

Programação de Computadores

PONTEIROS

Introdução

- As **variáveis** armazenam informações
 - Elas ocupam espaço na memória
 - Possuem um tipo

```
char   ch   = 'W';  
short  sol   = 25;  
int     num  = 45820;  
  
float   taxa = 0.25f;  
double  peso = 1.72E5;
```

'W'	0xFCF8	=	ch
25	0xFCF9	=	sol
45820	0xFD01	=	num
0.25f	0xFCFD	=	taxa
1.72E5	0xFD05	=	peso

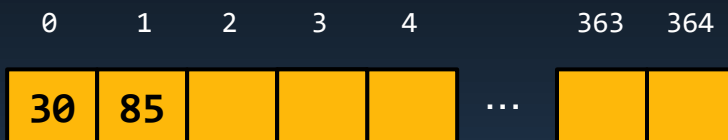
Introdução

- **Vetores** armazenam um conjunto de informações
 - Sob um único identificador
 - Todas do mesmo tipo

```
unsigned visitas[365];
```

```
visitas[0] = 30;
```

```
visitas[1] = 85;
```



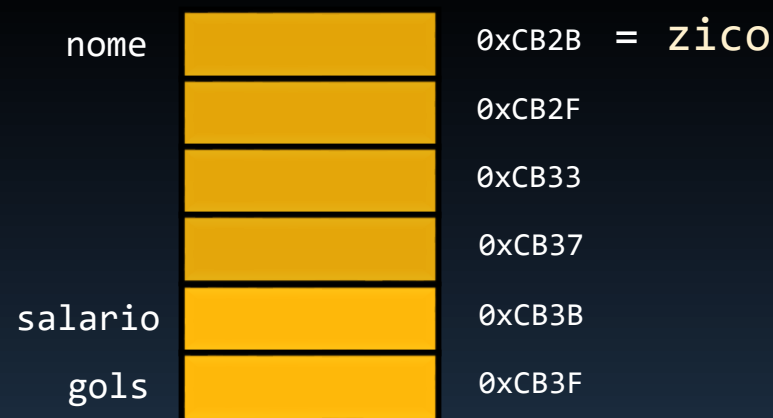
Memória

Introdução

- **Registros** armazenam várias informações
 - De tipos que podem ser diferentes
 - Definem um novo tipo de dado

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

```
Jogador zico;
```



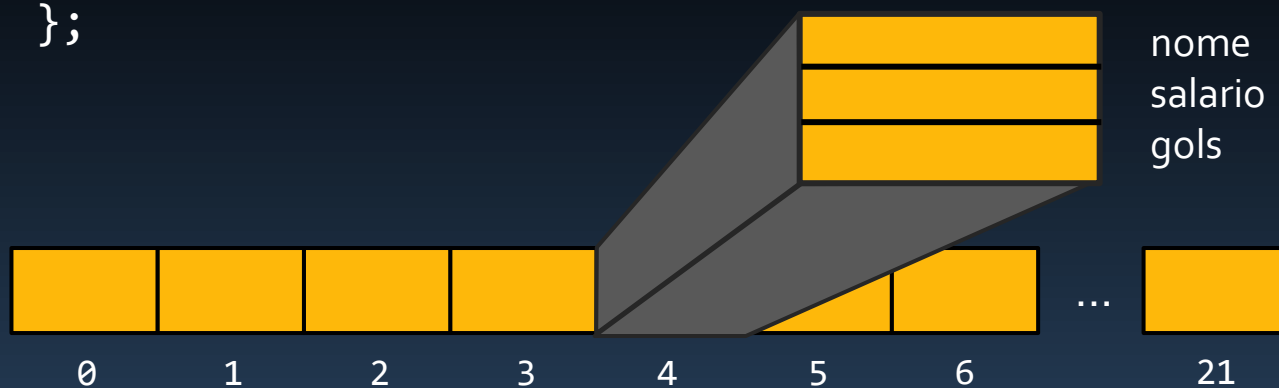
Memória

Introdução

- **Vetores** e **registros** são frequentemente combinados
 - Armazenam uma grande quantidade de informação

