# 9. OUTROS TIPOS DE DADOS

A linguagem C permite criar tipos de dados que são definidos pelo usuário de várias formas. Neste capítulo serão estudados os seguintes tipos:

- Estrutura (agrupamento de variáveis sob um nome);
- Criado através de *typedef* (define um novo nome para um tipo existente);
- União (permite que a mesma porção da memória seja definida por dois ou mais tipos diferentes de variáveis);
- Enumeração (lista de símbolos).

## 9.1. Estruturas

Em C uma estrutura é uma coleção de variáveis referenciadas por um nome, fornecendo uma maneira conveniente de se ter informações relacionadas agrupadas. Uma definição de estrutura forma um modelo que pode ser usado para criar variáveis de estruturas. As variáveis que compreendem a estrutura são chamadas membros da estrutura. Os membros da estrutura são em geral chamados de elementos ou campos. A forma geral de uma definição de estrutura é:

```
struct <identificador> {
        tipo <nome_variável>;
        tipo <nome_variável>;
        :
} <variáveis_estrutura>;
```

O exemplo a seguir mostra como utilizar uma *struct*:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
main() {
  struct endereco {        // "endereco" é o identificador
        char rua[40];              // \
        int num;                   //  \
        int complemento;           //   \
        char bairro[20];           //    > campos
        char cidade[20];           //   /
        char estado[3];            //  /
        char cep[10];              // /
  };  // como definição de estrutura é um comando, tem que ter o ";"
  endereco e1;         //declaração de variáveis do tipo "endereco";
  struct endereco e2;   // compilador C aloca memória para todos os campos
  clrscr();
  strcpy(e1.rua, "Avenida Ipiranga");       // inicialização dos campos de e1...
  e1.num = 1234;
  e1.complemento = 101;
  strcpy(e1.bairro, "Partenon");
  strcpy(e1.cidade, "Porto Alegre");
  strcpy(e1.estado, "RS");
  strcpy(e1.cep, "90000-123");
  strcpy(e2.rua, "Rua Lima e Silva");       // inicialização dos campos de e2...
  e2.num = 1987;
  e2.complemento = 308;
  strcpy(e2.bairro, "Cidade Baixa");
```

```
  strcpy(e2.cidade, "Porto Alegre");
  strcpy(e2.estado, "RS");
  strcpy(e2.cep, "90000-345");
  printf("\nDados: \n");
  printf("\n%s %d/%d", e1.rua, e1.num, e1.complemento);
  printf("\n%s", e1.bairro);
  printf("\n%s, %s/%s", e1.cep, e1.cidade, e1.estado);
  printf("\n\n%s %d/%d", e2.rua, e2.num, e2.complemento);
  printf("\n%s", e2.bairro);
  printf("\n%s, %s/%s", e2.cep, e2.cidade, e2.estado);
  getch();
}
```

Também é possível declarar uma ou mais variáveis ao definir a estrutura. Por exemplo:

```
struct endereco {          // "endereco" é o identificador
        char rua[40];              // \
        int num;                   //  \
        int complemento;           //   \
        char bairro[20];           //    > campos
        char cidade[20];           //   /
        char estado[3];            //  /
        char cep[10];              // /
 } e2, e3;  // variáveis
```

Se for necessário apenas uma variável estrutura, o nome da estrutura não é necessário. Isso significa que:

```
struct {
        char rua[40];              // \
        int num;                   //  \
        int complemento;           //   \
        char bairro[20];           //    > campos
        char cidade[20];           //   /
        char estado[3];            //  /
        char cep[10];              // /
 } e1;  // variável
```

declara uma variável chamada "e1" como definido pela estrutura que a precede.

Se o compilador C é compatível com o padrão C ANSI, a informação contida em uma estrutura pode ser atribuída a outra estrutura do mesmo tipo. Isto é, em lugar de ter de atribuir os valores de todos os elementos separadamente, é possível empregar um único comando de atribuição. Por exemplo:

```
e1 = e2; // considerando as variáveis declaradas no exemplo anterior
```

Talvez o uso mais comum de estruturas seja em matriz, ou vetor, de estruturas. Para declarar uma matriz de estruturas, primeiro é preciso definir uma estrutura e, então, declarar uma variável matriz desse tipo. Por exemplo, para declarar um vetor de estruturas com 100 elementos do tipo "endereco", definido anteriormente, deve-se escrever:

```
endereco vetor_end[100];
```

ou

```
struct endereco vetor_end[100];
```

Para manipular os dados do vetor, devem ser fornecidos o índice e o campo. Exemplo:

```
strcpy(vetor_end[0].rua, "Carlos Gomes");
strcpy(vetor_end[1].rua, "Goethe");
```

Quando se passa um elemento (campo) de uma variável estrutura para uma função, na verdade está se passando o valor desse elemento para a função. Ou seja, neste caso o que se tem é a passagem de parâmetro de uma variável simples. Esta passagem de parâmetro pode ser feita por valor ou por referência, como mostra o próximo exemplo.

