Programação de Computadores

ARMAZENAMENTO E REFERÊNCIAS

Judson Santos Santiago

Introdução

- Programas são compostos por vários arquivos
  - Que podem conter declarações e definições:
    - Variáveis
    - Constantes

O local em que os dados são declarados influencia em como e onde eles podem ser usados

```
// ginastica.h
void flexao(int);
void abdominal(int);
```

```
// ginastica.cpp
void flexao(int n)
{
   ...
}

void abdominal(int n)
{
   ...
}
```

```
// malhando.cpp
#include <iostream>
#include "ginastica.h"

int main()
{
   cout << "Exercícios "
        << "de hoje:"
        << endl;

   flexao(10);
   abdominal(20);

   return 0;
}
```

Introdução

- Dependendo de como e onde as variáveis e constantes são declaradas e definidas, muda:
  - A sua visibilidade dentro do código
    - Escopo
    - Ligação
  - O seu tempo de vida
    - Categoria de armazenamento
- Esses 3 conceitos impactam o uso dos dados

Armazenamento

- A categoria de armazenamento afeta

  "Quanto tempo o dado é mantido na memória"

  - C++ possui 4 tipos de armazenamento
    - Automático: pela duração da função
    - Estático: pela duração do programa
    - Thread: pela duração da thread
    - Dinâmico: por uma duração controlada pelo programador

Escopo e Ligação

- As categorias de armazenamento se relacionam com dois outros importantes conceitos de programação
  - Escopo
    - Descreve a visibilidade de um nome dentro de um arquivo
      - Uma variável definida em um escopo local é visível apenas dentro de um bloco, enquanto em um escopo global é visível por todo o arquivo
  - Ligação
    - Descreve como um nome pode ser compartilhado entre arquivos
      - Um nome com ligação externa pode ser compartilhado entre vários arquivos, enquanto um com ligação interna funciona apenas em um arquivo

Armazenamento Automático

- O armazenamento automático é usado para:
  - Parâmetros de funções e variáveis locais
    - Escopo local
    - Sem ligação

alocação das variáveis taxa_cambio e dolar acontece na entrada da função

```
float converte(float dolar)
{
    cout << "Digite a taxa de cambio:";
    float taxa_cambio;
    cin >> taxa_cambio;
    return taxa_cambio * dolar;
}
```

escopo da variável taxa_cambio

escopo da variável dolar

Armazenamento Estático

- O armazenamento estático é usado para:
  - Variáveis globais
    - Escopo: global
    - Ligação: externa ou interna

```
#include <iostream>
using namespace std;

int tamanho;                // estática com ligação externa
static int indice;          // estática com ligação interna

int main()
{
    ...
}
```

Uma variável estática não inicializada tem todos os seus bits ajustados para 0 (zero).

Armazenamento Estático

- Uma variável com ligação externa pode ser usada em outros arquivos através da declaração extern

```
// principal.cpp

int tamanho = 1000;


int main()
{
}

void exibir(int n)
{
}
```

```
// auxiliar.cpp

extern int tamanho;


int calcular(int n)
{
}

int ler()
{
}
```

A definição da variável só pode ocorrer uma vez, mas uma declaração extern pode ser feita para cada arquivo que pretende usar essa variável.

Armazenamento Estático

- Uma variável global declarada com static possui ligação interna, ou seja, seu escopo é limitado ao arquivo

```
// principal.cpp
int tamanho = 1000;

static int indice = 5;

int main()
{
}

void exibir(int n)
{
}
```

```
// auxiliar.cpp
extern int tamanho;

static int indice = 10;

int calcular(int n)
{
}

int ler()
{
}
```

Cada arquivo possui uma variável "indice" diferente e não há choque de nomes porque o escopo é limitado ao arquivo.