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

Jogador equipe[22];



Motivação

- O **tamanho dos vetores** precisa ser:

- Conhecido durante a compilação
- Uma constante inteira

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

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

- É possível fazer diferente?
 - Sim, usando **ponteiros** e **alocação dinâmica** de memória

Motivação

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

```
struct Bitmap
{
    BITMAPFILEHEADER bitmapHeader;
    BITMAPINFOHEADER bitmapInfo;
    unsigned char * imgData;
};
```

```
struct BITMAPFILEHEADER
{
    unsigned long  bfType;
    unsigned long  bfSize;
    unsigned short bfReserved1;
    unsigned short bfReserved2;
    unsigned long  bfOffBits;
};
```

```
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;
};
```

Motivação

- A **passagem de argumentos** em funções é feita por cópia

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

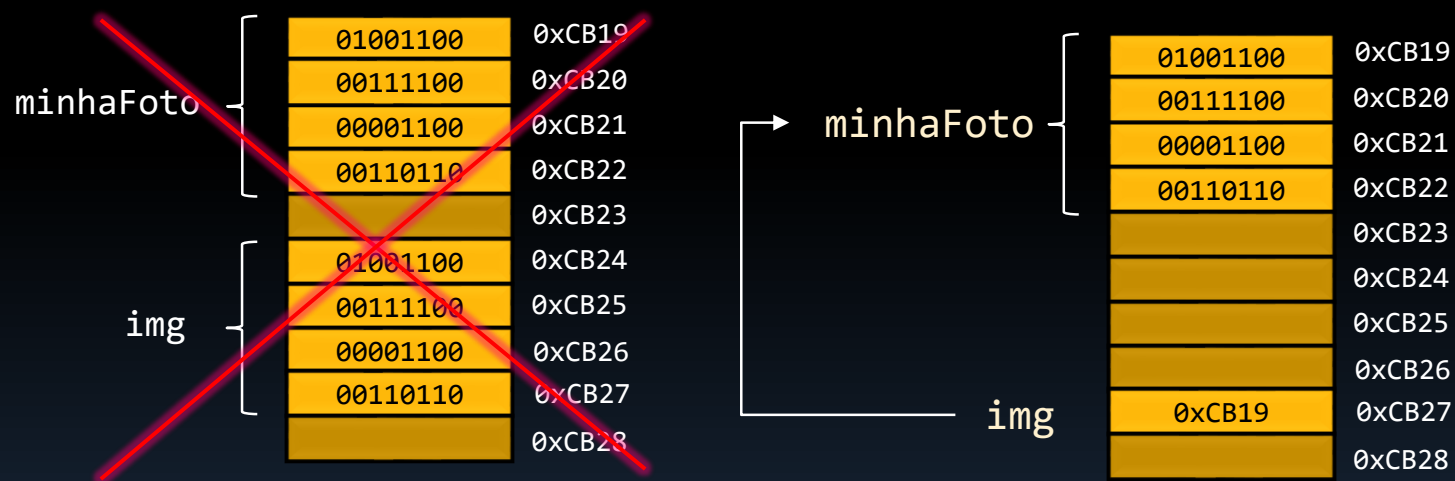
Faz cópia
da imagem
Bitmap

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

01001100	0xCB19 = minhaFoto
00111100	0xCB20
00001100	0xCB21
00110110	0xCB22
	0xCB23
01001100	0xCB24 = img
00111100	0xCB25
00001100	0xCB26
00110110	0xCB27
	0xCB28

Motivação

- Copiar grandes quantidades de dados **não é eficiente**



Solução: usar um **ponteiro** para registros

Motivação

- O **tempo de vida** de uma **variável local** é limitado
 - Está atrelado a função onde foi declarada