```
struct teste {
        char x;
        int y;
        float z;
        char s[10];
};
teste t1;

// Passagem de parâmetro por valor de campos da estrutura
func1(t1.x);      // passa o valor do caractere de x
func2(t1.y);      // passa o valor inteiro de y
func3(t1.z);      // passa o valor float de z
func4(t1.s);      // passa o endereço da string s
func1(t1.s[2]);   // passa o valor do caractere de s[2]

// Passagem de parâmetro por referência de campos da estrutura
func1(&t1.x);     // passa o endereço do caractere x
func2(&t1.y);     // passa o endereço do inteiro y
func3(&t1.z);     // passa o endereço do float z
func4(t1.s);      // passa o endereço da string s
func1(&t1.s[2]); // passa o endereço do caractere s[2]
```

Quando uma estrutura é usada como um argumento para uma função, a estrutura inteira é passada usando o método padrão de chamada por valor. Obviamente, isso significa que quaisquer alterações podem ser feitas no conteúdo da estrutura dentro da função para a qual ela é passada sem afetar a estrutura usada como argumento.

C também permite ponteiros para estruturas exatamente como permite ponteiros para outros tipos de variáveis. Como outros ponteiros, ponteiros para estrutura são declarados colocando * na frente do nome da estrutura. Há dois usos primários para ponteiros de estrutura: gerar uma chamada por referência para uma função e criar listas encadeadas e outras estruturas de dados dinâmicas usando o sistema de alocação de C.

Existe um prejuízo maior em passar todas as estruturas, exceto as mais simples, para funções: o tempo extra necessário para colocar (e tirar) todos os campos da estrutura na pilha. Em estruturas simples, com poucos campos, esse tempo extra não é tão grande. Entretanto, se vários campos são usados, ou se alguns dos campos são matrizes, a performance pode ser reduzida a níveis inaceitáveis. A solução para esse problema é passar apenas uma ponteiro para uma função.

Quando um ponteiro para uma estrutura é passado para uma função, apenas o endereço da estrutura é colocado (e tirado) da pilha. Isso contribui para chamadas muito rápidas a funções. Uma segunda vantagem, em alguns casos, é quando a função precisa referenciar o argumento real em lugar de uma cópia. Passando um ponteiro, é possível alterar o conteúdo dos campos reais da estrutura usada na chamada. Para encontrar o endereço da variável estrutura, deve-se colocar o operador & antes do nome da estrutura. Para acessar os campos de uma estrutura usando um ponteiro para a estrutura, deve-se usar o operador ->, como mostra o exemplo abaixo.

```
/* Mostra um relógio utilizando struct */
#include <stdio.h>
#include <conio.h>
#include <dos.h>

struct my_time {
  int hours;
  int minutes;
  int seconds;
};
```

```
void display(struct my_time *t);
void update(struct my_time *t);
void main(void) {
  int i;
  struct my_time systime;
  clrscr();
  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;
  for(i=0;i<70;i++) {
    update(&systime);
    display(&systime);
  }
  getch();
}
void update(struct my_time *t) {
  t->seconds++;
  if(t->seconds==60) {
    t->seconds = 0;
    t->minutes++;
   }
  if(t->minutes==60) {
    t->minutes = 0;
    t->hours++;
  }
  if(t->hours==24) t->hours = 0;
  sleep(1);
}
void display(struct my_time *t) {
  printf("%02d:", t->hours);
  printf("%02d:", t->minutes);
  printf("%02d\n", t->seconds);
}
```

Torna-se interessante comentar que um campo de uma estrutura também pode ser uma matriz, uni ou multidimensional. Um campo de uma estrutura que é uma matriz é tratado como nos exemplos anteriores, como mostra o exemplo a seguir.

