# Projeto e Análise de Algoritmos
# Linguagem C - Parte 3

Antonio Luiz Basile

Faculdade de Computação e Informática
Universidade Presbiteriana Mackenzie

February 19, 2018

# Estruturas

Todos os exemplos a seguir retirados do livro: Kernighan, Brian W.. C Programming Language - Pearson Education.

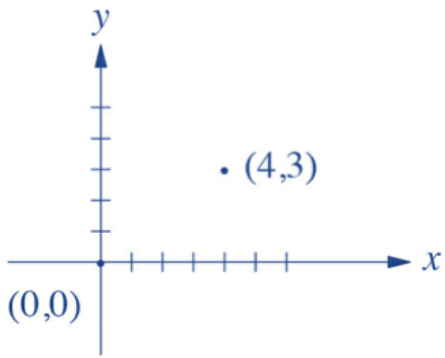


Figure: struct point (K&R)

```
struct point {
    int x;
    int y;
};
struct point pt = {4,3};
```

# Estruturas

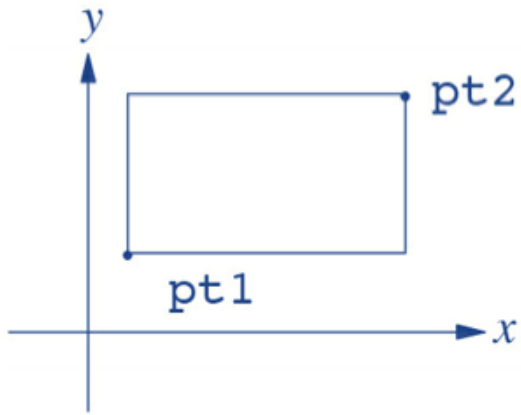


Figure: struct rect (K&R)

```
struct rect {
    struct point pt1;
    struct point pt2;
};

struct rect screen;
```

# Membro de Estrutura

Um membro de uma estrutura particular é referenciado numa expressão da seguinte forma:

*nomeEstrutura.membro*

Exemplo:

```
printf("%d,%d", pt.x, pt.y);

printf("%d", screen.pt1.x);
```

# Estruturas e Funções

Estruturas não podem ser comparadas, mas podem ser passadas para ou retornadas de funções.

```
struct point makepoint(int x, int y){
    struct point temp;
    temp.x = x;
    temp.y = y;
    return temp;
}
int main(){
    struct rect screen;
    struct point middle;
    screen.pt1 = makepoint(0, 0);
    screen.pt2 = makepoint(XMAX, YMAX);
    middle = makepoint((screen.pt1.x + screen.pt2.x)/2,
                       (screen.pt1.y + screen.pt2.y)/2);
...}
```

# Estruturas e Ponteiros

Se uma grande estrutura é passada para uma função, geralmente é mais eficiente passar um ponteiro que copiar toda a estrutura.

```
struct point origin, *pp;

pp = &origin;
printf("origin is (%d,%d)\n", (*pp).x, (*pp).y);

/*same as: */
printf("origin is (%d,%d)\n", pp->x, pp->y);
```

# Estruturas e Ponteiros

Vejamos agora 4 modos de expressar o mesmo.

```
struct rect r, *rp = &r;

/*as 4 expressões a seguir são equivalentes: */
r.pt1.x
rp->pt1.x
(r.pt1).x
(rp->pt1).x
```

# Vetores de Estruturas

Se formos criar um programa que conta quantas ocorrências de cada palavra-chave do C, precisaremos de dois vetores em paralelo:

```
char *keyword[NKEYS];
int keycount[NKEYS];
```

Um modo melhor é criarmos um vetor composto pela estrutura a seguir:

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

# Vetores de Estruturas

Observe que podemos declarar tudo de uma vez:

```
struct key {
    char *word;
    int count;
} keytab[NKEYS];
```

Ou declararmos em separado como abaixo:

```
struct key {
    char *word;
    int count;
};

struct key keytab[NKEYS];
```

# Vetores de Estruturas

Podemos inclusive inicializar o vetor:

```
struct key {
    char *word;
    int count;
} keytab[] = {
    "auto", 0,
    "break", 0,
    "case", 0,
    "char", 0,
    /* ... */
    "while", 0
};
```

# Estruturas Auto-Referenciais

É muito comum utilizarmos estruturas de dados auto-referenciais. Por exemplo, uma lista ligada é um conjunto de itens onde cada item é parte de um nó que também contém um link para um nó. Em C temos:

```
typedef struct node *link;

struct node {
    int item;
    link next;
};
```

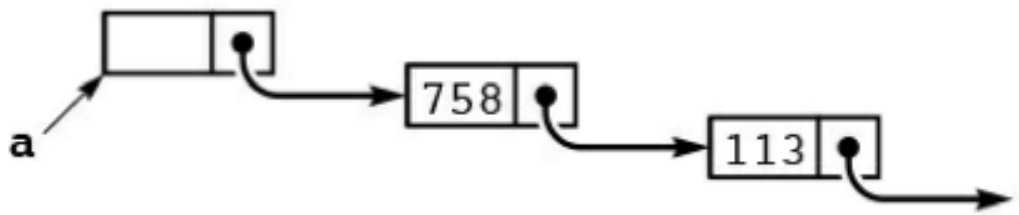


Figure: from: Sedgewick, R. Algorithms in C

# Declarações Difíceis

Abaixo seguem alguns exemplos de como declarações envolvendo ponteiros podem ser intrincadas. Como exercício, tente decifrá-las.

```
char **argv
int (*dias)[10]
int *dias[10]
void *comp()
void (*comp)()
char (*(*x())[])()
char (*(*x[3])())[5]
```

# Declarações Difíceis: Respostas

```
char **argv
/* argv é ponteiro para ponteiro para char. */
```

```
int (*dias)[10]
/* dias é ponteiro para vetor[10] de int. */
```

```
int *dias[10]
/* dias é vetor[10] de ponteiro para int. */
```

# Declarações Difíceis: Respostas

```
void *comp()
/* comp é função retornando ponteiro para void. */
```

```
void (*comp)()
/* comp é ponteiro para função retornando void. */
```

```
char (*(*x())[])()
/* x é função retornando ponteiro para vetor[] */
/* de ponteiros para função retornando char. */
```

```
char (*(*x[3])())[5]
/* x é um vetor[3] de ponteiros para função */
/* retornando ponteiros para vetor[5] de caracteres. */
```