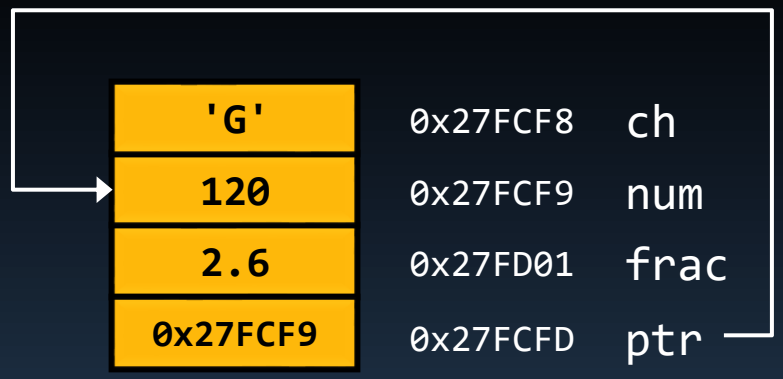
Programação de Computadores

# ALOCAÇÃO DINÂMICA DE MEMÓRIA

Judson Santiago

# Introdução

- Um ponteiro é um tipo especial que armazena endereços
  - Operador & recupera o endereço de uma variável
  - Operador * acessa o conteúdo apontado

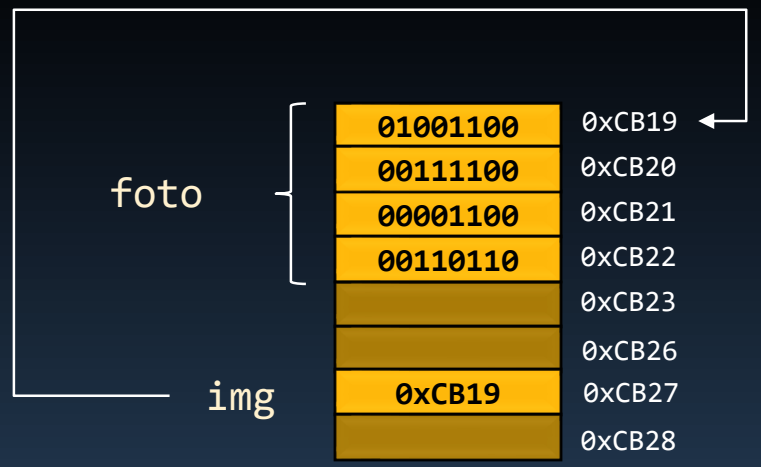



```
char  ch   = 'G';
int   num  = 120;
float frac = 2.6f;

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

# Introdução

- Ponteiros podem ser usados em parâmetros de funções
  - Evita cópia de grandes volumes de dados
  - As funções trabalham com os dados originais

```
void Tamanho(Bitmap * img)
{
    cout << img->altura  << " x "
         << img->largura << "\n";
}

int main()
{
    Bitmap foto = Carregar("Foto.bmp");
    Tamanho(&foto);
    ...
}
```



# Introdução

- Ponteiros guardam endereços de variáveis existentes
  - Variáveis são memórias rotuladas pelo compilador
  - Ponteiros fornecem uma segunda forma de acesso


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

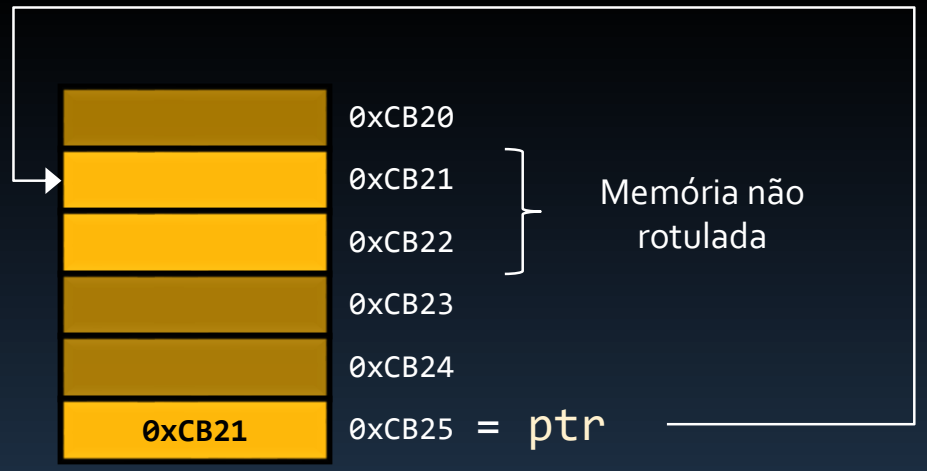
```
cout << *ptr; // 120
```

ou