Armazenamento Estático

- O armazenamento estático é usado também para:
  - Variáveis locais estáticas
    - Escopo: local
    - Ligação: sem ligação

```
int tamanho = 1000;          // estática com ligação externa
static int indice = 5;       // estática com ligação interna

void processar()
{
    static int cont;         // estática sem ligação (inicializada para 0)

    int temp = 0;            // automática sem ligação
}
```

Armazenamento Estático

- Uma variável local estática preserva seu conteúdo entre chamadas de funções
  - A inicialização acontece apenas uma vez

```
int main()
{
   for (int i = 0; i < 5; ++i)
      exibir();
}

void exibir()
{
    static int cont = 1;
    cout << cont++ << endl;
}
```

Armazenamento Thread

- O armazenamento thread é obtido com o uso da palavra-chave thread_local*
  - Voltado para programação usando concorrência
    - Multicore
    - Multiprocessor
    - Multithread
  - A variável persiste enquanto a thread em que ela foi declarada existir

* Não abordaremos programação de threads na disciplina

Armazenamento Dinâmico

- O armazenamento dinâmico é obtido através do uso dos operadores new e delete

```
int main()
{
    imagem * p = criar();

    ...

    usar(p);

    ...

    destruir(p);
}
```

```
imagem * criar()
{
    imagem * img = new imagem;

    return img;
}

void destruir(imagem * pt)
{
    delete pt;
}
```

O tempo de vida da memória alocada não está atrelado às funções

Referências

- Uma referência é um nome que atua como um apelido para uma variável previamente definida

```
int rato;
int & roedor = rato;  // roedor é um apelido para rato
```

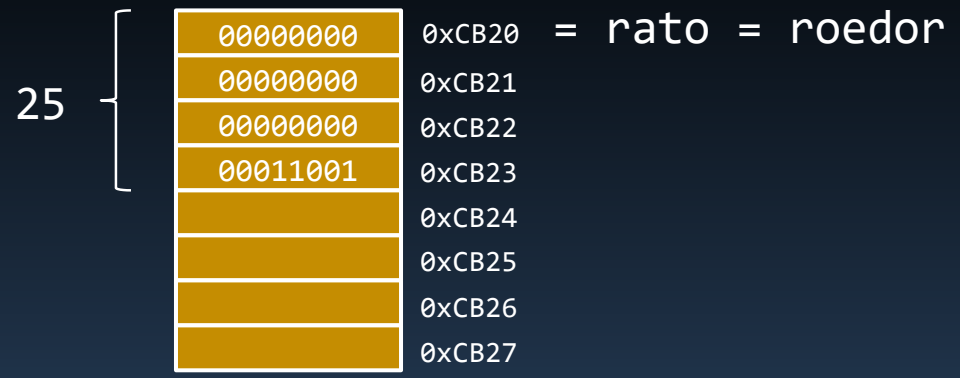
  - O símbolo & é usado para declarar uma referência
    - Neste contexto, & não é o operador de endereço

```
int * pt = &rato;  // ponteiro para int
int & rf = rato;   // referência para int
```

Referências

- A referência permite usar ambos os nomes para acessar o mesmo valor e a mesma posição de memória

```
int rato = 25;
int & roedor = rato;  // roedor é um apelido para rato
```



Referências

- O principal uso de referências é como parâmetro de funções
  - Ela representa uma alternativa ao uso de ponteiros
  - A função trabalha com o dado original

```
// protótipo da função pragas
void pragas(int & roedor)
{
   ...
}

int main()
{
   int rato = 25;
   pragas(rato);
   ...
}
```

Referências

```cpp
#include <iostream>
using namespace std;

int main()
{
     int ratos = 101;
     int & roedores = ratos;  // roedores é uma referência
     cout << "ratos = " << ratos;
     cout << ", roedores = " << roedores << endl;

     roedores++;

     cout << "ratos = " << ratos;
     cout << ", roedores = " << roedores << endl;
     cout << "endereço de ratos    = " << &ratos    << endl;
     cout << "endereço de roedores = " << &roedores << endl;
}
```

