Estruturas compostas

Estruturas são um tipo de dados definido pelo programador, capaz de armazenar, sob um msmo nome de variável, diversos dados inter-relacionados e possivelmente de tipos diferentes.

Por que usar estruturas?

Vamos imaginar um programa que trabalha com dados sobre alunos de uma universidade, tais como nome, registro acadêmico e código do curso.

Cada uma destas informações poderia ser armazenada em uma variável isolada. Se desejarmos manipular uma lista de alunos, então existe a opção de declarar vários vetores, um para cada um dos dados de nomes, registros acadêmicos e códigos de curso. Assim, para cada índice *i*, os dados nas posições *i* desses vetores seriam referentes ao aluno *i*.

Esta opção não parece muito interessante. O programa terá que lidar com um grande número de variáveis. Quando desejamos copiar ou alterar dados de um aluno, muitas variáveis precisam ser acessadas, e facilmente o programador esquecerá ou confundirá algumas delas. Além disso, será difícil introduzir futuras modificações, tais como adicionar um novo item à descrição do aluno. Por outro lado, como todas essas variáveis descrevem o mesmo aluno, é natural que elas sejam armazenadas de forma mais aglutinada, juntando informações de um mesmo aluno.

Queremos uma alternativa melhor, que permita reunir dados mais complexos formados pela união de várias variáveis inter-relacionadas. Desejamos armazenar todas estas variáveis sob um único nome, para nos permitir operar sobre elas como se fossem uma única variável de um tipo de dados mais abrangente.

Conceitos

Uma estrutura, ou registro, é formada por uma coleção de uma ou mais variáveis (provavelmente com tipos diferentes) declaradas juntas sob um único nome e manipuladas simultaneamente nas operações. Para diferenciar das variáveis convencionais, as variáveis que formam uma estrutura são chamadas de **atributos** da estrutura. Note que a estrutura é um conceito diferente do vetor, que é uma coleção de valores do *mesmo tipo*, tratados sob um mesmo nome de variável e acessíveis através de um índice.

A **Error! Reference source not found.** ilustra um exemplo de uma estrutura correspondente a um novo tipo de dados: `TipoAluno`, que agrupa todos os atributos de um aluno matriculado na universidade. Note que cada atributo possui seu próprio tipo de dados.

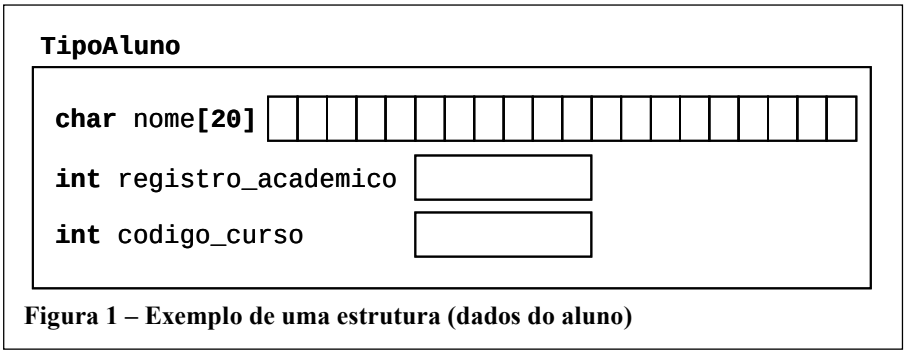

Figura 1 – Exemplo de uma estrutura (dados do aluno)

Declaração simples de estruturas

A declaração de estruturas é um pouco mais elaborada que as outras apresentadas até o momento. A forma geral da declaração está indicada abaixo:

```
struct {
   tipo1 atributo1;
   tipo2 atributo2;
   tipo3 atributo3;
   ...
} variavel;
```

A primeira palavra, **`struct`** (*structure*), indica que estamos declarando uma estrutura, ou registro. Seguem entre chaves **`{ }`** uma lista das declarações (tipo e nome) de cada um dos atributos que compõem a estrutura. Note como a declaração de um atributo segue a mesma sintaxe que a declaração de uma variável. O tipo do atributo pode ser qualquer tipo de dados da linguagem C, inclusive vetores, matrizes, ou qualquer outro tipo que aprenderemos no decorrer do curso. Em particular para atributos vetores ou atributos matrizes, a declaração será:

```
struct {
   ...
   tipo vetor[tamanho];
   tipo matriz[linhas][colunas];
   ...
} ...;
```

Por último, após as chaves, segue uma lista de um ou mais nomes de variáveis cujo tipo será essa estrutura.

