C recap

ARCOS Group

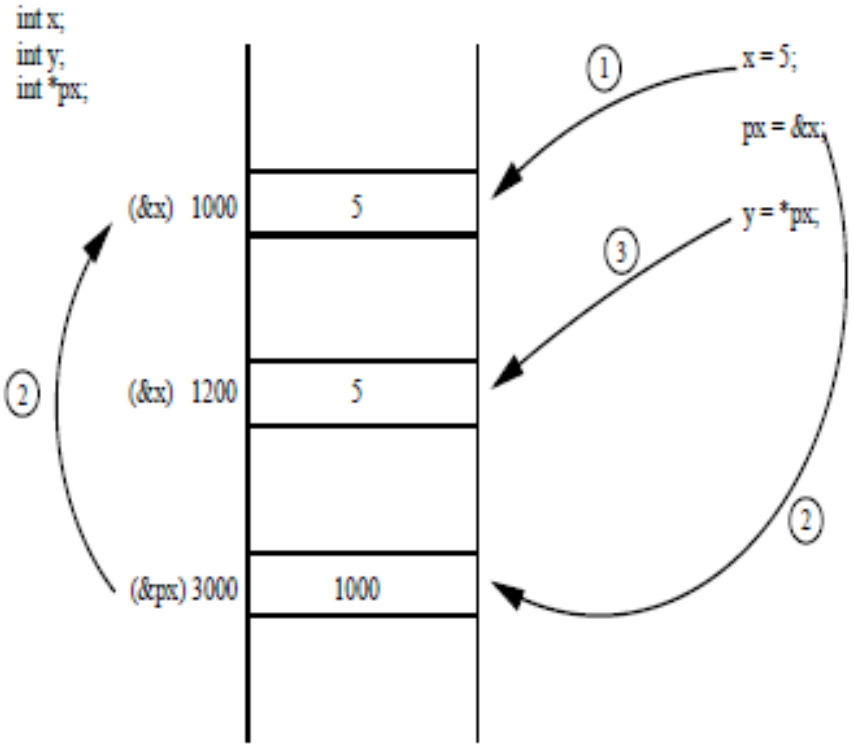
Distributed Systems

Bachelor In Informatics Engineering

Universidad Carlos III de Madrid

Pointers

- `*px` represents the content of the memory address which is targered by `px`
- `*` indirection operator is only relevant for pointers
- `&` is relevant for any variable (including pointers)
- A pointers is a variable which stores a memory address, never its value



Example

- In:

```
char *p;
char letra;
letra = 'a';
p = &letra;
*p = 'b';
```

- ¿How many bytes are used in `p`?
- ¿How many bytes are used in `*p`?
- ¿Does it correct?

Example

```
#include <stdio.h>

main()
{
    int x;
    int y;
    int *px;

    x = 5;
    px = &x;
    y = *px;

    printf("x = %dn", x);
    printf("y = %dn", y);
    printf("*px = %dn", *px);
```

Example

```
float n1;
float n2;
float *p1;
float *p2;

n1 = 4.0;
p1 = &n1;

p2 = p1;

n2 = *p2;

n1 = *p1 + *p2;
```

- ¿Value of `n1` and `n2`?

Example

Given the following code:

```
float a = 0.001, *b, *c;
b = &a;
c = b;
a = *c + *b;
```

¿Which of the following statements is correct?

A.- `b` and `c` are stored in the same memory address.

B.- `*c = 4;` modifies the content of variable `a`.

C.- `a` takes a indeterminated value.

D.- `c` stores the memory address of the variable `b`.

Example

Given the following piece of code:

```
int *a, x, y;
x = 1;
a = &x;
y = 2;
x = y * 2;
```

¿`printf("%d %d\n", y, *a);` ?

A.- 2 1

B.- 2 4

C.- Error.

D.- 2 and an indeterminated value.

Passing arguments with pointers

- When you pass a pointer to a function that is passed a copy of the data but the address pointed to.
- The use of pointers allows passing arguments by reference .
- Using pointers as arguments of functions allows the data to be altered within the function globally.

