Overview da Linguagem C para Programadores

Sistemas Operativos, 1º Semestre 2004-05

History of C

- UNIX developed c. 1969 -- DEC PDP-7 Assembly Language
- BCPL -- a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone – Martin Richards.
- A new language “B” (1970) – Ken Thompson.
- A totally new language “C” a successor to “B”. Created on 1971.
- By 1973 UNIX OS almost totally written in “C”.

Os criadores da linguagem C...

Dennis M. Ritchie

Brian Kernighan's

Quem escreveu a linguagem Java?

James Gosling

Porquê C, depois de Java?...

- C is both a high-level and a low-level language
- Better control of low-level mechanisms
- Performance better than Java (Unix, NT !)
- Java hides many details needed for writing OS code

  But,….
- Memory management responsibility
- Explicit initialization and error detection
- More room for mistakes

O que faz este programa em C ?

```
#include <stdio.h>
struct list{int data; struct list *next};
struct list *start, *end;
void add(struct list *head, struct list *list, int data);
int delete(struct list *head, struct list *tail);

void main(void){
 start=end=NULL;
 add(start, end, 2);    add(start, end, 3);
 printf("First element: %d", delete(start, end));
}

void add(struct list *head, struct list *tail, int data){
 if(tail==NULL){
  head=tail=malloc(sizeof(struct list));
  head->data=data; head->next=NULL;
 }
 else{
  tail->next= malloc(sizeof(struct list));
  tail=tail->next; tail->data=data; tail->next=NULL;
 }
}
int delete (struct list *head, struct list *tail){
 struct list *temp;
 int returnval = -1;
 if (head != null) returnval = head->data;
 if(head==tail){
  free(head); head=tail=NULL;
 }
 else{
  temp=head->next; free(head); head=temp;
 }
 return returnval;
}
```

Objectivos deste Tutorial

- To introduce some basic concepts about C language.
- To warn you about common mistakes made by beginners.
- You will be able to understand that earlier complicated program completely !
  - And write more complicated code

Primeiro Exemplo

```
#include <stdio.h>

void main(void)
{
    printf("Hello World. \n \t and you ! \n ");
            /* print out a message */
    return;
}
```

```
$Hello World.
        and you !
$
```

Explicação do Exemplo…

- `#include <stdio.h>` = include header file stdio.h
  - No semicolon at end
  - Small letters only – C is case-sensitive
- `void main(void){ … }` is the only code executed
- `printf(" /* message you want printed */ ");`
- `\n` = newline `\t` = tab
- \ in front of other special characters within `printf`.
  - `printf("Have you heard of \"The Rock\" ? \n");`

Como Compilar:

> cc myprog.c (executável em a.out)
> cc -o myprog myprog.c
> gcc -o myprog myprog.c

Lint -- A C program verifier
> lint myprog.c.

Simple Data Types

- data-type # bytes(typical) values short-hand

- **int 4 -2,147,483,648 to 2,147,483,647 %d**
- **char 1 -128 to 127 %c**
- **float 4 3.4E+/-38 (7 digits) %f**
- **double 8 1.7E+/-308 (15 digits long) %lf**
- **long 4 -2,147,483,648 to 2,147,483,647 %l**
- **short 2 -32,768 to 32,767 %d**

- Lookup:
  - signed / unsigned

C type	Size (bytes)	Lower bound	Upper bound
char	1	—	—
unsigned char	1	0	255
short int	2	−32768	+32767
unsigned short int	2	0	65536
(long) int	4	-2^{31}	$+2^{31}-1$
float	4	$-3.2\times10^{+38}$	$+3.2\times10^{+38}$
double	8	$-1.7\times10^{+308}$	$+1.7\times10^{+308}$

- ex:

```
int num=200;
printf("SO tem %d alunos…\n", num);
```

Outro Exemplo

```
#include <stdio.h>

void main(void)
{
      int nstudents = 0; /* Initialization, required */

      printf("Quantos alunos tem Sist.Operativos ?:");
      scanf ("%d", &nstudents);  /* Read input */
      printf("SO tem %d alunos.\n", nstudents);

      return ;
}
```

