Programação de Computadores

PONTEIROS

Judson Santos Santiago

Introdução

- As variáveis e constantes armazenam informações
  - Elas ocupam espaço na memória
  - Possuem um tipo
- Os tipos básicos armazenam valores:

Inteiros

```
char   ch   = 'W';
short  sol  = 25;
int    num  = 45820;
```

Ponto-flutuantes

```
float  taxa = 0.25f;
double peso = 1.729156E5;
```

Introdução

- Porém, com os tipos básicos não é possível armazenar um conjunto de informações
  - Como armazenar o peso de 22 jogadores?

```
float p1  = 80.2;
float p2  = 70.6;
float p3  = 65.5;
...
float p21 = 85.8;
float p22 = 91.0;
```

Criar 22 variáveis diferentes não é a melhor solução.

  - A solução é usar vetores:

```
float peso[22];
```

Introdução

- Com vetores não é possível armazenar um conjunto de informações de tipos diferentes
  - Como armazenar um cadastro completo de 22 jogadores? (nome, idade, altura, peso, gols, etc.)

```
char     nome[22][80];
unsigned idade[22];
unsigned altura[22];
float    peso[22];
unsigned gols[22];
```

Criar vários vetores não é a melhor solução.

  - A solução é usar registros

Introdução

- Vetores e registros são frequentemente combinados para armazenar uma grande quantidade de informação
  - Como armazenar um cadastro de 500 jogadores?

```
struct jogador {
   char     nome[40];
   unsigned idade;
   float    peso;
   unsigned gols;
};
```

  - A solução é usar vetores de registros

```
jogador cadastro[500];
```

Motivação

- Vetores de registros comumente precisam guardar uma quantidade de elementos conhecida apenas durante a execução do programa

```
int tam;
cin >> tam;
jogador cadastro[tam];
```

A quantidade de elementos de um vetor não pode ser uma variável

  - Solução: ponteiros com alocação dinâmica de memória

Motivação

- Registros armazenam grandes quantidades de informação
  - Bancos de dados, imagens, áudio, vídeos, etc.

```
struct BITMAPINFOHEADER
{
    unsigned long  biSize;
    long           biWidth;
    long           biHeight;
    unsigned long  biPlanes;
    unsigned short biBitCount;
    unsigned long  biCompression;
    unsigned long  biSizeImage;
    long           biXPelsPerMeter;
    long           biYPelsPerMeter;
    unsigned long  biClrUsed;
    unsigned long  biClrImportant;
};
```

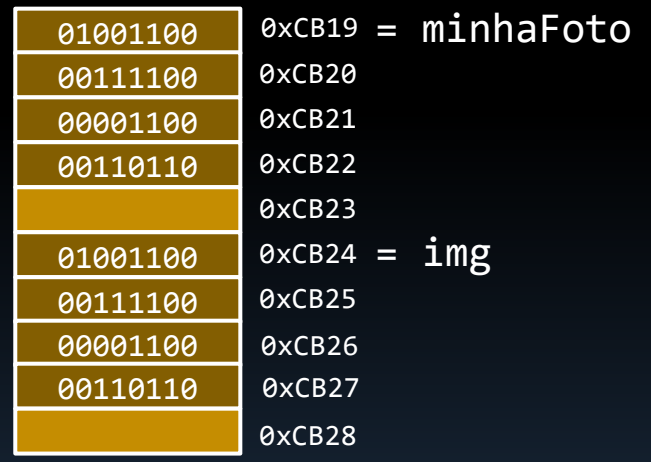
```
struct BITMAPFILEHEADER
{
    unsigned long  bfType;
    unsigned long  bfSize;
    unsigned short bfReserved1;
    unsigned short bfReserved2;
    unsigned long  bfOffBits;
};
struct Bitmap
{
    BITMAPFILEHEADER bitmapHeader;
    BITMAPINFOHEADER bitmapInfo;
    unsigned char * imgData;
};
```

Motivação

- Em C++, a passagem de argumentos é feita por cópia

```
int main()
{
    Bitmap minhaFoto;
    minhaFoto = CarregarImagem("C:\foto.bmp");
    TamanhoDaImagem( minhaFoto );
    ...
}
```