Observe que a declaração de uma variável com tipo estrutura segue a mesma lógica da declaração de variáveis padrões:

```
tipo variável;
```

No entanto, o tipo de fato é uma construção bem elaborada, que ocupa várias linhas:

```
struct {
   tipo1 atributo1;
   tipo2 atributo2;
   tipo3 atributo3;
   ...
} variavel1, variavel2;...;
```

tipo

variáveis

Um programa pode declarar um grande número de estruturas, e cada uma delas será tratada como um tipo de dados diferente. O compilador se encarrega de tratar todas elas apropriadamente.

Exemplo

Por exemplo, para declarar uma variável com informações sobre o aluno:

```
struct {
      char nome[50];
      int registro_academico;
      int codigo_curso;
} aluno;
```

Em seguida, o programa poderia declarar uma variável com dados de funcionário:

```
struct {
      char nome[50];
      float salario;
      int codigo_cargo;
} funcionario;
```

Esta segunda declaração para a variável `funcionario` é totalmente independente da declaração anterior para a variável `aluno`. As duas variáveis existirão simultaneamente, com estruturas diferentes. Note que as variáveis `aluno` e `funcionario` apresentam um atributo de mesmo none, chamado `nome`. Apesar destes atributos terem o mesmo nome, eles pertencem a estruturas diferentes. São variáveis armazenadas em lugares diferentes da memória. Alterar o nome do funcionário mantém o nome do aluno intacto.

Acesso ao conteúdo de estruturas

Para modificar ou ler um valor de um atributo, escrevemos o nome da variável que armazena a estrutura e o nome do atributo, separados por um ponto.

Para modificar o atributo `registro_academico` da variável `aluno`:

```
aluno.registro_academico = 991122;
```

Nova expressão de referência de memória

Para modificar ou ler atributos de uma estrutura, precisamos obter uma referência de memória para o atributo que desejamos acessar. Para isto, utilizamos o **operador de seleção de atributo**, que na linguagem C é representado pelo ponto '`.`'. Ao lado esquerdo do ponto escreve-se uma referência de memória para a estrutura, o lado direto contém o nome do atributo desejado.

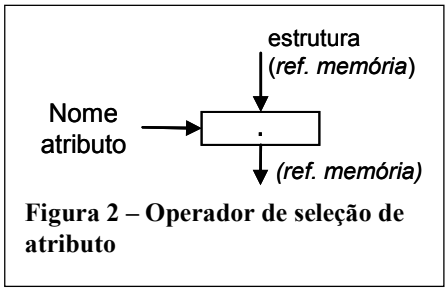

Figura 2 – Operador de seleção de atributo

O resultado da expressão de seleção de atributo pode ser utilizado em qualquer lugar onde se possa usar uma variável de mesmo tipo que o atributo selecionado.

Regras para acesso a estruturas

Atribuir o valor de todos os elementos

Não existe uma forma de modificar todos os atributos de uma estrutura em uma só linha. Cada atributo precisa ser acessado individualmente. Vamos supor que aluno é uma variável cujo tipo é a estrutura declarada anteriormente.

```
strcpy(aluno.nome, "Daniel");
aluno.registro_academico = 991122;
aluno.codigo_curso = 3;
```

Note que `aluno.nome` retorna uma referência de memória para um vetor de caracteres. Vetores não permitem o operador de atribuição, e por este motivo, utilizamos a função `strcpy` da biblioteca `strings.h` para escrever o texto "Daniel" no atributo `aluno.nome`.

Copiar estruturas

Além do operador de seleção de atributo, estruturas permitem o uso do operador de atribuição para copiar todo conteúdo de uma estrutura para outra de mesmo tipo. Supondo que `aluno1` e `aluno2` são duas variáveis declaradas como estrutura de mesmo tipo,

```
aluno1 = aluno2;
```

copiará literalmente todos os dados de `aluno2` para a variável `aluno1`.

Declaração com definição de tipo

Se desejarmos reutilizar a mesma estrutura para variáveis em outras partes do programa, é necessário re-escrever exatamente a mesma definição da estrutura em cada parte do programa. É um trabalho tedioso e sujeito a erros, além do que, se desejarmos alterar a estrutura, precisamos atualizar todas as declarações dela espalhadas por todo o código.

É possível dar um nome (um identificador) para uma estrutura. Desta forma, o compilador cria um novo tipo de dados baseado na definição da estrutura. Em outra parte do programa, este tipo poderá ser utilizado para declarar variáveis cujo tipo é esta mesma estrutura, sem necessidade de definir novamente todos os seus atributos. O nome da estrutura é escrito após a palavra **`struct`**.

```
struct nome_estrutura {
   tipo1 atributo1;
   tipo2 atributo 2;
   tipo3 atributo 3;
   ...
} variável;
```

Observe que o nome da estrutura é escrito antes das chaves `{ }`, enquanto que o nome da variável é escrito depois das chaves.

