Revisões sobre C

Conteúdo

1. Estrutura de um programa em C
2. Interface com a linha de comando
3. Controlo de fluxo
4. Operadores
5. Variáveis e tipo de dados
6. Directivas de preprocessamento
7. Apontadores
8. Funções: passagem por valor versus passagem por referência
9. Arrays e Strings
10. Estruturas e tipos definidos pelo programador
11. Alocação dinâmica de memória
12. Ficheiros
13. Makefiles

Hello World!!

```
#include <stdio.h>

int main() {
      printf("Hello, world!\n");
      return 0;
}
```

- Preprocessador: cpp
- Compilador: gcc

```
$ man gcc

$ gcc -Wall prog.c –o myprogram

$ ./myprogram
```

Interface com a linha de comando

```
#include <stdio.h>

int main(int argc, char *argv[]) {
        if (argc > 0)
                printf("Hello %s!\n", argv[1]);
        else
                printf("Hello World!\n");
        return 0;
}
```

$ *…comando para compilar….*

$ ./hello Pedro

Controlo de fluxo

- `if (x) { } else { }`
- `for (pre(); loop_invariant(); post()) { }`
- `while (x) { }`
- `do { } while (x);`
- `switch (x) { case 0: break; default: act(); }`

Operadores

- Aritméticos: `+, -, *, /, %`
- Relacionais: `<, >, <=, >=, ==, !=`
- Lógicos: `&&, ||, !, ?:`
- *Bitwise*: `&, |, ^, ~, <<, >>`

Tipos de dados

- Inteiros:
  - `char`: caracteres, tipicamente um byte
  - `int`, `short`, `long`: inteiros de diferentes tamanhos
  - Prefixos opcionais: `signed` ou `unsigned`
- Números de virgula flutuante:
  - `float`: número de virgula flutuante
  - `double`: número de virgula flutuante de precisão `dupla`
- Não há "*booleanos*":
  - 0 → false
  - outro valor → true

Exemplos

```
char a = 'A';
char b = 65;
char c = 0x41;

int i = -2343234;
unsigned int ui = 4294967295;
unsigned int uj = ((long) 1 << 32) - 1;
unsigned int uk = -1;

float pi = 3.14;
double long_pi = 0.31415e+1;
```

TPC:

- fazer `printf()` das variáveis acima com formatadores distintos (e.g., %d, %u; %ld, %x)
- Utilizar o `sizeof()` para para determinar o # de bytes dos diferentes tipos de dados

Enums

```
enum months {
        JANUARY, FEBRUARY, MARCH
};
```

Sequência de inteiros a começar do zero

```
enum days {
        SUNDAY,
        TUESDAY = 2,
        WEDNESDAY
};
```

A atribuição explícita de um valor faz com que a sequência continue a partir desse valor

Directivas de Pré-processamento

- Inclusão de outros ficheiros

```
#include <header.h>
```

- Definição de constantes

```
#define HOURS_PER_DAY 24
```

- Criação de macros:

```
#define CELSIUS(F) ((F - 32) * 5/9)
```

Apontadores

- Apontadores são variáveis que armazenam um endereço de memória

```
int foo;
int* ptr_foo;
int** ptr2_foo;
```

- Operações com apontadores:
  - `&` obter o endereço de uma variável
  - `*` aceder ao endereço de memória apontado pela variável apontador

Exemplos

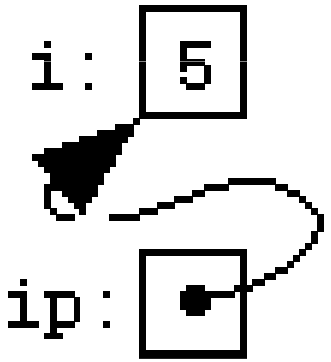
```
int x;

int* ptr; /* ptr aponta para uma posição de memória
             indefinida */

ptr = &x; /* ptr passou a apontar para o
             inteiro x */


*ptr = 3; /* ao inteiro apontado por x é atribuido o
             valor 3 */
```

Exemplos

```
int* ip;
int i = 5;
ip = &i;
printf("%d\n", *ip);
```

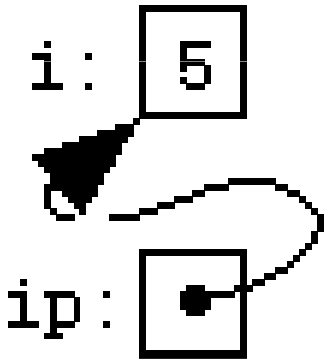
Exemplos

```
int* ip;
int i = 5;
ip = &i;
printf("%d\n", *ip);
```