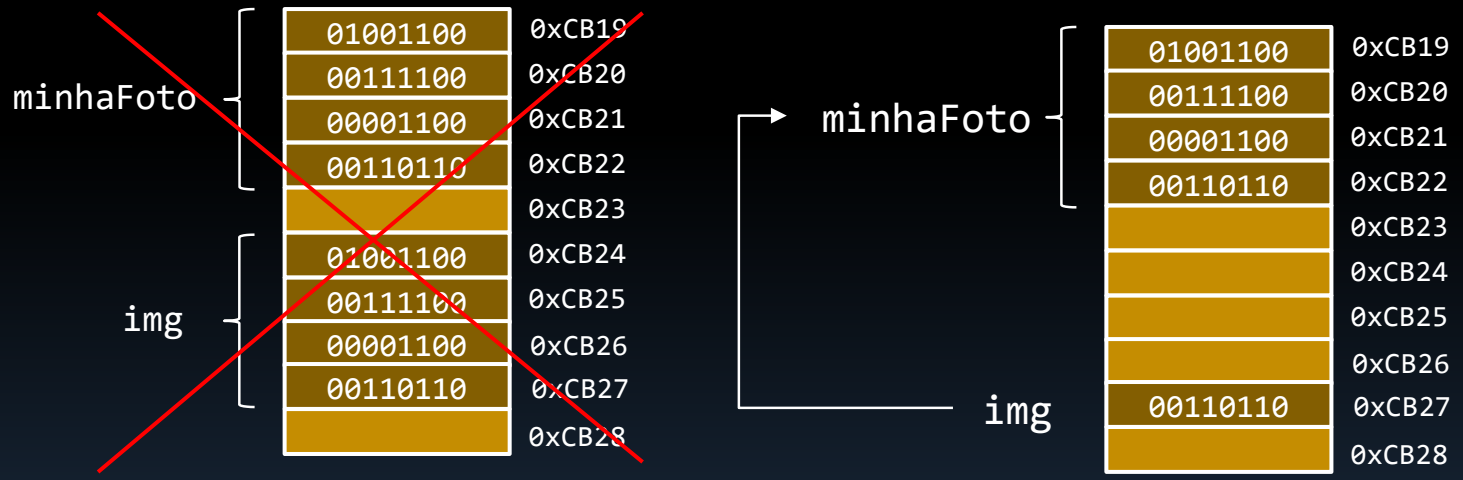
Faz uma cópia da imagem Bitmap




```
void TamanhoDaImagem( Bitmap img )
{
    cout << "Tam: " << img.bitmapInfo.biWidth  << "x" << img.bitmapInfo.biHeight;
}
```

Motivação

- Copiar uma grande quantidade de dados não é eficiente

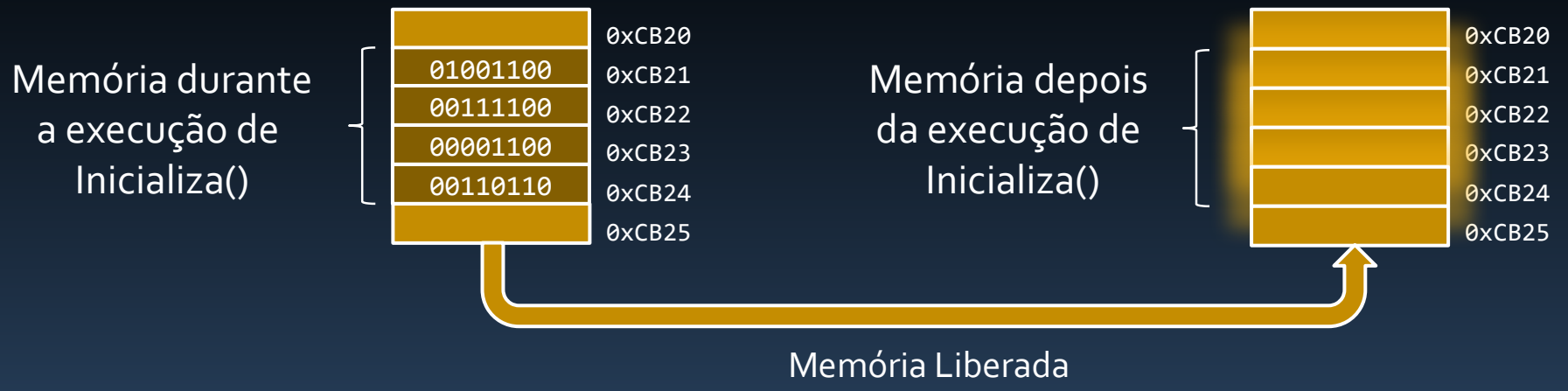


- Solução: usar um ponteiro para o registro

Motivação

- O tempo de vida de uma variável está atrelado ao seu escopo

```
void Inicializa()
{
    Bitmap explosion;
    explosion = CarregarImagem("BigExplosion.bmp");
}
```



Motivação

- Como alocar memória que continua "viva" após a execução de uma função?

```
void Inicializa()
{
   Bitmap explosion;
   explosion = CarregarImagem("BigExplosion.bmp");
}

void Desenha()
{
   DesenharImagem(explosion); // como usar explosion aqui?
}
```

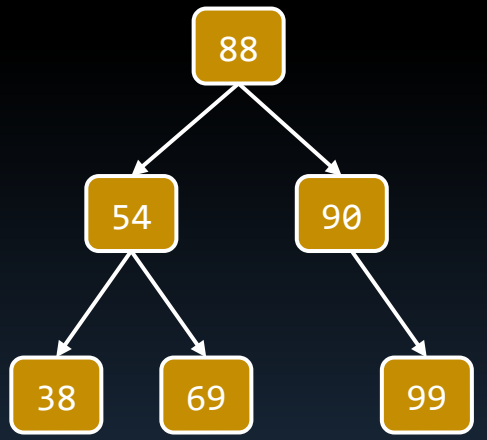
- Solução: ponteiros com alocação dinâmica de memória