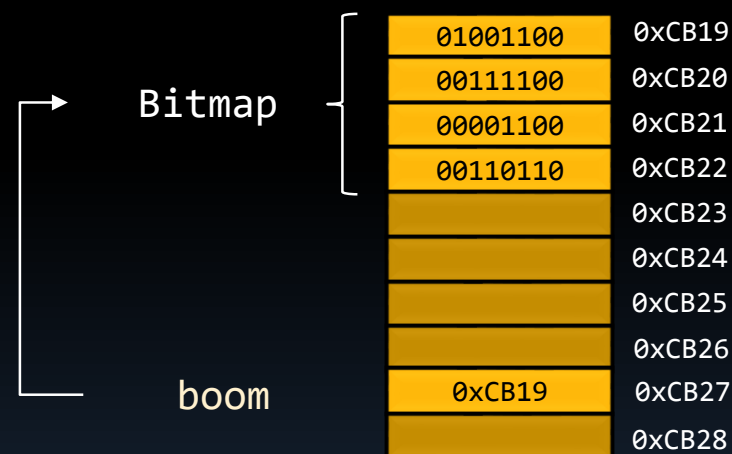
```
void Inicializar()
{
    Bitmap boom;
    boom = CarregarImagem("Explosion.bmp");
}

void Desenhar()
{
    DesenharImagem(boom); // como usar boom aqui?
}
```

Motivação

- Precisamos **alocar memória** que continua "viva"
 - Mesmo depois de finalizar a execução da função

```
void Inicializar()  
{  
    Bitmap * boom = new Bitmap;  
    ...  
}
```



Solução: **ponteiros** com **alocação dinâmica** de memória

Motivação

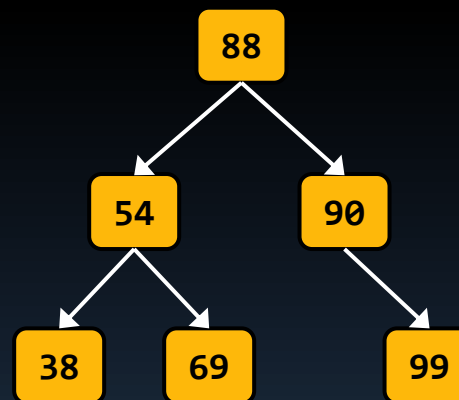
- Ponteiros são usados também em **estrutura de dados**
 - Solução mais eficiente de problemas
 - Organização de dados