// Roedores.cpp

Referências

- Saída do Programa:

```
ratos = 101, roedores = 101
ratos = 102, roedores = 102
endereço de ratos    = 0x0065fd48
endereço de roedores = 0x0065fd48
```

- Observe a diferença entre o uso de & como referência e como operador de endereço

```
int & roedores = ratos;  // roedores é uma referência

cout << "endereço de roedores = " << &roedores << endl;
```

Referências e Ponteiros

- Uma referência parece com um ponteiro
  - Ambos permitem acessar e modificar dados "apontados"

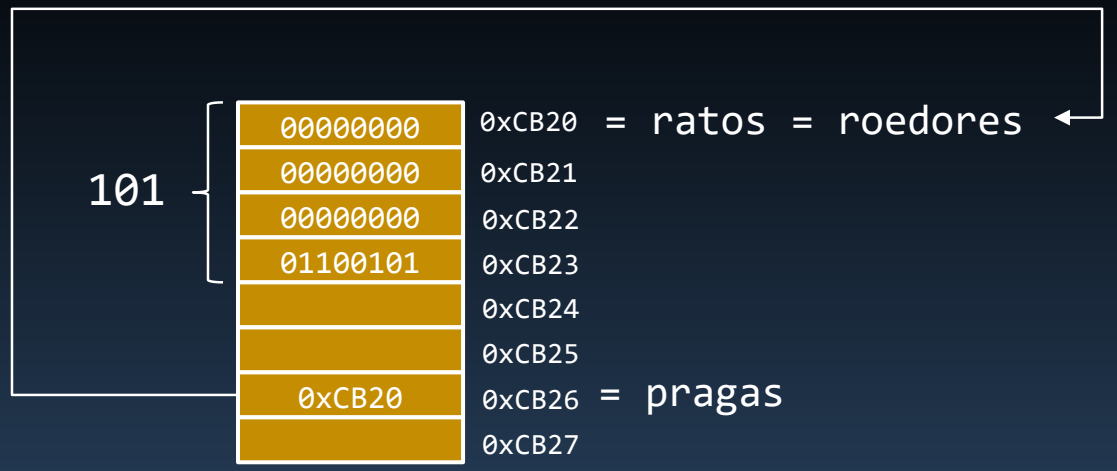
```
int ratos = 101;
int & roedores = ratos;    // roedores é uma referência
int * pragas   = &ratos;   // pragas é um ponteiro
```

  - O valor 101 pode ser acessado usando ratos, roedores ou *pragas
  - O endereço de 101 pode ser obtido com &ratos, &roedores ou pragas

Referências e Ponteiros

- Mas internamente a linguagem C++ trata referências e ponteiros de forma diferente

```
int ratos = 101;
int & roedores = ratos;    // roedores é uma referência
int * pragas   = &ratos;   // pragas é um ponteiro
```



Referências e Ponteiros

- Existem também diferenças no uso:
  - Uma referência deve ser sempre inicializada

```
int ratos = 101;
int & roedores;    x  // roedores é uma referência
roedores = ratos;  x  // tarde demais
```

  - Ponteiros podem receber valores a qualquer momento

```
int ratos = 101;
int * pragas;          // pragas é um ponteiro
pragas = &ratos;       // ok
```

Referências e Ponteiros

```cpp
#include <iostream>
using namespace std;

int main()
{
     int ratos = 101;
     int & roedores = ratos;  // roedores é uma referência
     cout << "ratos = " << ratos;
     cout << ", roedores = " << roedores << endl;
     cout << "endereço de ratos    = " << &ratos << endl;
     cout << "endereço de roedores = " << &roedores << endl;

     int coelhos = 50;
     roedores = coelhos;   // roedores é agora uma referência para coelhos?

     cout << "coelhos = " << coelhos;
     cout << ", ratos = " << ratos;
     cout << ", roedores = " << roedores << endl;
     cout << "endereço de coelhos  = " << &coelhos << endl;
     cout << "endereço de roedores = " << &roedores << endl;
}
```