Motivação

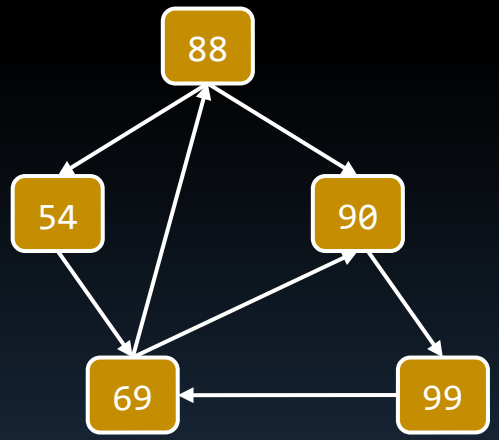
- Ponteiros são usados para organizar dados na memória para a solução mais eficiente de problemas

Lista Circular

Lista Duplamente Encadeada



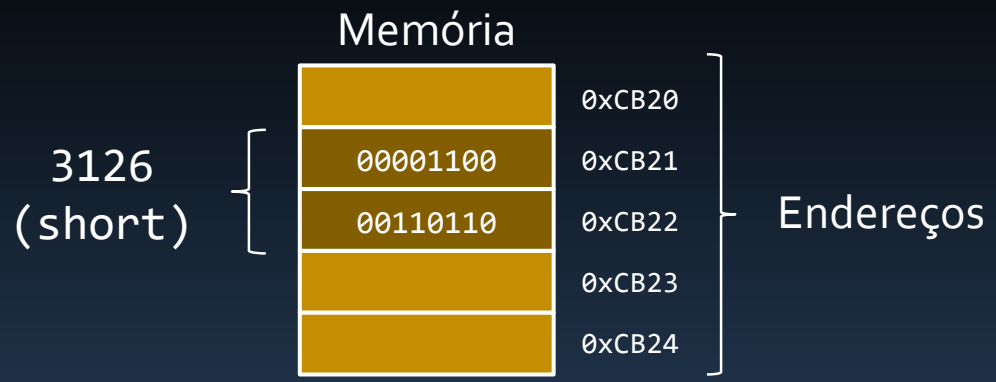
Árvore



Grafo

Variáveis

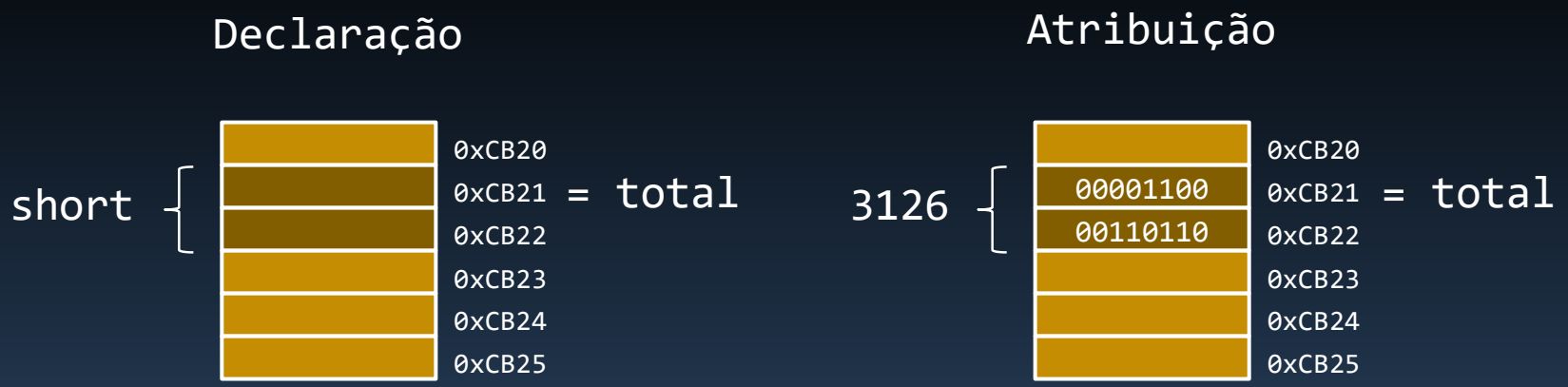
- Ao armazenar dados um programa gerencia:
  - Onde a informação está armazenada
  - Que tipo de informação é armazenada
  - O valor que é mantido lá



Variáveis

- A declaração de uma variável em um programa realiza os passos necessários para o armazenamento de dados