Example

```
#include <stdio.h>
double sum_array(double  v[], int n) {
  int i;
  double s = 0.0;
  for (i=0;i<n;i++) {
    s = s + v[i];
  }
  return s;
}

int main() {
  double w[10];
  for (int i=0;i<10;i++) {
        w[i] = 1.0 * i;
  }
  printf("Sum %lf\n", sum_array(w,10));
  return 0;
}
```

Example

```
#include <stdio.h>

void swap(int *a, int *b); /* prototype*/

main() {
   int x = 2;
   int y = 5;

   printf(”Before x = %d, y = %d\n", x, y);
   swap(&x, &y);
   printf(”After x = %d, y = %d\n", x, y);
}

void swap(int *a, int *b) {
   int temp;

   temp = *b;
   *b = *a;
   *a = temp;

   return;
}
```

Example

Given the following code:

```
void f(int *p){
     p = NULL;
}
int  main(void){
     int i=1, *ptr;
     ptr = &i;
     f(ptr);
}
```

¿Which is the value of ptr after calling function f?

A.- Memory address of i

B.- NULL

C.- 1

D.- Error

Example

Given the following code:

```
float avg(int a, float x[]);
int n;
float v[25];
float m;
```

¿Which of the following statements is correct?

A.- `m = avg(v, n);`

B.- `m = avg(n, *v);`

C.- `m = avg(n, v[25]);`

D.- `m = avg(n, v);`

Example

Given to following fragment:

```
int v[10] = {1,2,3,4,5,6,7,8,9,10};
int i, *a;
for (i = 0; i < 10; i ++) {
a = &v[i];
*a = v[0] + i;
}
```

¿Which of the following statements is correct?

A.- `v` is wrong defiened;

B.- `v[5]` = 7.

C.- `v[3]` = 4.

D.- `a = &v[i]` is incorrect.

Example

¿Which of the following statements allocs memory for a vector of 10 elements of type `float`?

A.- `(float) malloc (10);`

B.- `(float *) malloc(10);`

C.- `(float *) malloc(10 * sizeof(float));`

D.- `(float *) malloc(10 * sizeof(float *));`

Example

Given the following variables:

```
int x, *p1, **p2;
```

¿Which is the correct statement if we want to store 4 in variable x?

A.- `p1 = &p2; *p2 = &x; *p1 = 4;`

B.- `p2 = &x; *p2 = 4;`

C.- `p2 = p1; p1 = &x; *p2 = 4;`

D.- `p2 = &p1; p1 = &x; **p2 = 4;`

Structs

```
struct struct-name
   {
      type1 item_1;
      type2 item_2;
      .
      .
      typeN item_N;
   };
```

Pointers to structs

```
struct date
   {
      int month;
      int day;
      int year;
   };
```

- We define `struct date *p;`
  - It is not possible use `p` if it does not target to a variable of type `struct date`
- We can allocate dynamic memory

```
p = (struct date *) malloc(sizeof(struct date));
```

Example

Given the following definiction:

```
struct complex{
   float real;
   float imag;
   };
struct complex p1, *p2;
```

¿Which of the following statements is correct?

A.- `p1 = &p2;`

B.- `printf("%d", p1.real);`

C.- `printf("%f", p1->imag);`

D.- `printf("%f", p2.real);`

Example

Given the following code

```
struct something{
    double x;
    double y;
};
struct somthing p1;
struct somthing components[100];
```

¿Which of the following statements is correct?