Esta declaração é particularmente interessante, pois além de criar novas variáveis, ela também *define um novo tipo de dados*, que passa a existir junto com os tipos de dados nativos da linguagem C. Desta forma, o programador pode completar o repositório de tipos

de dados com novas estruturas necessárias para organizar seu código. Nas demais declarações de variáveis, ao invés de repetir toda lista de atributos, basta utilizar a forma abreviada com apenas o nome da estrutura:

```
struct nome_estrutura variavel;
```

Apesar da forma abreviada, a palavra chave **struct** continua obrigatória para deixar explícito tratar-se de uma declaração de variável tipo estrutura.

É possível definir estruturas nomeadas, sem sequer declarar variáveis no instante da declaração. Neste caso, o compilador apenas lembra a definição do novo tipo de dados formado pela estrutura. Se alguma variável vier as ser declarada com este tipo mais adiante no programa, a definição estará disponível.

```
struct nome_estrutura {
   tipo1 atributo1;
   tipo2 atributo2;
   tipo3 atributo3;
   ...
};
```

O programador experiente prefere definir estruturas no início do programa, sem neste momento declarar variáveis. Desta forma, o código fica mais organizado: um trecho de definições de estruturas no início do programa e, mais tarde, declaração de variáveis conforme a necessidade. Isto facilita o reaproveitamento de uma declaração de estrutura para se declarar várias variáveis.

Curiosamente, a linguagem C permite definir novas estruturas sem nome nem declaração de variáveis:

```
struct {
     char nome[50];
     int registro_academico;
     int codigo_curso;
};
```

Este tipo de sentença é totalmente inútil em um programa, pois não cria variáveis, nem introduz um novo tipo de estrutura. Ou seja, esta declaração não surte efeito.

Exemplo

Um programa que lê dados de dois alunos e em seguida seleciona e imprime aquele que está há mais tempo na universidade (e portanto com um número de registro acadêmico menor).

Código Fonte

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

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

     // Declaração do tipo estrutura aluno
     struct TipoAluno {
          char sobrenome[50];
          int registro_academico;
          int codigo_curso;
     };

     // Declarar várias variáveis com estruturas TipoAluno
     struct TipoAluno aluno1;
     struct TipoAluno aluno2;
     struct TipoAluno aluno_selecionado;

     // Ler dados do primeiro aluno
     printf("Aluno 1 (sobrenome, RA, cod. curso): ");
     scanf("%s", aluno1.sobrenome);
     scanf("%d", &aluno1.registro_academico);
     scanf("%d", &aluno1.codigo_curso);

     // Ler dados do segundo aluno
     printf("Aluno 2 (sobrenome, RA, cod. curso): ");
     scanf("%s", aluno2.sobrenome);
     scanf("%d", &aluno2.registro_academico);
     scanf("%d", &aluno2.codigo_curso);

     // Selecionar o aluno com menor registro academico
     if (aluno1.registro_academico < aluno2.registro_academico) {
          aluno_selecionado = aluno1;
     } else {
          aluno_selecionado = aluno2;
     }

     // Imprimir o aluno selecionado
     printf("Sobrenome: %s \n", aluno_selecionado.sobrenome);
     printf("RA:        %d \n", aluno_selecionado.registro_academico);
     printf("Curso:     %d \n", aluno_selecionado.codigo_curso);

     return 0;
}
```

Consulte: EstruturasCompostas\Aluno01\Aluno01.vcproj

Descrição passo a passo

```
struct TipoAluno {
      char sobrenome[50];
      int registro_academico;
      int codigo_curso;
};
```

Primeiro, define-se uma estrutura que representa dados de um aluno, formada por três atributos: `sobrenome`, `registro_acadêmico` e `codigo_curso`. Esta estrutura passará a ser chamada `TipoAluno`. Note que, neste momento, nenhuma variável é criada. A definição apenas instrui o compilador sobre a existência desta estrutura de dados.

```
struct TipoAluno aluno1;
struct TipoAluno aluno2;
struct TipoAluno aluno_selecionado;
```

Neste momento, estão sendo criadas três variáveis contendo estruturas de dados. As três variáveis têm o mesmo tipo, *i.e.* a mesma estrutura, mas são independentes e, portanto, podem representar alunos diferentes.

Note como o uso da estrutura `TipoAluno` simplifica o código do programa: as informações inter-relacionadas de cada aluno estão agrupadas na mesma variável.

```
printf("Aluno 1 (sobrenome, RA, cod. curso): ");
scanf("%s", aluno1.sobrenome);
scanf("%d", &aluno1.registro_academico);
scanf("%d", &aluno1.codigo_curso);
```

Lê os três atributos da estrutura.

```
if (aluno1.registro_academico < aluno2.registro_academico) {
      aluno_selecionado = aluno1;
} else {
      aluno_selecionado = aluno2;
}
```

Compara o registro acadêmico dos dois alunos. Em seguida, armazena em `aluno_selecionado` uma cópia completa do aluno que possui menor registro acadêmico.