```
struct exemplo {
        int a[10][10];
        float b;
} y;
y.a[1][2] = 8;
y.a[3][7] = 1;
y.a[9][0] = 6;
```

Campos de uma estrutura também podem ser uma estrutura. Neste caso, se diz que as estruturas estão **aninhadas**. O padrão ANSI C especifica que as estruturas podem ser aninhadas até 15 níveis [SCH 96]. Por exemplo, a estrutura "address" é aninhada em "emp" no seguinte código:

```
struct emp {
        struct addr address; // estrutura aninhada
        float wage;
} worker;
strcpy(worker.address.street, "Wall Street");
worker.address.zip = 93456;
```

Para finalizar, é importante lembrar que *struct* também pode ser usada como retorno de uma função. Por exemplo:

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

```
struct endereco {
        char rua[40];
        int num;
        int complemento;
};
endereco InicDados(void);
main() {
  struct endereco e;
  clrscr();
  e = InicDados();
  printf("\n%s %d/%d", e.rua, e.num, e.complemento);
  getch();
}
endereco InicDados(void) {
  endereco e;
  strcpy(e.rua, "Rua Ferreira Viana");
  e.num = 893;
  e.complemento = 101;
  return e;
}
```

## 9.2. Definição de Novos Nomes aos Tipos de Dados

C permite que sejam definidos explicitamente novos nomes aos tipos de dados, utilizando a palavra-chave ***typedef***. Na verdade não é realmente criada uma nova classe de dados, mas, ao contrário, é definido um novo nome para um tipo já existente. Esse processo pode ajudar a tornar programas dependentes da máquina um pouco mais portáveis. Se forem definidos novos nomes de tipo para cada tipo de dado dependente da máquina usado no programa, apenas os comandos *typedef* teriam que ser mudados quando o programa fosse compilado em um novo ambiente. Também pode auxiliar numa documentação do código, permitindo nomes mais descritivos aos tipos de dados padrões. A forma geral de um comando *typedef* é:

*typedef <tipo> <novo_nome>;*

onde <tipo> é qualquer tipo de dados permitido e <novo_nome> é o novo nome para esse tipo. É importante ressaltar que o novo nome definido é uma opção, não uma substituição ao nome do tipo existente.

Por exemplo, é possível criar um novo nome para *float* usando

*typedef float real;*

Esse comando diz ao compilador para reconhecer "real" como outro nome para *float*. Depois, é possível criar uma variável *float* usando "real":

*real x;*

Aqui "x" é uma variável de ponto flutuante do tipo "real", que é uma outra palavra para *float*.

Agora que "real" foi definido, ele pode ser usado no lado direito de um outro *typedef*. Por exemplo,

*typedef real pontoflutuante;*

diz ao compilador para reconhecer "pontoflutuante" como um outro nome para "real", que é um outro nome para *float*.

Existe uma aplicação de *typedef* que pode ser útil: *typedef* pode ser usada para simplificar a declaração de variáveis estrutura, união ou enumeração. Por exemplo, considerando:

```
struct mystruct {
        unsigned x;
        float f;
};
```

Para declarar uma variável do tipo "mystruct", é usada uma declaração como esta:
*struct mystruct s;*
que exige o uso de dois identificadores: *struct* e "mystruct". No entanto, se for aplicado *typedef* à declaração de "mystruct",
```
typedef struct {
        unsigned x;
        float f;
}mystruct;
```
então pode-se declarar variáveis deste tipo de estrutura usando a seguinte declaração:
*mystruct s;*

Pode-se observar que através do uso de *typedef* é criado um novo identificador de tipo composto de um único nome. O uso de *typedef* pode tornar o código mais fácil de ler e mais fácil de portar para um novo equipamento. Mas é importante lembrar que não é criado um tipo de dados novo [SCH 96].

## 9.3. Enumeradores

O tipo *enum* foi elaborado para que o programador torne mais legível seu código. Uma enumeração é um conjunto de constantes inteiras que especifica todos os valores legais que uma variável desse tipo pode ter. Na declaração de tipo enumerado coloca-se o nome do tipo, os valores que uma variável deste tipo receberá e os valores com que o programa tratará essas variáveis. A forma geral para enumeração é:
*enum <identificador> { <lista_de_enumeração> } <lista_de_variáveis>;*

Em outras palavras, enumerações permitem definir tipos inteiros cujos valores são representados por identificadores. Por exemplo:
```
enum dia_semana {
        domingo = 1,    // tratará como 1
        segunda,        // tratará como 2
        terça,          // tratará como 3
        quarta,         // tratará como 4
        quinta,         // tratará como 5
        sexta,          // tratará como 6
        sabado          // tratará como 7
};
```

Em C o nome da enumeração (ou identificador) é tratado como um tipo conhecido a partir da sua definição. Assim, para declarar-se variáveis deste tipo deve-se usar seu rótulo específico, como por exemplo:
```
dia_semana dia_de_feira, dia_do_aniversario;
dia_de_feira = sabado;
dia_do_aniversario = quarta;
```

O ponto-chave para o entendimento de uma enumeração é que cada símbolo representa um valor inteiro. Dessa forma, eles podem ser usados em qualquer lugar em que um inteiro pode ser usado. A cada símbolo é dado um valor maior em uma unidade do precedente. O valor do primeiro símbolo da enumeração é zero. Uma suposição errônea sobre enumerações é que os símbolos podem ser enviados para a saída e recebidos da entrada diretamente. Isso não acontece. Os próximos exemplos mostram as operações permitidas ou não com enumerações [GRA 91, SCH 96].
```
// Enumeração das moedas usadas nos Estados Unidos
enum coin {
penny,          // tratará como 0
nickel,         // tratará como 1
dime,           // tratará como 2
```