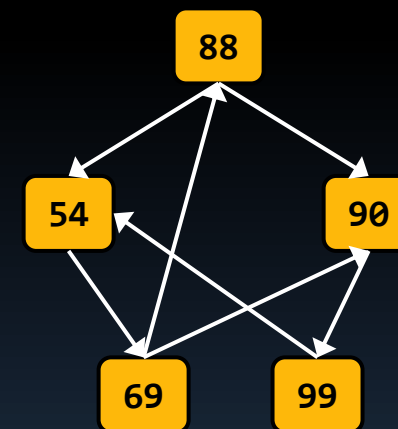
Fila



Lista



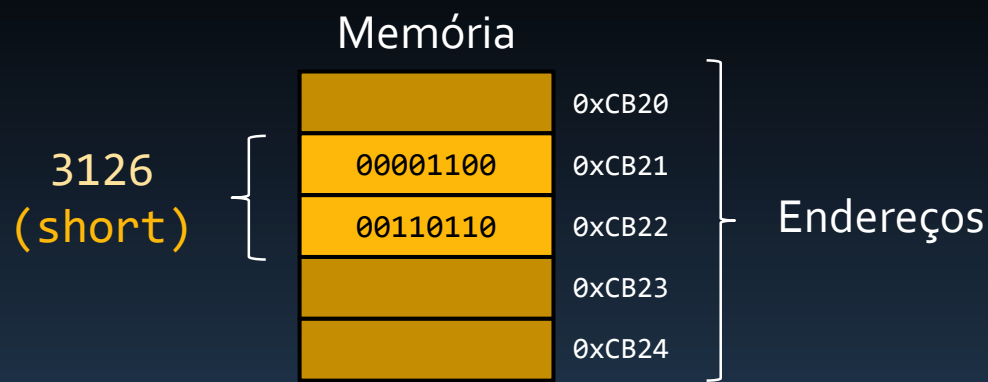
Árvore



Grafo

Variáveis

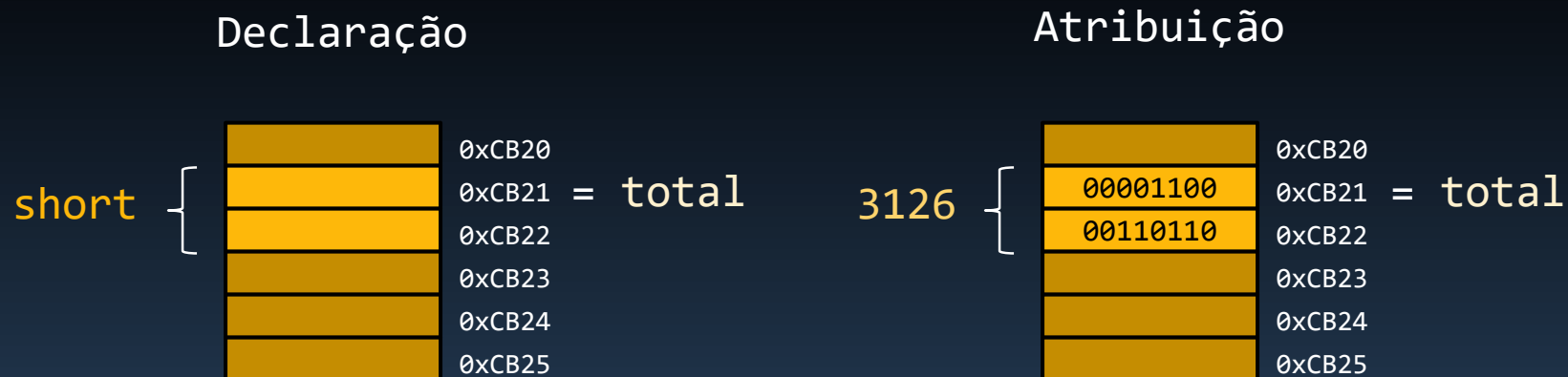
- Ao **armazenar dados**, um programa gerencia:
 - Onde a informação está armazenada
 - Qual o tipo da informação
 - Que valor é mantido lá



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
```

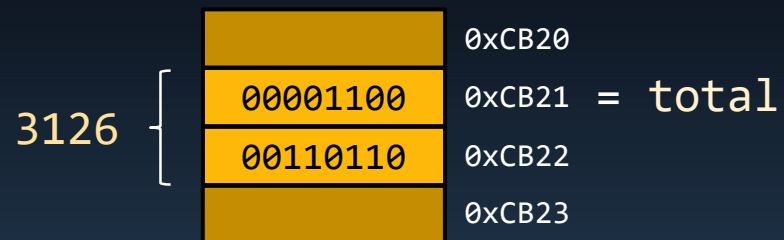


Endereços de Variáveis

- 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

```
#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;