```
printf("Sobrenome: %s \n", aluno_selecionado.sobrenome);
printf("RA:        %d \n", aluno_selecionado.registro_academico);
printf("Curso:     %d \n", aluno_selecionado.codigo_curso);
```

Por fim, imprime os atributos da variável `aluno_selecionado`.

Combinação de Estruturas e Vetores

Estruturas encadeadas

Uma estrutura é definida como:

```
struct nome_estrutura {
   tipo1 atributo1;
   tipo2 atributo2;
   tipo3 atributo3;
   ...
};
```

Note que o tipo de um atributo pode ser literalmente qualquer outro tipo, inclusive uma outra estrutura. Desta forma, é possível organizar os atributos dentro de uma estrutura. Ao invés de:

```
struct TipoAluno {
      char nome[50];
      int codigo_curso;
      int dia_matricula;
```

```
        int mes_matricula;
        int ano_matricula;
        ...
};
```

os três atributos da data de matrícula poderiam ser definidos como uma outra estrutura:

```
struct {
        char nome[50];
        int codigo_curso;
        struct {
                int dia;
                int mes;
                int ano;
        } data_matricula;
        ...
} aluno;
```

O encadeamento de estruturas pode ser realizado quantas vezes for conveniente para organizar o programa.

O acesso à data de matrícula é realizado aplicando-se duas vezes o operador de seleção de atributos: primeiro, para selecionar o atributo `data_matricula` dentro da estrutura `aluno`, depois para selecionar o atributo `dia` dentro da estrutura `data_matricula`:

```
aluno.data_matricula.dia = 2;
aluno.data_matricula.mes = 3;
aluno.data_matricula.ano = 1999;
```

A mesma idéia poderia ser aplicada para organizar os atributos do endereço do aluno. No entanto, o programador mais experiente prefere a seguinte abordagem, definindo no início do programa as estruturas, para só então declarar variáveis quando realmente precisar delas:

```
struct TipoData {
        int dia;
        int mes;
        int ano;
};
struct TipoAluno {
        char nome[50];
        int codigo_curso;
        struct TipoData data_matricula;
        ...
};
...
struct TipoAluno aluno;
```

Note que, desta forma, é possível reaproveitar a definição da estrutura `TipoData` para outras finalidades, como por exemplo, definir `TipoFuncionario`:

```
struct TipoFuncionario {
        char nome[50];
        int codigo_cargo;
        struct TipoData data_contratacao;
        ...
};
```

No entanto, não é possível declarar um atributo do mesmo tipo que a própria estrutura que está sendo definida. Vamos supor que para um funcionário, também desejamos armazenar dados sobre seu chefe:

```
struct TipoFuncionario {
     char nome[50];
     ...
     struct TipoFuncionario chefe;   // ERRADO!
     ...
};
```

Vetor de estruturas

Vamos supor que nosso programa tenha que manipular os dados de todos os alunos matriculados no curso de linguagem C. Como representar estes dados? Primeiro, note que o problema envolve uma coleção de itens do mesmo tipo, ou seja, de alunos. A solução será um vetor de alunos. Para descrever um aluno, podemos utilizar a estrutura já mencionada anteriormente.

Uma lista de 10 alunos será declarada como:

```
struct TipoAluno lista_alunos[10];
```

Cada aluno poderá ser selecionado usando o operador de índice. Em seguida, um atributo deste aluno é acessado com o operador de seleção de atributo.

Por exemplo, para atribuir o código do curso e a data de matrícula do terceiro aluno do vetor:

```
lista_alunos[2].codigo_curso = 3;
lista_alunos[2].data_matricula.dia = 2;
lista_alunos[2].data_matricula.mes = 3;
lista_alunos[2].data_matricula.ano = 1999;
```

Uniões

Uma estrutura armazena, simultaneamente, todos os seus atributos. É possível acessar cada um deles a qualquer momento, sem que isto modifique ou interfira nos demais atributos. Uniões também agregam atributos mas se comportam de forma diferente. Uma união é um agrupamento de variáveis de tipos distintos, mas que não podem coexistir simultaneamente. Apenas um dos tipos pode estar armazenado na variável de cada vez.

As uniões são usadas quando não sabemos de antemão o tipo de uma variável. Então declaramos na união todos os tipos que a variável poderá assumir. Note que, a rigor, estamos criando variáveis diferentes dentro da união, mas estará ativa somente aquela que apresenta o tipo desejado.

Como apenas uma variável é utilizada de cada vez, todas elas *compartilham o mesmo espaço na memória*.

Definição de uniões

A definição de uniões é sintaticamente semelhante à definição de estruturas:

```
union nome_uniao {
```

```
    tipo1 variavel1;
    tipo2 variavel2;
    tipo3 variavel3;
    ...
} variavel;
```

Portanto, muito cuidado é necessário para não confundir estruturas (`structs`) com uniões (`unions`)! A definição começa com a palavra **`union`**. Segue o identificador da união (opcional), indicando ao compilador que este deve lembrar esta definição como um novo tipo de dados para ser utilizado em outras partes do programa, em declarações abreviadas.

Note que ao invés de lista de atributos, agora a lista é de variáveis. Apenas uma delas estará ativa de cada vez. Após as chaves, pode-se declarar (opcionalmente) variáveis que armazenam uma das opções definidas nesta união.

Mais tarde, caso a definição tenha dado um nome à estrutura, outras variáveis poderão ser declaradas de forma abreviada:

```
union nome_uniao variavel;
```

Por exemplo, desejamos trabalhar com números, mas não sabemos de antemão se eles serão inteiros ou fracionários. Por isto, declaramos um novo tipo de dados, chamado `NumeroFlexivel`, capaz de armazenar valores de um dos dois tipos:

```
union NumeroFlexivel {
   int   inteiro;
   float real;
};
```

Regras de uso

Ao acessar uma variável de tipo **`union`**, precisamos indicar também qual a variável que desejamos realmente acessar. A sintaxe é a mesma que aquela para o acesso de atributos em estruturas.

Por exemplo, para declarar e atribuir um valor inteiro à variável `numero`:

```
union NumeroFlexivel numero;
numero.inteiro = 10;
```

Para atribuir um valor real à variável `numero`:

```
numero.real = 4.0;
```

Importante: como `numero.real` e `numero.inteiro` de fato *compartilham a mesma posição na memória*, a atribuição de um valor fracionário destrói o valor inteiro e vice versa! Não existe uma forma de "consultar" uma união sobre qual tipo ela está armazenando em um dado momento. Tampouco existe uma forma para o programa decidir automaticamente a qual variável da união ele deve atribuir um valor. Este controle fica por conta do programador.

As uniões tornam o código fonte potencialmente confuso e, portanto, devem ser utilizadas com cautela.

Exemplo

Uniões têm utilidade em declarações de estruturas complexas, nas quais alguns atributos específicos existem somente dependendo de certos dados.

Neste exemplo, declararemos uma estrutura `TipoPessoa` que pode representar alunos, professores e funcionários! Suponha que `TipoData` e `TipoEndereco` sejam estruturas declaradas anteriormente no programa.

Este exemplo ilustrará um uso completo de encadeamento de estruturas, vetores e uniões!

Código fonte:

```
struct TipoPessoa {
     char nome[50];
     struct TipoData data_nascimento;
     struct TipoEndereco endereco_residencia;
     union {
          struct {
               struct TipoData data_matricula;
               int codigo_curso;
          } aluno;
          struct {
               int codigo_contrato;
               struct TipoData data_contratacao;
               int codigo_cargo;
          } funcionario;
          struct {
               char area_pesquisa[40];
               char nome_departamento[40];
          } professor;
     } dados_especificos;
} pessoa;
```

Descrição passo a passo:

```
struct TipoPessoa {
...
} pessoa;
```

A declaração define uma nova estrutura, chamada `TipoPessoa`, que contém atributos que são comuns para alunos, professores e funcionários, bem como outros atributos específicos. Além disso, cria-se uma variável `pessoa` tendo como tipo essa estrutura.

```
char nome[50];
struct TipoData data_nascimento;
struct TipoEndereco endereco_residencia;
```

São declarados alguns atributos comuns a todas as pessoas: o nome, data de nascimento e o endereço. Note que as duas últimas são estruturas, que supomos que já tenham sido declaradas antes no mesmo programa.

Cada um destes atributos pode ser acessado como:

- pessoa.nome
- pessoa.data_nascimento:
    - pessoa.data_nascimento.dia,
    - pessoa.data_nascimento.mes,
    - pessoa.data_nascimento.ano
- pessoa.endereco:
    - pessoa.endereco.rua
    - pessoa.endereco.numero
    - pessoa.endereco.cidade
- etc...
-

Seguindo, temos:

```
union {
   ...
} dados_especificos;
```

O quarto atributo, dados_especificos, contém os dados que são específicos de alunos, professores ou funcionários.