$Quantos alunos tem Sistemas Operativos ?:
200 (enter)
SO tem 200 alunos.
$

Type Conversion

```
#include <stdio.h>
void main(void)
{
      int i,j = 12;      /* i not initialized, only j */
      float f1,f2 = 1.2;

      i = (int) f2;      /* explicit: i <- 1, 0.2 lost */
      f1 = i;            /* implicit: f1 <- 1.0 */

      f1 = f2 + (float) j;/* explicit: f1 <- 1.2 + 12.0 */
      f1 = f2 + j;       /* implicit: f1 <- 1.2 + 12.0 */
}
```

- Explicit conversion rules for arithmetic operation `x=y+z;`
  - convert `y or z` as
    - `double <- float <- int <- char, short`
  - then type cast it to `x` 's type
- Moral: stick with explicit conversions - no confusion !

Like Java, like C

- Operators same as Java:
  - Arithmetic
    - `int i = i+1; i++; i--; i *= 2;`
    - `+, -, *, /, %,`
  - Relational and Logical
    - `<, >, <=, >=, ==, !=`
    - `&&, ||, &, |, !`
- Syntax same as in Java:
  - `if ( ) { } else { }`
  - `while ( ) { }`
  - `do { } while ( );`
  - `for(i=1; i <= 100; i++) { }`
  - `switch ( ) {case 1: … }`
  - `continue; break;`

Outro Exemplo

```
#include <stdio.h>
#define Objectivo 15    /* C Preprocessor -
                  - substitution on appearance */
                        /* like Java 'final' */
void main(void)
{
      float nota=1;
      /* if-then-else as in Java */
      if (nota <= Objectivo){ /*replaced by 15*/
         printf("Este aluno deve ser do Benfica!\n");
      }
      else printf("Bom Aluno… Assim é que é !\n");

      return;
}
```

One-Dimensional Arrays

```
#include <stdio.h>

void main(void)
{
    int number[12]; /* 12 cells, one cell per student */
    int index, sum = 0;
           /* Always initialize array before use */
    for (index = 0; index < 12; index++) {
      number[index] = index;
    }

    for (index = 0; index < 12; index = index + 1) {
      sum += number[index];  /* sum array elements */
    }

    return;
}
```

Arrays in C

```
int number[12];
int x = 0;

number[12] = 100;  /* what will happen here? */

                   /* and what if it was in Java? */
```

More Arrays

- Strings

```
char name[6];
name = {'U','N','I','X','\0'};
                    /* '\0'= end of string */
printf("%s", name); /* print until '\0' */
```

  – Functions to operate on strings
    - `strcpy, strncpy, strcmp, strncmp, strcat, strncat, strstr,strchr`
    - `#include <strings.h>` at program start

- Multi-Dimensional arrays

```
int points[3][4];
points [1][3] = 12;  /* NOT points[3,4] */
printf("%d", points[1][3]);
```

Like Java, somewhat like C

- Type conversions
  – but you can typecast from any type to any type
    - `c = (char) some_int;`
  – So be careful !
- Arrays
  – *Always* initialize before use
  – `int number[12];`
    `printf("%d", number[20]);`
    - produces undefined output, may terminate, may not even be detected.
- Strings are terminated by `'\0'` character

```
char name[6] = {'U','N','I','X','\0'};
                    /* '\0'= end of string */
printf("%s", name); /* print until '\0' */
```

Memory Layout

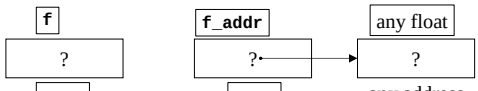
```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```

5	10	12.5	9. 8	c	d
4300	4304	4308	4312	4316	4317

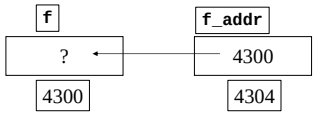
Pointers made easy - 1

- *Pointer* = variable containing address of another variable

```
float f;        /* data variable */
float *f_addr;  /* pointer variable */
```

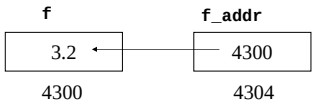



```
f_addr = &f; /* & = address operator */
```

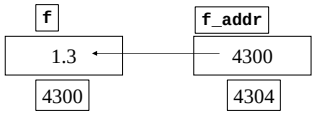


Pointers made easy - 2