    return 0;
}
```


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

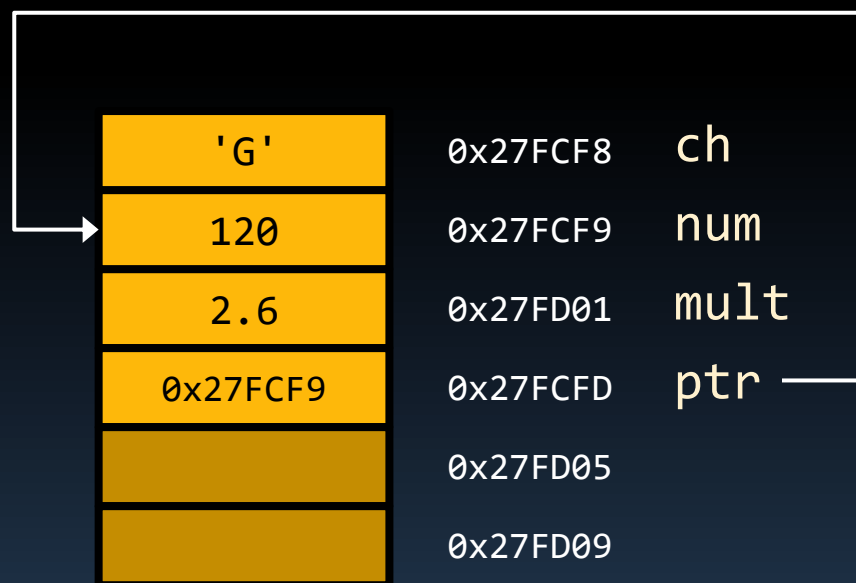
- Um **ponteiro** é um tipo especial que armazena **endereços**

'G'	0x27FCF8	ch	<code>char ch = 'G';</code>
120	0x27FCF9	num	<code>int num = 120;</code>
2.6	0x27FD01	mult	<code>float mult = 2.6;</code>
0x27FCF9	0x27FCFD	ptr	<code>int * ptr = &num;</code>
	0x27FD05		
	0x27FD09		

- 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;  
int * ptr = &num;
```

Ponteiros

- A **declaração de um ponteiro** segue o seguinte padrão:

O tipo do elemento apontado

Nome do ponteiro

```
int * ptr;
```

Operador de indireção

Ponteiros

- O **ponteiro** armazena um endereço

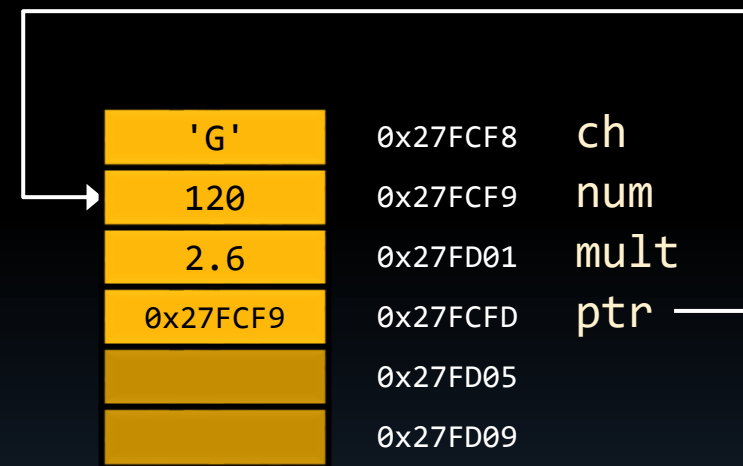
- O **operador de indireção *** acessa o conteúdo apontado

```
// declaração do ponteiro
```

```
int * ptr = &num;
```

```
cout << ptr;    // endereço armazenado no ponteiro
```

```
cout << *ptr;   // conteúdo apontado
```



Ponteiros

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

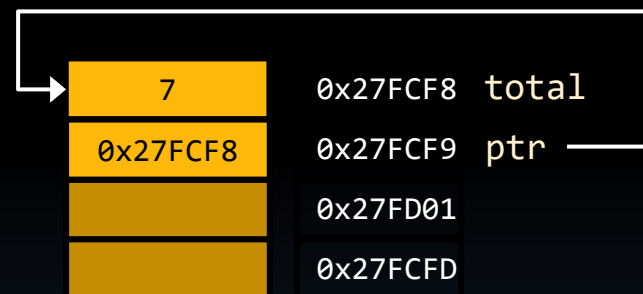
```
int main()
{
    int total = 6;      // declara uma variável
    int * ptr;          // declara um ponteiro

    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;
}
```



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 nome da variável representa um **valor**
 - A **localização** do valor é obtido por um operador (&)