```
cout << num;  // 120
```

# Introdução

- O verdadeiro poder dos ponteiros está em apontar para memória não rotulada, alocada durante a execução do programa



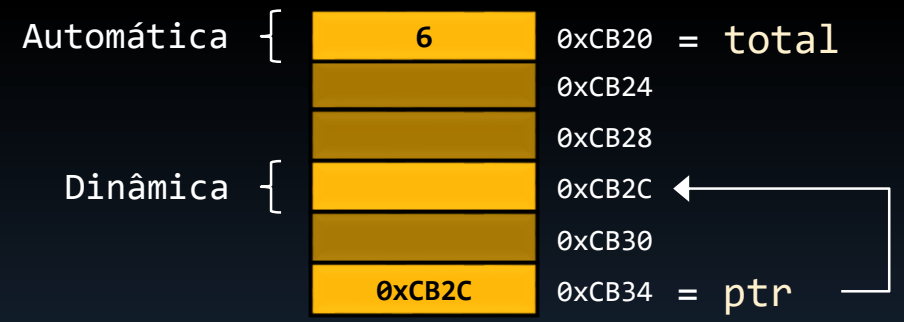
# Alocação de Memória

- Existem duas formas de alocar memória:
  - Alocação automática
    - Declaração de variáveis

```
int total = 6;
```

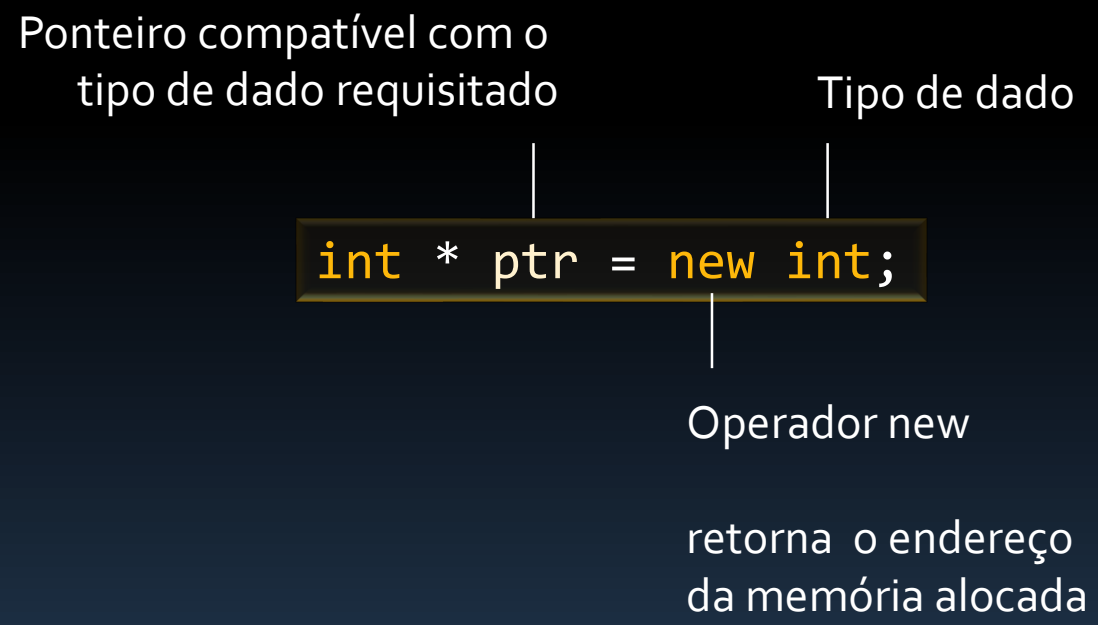
  - Alocação dinâmica
    - Operador new

```
int * ptr = new int;
```



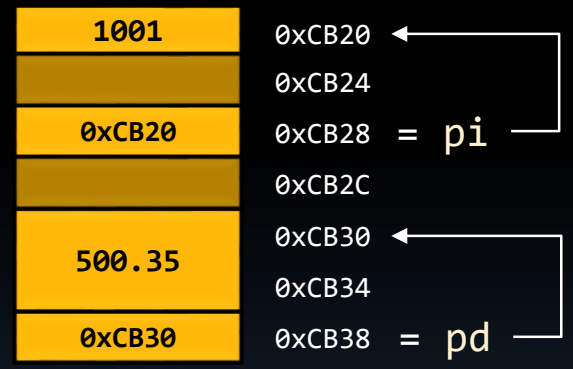
# Alocação de Memória

- A alocação de memória é feita com o operador new



# Alocação de Memória