Exemplos

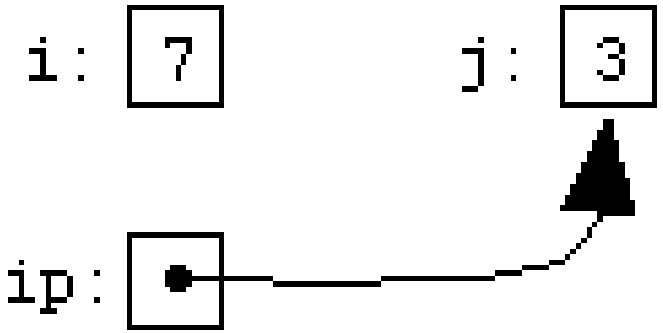
```
*ip = 7;
```




```
int j = 3;
ip = &j;
```

Exemplos

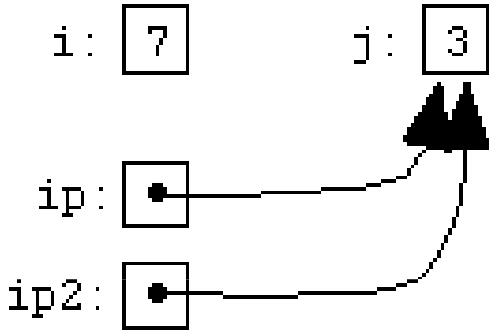
```
*ip = 7;
```



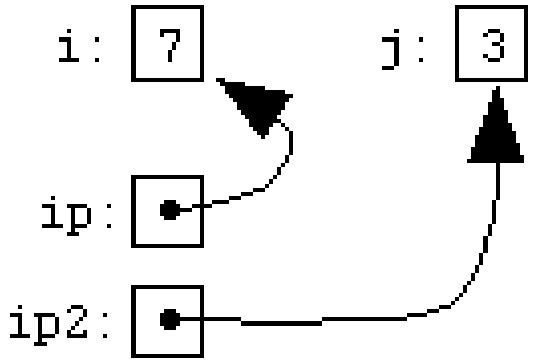

```
int j = 3;
ip = &j;
```

Exemplos

```
int* ip2;
ip2 = ip;
```




```
ip = &i;
```



Funções

- Passagem de parâmetros por valor

```
void swap_value(int n1, int n2) {
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

Funções

- Passagem de parâmetros por valor

```
void swap_value(int n1, int n2) {
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}

// swap_value() troca n1 por n2 apenas no âmbito local!!
```

Funções

- Passagem de parâmetros por referência

```
void swap_reference(int* p1, int* p2) {
    int temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

Funções

- Passagem de parâmetros por referência

```
void swap_reference(int* p1, int* p2) {
    int temp;
    temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}
```

```
/* swap_reference() troca os valores das posições de memória apontadas por p1 e p2 */
```

Arrays e Strings

- Arrays

```
#define DIM 10
int A[DIM];
int i;
for (i = 0; i < 10; ++i) {
     A[i] = i * i;
}
```

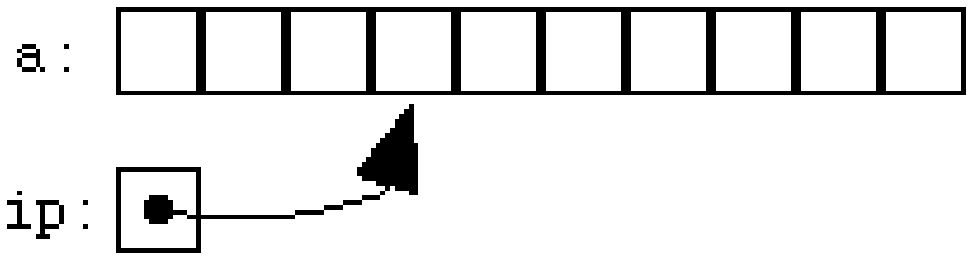
- Strings

```
char name[] = "CC222";
char name2[] = {'C', 'C', '1', '0', '6', 0}; // '\0' = 0
char *name3 = "CC222";

name[4] = '1';
name2[4] = '1';
name3[4] = '1'; // incorrecto!!!
```

- Funções para operar sobre strings em `string.h`
  - `strlen(), strcmp(), strstr(), strncmp(), strtok()…`