// Coelhos.cpp

Referências e Ponteiros

- Saída do Programa:

```
ratos = 101, roedores = 101
endereço de ratos    = 0x0065fd44
endereço de roedores = 0x0065fd44

coelhos = 50, ratos = 50, roedores = 50
endereço de coelhos  = 0x0065fd48
endereço de roedores = 0x0065fd44
```

- A instrução causou uma atribuição de valor
  - Não é possível redefinir uma referência

```
int coelhos = 50;
roedores = coelhos;   // podemos mudar a referência? NÃO
```

Usos de Referências

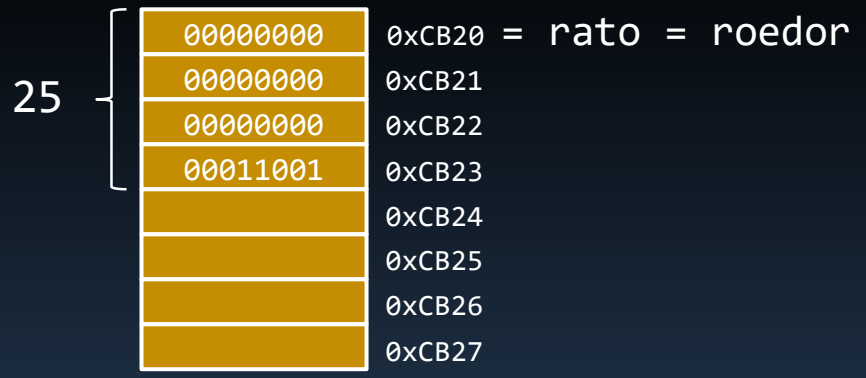
- As principais aplicações das referências são:
  - Em parâmetros de função
    - Evita cópia dos argumentos da função
  - Com registros e objetos
    - Normalmente armazenam grandes quantidades de informação
    - Evita cópia dos registros dentro do programa
    - Especialmente em chamadas de funções

Referências e Funções

- Referências como parâmetros de funções
  - Cria-se um apelido para uma variável da função chamadora
  - Isto se chama passagem por referência

```
void pragas(int & roedor)
{
    ...
}

int main()
{
    int rato = 25;
    pragas(rato);
    ...
}
```