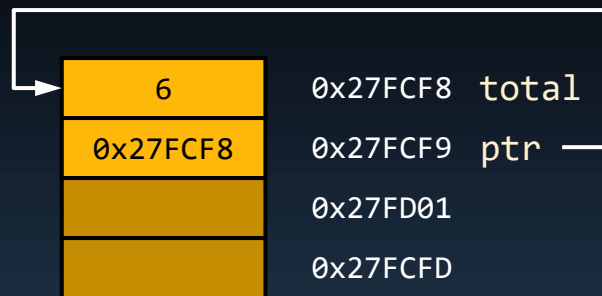
```
int total = 6;      // variável total
cout << total;      // total se refere ao valor
cout << &total;     // &total se refere ao endereço
```

6	0x27FCF8 total
	0x27FCF9
	0x27FD01
	0x27FCFD

Variável *versus* Ponteiro

- Ao usar um ponteiro:
 - O nome do ponteiro representa uma **localização**
 - O **valor** contido nesta localização é obtido por um operador (*)

```
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 = 2.6;
```

```
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

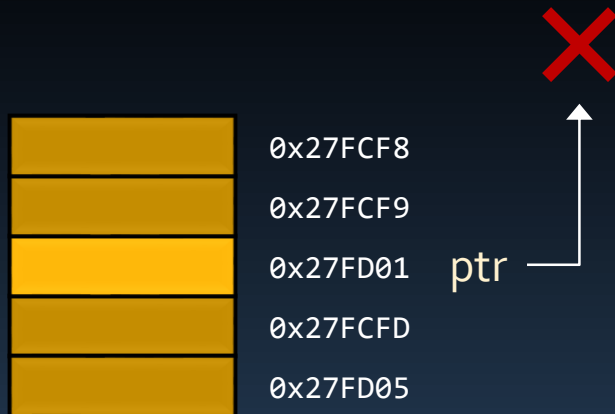
```
// p1 é um ponteiro para int, p2 é um int
int * p1, p2;
```

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

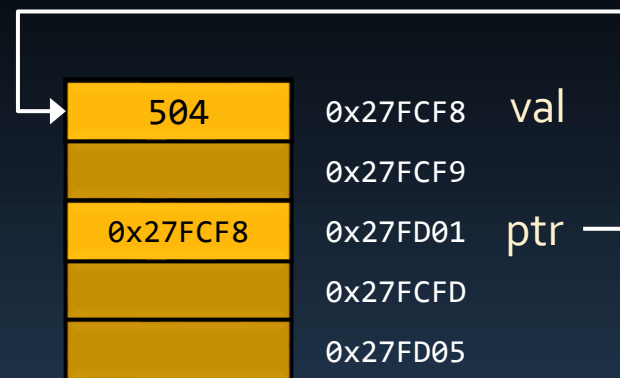
Declaração de Ponteiros

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

```
int * ptr;  
*ptr = 504;
```




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



Atribuição de Valores

- Os ponteiros guardam **endereços**
 - Um endereço é um inteiro de 4 ou 8 bytes [†]
 - **Endereços** são valores inteiros mas não têm o **tipo int**


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

- É possível atribuir usando um **type cast**

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

Atribuição de Valores

- O **type cast** converte o inteiro para endereço e indica o **tipo do valor** presente lá