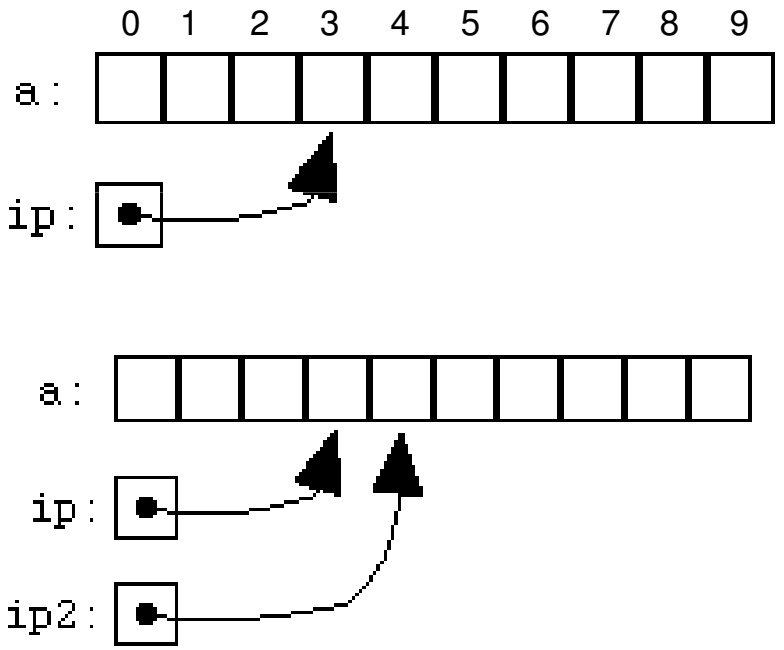
Arrays

```
int *ip;
int a[10];
ip = &a[3];
```



Arrays

```
int *ip;
int a[10];
ip = &a[3];

int *ip2;
ip2 = ip + 1;
```



Arrays vs Strings

- Exemplo: comparação de strings - my_strcmp()

```
int my_strcmp(char* str1, char str2[]){
        char* p1 = &str1[0]; // ou: char* p1 = str1;
        char* p2 = str2; // ou: char* p2 = &str2[0];

        while(1){
                if(*p1 != *p2) return *p1 - *p2;
                if(*p1 == '\0' || *p2 == '\0')
                        return 0;
                p1++;
                p2++;
        }
}
```

Estruturas

```
struct list_elem{
        int data;
        struct list_elem* next;
};


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

        struct list_elem le = {10, NULL};

        le.data = 20;

        return 0;
}
```

Typedefs

```
typedef struct list_elem{
        int data;
        struct list_elem* next;
} list_elem_t;
```

- Dar um novo nome a um tipo
- Sintaxe: `typedef` *`type alias`*
- Utilização igual à de qualquer tipo pré-definido:

```
list_elem_t x;
```

Exemplo

```
typedef struct directory_entry {
   char type;
   char name[MAX_NAME_LENGTH];
   unsigned char day;
   unsigned char month;
   unsigned char year;
   int size;
   int first_block;
} dir_entry;
```

```
dir_entry x;
dir_entry* px = &x;

x.day = 28;
px->month = 2; // equivalente a (*px).month = 2;
```

Alocação dinâmica de memória

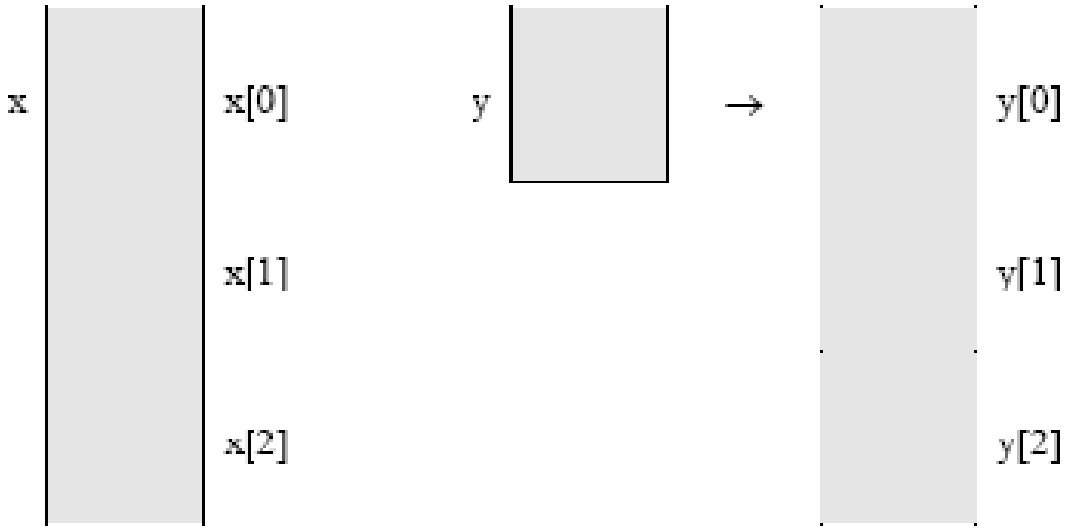
- Alocação com `malloc()`
- Libertação de memória com `free()`

```
void* malloc(int)
void free(void*)
```

- Gestão de memória da heap é reponsabilidade do programador (não há *garbage collection* como no java)
- Não libertar memória (`free()`) origina memory leaks
- Chamar `free()` para a mesma posição de memória mais que uma vez pode *crachar* o programa