A.- `components[5] = p1.x;`

B.- `components[i] = p1.x;`

C.- `components[5].x = p1->x;`

A.- `componeents[5].x = p1.y;`

Numeric datatypes

- `char <= short <= int <= long <= long long int`

arch	Size:	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386		1	2	4	4	4	8	1	2	4	8
alpha		1	2	4	8	8	8	1	2	4	8
armv4l		1	2	4	4	4	8	1	2	4	8
ia64		1	2	4	8	8	8	1	2	4	8
m68k		1	2	4	4	4	8	1	2	4	8
mips		1	2	4	4	4	8	1	2	4	8
ppc		1	2	4	4	4	8	1	2	4	8
sparc		1	2	4	4	4	8	1	2	4	8
sparc64		1	2	4	4	4	8	1	2	4	8
x86_64		1	2	4	8	8	8	1	2	4	8

- `float <= double <= long double`

sizeof()

- The function **sizeof()** pallows get the variable/datatype size (**in bytes**)
- Not recommended for strings
- Examples:
  - sizeof(char) [returns1]
  - long a;

    sizeof(a) [returns 4]

We don't have booleans (C90)

- But we can use numbers:
  - 0 → False
  - >= 1 → True

- Caution:

```
#include <stdio.h>
int main() {
  int x, y;
  x=1;
  y=2;
  if (x=y) { printf("Same\n"); }
  return 0;
}
```

Pointers

- A **pointer variable** stores a memory address
- **&** obtains the memory address of a variable
- * represents:
  - Si se encuentra en la definición de la variable, que es un puntero.
  - Si se encuentra en el código *"contenido de"*

- Pointer declaration:
  - type_ptr *nb_ptr;
    - nb_ptr variable which stores a memory address,
    - type_ptr the base datatype of the pointer

Pointer example

```
int x = 3;
int *p;        /* p: pointer of int */
*p = 7;        CRASH!

int x = 3;
int *p = &x; /* p targets to x */
int *q;
*p = 7;        /* x stores 7 */
q = p;         /* q targets to the same memory address of p */
p = 0;         /* p does not target x. Null pointer */
p = NULL;      /* Null pointer as well */
*p = 7;        /* Not defined behaviour */
*q = 0;        /* x stores 0 */
```

Pointers and parameters

```
#include <stdio.h>

void aggregate(int * s, int x) {
  *s += x;
}

int main() {
  int sum= 0;
  int i;
  for (i=0;i<10;i++) {
    aggregate(&sum,i);
  }
  printf("The sum is %d\n",sum);
}
```

Example: Arrays

```
int main() {
  int v[100];
  double w[] = { 1.0, 3.5, 7.0 }; /* Size = 3 */
  float y[];  /* ERROR. Size not defined */

  v[0] = 3;
  v[10] = v[0];
  v[-1] = 0;  /* ERROR */
  v[100]= 17; /* ERROR */
  return 0;
}
```

Pointer arithmetics

- It is possible to perform arithmetic operation over a pointer using integers
  - Always dependent of the variable size of the pointer.

```
int v[10];
int *p = v;   /* targets p = &v[0] */
p = p + 2;    /* p targets v[2] */
p += 3;       /* p targets v[5] */
p--;          /* p targets v[4] */
*p = 5;       /* v[4] = 5 */
```

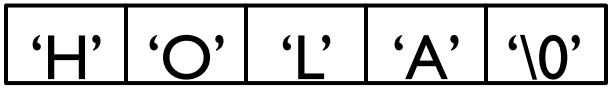
Vector sum

```
#include <stdio.h>

double add_array(double * v, int n) {
  double *end = v + n;
  double s = 0.0;
  for (;v!=end;v++) {
    s += *v;
  }
  return s;
}

int main() {
  double w[10];
  for (int i=0;i<10;i++) { w[i] = 1.0 * i; }
  printf("sum %lf\n", add_array(w,10));
  return 0;
}
```

Strings

- String in C are represent as an `array` of `char`.
- We miss JAVA ☹
  - A character per index.
  - Limits by the char with value 0 (represented as `'\0'`).
  - Lenght or size: number of characters plus one.
  - The string.hlibrary includes the following main functions::
    - strlen(): Para conocer el tamaño de una cadena
    - strcat(): concat
    - strcpy(): copy
    - strcmp(): compare
    - strchr(): find a character inside a string
    - strstr(): find a string in another
    - sprintf(): usefull to include variables as char in a string: sprintf (string, "%d %f", int,float)
    - sscanf(): get a variable from a string

