Tipos Enumerados e Registros

Introdução

Os tipos enumerados e registros são estruturas fundamentais na programação que permitem aos desenvolvedores criar tipos de dados personalizados para representar informações de maneira mais organizada e intuitiva. Enquanto os tipos enumerados permitem definir um conjunto finito de valores nomeados, os registros (ou structs) possibilitam agrupar diferentes tipos de dados relacionados em uma única unidade lógica.

Estas estruturas são essenciais para o desenvolvimento de programas bem estruturados, pois facilitam a modelagem de entidades do mundo real e tornam o código mais legível e manutenível. Neste material,

exploraremos os conceitos, sintaxes e aplicações dos tipos enumerados e registros, com foco especial na linguagem C, embora os conceitos sejam aplicáveis a diversas linguagens de programação.

Tipos Enumerados

Definição e Conceitos Básicos

Um tipo enumerado (enum) é um tipo de dado definido pelo usuário que consiste em um conjunto de constantes nomeadas, chamadas enumeradores. Os tipos enumerados são utilizados para representar um conjunto finito de valores discretos, como dias da semana, meses do ano, estados de um processo, entre outros.

Em linguagens como C, os enumeradores são, na realidade, constantes inteiras. O compilador associa automaticamente valores inteiros sequenciais aos enumeradores, começando por 0 para o primeiro enumerador, 1 para o segundo, e assim por diante, a menos que valores específicos sejam atribuídos explicitamente.

A sintaxe básica para definir um tipo enumerado em C é:

```
enum nome_do_tipo {
    identificador1,
    identificador2,
    ...
    identificadorN
};
```

Declaração e Uso de Variáveis Enumeradas

Para declarar uma variável de um tipo enumerado:

```
enum dias_semana amanha = TERCA;
```

Também é possível usar typedef para criar um nome de tipo mais conciso:

```
typedef enum {
    DOMINGO,
    SEGUNDA,
    …
} DiaSemana;
DiaSemana hoje = SEGUNDA;
```

Operações com Tipos Enumerados

Os tipos enumerados em C são tratados como inteiros, o que permite realizar operações aritméticas e comparações:

```
enum dias_semana hoje = SEGUNDA;
enum dias_semana amanha = hoje + 1;  // amanha = TERCA
if (hoje < amanha) {
    printf("Hoje vem antes de amanha
");
}
// Iterando pelos dias da semana
for (enum dias_semana dia = DOMINGO; dia <= SABADO; dia++) {
    // Fazer algo com cada dia
}
```

No entanto, é importante ter cuidado com essas operações, pois elas podem resultar em valores que não correspondem a nenhum enumerador válido.

Vantagens dos Tipos Enumerados

1. Legibilidade: Tornam o código mais legível ao usar nomes significativos em vez de números mágicos.
2. Segurança de tipo: Ajudam a prevenir erros ao restringir os valores possíveis.
3. Documentação: Servem como documentação do código, tornando explícito o conjunto de valores válidos.
4. Manutenibilidade: Facilitam a manutenção, pois alterações nos valores são centralizadas.

Limitações dos Tipos Enumerados em C

1. Não são tipos fortemente tipados: O compilador C permite atribuir qualquer valor inteiro a uma variável enum.
2. Não possuem namespace: Os nomes dos enumeradores são globais ao escopo em que são definidos.
3. Não suportam métodos ou propriedades como em linguagens orientadas a objetos.

Registros (Structs)

Definição e Conceitos Básicos

Um registro (struct) é um tipo de dado composto que agrupa variáveis de diferentes tipos sob um único nome. Cada variável dentro de um registro é chamada de campo ou membro. Os registros são utilizados para representar entidades complexas do mundo real, como pessoas, produtos, contas bancárias, entre outros.
Diferentemente dos arrays, que armazenam elementos do mesmo tipo, os registros podem conter elementos de tipos diferentes, o que os torna ideais para modelar objetos com múltiplas características.
A sintaxe básica para definir um registro em C é:

```
struct nome_do_tipo {
    tipo1 campo1;
    tipo2 campo2;
    ...
```