```
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
```

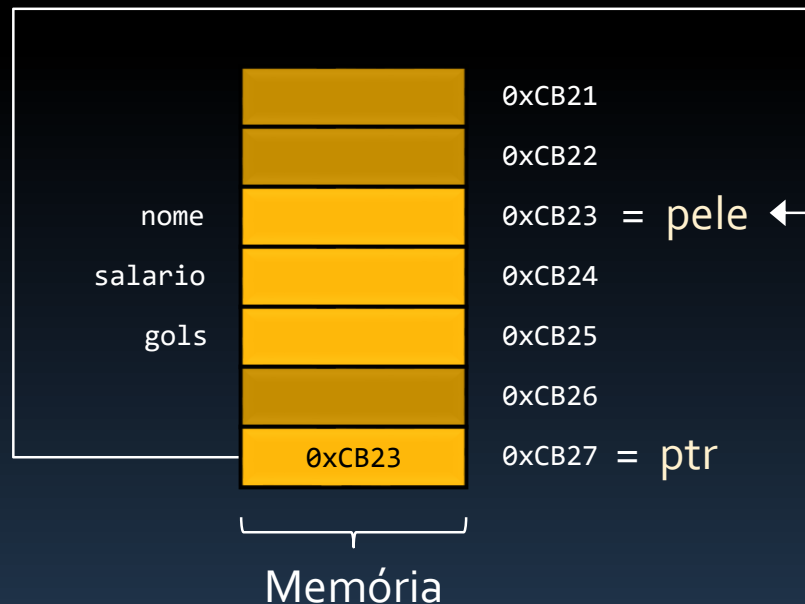
Ponteiros e Registros

- 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;   // salário 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;   // salário do jogador  
cout << ptr->gols;      // número de gols do jogador
```


Ponteiros e Vetores

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

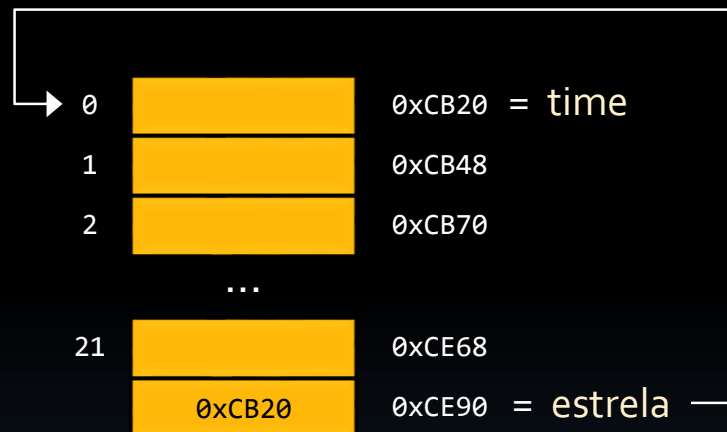
```
struct Jogador {
    char nome[20];
    float salario;
    unsigned gols;
};
```

```
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";
```

```
}
```



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:

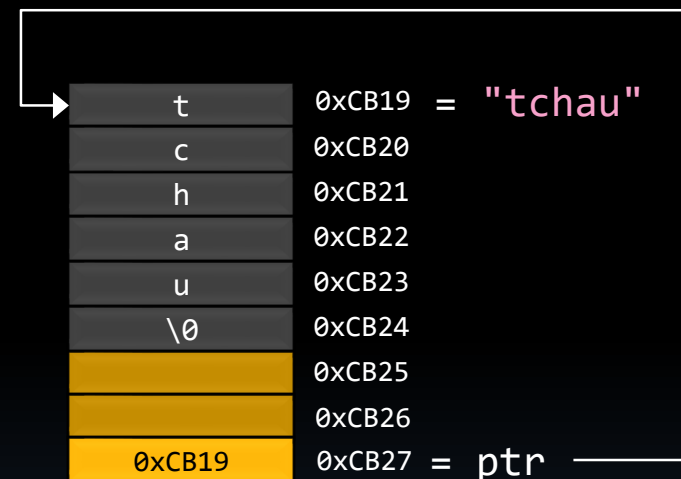
```
Jogador * estrela = time;    // nome do vetor é um endereço
```