Strings




```
char c1[5] = { 'h', 'o', 'l', 'a', 0 };
char c2[5] = "hola";
char c3[] = "hola";
char c4[100] = "hola";
char *c5 = c2;          /* c5 targets string c2 */
c5 = c1;                /* c5 targets string c1 */
c3 = c2;                /* ERROR */
```

Example

```
#include <stdio.h>
#define SIZE_LINE   80
int length(char string[]);

main() {
   char line[SIZE_LINE];
   int num_car;

   while (gets(linea) != NULL)
   {
      num_car = length(line);
      printf("Length %d chars\n", num_car);
   }
}
int length (char string[]){
   int j = 0;

   while (string[j] != '\0')
      j++;
   return(j);
}
```

Example

```
int length(char string[]){
   int j = 0;

   while (string[j] != '\0')
      j++;
   return(j);
```

```
int length (char string []){
   int j = 0;

   while (*string!= '\0'){
      j++;
      string++;
   return(j);
```

```
int length (char *string){
   int j = 0;

   while (string[j] != '\0')
      j++;
   return(j);
```

```
int length (char *string){
   int j = 0;

   while (*string) {
      string++;
      j++;
}
   return(j);
```

Example: matching

```
int count(char * c, char x) {
  int n=0;
  char * q = c;
  while (q!=NULL) {
    if (*q==x) {
      n++;
    }
    q++;
   }
   return n;
}
```

Differences between string and `arrays`

- The library string.h should be used to strings, not arrays of data

  - Never we can use `strlen()` to get the lenght of an `array`.
  - **String**: limit by a special character (`'\0'`)
  - **Array**: we need use another variable in order to "remember" the lenght, usually an integer.

Void

- We count with generic pointers.
  - `void * p; /* represent anything in memory */`
- Rules:
  - We can copy to a generic pointer everything.
    - `char * q = var;`
    - `p = q;`
  - Is not possible get values directly from a generic pointer
    - `*p = 'x'; /* ERROR */`
  - We can copy fron a generic pointer, but we need declare a casting.
    - `q = (char*)p;`

Dynamic memory

- Need to include stdlib.h
  - Allocate memory

```
void *malloc(size_t size);
```

  - Free the memory

```
void free(void *ptr)
```

  - Example:

```
char *q = (char*) malloc(nbytes);
int *p = (int*) malloc(N*sizeof(int));
```

  - Data:
    - `q[0] = 'a';  p[0]=1;`
  - Free memory in C **is NOT AUTOMATIC**
    - `free(p)`

Dynamic `Arrays` of multiple dimensions

- We allocate dynamic `arrays` of multiple dimensions:
  - Ej: dynamic `array` of 2 dimensions (NxN)

```
int **a, *b, i;
/* we allocate the first level */
a = (int **)malloc(sizeof(int*)*N);
/* we allocate N vectors, one per each index in a */
for (i=0; i<N; i++){
    a[i] = (int *)malloc(sizeof(int)*N);
}
```

Preprocessor directives

- Directives are commands that tell the preprocessor to skip part of a file, include another file, or define a constant or macro.
- The most common directives are:
  - #include
    - Includes files in the current code
  - #define
    - Macros and constants
  - #ifdef/#ifndef .. #else .. #endif
    - Conditional compilation
- Could be set at compilation time (see the example)

Example

```
#ifdef DEBUG
   printf("Variable x = %d\n", x);
#endif
```

At compilation time:

```
gcc –c example.c -DDEBUG
```

Constants

- Two ways:
  - Preprocesor sentence **`#define`**
    - Before linking, the compiler replace the labels with the real value

```
#define      N      1000
```

  - Variables as constants.
    - Datatype checking

```
const int N=1000;
```

- Second ways is better, althought the first one large used currently

Command line arguments

./application arg_1 arg_2 arg_3 ... arg_n

- The prototype of the main function is the following one
  - `int main(int argc, char *argv[])`
- argc represent the number of arguments in the command line.
  - `argv[0]` = name of the program.
  - `argv[1]` = first argument in the program.
  - `argv[i]` = i argument in the program.
- Note: an argument is all except blank spaces!!!

