Structs

Registros (Estruturas) em C

- As estruturas (*struct*) em C permitem agrupar dados heterogéneos que estão naturalmente relacionados.
- Permitem-nos guardar tipos de dados mais complexos que *int* ou *arrays*.
- Permitem definir tipos complexos e associar dados.

Declaração

- Uma *struct* é declarada com a *keyword* do mesmo nome.
  Por exemplo, se quiser guardar um registo de uma pessoa posso declarar a seguinte estrutura,

  ```
  struct pessoa {
    int idade;
    char nome[30];
  }
  ```
- Podemos então usar esta estrutura no nosso código:

  ```
  int main(){
    struct pessoa p;
    p.idade = 18;
    p.nome = "Hugo";

    printf("Idade: %d, Nome: %s", p.idade, p.nome);
    return 0;
  }
  ```

Typedef

- A palavra *typedef* permite-nos dar alcunhas aos nossos tipos, por exemplo, se fizermos `typedef int inteiro` podemos escrever:

  ```
  typedef int inteiro;

  int main(){
    inteiro i = 0;
  }
  ```

- O typedef é muito útil quando usado em conjunto com *structs* para que não tenhamos de repetir a palavra *struct* tantas vezes.

  ```
  struct spessoa{
    int idade;
    char nome [30];
  }

  typedef struct spessoa Pessoa;

  int main(){
    Pessoa p; //Usamos os typedef em vez do "nome completo" do tipo
  }
  ```
- Isto pode ainda ser abreviado:

  ```
  typedef struct spessoa{
    int idade;
    char nome[30];
  } Pessoa;
  ```

Estruturas dentro de estruturas

- As estruturas são apenas uma coleção de campos relacionados e, se podemos guardar um *int*, também podemos guardar outra estrutura.

  Imagine-se que queremos guardar a data de nascimento da nossa Pessoa:

```
typedef struct sdata{
  int dia;
  int mes;
  int ano;
} Data;

typedef struct spessoa{
  int idade;
  char nome[30];
  Data nascimento;
} Pessoa;

int main(){
  Pessoa p;
  p.idade = 18;
  p.nascimento.dia = 25;
  p.nascimento.mes = 4
  p.nascimento.ano = 1997;

  return 0;
}
```

Passar por copia vs referência

- Quando passamos uma instância da nossa estrutura para uma função podemos fazê-lo por cópia ou por referência.
  - Isto tem as mesmas consequências de passar qualquer outra variável por cópia ou referência.
- Uma diferença importante é que se passarmos a estrutura por cópia esta vai ter de ser completamente copiada para a função que a recebe, tendo um desempenho menor.
- Por esta razão, é costume declarar apontadores para as estruturas e trabalhar com eles.

```
int main(){
Pessoa p;
Pessoa *pes;
pes = &p;
(*pes).idade = 18;
}
```

- Porque estamos a usar um apontador, temos de aceder ao valor apontado (*pes) para alterarmos o conteúdo.
- Mas como esta sintaxe é muito chata de escrever existe um forma equivalente de o fazer: `p->idade`.

```
void birthday(Pessoa p){
  p.idade +=1;
}

void birthday_prt(Pessoa *p){
  p->idade += 1;
}

int main(){
  Pessoa p = (Pessoa){.idade = 18};
  printf("%d\n", p.idade);
  birthday(p);
  printf("%d\n", p.idade);
  birthday_prt(&p);
  printf("%d\n", p.idade)
```

```
  return 0;
}
```

- Podemos, então, ter apontadores para estruturas dentro da nossa estrutura.
- Para ilustrar a diferença e data de nascimento será “normal” enquanto que a de óbito será um apontador:

```
typedef struct data {
  int dia;
  int mes;
  int ano;
} Data;

typedef struct pessoa {
  int idade;
  char nome [30];
  Data nascimento;
  Data * falecimento;
}


int main (){
  Pessoa *p = malloc(sizeof(struct pessoa));
  p->idade = 18;
  p->nascimento.dia = 25;
  p-> nascimento-> ano = 2037;

  return 0;
}
```

- Como se pode verificar, o padrão mantem-se:
    - Se a variável é um apontador usa-se `->` caso contrário usa-se o `.`.