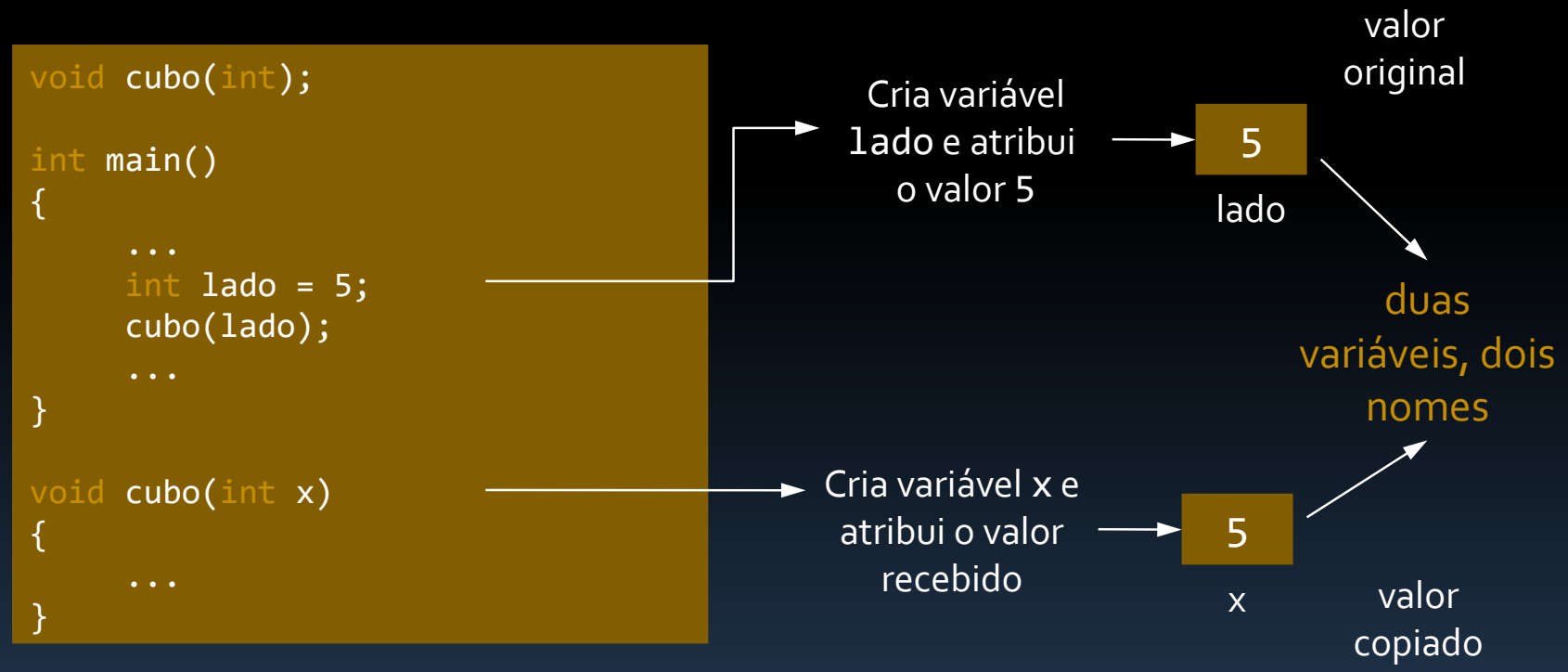
Referências e Funções

- Passagem de argumentos por valor

```
void cubo(int);

int main()
{
     ...
     int lado = 5;
     cubo(lado);
     ...
}

void cubo(int x)
{
     ...
}
```



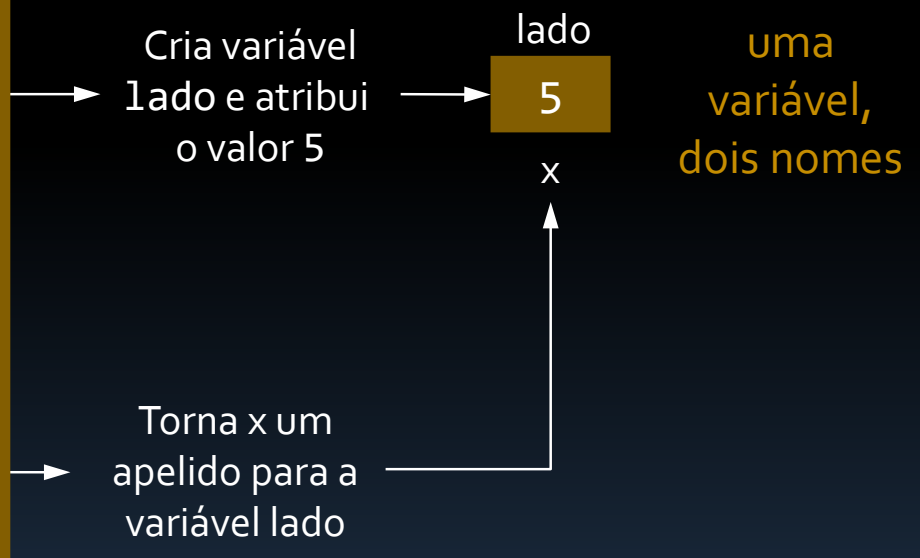
Referências e Funções

- Passagem de argumentos por referência

```
void cubo(int &);

int main()
{
      ...
      int lado = 5;
      cubo(lado);
      ...
}

void cubo(int & x)
{
      ...
}
```