Libraries

- For example: the function
  - `double pow(double x, double y);`
- Its prototype is defined in `<math.h>`
- Its code in the library `libm.a` or `libm.so`
- If you want use this function, we hato to include the reference to the library (its prototype)
  - `#include <math.h>`
- In order to create an executable
  - `cc  example.c -lm`
- Static libraries
  - `ar rcs libexample.a file1.o file2.o`

Some standart libraries in C

- **pthread.h**
  - POSIX Threads.
  - We need link with the libpthread.a library, so we add –lpthread at compilation time
- **stdio.h**
  - Standart I/O.
- **stdlib.h**
  - Dynamic memory functions and constants.
- **string.h**
  - Needed if you want to handle string.
- **time.h**
  - Time library
- **errno.h**
  - System error. Needed by perror()
- **math.h**
  - Mathematical constant and functions
  - We need link with the libm.a library, so we add –lm at compilation time

Compilation process

sum.h

```
double sum_array(double  v[], int
n);
```

sum.c

```
#include <stdio.h>
#include "sum.h>

double sum_array(double  v[], int n)
  {
  int i;
  double s = 0.0;
  for (i=0;i<n;i++) {
    s = s + v[i];
  }
  return s;
}
```

prog.c

```
#include <stdio.h>
#include "sum.h"

int main() {
  double w[10];
  for (int i=0;i<10;i++) {
        w[i] = 1.0 * i;
  }
  printf("sum %lf\n", sum_array(w,10));
  return 0;
}
```

Compilation process

- cc –c suma.c
- cc –c prog.c
- cc –g –o exec suma.o prog.o

Compilatin process

- Used by `gcc`:
  - **Step 1**. Creation of object files (.o)
    - gcc -c <file.c> <flags>

      Example: gcc -c hello.c –DDEBUG –I/path/interfaces

  - **Step 2**. Creation the executable file
    - gcc -o <application name> <object files> <library>

      Example:

```
gcc -o file file1.o ../file2.o -lpthread –lm
gcc -o file file1.o ../file2.o -lpthread –L/path/library
```

Compilation process

- Using `make`:
  - `./configure <opciones>`
  - `make`
- `Makefile`

Makefile

```
CFLAGS = -g -Wall
CC = gcc
LIBS =  -lm
INCLUDES =
OBJS = a.o b.o c.o
SRCS = a.c b.c  c.c prog1.c prog2.c
HDRS =  abc.h

all: prog1 prog2

# The variable $@ has the value of the target. In this case $@ = psort
prog1: prog1.o ${OBJS}
          ${CC} ${CFLAGS} ${INCLUDES} -o $@ prog1.o ${OBJS} ${LIBS}

prog2: prog2.o ${OBJS}
          ${CC} ${CFLAGS} -o $@ prog2.o ${OBJS} ${LIBS}

.c.o:
          ${CC} ${CFLAGS} ${INCLUDES} -c $<

depend:
          makedepend ${SRCS}

clean:
          rm *.o core *~

# DO NOT DELETE
```

Bibliography

- **Problemas resueltos de programación en C**
  F. García, J. Carretero, A. Calderón, J. Fernández, J. M. Pérez.
  Thomson, 2003.
  ISBN: 84-9732-102-2.

- **El lenguaje de programación C. Diseño e implementación de programas**
  J. Carretero, F. García, J. Fernández, A. Calderón
  Prentice Hall, 2001

Links of interest

- Are you Ready For C99?
  - http://www.kuro5hin.org/?op=displaystory;sid=2001/2/23/194544/139
- The New ISO Standard for C (C9X)
  - http://home.datacomm.ch/t_wolf/tw/c/c9x_changes.html