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
```

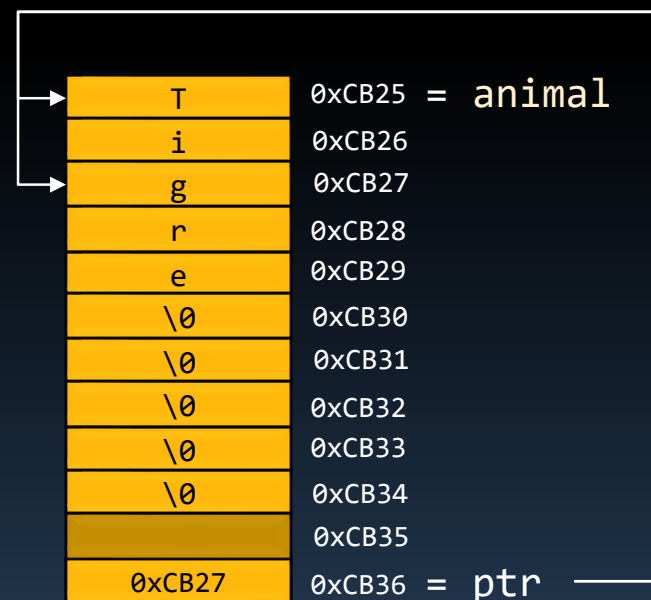
Ponteiros e Strings

- Já 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;
```



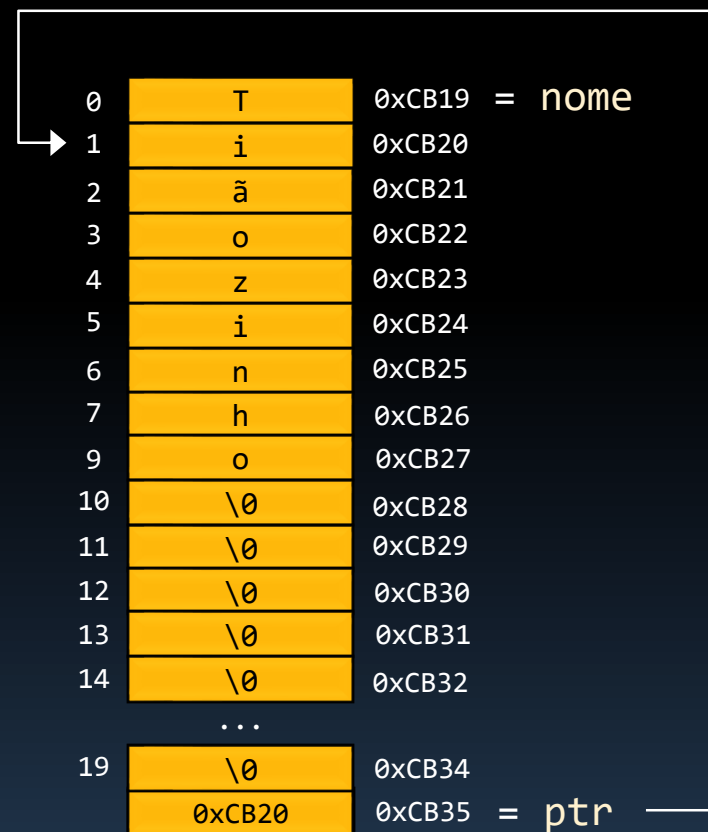
Ponteiros e Strings

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

int main()
{
    // inicializa vetor com a constante string
    char nome[20] = "Joãozinho";
    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;
}
```



Ponteiros e Strings

- Saída do programa:

```
Joãozinho  
Pãozinho  
Tiãozinho
```

- 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;
```

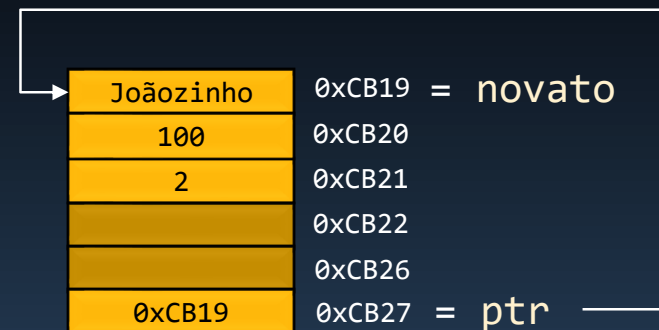
Ponteiros e Funções

- 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 = {"Joãozinho", 100, 2};  
    Exibir(&novato);  
    ...  
}
```

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



Resumo

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