```
quarter=100,    // tratará como 100
half_dollar,    // tratará como 101
dollar          // tratará como 102
};
coin money;
money = dollar;
printf("%s", money);      // isso NÃO funciona
strcpy(money, "dime");  // isso NÃO funciona
switch (money) {
        case penny: printf("penny");
                    break;
        case nickel: printf("nickel");
                    break;
        case dollar: printf("dollar");
                    break;
}
```

## 9.4. Uniões

Em C uma *union* é uma posição de memória que é compartilhada por duas ou mais variáveis diferentes, geralmente de tipos diferentes, em momentos diferentes. A definição de uma *union* é semelhante à definição de estrutura. Sua forma geral é:

```
union <identificador> {
        <tipo1> <nome_var1>
        <tipo2> <nome_var2>
        :
} <variáveis_união>;
```

Por exemplo,

```
union u_type {
        int i;
        char ch;
};
```

Essa definição não declara quaisquer variáveis. É possível declarar uma variável colocando seu nome no final da definição ou usando um comando de declaração separado. Para declarar uma variável "*union* cnvt" do tipo "u_type", usando a definição dada há pouco, é necessário escrever:

```
union u_type cnvt;
```

Neste exemplo, na "*union* cnvt", tanto o inteiro "i" como o caracter "ch" compartilham a mesma posição de memória. Entretanto, obviamente, "i" ocupa dois *bytes* e "ch" ocupa apenas um. A figura 9.1 mostra como "i" e "ch" compartilham o mesmo endereço. A qualquer momento, é possível se referir ao dado armazenado em "cnvt" como um inteiro ou um caracter.

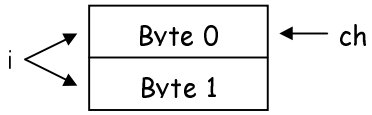


Figura 9.1 – Como "i" e "ch" utilizam a "*union* cnvt"[SCH 96]

Quando uma *union* é declarada, o compilador cria automaticamente uma variável grande o bastante para conter o maior tipo de variável da *union*. No exemplo, supondo inteiros de dois *bytes*, "cnvt" tem dois *bytes* de comprimento para poder armazenar "i", embora "ch" exija somente um *byte*.

Para acessar um elemento da *union*, deve-se usar a mesma sintaxe que seria usada para estruturas: o operador ponto e o operador seta. No caso, para atribuir o inteiro 10 ao elemento "i" de "cnvt" utiliza-se:

```
cnvt.i = 10;
```

A seguir, um ponteiro para "cntv" é passado para uma função:

```
void func1(union u_type *un) {
        un->i = 10; // atribui 10 a cnvt usando uma função
}
```

Usar uma *union* pode ajudar na produção de código independente da máquina (portável). Como o compilador não perde o tamanho real das variáveis que perfazem a união, nenhuma dependência da máquina é produzida. Não é necessário se preocupar com o tamanho de *int*, *long*, *float* ou o que quer que seja.

*Unions* são usadas freqüentemente quando conversões de tipo são necessárias porque é possível referenciar os dados contidos na *union* de maneiras diferentes. Por exemplo, pode-se usar uma *union* para manipular os bytes que constituem um *double*, a fim de alterar sua precisão ou para realizar um tipo de arredondamento incomum.

Para se ter uma idéia da utilidade de uma *union* quando conversões não padrão de tipos são necessárias, se pode considerar o problema de escrever um inteiro em um arquivo. A biblioteca padrão de C não contém funções projetadas para especificamente escrever um inteiro em um arquivo. Embora seja possível escrever qualquer tipo de dado (incluindo um inteiro) em um arquivo, usar *fwrite()* é muito para uma operação tão simples.

Contudo, usando uma *union*, pode-se criar facilmente uma função chamada *putw()*, a qual escreve a representação binária de um inteiro em um arquivo, um *byte* por vez, como mostra o trecho de código a seguir.

```
union pw { // union: inteiro e vetor de caracter de dois bytes
        int i;
        char ch[2];
}
putw(union pw word, FILE *fp) {
        putc(word->ch[0], fp); // escreve a primeira metade
        putc(word->ch[1], fp); // escreve a segunda metade
}
```

Embora possa ser chamada com um inteiro, *putw()* ainda pode usar a função *putc()* para escrever um inteiro em um arquivo em disco um *byte* por vez [SCH 96].