```
#include <iostream>
using namespace std;
int main()
{
    int * pi = new int;
    *pi = 1001;
    cout << "Valor inteiro = " << *pi << endl;
    cout << "Localização = " <<  pi << endl << endl;

    double * pd = new double;
    *pd = 500.35;
    cout << "Valor ponto-flutuante = " << *pd << endl;
    cout << "Localização = " << pd << endl << endl;

    cout << "Tamanho de pi  = " << sizeof(pi)  << endl;
    cout << "Tamanho de *pi = " << sizeof(*pi) << endl << endl;
    cout << "Tamanho de pd  = " << sizeof(pd)  << endl;
    cout << "Tamanho de *pd = " << sizeof(*pd) << endl;
}
```




NewSize.cpp

# Alocação de Memória

- Saída do programa:

**Plataforma 32-bits**

```
Valor inteiro = 1001
Localização = 00114CD0

Valor ponto-flutuante = 500.35
Localização = 00114DA8

Tamanho de  pi = 4
Tamanho de *pi = 4
Tamanho de  pd = 4
Tamanho de *pd = 8
```

**Plataforma 64-bits**

```
Valor inteiro = 1001
Localização = 000001889E409FD0

Valor ponto-flutuante = 500.35
Localização = 000001889E40FD50

Tamanho de  pi = 8
Tamanho de *pi = 4
Tamanho de  pd = 8
Tamanho de *pd = 8
```

# Liberando Memória

- Toda memória alocada deve ser liberada ao final do uso
  - Deve-se manter new e delete sempre balanceados

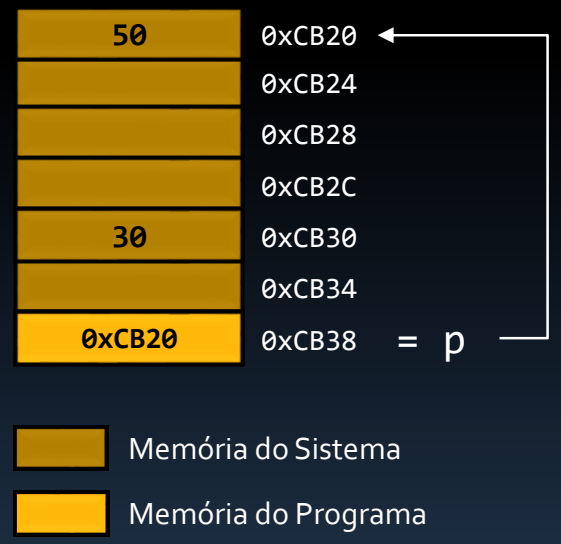
```
int * p = new int;  // aloca memória com new

...                 // usa memória

delete p;           // libera memória com delete
```

- O operador delete permite retornar a memória não mais utilizada para o sistema

# Liberando Memória

- O delete libera a memória mas não destrói o ponteiro
  - O ponteiro pode ser reutilizado para novas alocações

```
int * p = new int;
*p = 30;
cout << *p;
delete p;

--------------------

p = new int;
*p = 50;
cout << *p;
delete p;
```



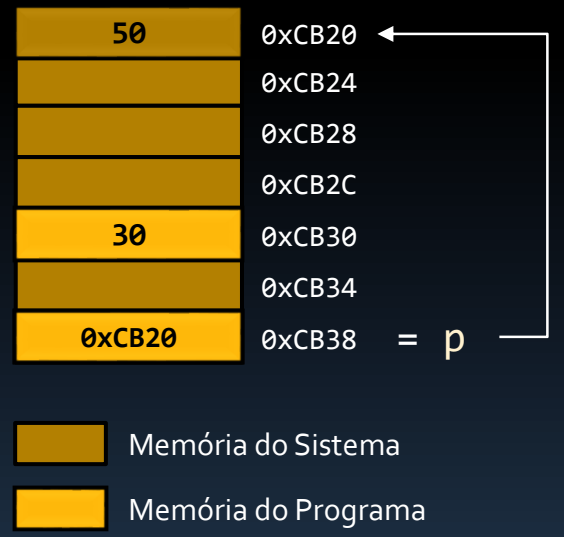
# Liberando Memória

- O new deve ser sempre balanceado com um delete
  - Caso contrário tem-se um vazamento de memória