Exemplo:

```
struct pessoa {
    char nome[50];
```

```
    tipoN campoN;
};
```

```
    int idade;
    float altura;
    float peso;
};
```

Assim como com enums, é comum usar typedef para criar um nome de tipo mais conciso:

```
typedef struct {
    char nome[50];
    int idade;
    float altura;
    float peso;
} Pessoa;
```

Declaração e Inicialização de Registros

Para declarar uma variável de um tipo registro:

```
struct pessoa p1;
// Ou usando typedef
Pessoa p2;
```

Inicialização de registros:

```
// Inicialização por membros
struct pessoa p1 = {"João Silva", 30, 1.75, 70.5};

// Inicialização por designadores (C++ em diante)
struct pessoa p2 = {
    .nome = "Maria Santos",
    .idade = 25,
    .altura = 1.65,
    .peso = 60.0
};
```

```
// Inicialização membro a membro
struct pessoa p3;
strcpy(p3.nome, "Pedro Oliveira");
p3.idade = 40;
p3.altura = 1.80;
p3.peso = 85.0;
```

Acesso aos Campos de um Registro

Para acessar os campos de um registro, utiliza-se o operador ponto (.):

```
struct pessoa p1;
strcpy(p1.nome, "João Silva");
p1.idade = 30;
```

```
p1.altura = 1.75;
p1.peso = 70.5;
printf("Nome: %s", p1.nome);
printf("Idade: %d anos", p1.idade);
printf("Altura: %.2f m", p1.altura);
printf("Peso: %.1f kg", p1.peso);
```

Quando se trabalha com ponteiros para registros, utiliza-se o operador seta (->):

```
struct pessoa *ptr = &p1;
printf("Nome: %s", ptr->nome);  // Equivalente a (*ptr).nome
printf("Idade: %d anos", ptr->idade);
```

Arrays de Registros

É comum utilizar arrays de registros para armazenar coleções de objetos do mesmo tipo:

```
struct pessoa equipe[10];  // Array de 10 pessoas
for (int i = 0; i < 10; i++) {// Iterando pelo array
    printf("Pessoa %d: %s, %d anos", i+1, equipe[i].nome, equipe[i].idade);
}
```

Registros Aninhados

Registros podem conter outros registros como membros, permitindo criar estruturas de dados mais complexas:

```
struct endereco {
    char rua[50];
    int numero;
    char cidade[30];
    char estado[3];
    char cep[10];
};
struct pessoa {
    char nome[50];
    int idade;
    struct endereco residencia;  // Registro aninhado
};
// Acesso a campos de registros aninhados
struct pessoa p;
strcpy(p.residencia.cidade, "São Paulo");
p.residencia.numero = 123;
```

Passagem de Registros para Funções

Registros podem ser passados para funções por valor ou por referência:

1. Passagem por valor:

```
void imprime_pessoa(struct pessoa p) {
    printf("Nome: %s, Idade: %d", p.nome, p.idade);
}
// Chamada
imprime_pessoa(p1);
```

2. Passagem por referência (usando ponteiros):

```
void atualiza_idade(struct pessoa *p, int nova_idade) {
    p->idade = nova_idade;
}
// Chamada
atualiza_idade(&p1, 31);
```

A passagem por referência é geralmente mais eficiente para registros grandes, pois evita a cópia de todos os dados.

Retorno de Registros por Funções

Funções também podem retornar registros:

```
struct pessoa cria_pessoa(char *nome, int idade, float altura, float peso) {
    struct pessoa p;
    strcpy(p.nome, nome);
    p.idade = idade;
    p.altura = altura;
    p.peso = peso;
    return p;
}
// Uso
struct pessoa p1 = cria_pessoa("João Silva", 30, 1.75, 70.5);
```

Comparação entre Registros

Em C, não é possível comparar registros diretamente usando operadores como == ou !=. É necessário comparar cada campo individualmente:

- strcmp(p1.nome, p2.nome) == 0 && p1.idade == p2.idade;

Atribuição entre Registros

A atribuição entre registros do mesmo tipo é permitida e copia todos os campos:

```
struct pessoa p1 = {"João Silva", 30, 1.75, 70.5};
struct pessoa p2;
p2 = p1;  // Todos os campos de p1 são copiados para p2
```