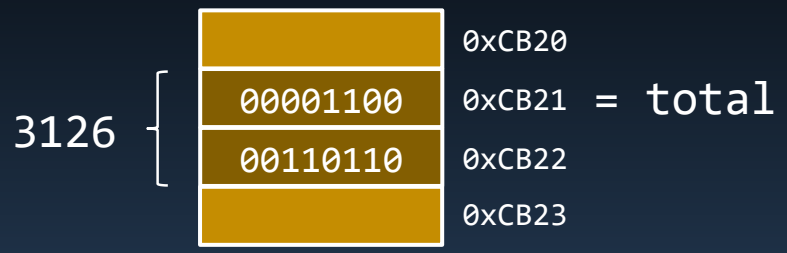
```
short total;   // declaração de variável
total = 3126;  // atribuição de valor
```



Endereços de Variáveis

- Todo nome de variável está associado a um endereço
  - O operador de endereço & obtém a sua localização

```
short total;       // declaração
total = 3126;      // atribuição

cout << total;     // valor
cout << &total;    // localização
```



Endereços de Variáveis

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

int main()
{
    int copos = 6;
    double cafe = 4.5;

    cout << "Valor de copos = " << copos << endl;
    cout << "Endereço de copos = " << &copos << endl;

    cout << "Valor de cafe = " << cafe << endl;
    cout << "Endereço de cafe = " << &cafe << endl;
}
```

// Endereço.cpp

Endereços de Variáveis

- Saída do programa:

Execução 1

```
Valor de copos = 6
Endereço de copos = 0027FCF8
Valor de cafe = 4.5
Endereço de cafe = 0027FCE8
```

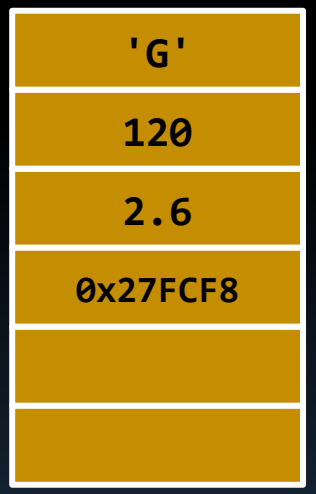
Execução 2

```
Valor de copos = 6
Endereço de copos = 0021F8FC
Valor de cafe = 4.5
Endereço de cafe = 0021F8EC
```

- Os endereços mudam mas os valores são os mesmos

Ponteiros

- Um ponteiro é um tipo especial que armazena endereços



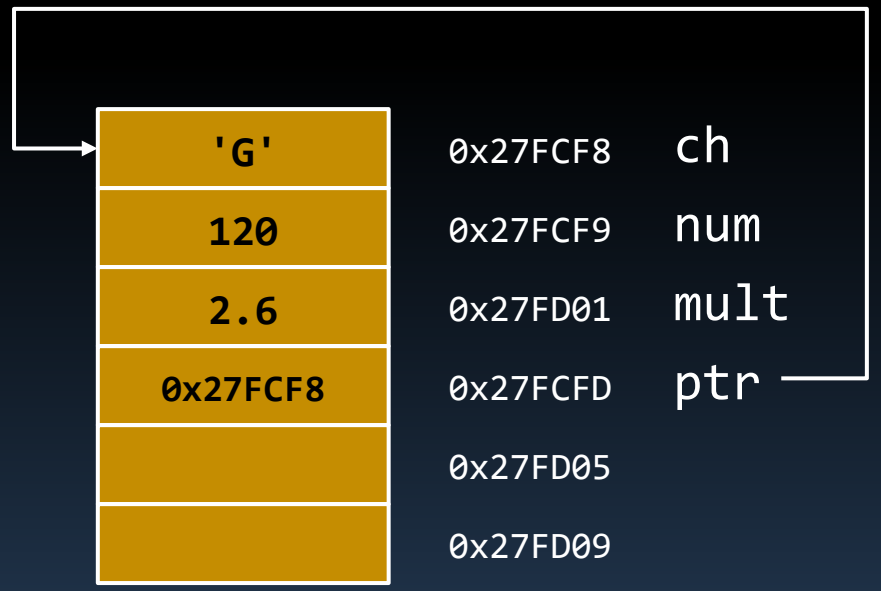
0x27FCF8	ch
0x27FCF9	num
0x27FD01	mult
0x27FCFD	ptr
0x27FD05	
0x27FD09	