```
*f_addr = 3.2;    /* indirection operator */
```




```
float g=*f_addr; /* indirection:g is now 3.2 */

f = 1.3;
```



Pointers made easy - 3

IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it.

So ...

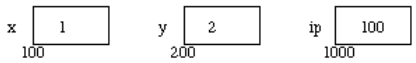
```
int *ip;
*ip = 100; // will generate an error (program crash!!).
```

The correct use is:

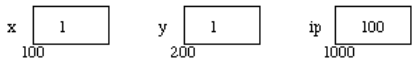
```
int *ip;
int x;
ip = &x;
*ip = 100;
```

int x = 1, y =2;
int *ip;

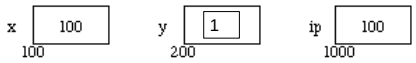
ip = &x;



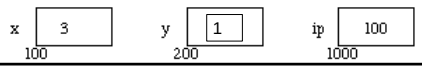
y = *ip;



x = ip;



*ip = 3



Pointer Example

```
#include <stdio.h>

void main(void) {
      int j;
      int *ptr;

      ptr=&j;   /* initialize ptr before using it */

      *ptr=4;     /* j <- 4 */

      j= j + (*ptr);     /* j <- ??? */
}
```

Dynamic Memory Allocation

- Explicit allocation and de-allocation

```
#include <stdio.h>

void main(void) {
      int *ptr, *ptr_aux;
      int x;
      /* allocate space to hold an int */
      ptr = malloc(sizeof(int)*10);
      ptr_aux=ptr;
      *ptr=4;
      ptr++;
      *ptr=5;
      ptr += 2;
      *ptr=6;
      x= *(ptr_aux+3);
      free(ptr_aux);
      /* free up the allocated space */
}
```

Elementary file handling

```
#include <stdio.h>

void main(void) {
      /* file handles */
    FILE *input_file=NULL;
    int ret;

      /* open files for writing*/
    input_file = fopen("cwork.dat", "w");

      /* write some data into the file */
    ret=fprintf(input_file, "Hello there");
     // ver o que devolve esta funcao na variavel ret
      /* don't forget to close file handles */
    fclose(input_file);
    return;
}
```

Error Handling

- unlike Java, no explicit exceptions
- need to manually check for errors
  - Whenever using a function you've not written
  - Anywhere else errors might occur

Functions - why and how ?

- If a program is too long.
- Modularization – easier to:
  - code
  - debug
- Code reuse.
- Passing arguments to functions:
  - By value
  - By reference
- Returning values from functions:
  - By value
  - By reference

Functions – Basic Example

```
#include <stdio.h>
int sum(int a, int b);
            /* function prototype at start of file */

void main(void){
   int a=4, b=5;
   int total = sum(a,b); /* call to the function */

   printf("The sum of 4 and 5 is %d, %d", total,a);
}
//-----------------------------------------------------
int sum(int a, int b){   /* the function itself
                       - arguments passed by value*/
      a = 10;
      return (a+b);      /* return by value */
}
```

Arguments by Reference

```
#include <stdio.h>
int sum(int *pa, int *pb);
            /* function prototype at start of file */

void main(void){
   int a=4, b=5;
   int *ptr = &b;
   int total = sum(&a,ptr); /* call to the function */

   printf("The sum of 4 and 5 is %d", total);
}
//-----------------------------------------------------
int sum(int *pa, int *pb){   /* the function itself
                        - arguments passed by reference */
      *pa=10;
      return (*pa+*pb);      /* return by value */
}
```

Why pointer arguments?!

```
#include <stdio.h>

void swap(int, int);

main() {
  int num1 = 5, num2 = 10;
  swap(num1, num2);
  printf("num1 = %d and num2 = %d\n", num1, num2);
}
//-----------------------------------------------
void swap(int n1, int n2) { /* passed by value */
  int temp;

  temp = n1;
  n1 = n2;
  n2 = temp;
}
```

This is why…

```
#include <stdio.h>

void swap(int *, int *);

main() {
  int num1 = 5, num2 = 10;
  swap(&num1, &num2);
  printf("num1 = %d and num2 = %d\n", num1, num2);
}
//-----------------------------------------------
void swap(int *n1, int *n2) { /* passed and returned by
                                    reference */
  int temp;

  temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}
```

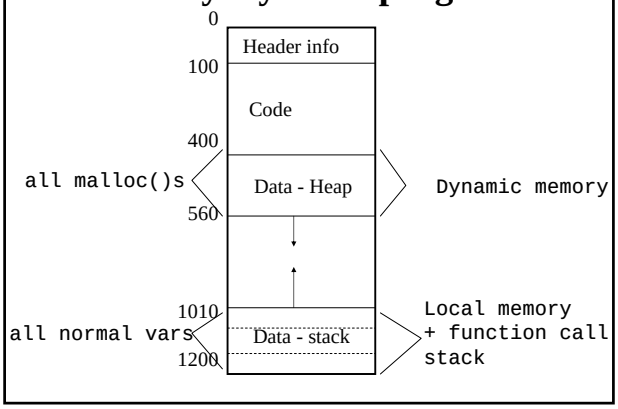
What's wrong with this ?

```
#include <stdio.h>

void dosomething(int *ptr);

main() {
      int *p=NULL;
      dosomething(p);
      printf("%d", *p);     /* will this work ? */
}
//-----------------------------------------------------
void dosomething(int *ptr){ /* passed and returned by
                                  reference */
  int temp=32+12;
  ptr = &temp;
}
/* compiles correctly, but gives run-time error */
```

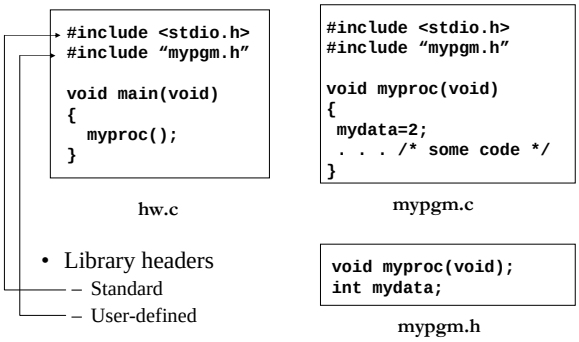
Passing and returning Arrays

```
#include <stdio.h>

void init_array(int array[], int size) ;

void main(void) {
  int list[5];

  init_array(list, 5);
  for (i = 0; i < 5; i++)
    printf("next:%d", list[i]);
}
//-----------------------------------------------
void init_array(int *array, int size) { /* why size ? */
          /* arrays ALWAYS passed by reference */
 int i;
 for (i = 0; i < size; i++)
    array[i] = 0;
    *(array+i) =0;
}
```

Memory layout of programs



Program with multiple files



Externs

```
#include <stdio.h>

extern char user2line [20];   /* global variable defined
                                 in another file */
char user1line[30];           /* global for this file */
void dummy(void);

void main(void) {
 char user1line[20];       /* different from earlier
                              user1line[30] */
 . . .                     /* restricted to this func */
}
//-----------------------------------------------------
void dummy(){
 extern char user1line[];  /* the global user1line[30] */
 . . .
}
```

Structures

- Equivalent of Java's classes with only data (no methods)…

```
#include <stdio.h>

struct birthday{
    int month;
    int day;
    int year;
  };

main() {
  struct birthday mybday;     /* - no 'new' needed ! */
                              /* then, it's just like Java ! */
  mybday.day=1; mybday.month=1; mybday.year=1977;
  printf("I was born on %d/%d/%d", mybday.day,
                               mybday.month, mybday.year);
}
```

More on Structures

```
struct person{
    char name[41];
    int age;
    float height;
    struct {        /* embedded structure */
      int month;
      int day;
      int year;
    } birth;
 };

struct person me;

me.birth.year=1977;……

struct person class[60];
      /* array of info about everyone in class */

class[0].name="Gun"; class[0].birth.year=1971;……
```

Passing/Returning a structure

```
/* pass struct by value */
void display_year_1(struct birthday mybday) {
  printf("I was born in %d\n", mybday.year);
}                 /* - inefficient: why ? */
. . . .
/* pass struct by reference */
void display_year_2(struct birthday *pmybday) {
  printf("I was born in %d\n", pmybday->year);
      /* warning ! '->', not '.', after a struct pointer*/
}
. . . .
/* return struct by value */
struct birthday get_bday(void){
  struct birthday newbday;
  newbday.year=1971;  /* '.' after a struct */
  return newbday;
}                 /* - also inefficient: why ? */
```

Synonym for a data type