Tamanho de Registros

O tamanho de um registro não é necessariamente a soma dos tamanhos de seus campos devido ao alinhamento de memória:

```
struct exemplo {
    char c;     // 1 byte
    int i;      // 4 bytes
    double d;   // 8 bytes
};
// sizeof(struct exemplo) pode ser maior que 13 bytes
```

O compilador pode inserir bytes de preenchimento (padding) entre os campos para garantir o alinhamento adequado, o que pode resultar em um tamanho total maior que a soma dos tamanhos individuais.

Usos Avançados de Registros

1. Uniões dentro de registros:

```
struct valor {
    enum { INTEIRO, REAL } tipo;
    union {
        int i;
        float f;
    } dados;
};
```

2. Campos de bits:
    - unsigned int flag1 : 1; // Usa apenas 1 bit
3. Registros auto-referenciados (para estruturas de dados como listas ligadas):
    - struct no *proximo; // Ponteiro para outro nó

Combinando Tipos Enumerados e Registros

Os tipos enumerados e registros podem ser combinados de maneira eficaz para criar estruturas de dados mais expressivas e seguro, aproveitando as vantagens de ambas as estruturas.

Diferenças entre Registros em Diversas Linguagens

Embora o conceito de registros seja similar em diferentes linguagens de programação, existem algumas diferenças importantes:

1. C: Usa a palavra-chave struct, sem encapsulamento ou métodos.
2. C++: Estende o conceito de struct para incluir métodos, construtores e herança.
3. Pascal: Usa a palavra-chave record, com sintaxe ligeiramente diferente.
4. Java: Não tem registros tradicionais, mas introduziu o conceito de "record" no Java 16 como um tipo de

classe imutável.
5. Python: Usa classes ou namedtuples, e a partir do Python 3.7, dataclasses.

Boas Práticas no Uso de Tipos Enumerados e Registros

1. Nomeação clara e significativa para tipos, enumeradores e campos.
2. Documentação adequada, especialmente para registros complexos.
3. Organização lógica dos campos em registros, agrupando campos relacionados.
4. Uso de typedef para criar nomes de tipos mais concisos.
5. Validação de valores ao trabalhar com enumerados, especialmente quando os valores vêm de fontes externas.
6. Criação de funções auxiliares para operações comuns com registros (inicialização, cópia, comparação).
7. Consideração do alinhamento de memória ao definir a ordem dos campos em registros para otimizar o uso de memória.

Aplicações Práticas

Tipos Enumerados e Registros são amplamente utilizados em diversas áreas da programação:
1. Modelagem de dados: Representação de entidades do mundo real (pessoas, produtos, transações).
2. Interfaces gráficas: Definição de cores, estilos, estados de componentes.
3. Protocolos de comunicação: Definição de tipos de mensagens, códigos de erro.
4. Compiladores e interpretadores: Representação de tokens, nós de árvores sintáticas.
5. Jogos: Estados de jogo, tipos de personagens, itens.
6. Sistemas embarcados: Configuração de hardware, estados de máquinas de estado.

Conclusão

Os tipos enumerados e registros são ferramentas fundamentais na programação estruturada, permitindo a criação de tipos de dados personalizados que melhor representam os conceitos do domínio do problema. Enquanto os tipos enumerados oferecem uma maneira elegante de definir conjuntos de constantes nomeadas, os registros permitem agrupar dados relacionados em uma única unidade lógica.
O uso adequado dessas estruturas resulta em código mais legível, manutenível e menos propenso a erros. Além disso, elas servem como base para conceitos mais avançados em linguagens orientadas a objetos, como classes e objetos.
Ao dominar os tipos enumerados e registros, os programadores adquirem ferramentas poderosas para modelar dados de forma eficiente e expressiva, contribuindo para o desenvolvimento de software de alta qualidade.

Referências

- 📖 Livros e Apostilas
    - KERNIGHAN, B. W.; RITCHIE, D. M. C: A Linguagem de Programação Padrão ANSI. Campus.