```
char ch = 'G';
int num = 120;
float mult = 2.6;
char * ptr = &ch;
```

- Cada variável tem um endereço, incluindo o ponteiro

Ponteiros

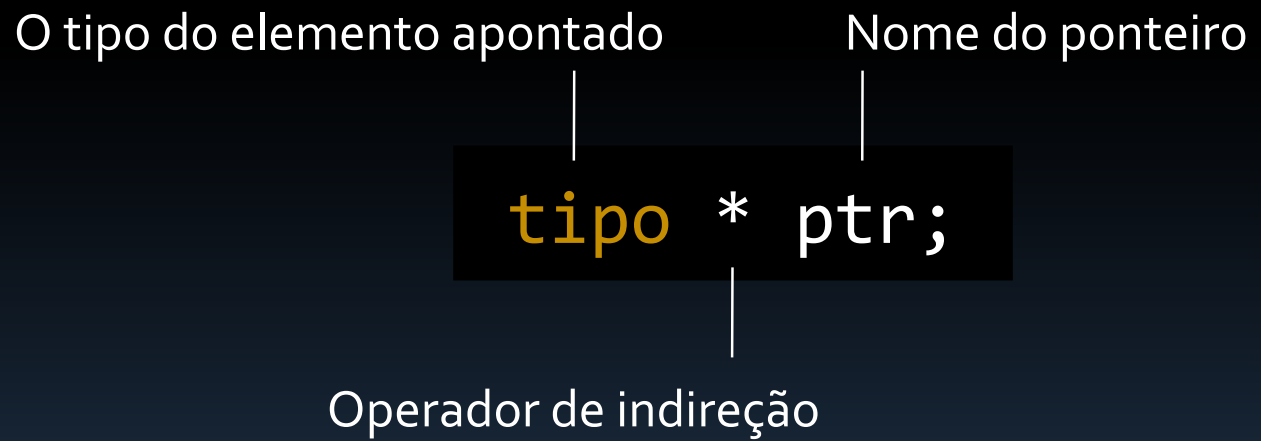
- Como o ponteiro contém um endereço de memória, diz-se que ele aponta para aquela posição de memória




```
char ch = 'G';
int num = 120;
float mult = 2.6;
char * ptr = &ch;
```

Ponteiros

- A declaração de um ponteiro segue o seguinte padrão:



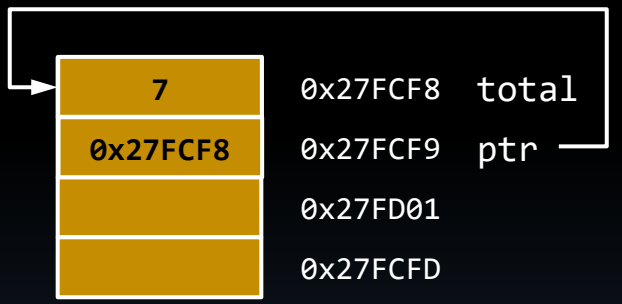
Ponteiros

- O ponteiro armazena um endereço
- O operador de indireção *
  acessa o conteúdo apontado


```
// declaração do ponteiro
char * ptr = &ch;

cout << ptr;   // endereço armazenado no ponteiro
cout << *ptr;  // conteúdo apontado
```

Ponteiros

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

int main()
{
    int total = 6;      // declara uma variável
    int * ptr;          // declara um ponteiro

    ptr = &total;       // atribui endereço de total

    cout << "Conteúdo de total  = " << total << endl;
    cout << "Conteúdo apontado  = " << *ptr << endl;

    cout << "Endereço de total  = " << &total << endl;
    cout << "Conteúdo de ptr    = " << ptr << endl;

    *ptr = *ptr + 1;      // altera valor
    cout << "Agora total vale = " << total << endl;
}
```



// Apontador.cpp

Ponteiros

- Saída do Programa:

```
Conteúdo de total  = 6
Conteúdo apontado  = 6
Endereço de total  = 0034FBBC
Conteúdo de ptr    = 0034FBBC
Agora total vale = 7
```

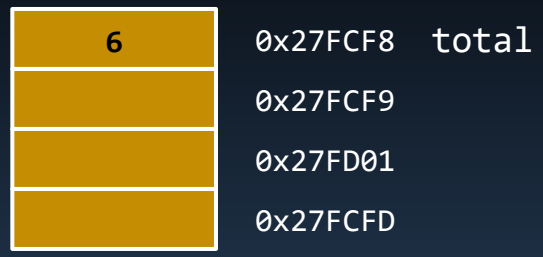
- A alteração de *ptr mudou o valor da variável apontada

```
*ptr = *ptr + 1;  // altera valor
cout << "Agora total vale = " << total << endl;
```

Variável *versus* Ponteiro

- Ao usar uma variável comum:
  - O valor é um elemento que possui um nome
  - A localização do valor é um elemento derivado (&)

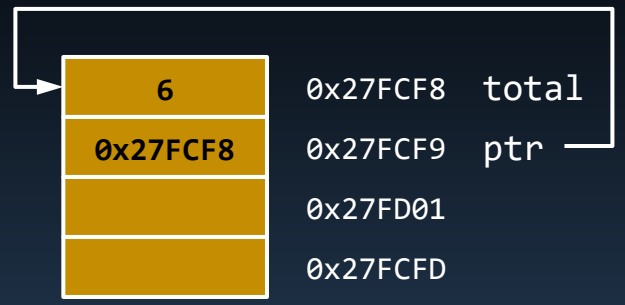
```
int total = 6;      // variável total
cout << total;      // total se refere ao valor
cout << &total;     // &total se refere ao endereço
```