```
typedef int Employees;

Employees my_company;  /* same as int my_company; */

typedef struct person Person;

Person me;             /* same as struct person me; */

typedef struct person *Personptr;

Personptr ptrtome;     /* same as struct person *ptrtome;*/
```

- Easier to remember
- Clean code

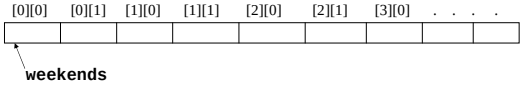
More pointers

```
int month[12]; /* month is a pointer to base address 430*/
int *ptr;
month[3] = 7;  /* month address + 3 * int elements
               => int at address (430+3*4) is now 7 */

ptr = month + 2; /* ptr points to month[2],
               => ptr is now (430+2 * int elements)= 438 */
ptr[5] = 12;
                /* ptr address + 5 int elements
               => int at address (434+5*4) is now 12.
                   Thus, month[7] is now 12 */

ptr++;          /* ptr <- 438 + 1 * size of int = 442 */
(ptr + 4)[2] = 12;   /* accessing ptr[6] i.e., array[9] */
```

- Now , **`month[6], *(month+6), (month+4)[2], ptr[3], *(ptr+3)`** are all the same integer variable.

2-D Arrays

- 2-dimensional array
  `int weekends[52][2];`



- **`weekends[2][1]`** is same as **`*(weekends+2*2+1)`**
  - NOT **`*weekends+2*2+1`** :this is an int !

argc and argv parameters

```
#include <stdio.h>
        /* program called with cmd line parameters */
void main(int argc, char *argv[]) {
  int ctr;
  printf("Nº Argumentos= %d \n",argc);
  for (ctr = 0; ctr < argc; ctr = ctr + 1) {
    printf("Argument #%d is -> | %s|\n", ctr, argv[ctr]);
  }     /* ex., argv[0] == the name of the program */
}
```

```
>teste p1 p2 p3
Nº Argumentos= 4
Argument #0 is -> | teste
Argument #1 is -> | p1
Argument #1 is -> | p2
Argument #1 is -> | p3
```

Strings

```
#include <stdio.h>

main() {
  char msg[10]; /* array of 10 chars */
  char *p;      /* pointer to a char */
  char msg2[]="Hello"; /* msg2 = 'H''e''l''l''o''\0' */

  strcpy(msg,"Bonjour");

  msg = "Bonjour"; /* ERROR. msg has a const address.*/
  p   = "Bonjour"; /* address of "Bonjour" goes into p */

  msg = p; /* ERROR. Message has a constant address. */
           /* cannot change it. */
  p = msg; /* OK */

  p[0] = 'S', p[1] = 'O',p[2]='\0';
           /* *p and msg are now "SO" */
}
```

O que faz este programa em C?

```
#include <stdio.h>
struct list{int data; struct list *next};
struct list *start, *end;
void add(struct list *head, struct list *list, int data};
int delete(struct list *head, struct list *tail);

void main(void){
 start=end=NULL;
 add(start, end, 2);
 add(start, end, 3);
 printf("First element: %d", delete(start, end));
}

void add(struct list *head, struct list *tail, int data){
 if(tail==NULL){
  head=tail=malloc(sizeof(struct list));
  head->data=data; head->next=NULL;
 }
 else{
  tail->next= malloc(sizeof(struct list));
  tail=tail->next; tail->data=data; tail->next=NULL;
 }
}
int delete (struct list *head, struct list *tail){
 struct list *temp;
 int returnval = -1;
 if (head != NULL) returnval=head->data;
 if(head==tail){
  free(head); head=tail=NULL;
 }
 else{
  temp=head->next; free(head); head=temp;
 }
 return returnval;
}
```

Já consegue entender este exemplo?....

Final Tips

- **Always initialize anything before using it. (especially pointers!)**
- **Don't use pointers after freeing them.**
- **When you `malloc()` don't forget to `free()`.**
- **Don't return a function's local variables by reference.**
- **No exceptions – so check for errors everywhere.**
- **An array is also a pointer, but its value is immutable.**