    - DEITEL, H. M.; DEITEL, P. J. C: Como Programar. Pearson.
    - SCHILDT, H. C Completo e Total. Makron Books.
    - BACKES, A. Linguagem C: Completa e Descomplicada. Elsevier.
- 🌐 Recursos Online
    - IC-Unicamp - Tipos Enumerados e Registros - https://www.ic.unicamp.br/~zanoni/teaching/mc102/2013-2s/aulas/aula17.pdf
    - PUCRS - Programação C - Structs - https://www.inf.pucrs.br/~pinho/LaproI/Structs/Structs.htm
    - Embarcados - Struct - Registros em Linguagem C - https://embarcados.com.br/struct-registros-em-linguagem-c/
    - IC-Unicamp - Aula 21 - Registros - https://ic.unicamp.br/~ducatte/mc102/aula21.comrespostas.pdf
- 📽️ Vídeos e Cursos
    - Registros, Tipos e Enumerados em C - Aula 16 - https://www.youtube.com/watch?v=VVLRSRMXxmk

Anexo - Resumo Estruturado/Mapa Mental Gerado por IA

1. Introdução

- Estruturas fundamentais para criar **tipos personalizados**
- Representam informações de forma **organizada e intuitiva**
- Auxiliam na **modelagem de entidades do mundo real**

---

2. Tipos Enumerados (enum)

- **Definição**: conjunto finito de valores nomeados (constantes inteiras).
- **Exemplos**: dias da semana, meses, estados de um processo.

Declaração:
enum dias_semana { DOMINGO, SEGUNDA, ... SABADO };

- **Uso de typedef** para nomes mais curtos.

Operações

- Podem ser tratados como inteiros (soma, comparação, iteração).
- Atenção a valores inválidos fora do conjunto.

Vantagens

1. Maior **legibilidade** do código
2. **Segurança de tipo** (restrição de valores)
3. Servem como **documentação implícita**
4. Melhor **manutenção**

Limitações em C

- Não são fortemente tipados
- Enumeradores não possuem namespace próprio
- Não têm métodos ou propriedades

---

3. Registros (structs)

- **Definição**: agrupam variáveis de diferentes tipos sob um único nome.

Exemplo:
struct pessoa { char nome[50]; int idade; float altura; };

- **Declaração e inicialização**: por membros, designadores ou manual.
- **Acesso aos campos**: operador . (ponto) ou -> (ponteiros).

Extensões

- **Arrays de registros** (coleções de entidades)
- **Registros aninhados** (ex.: endereço dentro de pessoa)
- **Passagem a funções**: por valor ou referência
- **Retorno de registros** por funções
- **Atribuição entre registros**: copia todos os campos
- **Comparação**: precisa ser campo a campo
- **Tamanho**: pode variar devido a **padding/alinhamento**

Usos avançados

- Uniões dentro de registros
- Campos de bits
- Estruturas auto-referenciadas (listas ligadas, árvores, etc.)

---

4. Combinação Enum + Struct

- Permite **estruturas mais expressivas e seguras**
- Exemplo: um campo enum para indicar tipo e um struct para armazenar dados

---

5. Diferenças entre Linguagens

- **C**: struct simples, sem métodos
- **C++**: structs com métodos, herança, construtores
- **Pascal**: usa record
- **Java**: introduziu record (imutável) no Java 16
- **Python**: usa namedtuple, dataclass ou classes

---

6. Boas Práticas

- Nomeação clara e significativa
- Documentação dos tipos e campos
- Agrupamento lógico de campos relacionados
- Uso de typedef para clareza
- Validação de valores de enums (especialmente vindos de fora)
- Funções auxiliares para inicializar/copiar registros

- Considerar alinhamento de memória ao definir campos

---

7. Aplicações Práticas

- **Modelagem de dados** (pessoas, produtos, transações)
- **Interfaces gráficas** (cores, estilos, estados)
- **Protocolos de comunicação** (códigos de erro, tipos de mensagens)
- **Compiladores** (tokens, árvores sintáticas)
- **Jogos** (tipos de personagens, estados do jogo)
- **Sistemas embarcados** (configurações de hardware, máquinas de estado)