Variável *versus* Ponteiro

- Ao usar um ponteiro:
  - A localização é um elemento que possui um nome
  - O valor é um elemento derivado (*)

```
int * ptr = &total; // ponteiro ptr
cout << ptr;        // ptr se refere ao endereço
cout << *ptr;       // *ptr se refere ao valor
```



Declaração de Ponteiros

- Por que não se declara um ponteiro da mesma forma que um int, char ou float?

```
char ch = 'G';
int num = 120;
float f = 2.1;
```

```
pointer p = 0x27FCF8;

// cout não sabe o tipo de *p
cout << *p;
```

- Não é suficiente dizer que uma variável é um ponteiro, é preciso também especificar para que tipo de dado ele aponta

```
char  * pc = &ch;
int   * pi = &num;
float * pf = &f;
```

Declaração de Ponteiros

- Na declaração, o uso de espaços ao redor do * é opcional

```
int *ptr;  // enfatiza que *ptr é um int
int* ptr;  // enfatiza que ptr é um endereço de um int
int * ptr; // estilo neutro
```

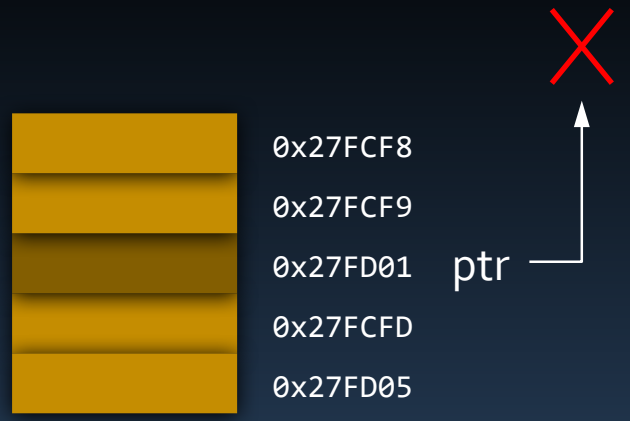
- Porém, cuidado com declarações múltiplas
  - Elas adotam o primeiro estilo

```
// p1 é um ponteiro para int, p2 é um int
int * p1, p2;

// p1 e p2 são ponteiros para int
int *p1, *p2;
```

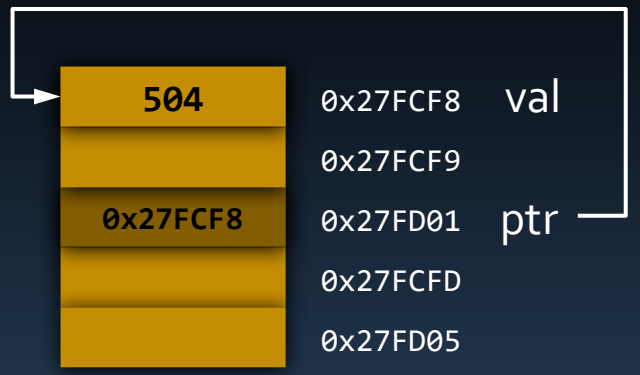
Declaração de Ponteiros

- Ao declarar um ponteiro o computador não aloca automaticamente memória para guardar o valor apontado

```
long * ptr;
*ptr = 504;
```




```
long val;
long * ptr = &val;
*ptr = 504;
```



Atribuição de Valores

- Os ponteiros guardam endereços
  - Endereços são valores inteiros mas não têm o tipo int:
  - Um endereço é um inteiro de 4 bytes (ou 8 bytes) †
  - O tipo int é um inteiro que pode ter 2 bytes (antigo MS-DOS)

```
int * p  = 0xB800;      // inválido, mistura de tipos
```

- É possível converter um inteiro para um endereço usando um type cast

```
int * p  = (int *) 0xB800;
```

Atribuição de Valores

- O type cast converte para um endereço e indica também o tipo do valor apontado

```
char * p = (char *) 0xB800;   // endereço de um char
```

- Ao usar o operador &, o tipo do endereço já é fornecido pelo tipo da variável

```
char ch  = 'G';
char * p = &ch;  // tipo = endereço de char
```

Ponteiros e Registros

- Um ponteiro pode apontar para tipos criados pelo programador (registros, uniões e enumerações)

```
struct jogador
{
    char nome[20];
    float salario;
    unsigned gols;
};

jogador pele;

jogador * ptr = &pele;
```

Ponteiros e Registros

- Os campos de um registro são acessados com o operador (.)