Referências com Registros

- Referências foram inicialmente criadas para trabalhar com registros e objetos

```
struct atleta
{
   int   acertos;
   int   tentativas;
   float percentual;
};

atleta rick = { 13, 14 };
atleta john = { 10, 16 };

void calcular(atleta & atl);
void exibir(const atleta & atl);
```

Registro atleta

Variáveis do tipo atleta

Funções usando referências

Referências com Registros

```
#include <iostream>
using namespace std;

struct atleta
{
   int   acertos;
   int   tentativas;
   float percentual;
};

void calcular(atleta & atl);
void exibir(const atleta & atl);
atleta & acumular(atleta & soma, const atleta & atl);

int main()
{
   atleta rick = { 13, 14 };
   atleta john = { 10, 16 };
   atleta mark = { 7, 9 };
   atleta time = { 0, 0 };
   ...
```

// Atleta.cpp

Referências com Registros

```
    ...

    calcular(rick);                         // rick é um atleta
    cout << "mostrar Rick:\n" ;
    exibir(rick);

    acumular(time, rick);                   // não usa o retorno
    cout << "mostrar time:\n";
    exibir(time);

    cout << "John no time:\n";
    exibir(acumular(time, john));           // usa retorno como argumento

    atleta todos = acumular(time, mark);    // usa retorno em atribuição
    cout << "mostrar todos:\n";
    exibir(todos);
    cout << "mostrar time:\n";
    exibir(time);

    return 0;
}
```

// Atleta.cpp

Referências com Registros

```
void calcular(atleta & atl)
{
   if (atl.tentativas != 0)
      atl.percentual = 100.0f * float(atl.acertos) / float(atl.tentativas);
   else
      atl.percentual = 0;
}

void exibir(const atleta & atl)
{
   cout << "Acertos: "     << atl.acertos    << "  ";
   cout << " Tentativas: " << atl.tentativas << "  ";
   cout << " Percentual: " << atl.percentual << "\n\n";
}

atleta & acumular(atleta & soma, const atleta & atl)
{
   soma.tentativas += atl.tentativas;
   soma.acertos += atl.acertos;
   calcular(soma);
   return soma;
}
```

// Atleta.cpp

Referências com Registros

- Saída do Programa:

```
mostrar rick:
    Acertos: 13      Tentativas: 14     Percentual: 92.8571

mostrar time:
    Acertos: 13      Tentativas: 14     Percentual: 92.8571

John no time:
    Acertos: 23      Tentativas: 30     Percentual: 76.6667

mostrar todos:
    Acertos: 32      Tentativas: 47     Percentual: 68.0851

mostrar time:
    Acertos: 32      Tentativas: 47     Percentual: 68.0851
```

Resumo

- Existem 4 classes de armazenamento de dados
  - Automático – variáveis locais
  - Estático – variáveis globais
  - Thread – programação concorrente
  - Dinâmico – alocação dinâmica de memória

- Uma variável estática pode ser criada declarando-a como global ou através do palavra-chave static

Resumo

- Se uma função usa dados sem modificá-los:
  - Se o dado é pequeno, como os de tipo básico, passe por valor

```
double soma(double a, double b);
```

  - Se é um vetor, use um ponteiro porque é a sua única escolha (use const para o ponteiro)

```
void mostraVetor(const double vet[], int tam);
```

  - Se o dado é um registro, use um ponteiro ou uma referência (use const para evitar modificação)

```
void exibir(const atleta & atl);
```

Resumo

- Se a função modifica os dados originais:
  - Se o dado é tipo básico, use um ponteiro porque isso deixa claro a intenção de modificá-lo

```
void atualiza(int * num);
```

  - Se é um vetor, use um ponteiro porque é a sua única escolha

```
void mostraVetor(int vet[], int tam);
```

  - Se o dado é um registro, use um ponteiro ou uma referência

```
void calcular(atleta & at1);
```