()` or `return` terminate the current process.

`int wait(int * status)`: to wait for the *first* child that terminates (call it several times if you have generated more than one child). [2]

[2] See more about process control in Section 26 of the GNU online manual.

Process control (2)

The OS assigns PIDs in increasing order but their values are unpredictable.

In C, `fork` is *called once but returns twice*.

The return value is -1 if the process creation failed, 0 in the child process, and the child's PID in the parent process.

Process control (3)

When `fork` returns, the child address space is a copy of the parent address space, but the parent and the child have different PIDs.

The return values of `fork` allow you to make conditional statements.

The address spaces of the child and the parent are *private*, i.e. the parent cannot see the changes made by the child, and vice versa.

The parent and the child run concurrently. The execution order is unpredictable.

Example: private but identical address spaces

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
int main() {
  int j = 0;
  printf("PID = %d, j = %d, &j = %p \n", getpid(), j, (void *) &j);
  int pid  = fork();
  if (!pid) j = 1;
  else j = 2;
  printf("PID = %d, j = %d, &j = %p \n", getpid(), j, (void *) &j);
}
```

Output:

```
./a.out
PID = 2789388, j = 0, &j = 0x7fff5ea69010
PID = 2789388, j = 2, &j = 0x7fff5ea69010
PID = 2789389, j = 1, &j = 0x7fff5ea69010
```