```
struct {
      struct TipoData data_matricula;
      int codigo_curso;
} aluno;
```

A primeira opção da união são dados específicos do aluno que, por serem vários dados de tipos diferentes, são declarados como uma estrutura. Cada um destes atributos pode ser acessado como:

- pessoa.dados_especificos.aluno.data_matricula
    - pessoa.dados_especificos.aluno.data_matricula.dia
    - pessoa.dados_especificos.aluno.data_matricula.mes
    - pessoa.dados_especificos.aluno.data_matricula.ano
- pessoa.dados_especificos.aluno.codigo_curso

```
struct {
      int codigo_contrato;
      struct TipoData data_contratacao;
      int codigo_cargo;
} funcionario;
```

A segunda opção da união mostra dados específicos do funcionário que, por serem também dados de tipos diferentes, são declarados como uma estrutura.

Cada um destes atributos pode ser acessado como:

- pessoa.dados_especificos.funcionario.codigo_contrato
    - pessoa.dados_especificos.funcionario.data_contratacao.dia
    - pessoa.dados_especificos.funcionario.data_contratacao.mes
    - pessoa.dados_especificos.funcionario.data_contratacao.ano
- pessoa.dados_especificos.funcionario.codigo_cargo

```
struct {
      char area_pesquisa[40];
      char nome_departamento[40];
} professor;
```

A terceira opção da união são dados específicos do professor que, por serem dados de tipos diferentes, são também declarados como uma estrutura.

Cada um destes atributos pode ser acessado como:

- `pessoa.dados_especificos.professor.area_pesquisa`
- `pessoa.dados_especificos.professor.nome_departamento`

Nomeando Tipos

Ao criar novos tipos de dados, a descrição destes novos tipos pode ser bastante verbosa e técnica. Seria interessante poder criar novos nomes mais curtos, ou mais intuitivos, para os tipos utilizados freqüentemente. Uma declaração **`typedef`** permite definir novos nomes para tipos de dados já existentes. Outra aplicação típica do **`typedef`** é criar sinônimos que ocultam detalhes complicados da declaração dos tipos.

Durante a compilação, os sinônimos são substituídos novamente pelo tipo original.

Por que nomear tipos?

Vamos criar uma estrutura que representa um número complexo. Ela é definida com o nome *`TipoComplexo`*. A definição de tipo por si só ainda não cria variáveis. Por este motivo, o compilador guarda esta definição sob o nome *`TipoComplexo`*.

```
struct TipoComplexo {
     double real;
     double imaginario;
};
```

Para criar duas variáveis `a` e `b` do tipo *`TipoComplexo`*, utilizamos uma declaração resumida, pois o compilador já conhece **`struct`** *`TipoComplexo`*.

```
struct TipoComplexo a, b;
```

Em cada declaração de variáveis, é necessário adicionar a palavra `struct`, para evidenciar tratar-se de uma estrutura. Seria interessante poder criar estas variáveis com uma declaração mais curta. Além disso, não queremos que o programador precise se preocupar com o fato do número complexo estar sendo implementado como uma estrutura. Desejamos manter isto oculto.

Declaração de sinônimos

Para criar um sinônimo para um novo tipo:

```
typedef tipo_original sinônimo;
```

Onde *tipo_original* é um tipo já conhecido pelo compilador (tanto um tipo interno primitivo quanto um outro tipo qualquer definido pelo programador).

Exemplo simples

Ao invés de **int** para declarar números inteiros, gostaríamos de utilizar a palavra "inteiro". Podemos utilizar **typedef** para criar um sinônimo:

```
typedef int inteiro;
```

Agora, para declarar uma variável utilizando o sinônimo, simplesmente escrevemos o novo nome como o tipo.

```
inteiro numero;
```

A variável numero é declarada como sendo do tipo *inteiro*, que de fato será interpretado pelo compilador como um tipo *int*.

Exemplo mais elaborado

Ao invés de *unsigned long long int* para declarar números inteiros muito grandes, gostaríamos de utilizar a palavra *gigante*, que é um nome mais curto e fácil de digitar:

```
typedef unsigned long long int gigante;
```

Para declarar uma variável:

```
gigante numero;
```

A variável numero é declarada como do tipo *gigante*, que de fato será interpretado pelo compilador como um tipo *unsigned long long int*.

Exemplo com estruturas

Ao invés de **struct** TipoComplexo para declarar números complexos, gostaríamos de utilizar a palavra *complexo*, que é uma descrição mais curta e também esconde o fato que o tipo *complexo* é, na realidade, uma estrutura. Poderíamos então usar a declaração:

```
typedef struct TipoComplexo complexo;
```

E agora, para declarar um número complexo usaríamos:

```
complexo numero;
```

A variável numero é declarada como tipo *complexo*, que de fato será interpretado pelo compilador como um tipo **struct** *TipoComplexo*.

Declaração de novos tipos

A forma mais fácil de se declarar um sinônimo para um vetor, uma estrutura ou uma união, é primeiro defini-la em alguma parte do programa sem criar variáveis e, mais à frente, utilizar o **typedef** para associar um novo nome (sinônimo) a este tipo.

Esta operação também pode ser realizada em uma única sentença em C:

```
typedef declaração_completa sinônimo;
```

A declaração completa é a definição de um tipo que já foi encontrado antes.

Exemplo

Para o número complexo, definimos um novo tipo, chamado complexo. Ao invés de:

```
struct TipoComplexto {
     double real;
     double imaginario;
};
typedef struct TipoComplexo complexo;
```

podemos escrever uma única declaração:

```
typedef struct {
     double real;
     double imaginario;
} complexo;
```

Agora podemos declarar variáveis:

```
 complexo numero;
```

Enumerações

As enumerações são um tipo de dados através do qual se pode representar um único valor de um pequeno conjunto discreto e finito de alternativas.

Por que usar enumerações?

Por exemplo, imaginemos um programa simples para controlar o recebimento em um caixa de restaurante. A forma de pagamento pode ser: *dinheiro*, *cheque* e *vale refeição*. Como representar a forma de pagamento em uma variável? Poderíamos:

Opção 1: Declarar a variável forma_pagamento como *char* e impor a seguinte convenção: armazenaremos o valor 'd' para dinheiro, 'c' para cheque e 'v' para vale refeição.

Opção 2: Declarar variável `forma_pagamento` como *`int`* e definir uma convenção semelhante à opção anterior: 1 para dinheiro, 2 para cheque e 3 para vale refeição.

Opção 3: Declarar três variáveis de tipo inteiro: `dinheiro`, `cheque` e `vale_refericao`. Estabelecemos que todas elas contêm o valor zero, exceto aquela correspondente à forma de pagamento, que conterá 1.

Um código típico para a opção 1 seria:

```
char forma_pagamento;
switch (forma_pagamento) {
     case 'd':
          // código para forma de pagamento em dinheiro
          // ...
          break;
     case 'c':
          // código para forma de pagamento em cheque
          // ...
          break;
     case 'v':
          // código para forma de pagamento em vale
          // ...
          break;
}
```

As três opções resolvem nosso problema inicial, mas todas elas contêm alguns vícios que gostaríamos de evitar.

Definição do significado de valores: Na opção 1, como nosso programa se comportaria se `forma_pagamento` contém o valor 'D' (em maiúsculo)? Será que devemos interpretar como dinheiro? Note que isto não está explícito na convenção adotada.

Manutenção do significado dos valores: O fato de existir uma convenção para se interpretar o valor das variáveis já é uma dificuldade em si. É necessário documentá-la e controlar seu uso. Mesmo os melhores programadores podem esquecer uma regra ou confundir algum valor, principalmente se o programa possuir várias convenções. Existe aqui um potencial para se introduzir erros extremamente difíceis de serem localizados.

Estender o conjunto de significados: Para o pagamento por cartão de débito, qual valor adicionar na opção 1? Existem duas alternativas óbvias: '`c`' (cartão) ou '`d`' (débito). Mas ambos valores já foram atribuídos para cheque e dinheiro. Com um pouco de imaginação, seria possível definir '`k`' ou '`a`' para cartão (*k*artão ou c*a*rtão), mas o código só ficará ainda mais confuso.

Alteração consistente de valores: Caso seja necessário alterar o valor associado com dinheiro de '`d`' para '`$`', como garantir que todos as linhas de código sejam atualizadas, sem esquecer nenhuma?

Enumerações

Quando precisamos representar uma opção dentro de um pequeno conjunto de possibilidades finito e pré-determinado, a solução na linguagem C é usar uma **enumeração**. O valor de uma variável enumerada será sempre uma das opções dadas. A enumeração

permite ao programador associar um identificador para cada opção do conjunto de possibilidades. O identificador deve ser uma palavra que descreve o siginificado da opção.

Por exemplo, ao invés de usar códigos '`c`', '`d`' e '`v`' para representar "cartão de crédito", "dinheiro" e "vale refeição", poderíamos utilizar identificadores como `cartao_credito`, `dinheiro` e `vale_refeicao`. O código fonte ficará muito mais claro e intuitivo.

Se nada dito em contrário, o compilador C atribui o valor 0 à primeira opção de uma enumeração, 1 para a segunda, e assim por diante. Mas o programador não precisa mais se preocupar com isto. Basta utilizar os identificadores que ele escolheu para representar as opções que o compilador C se encarregará de realizar automaticamente a tradução para número e de manter a coerência em todo o programa.

Isto resolve o problema da alteração consistente de valores e da extensão e manutenção do conjunto de significados discutidos no tópico anterior.

Declaração de enumerações

As declarações de enumeração seguem a sintaxe a seguir:

```
enum nome_enumeracao {
   opcao1,
   opcao2,
   ...
} variavel;

enum nome_enumeracao {
   opcao1,
   opcao2,
   ...
} variavel = opcao;
```

A palavra **`enum`** indica que a declaração será uma enumeração. Seguem o nome da enumeração (opcional), indicando que o compilador deve lembrar esta definição para poder utilizá-la em declarações abreviadas em outras partes do programa. Neste caso, a definição também *cria um novo tipo de dados*, que passa a existir junto com os tipos de dados nativos da linguagem C. Entre chaves `{ }`, está uma lista de identificadores descrevendo as opções que a enumeração poderá aceitar. Após as chaves, pode-se declarar (opcionalmente) variáveis que armazenam uma das opções definidas nesta enumeração.

Observe que a declaração de uma variável de tipo enumeração segue a mesma lógica da declaração de variáveis padrões:

```
enum opcoes variável;
```

onde `opcoes` é uma enumeração, na forma:

```
enum opcoes {opcao1, opcao2, ....};
```

Um programa pode declarar várias variáveis de enumerações diferentes.

Regras de uso

Ao criar uma variável contendo a forma de pagamento:

```
enum {dinheiro, cheque, vale_refeicao, cartao} forma_pagamento;
```

O compilador automaticamente convencionará o valor 0 para `dinheiro`, 1 para `cheque`, 2 para `vale_refeicao` e 3 para `cartao`. Outros valores que não sejam um destes quatro serão tratados como erros de compilação.

O código abaixo,

```
forma_pagamento = cheque;
```

será equivalente à

```
forma_pagamento = 1;
```

Note como a primeira forma é muito mais elegante, inteligível e robusta para se programar!

É comum adicionar no início da enumeração um valor correspondente a *não-definido*, como um valor padrão da enumeração. Por ser o primeiro da lista, ele também será associado com o número zero, que também será interpretado como falso se usado como uma condição.

```
enum {desconhecido, dinheiro, cheque, vale_refeicao, cartao} forma_pagamento
= desconhecido;
```

Um código para verificar a forma de pagamento ficará semelhante ao ilustrado abaixo:

```
switch (forma_pagamento) {
      case dinheiro:
            // ...
            break;
      case cheque:
            // ...
            break;
      case vale_refeicao:
            // ...
            break;
      case cartao:
            // ...
            break;
      case desconhecido:
            // ...
            break;
}
```

Note como o **switch** é a estrutura condicional ideal para trabalhar com enumerações!

Enumerações indexadas

Em algumas situações, podemos desejar controlar manualmente o processo de indexação.

```
enum {opcao1 = indice1, opcao2 = indice2, ....} variavel;
```

Se apenas algumas das opções forem indexadas manualmente, o compilador indexará as demais com valores a partir de 0.

Observações:

- O índice pode ser qualquer valor inteiro, inclusive valor negativo.
- Duas opções podem receber o mesmo índice. Neste caso, todos os identificadores associados com o mesmo índice representam a mesma opção.

Exemplo:

Vamos definir o valor de cada uma das formas de pagamento segundo nossa própria convenção:

```
enum {dinheiro = 'd', cheque = 'c', vale_refeicao = 'v', cartao = 'a'}
forma_pagamento;
```

Aplicações de Tipos Enumerados

Máquinas de Estado

Programas com finalidade de controlar algum tipo de equipamento costumam representar as possíveis formas de operação como um "estado". Um exemplo simples é o semáforo, que pode apresentar três estados: vermelho, amarelo e verde. A forma ideal de representar o estado do semáforo é justamente o tipo enumerado.

```
enum {vermelho, amarelo, verde} estado_semaforo;
```

Uma porta automática, controlada por um programa, pode estar em um de quatros estados: `fechada, abrindo, aberta, fechando`. Poderíamos escrever:

```
enum {fechada, abrindo, aberta, fechando} estado_porta;
```

Lista de opções

As enumerações são ideais para se declarar variáveis (ou atributos de estruturas) que representam uma escolha dentro de um pequeno conjunto de opções.

Dados armazenados em uniões

Quando usamos uniões, não é possível consultar qual valor está realmente sendo armazenado. Enumerações são uma boa alternativa para identificar o conteúdo de uma união.

```
struct TipoPessoa {
      ...
      enum {valorAluno, valorFuncionario, valorProfessor} valor_especifico;
      union {
            struct {
                  ...
            } aluno;
            struct {
                  ...
            } funcionario;
            struct {
                  ...
            } professor;
      } dados_especificos;
} pessoa;
```

Neste caso, a enumeração declara um atributo que guarda o tipo do valor que está armazenado na união. Por exemplo, se `pessoa.valor_especifico == valorAluno`, então sabemos que a variável `pessoa` armazena dados de um aluno e devemos acessar somente `pessoa.dados_especificos.aluno` e não `pessoa.dados_especificos.professor` ou `pessoa.dados_especificos.funcionario`.