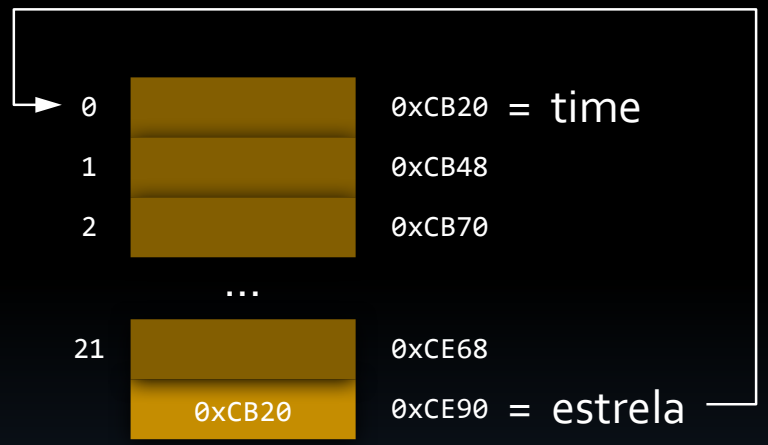
```
jogador pele;
cout << pele.nome;       // nome do jogador
cout << pele.salario;    // salario do jogador
cout << pele.gols;       // número de gols do jogador
```

- Os campos de um ponteiro para registro usam o operador (->)

```
jogador * ptr = &pele;
cout << ptr->nome;      // nome do jogador
cout << ptr->salario;   // salario do jogador
cout << ptr->gols;      // número de gols do jogador
```

Ponteiros e Vetores

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

struct jogador {
    char nome[20];
    float salario;
    unsigned gols;
};

int main() {
     jogador time[22];
     jogador * estrela = &time[0];

     cout << "Digite o nome, salario e gols de dois jogadores: ";
     cin >> time[0].nome; cin >> time[0].salario; cin >> time[0].gols;
     cin >> time[1].nome; cin >> time[1].salario; cin >> time[1].gols;

     cout << "\nO jogador estrela do time é " << estrela->nome << "!\n";
}
```



// Estrela.cpp

Ponteiros e Vetores

- Saída do programa:

```
Digite o nome, salário e gols de dois jogadores:
Bebeto 200000 600
Romario 300000 800

O jogador estrela do time é Bebeto!
```

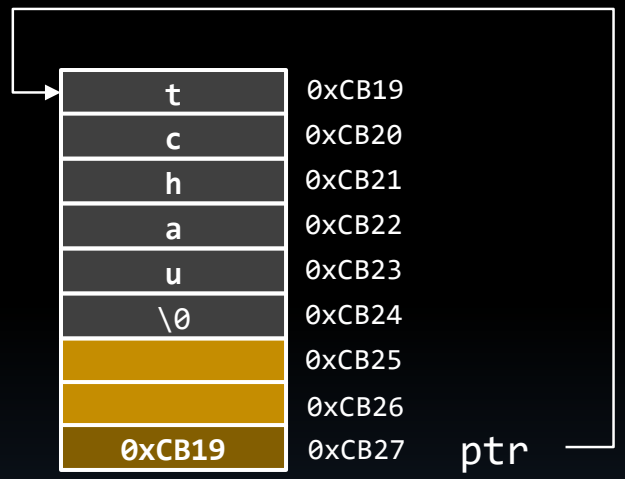
- O endereço do primeiro elemento pode ser obtido assim:

```
jogador * estrela = time;    // nome do vetor é um endereço
```

Ponteiros e Strings

- Uma constante string é um char * (endereço do primeiro caractere)

```
cout << "tchau";

char * ptr = "tchau";     // ptr aponta para a constante
cout << ptr;              // ptr = endereço de um char
```



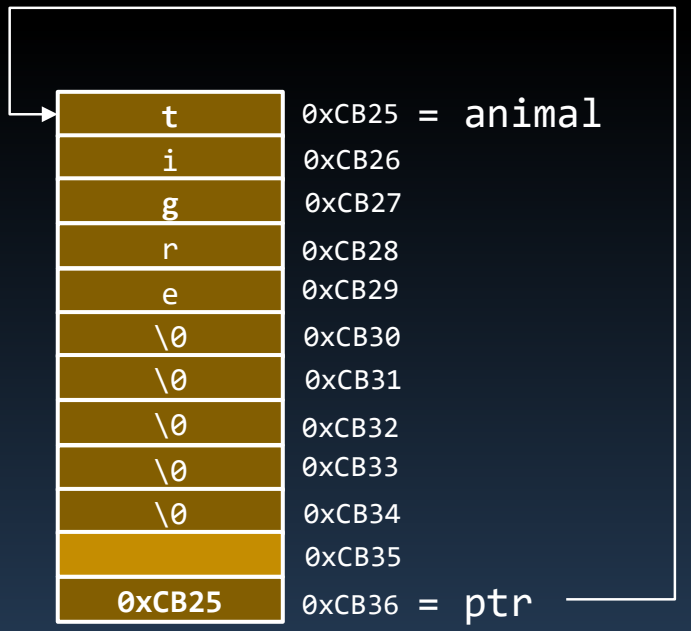
- O ponteiro não pode ser usado para alterar uma constante
  - É recomendável indicar isso na declaração do ponteiro