Alocação dinâmica de memória

```
int x[3];
int *y = malloc(3*sizeof(int));
```



Qual é a origem do problema?

```c
// segmentation fault!

#include <stdio.h>
#include <string.h>

int main() {
        char *x;

        strcpy(x,"Hello world!");
        printf("Valor de x = %s\n", x);

        return 0;
}
```

Correcção do problema

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main() {
        char *x;

        x = (char*) malloc(20 * sizeof(char));

        strcpy(x,"Hello world!");
        printf("Valor de x = %s\n", x);

        return 0;
}
```

Ficheiros

- Abertura/fecho

```
FILE* fopen(char* nome, char* modo);
int fclose(FILE* fp);
```

- Leitura/escrita formatada para ficheiros

```
int fprintf(FILE* , char* format, …);
int fscanf(FILE* , char* format, …);
```

- O que é `stdin`, `stdout` e `stderr`?

- Leitura/escrita binária

```
size_t fread(void* ptr, size_t size, size_t nitems, FILE* stream);
size_t fwrite(void* ptr, size_t size, size_t nitems, FILE* stream);
```

- (Re-)Posicionamento da leitura/escrita num ficheiro

```
int fseek(FILE* ptr, long offset, int whence);
```

Leitura de ficheiros

```
int main (int argc, char* argv[]){
          int l, c, i, j;
          FILE* f;
          int* matrix;

          f= fopen(argv[1], "r");

          fscanf(f,"%d %d", &l, &c);

          matrix = malloc(l*c*sizeof(int));

          for(i=0; i< l; i++){
                  for(j=0; j< c; j++){
                          fscanf(f,"%d ", matrix + i*c +j );
                  }
                  printf("\n");
          }

          fclose(f);
          return 0;
 }
```

Escrita para ficheiros

```
int main (int argc, char* argv[])
  {
          int l, c, i, val;
          FILE* f:
          int* mp;

          l = atoi(argv[1]);
          c = atoi(argv[2]);

          mp = malloc(l*c*sizeof(int));

          f= fopen(argv[3], "w");
          fprintf(f, "%d %d ", l, c);

          for(i=0; i< l*c; i++){
                  scanf("%d", &val);
                  matrix[i]= val;
                  fprintf(f, "%d ", val);
          }
          fprintf(f, "\n");
          fclose(f);

          return 0;
  }
```

Escrita binária para ficheiros

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[] ) {
        int l, c, i, val;
        FILE* f;
        int* matrix;

        l = atoi(argv[1]);
        c = atoi(argv[2]);

        matrix = malloc(l*c*sizeof(int));

        for(i=0; i< l*c; i++){
                scanf("%d", &val);
                matrix[i]= val;
        }

        f= fopen(argv[3], "w");

        if (fwrite(&l, sizeof(int), 1, f) != 1) printf("ERROR!\n");
        if (fwrite(&c, sizeof(int), 1, f) !=1) printf("ERROR!\n");
        if (fwrite(matrix, sizeof(int), l*c, f) !=l*c) printf("ERROR!\n");

        fclose(f);

  return 0;
}
```

Compilação

- Flags

```
prompt> gcc -o hw hw.c  # -o: to specify the executable name
prompt> gcc -Wall hw.c  # -Wall: gives much better warnings
prompt> gcc -g hw.c     # -g: to enable debugging with gdb
prompt> gcc -O hw.c     # -O: to turn on optimization
```

- Compilação separada

```
# we are using -Wall for warnings, -O for optimization
prompt> gcc -Wall -O -c hw.c
prompt> gcc -Wall -O -c helper.c
prompt> gcc -o hw hw.o helper.o -lm
```

Makefiles

- Makefile

```
hw: hw.o helper.o
    gcc -o hw hw.o helper.o -lm

hw.o: hw.c
    gcc -O -Wall -c hw.c

helper.o: helper.c
    gcc -O -Wall -c helper.c

clean:
    rm -f hw.o helper.o hw
```

```
prompt> make
```

- Forma genérica das regras

```
target: prerequisite1 prerequisite2 ...
    command1
    command2
    ...
```

Makefiles

```
# specify all source files here
SRCS = hw.c helper.c
# specify target here (name of executable)
TARG = hw
# specify compiler, compile flags, and needed libs
CC   = gcc
OPTS = -Wall -O
LIBS = -lm

# this translates .c files in src list to .o's
OBJS = $(SRCS:.c=.o)

# all is not really needed, but is used to generate the target
all: $(TARG)

# this generates the target executable
$(TARG): $(OBJS)
    $(CC) -o $(TARG) $(OBJS) $(LIBS)

# this is a generic rule for .o files
%.o: %.c
    $(CC) $(OPTS) -c $< -o $@

# and finally, a clean line
clean:
    rm -f $(OBJS) $(TARG)
```