```
const char * ptr = "tchau";  // ponteiro para valor constante
```

Ponteiros e Strings

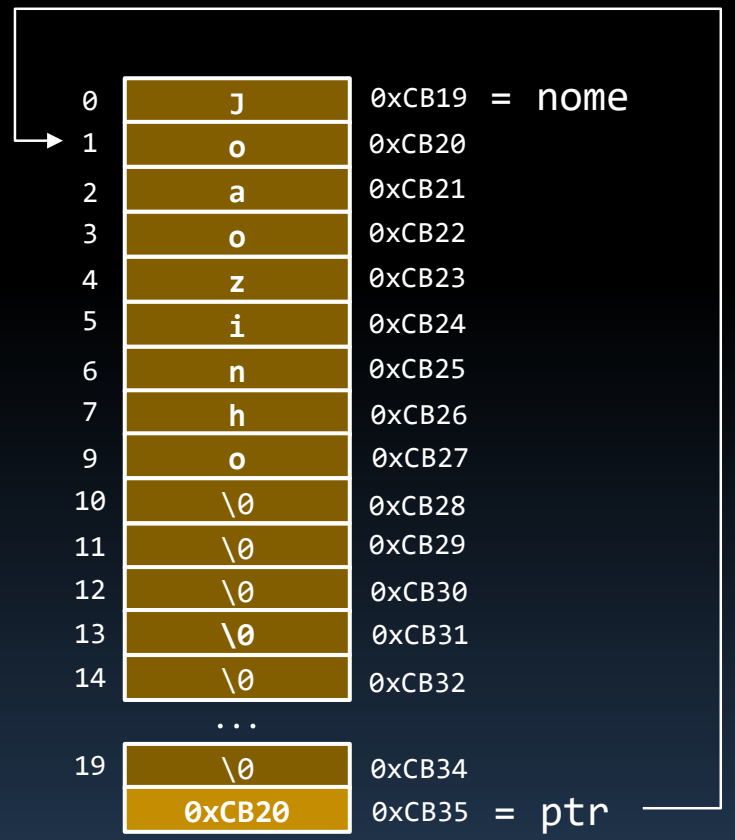
- O vetor de caracteres:
  - Armazena uma cópia da constante
  - Pode ser modificado

```
char animal[10] = "tigre";
cout << animal;

char * ptr = animal;
*ptr = 'T';
cout << animal;

ptr = &animal[2];
cout << ptr;
```



Ponteiros e Strings

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

int main()
{
    // inicializa vetor com a constante string
    char nome[20] = "Joaozinho";
    cout << nome << endl;

    // manipula elementos com um ponteiro
    char * ptr = &nome[1];
    *ptr = 'P';
    cout << ptr << endl;

    nome[0] = 'T';
    *ptr = 'i';
    cout << nome << endl;

}
```



// StringPtr.cpp

Ponteiros e Strings

- Saída do programa:

```
Joaozinho
Paozinho
Tiaozinho
```

- Um ponteiro pode apontar para elementos de um vetor

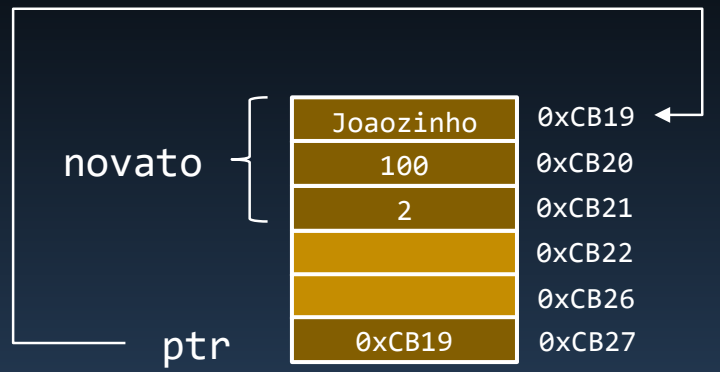
```
// guarda endereço do segundo caractere
char * ptr = &nome[1];

// o nome do vetor é o endereço do primeiro elemento
char * ptr = nome;
```

Ponteiros e Funções

- Ponteiros podem ser usados em parâmetros de funções
  - Evita cópia de um grande volume de informações
  - Pode ser usado para modificar os dados originais

```
void exibir(jogador * ptr) {
    cout << ptr->nome << " "
         << ptr->salario << " "
         << ptr->gols << endl;
}

int main() {
    jogador novato = {"Joaozinho", 100, 2};
    exibir(&novato);
    ...
}
```

```
struct jogador
{
    char nome[20];
    float salario;
    unsigned gols;
};
```



Resumo

- Ponteiros são variáveis que armazenam endereços
- O primeiro uso importante:
  - Para guardar o endereço de uma variável
    - Permite passar o endereço de variáveis para funções
    - Evita cópia de grandes quantidades de dados