```
int * p = new int;
*p = 30;
cout << *p;
```

---

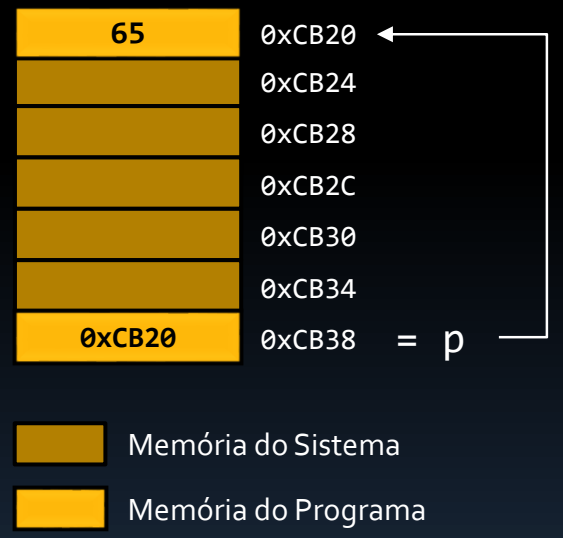
```
p = new int; // memory leak
*p = 50;
cout << *p;
delete p;
```



# Liberando Memória

- É fácil esquecer o delete

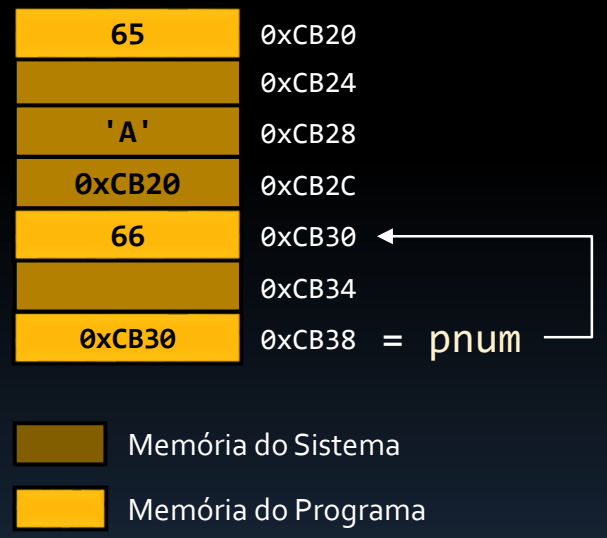
```
void PrintNum(char ch)
{
    int * p = new int { ch };
    cout << *p;
    // esqueceu de deletar memória
    // antes de sair da função
}

int main()
{
    PrintNum('A');
    ...
}
```



# Liberando Memória

- Cuidado com funções que alocam memória

```
int * GetNum(char ch)
{
    int * p = new int { ch };
    return p;
}

int main()
{
    int * pnum;
    pnum = GetNum('A');
    cout << *pnum;
    // esqueceu de deletar
    pnum = GetNum('B');
}
```



# Liberando Memória

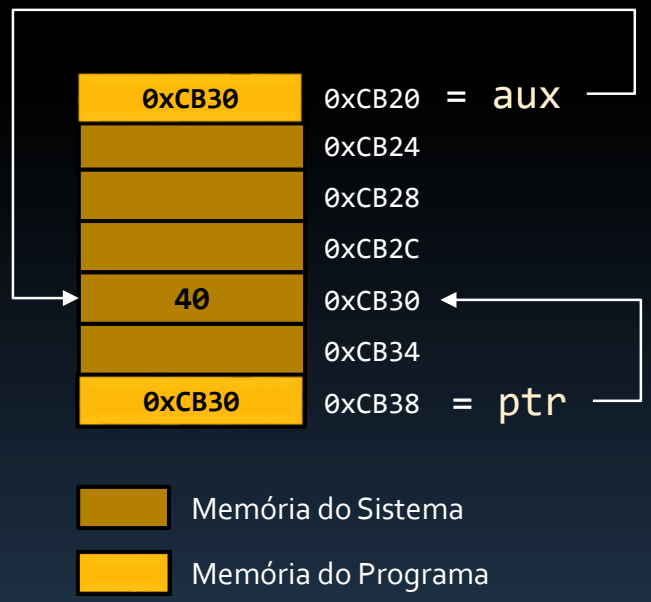
- Não se pode liberar o mesmo bloco de memória duas vezes

```
int * p = new int;  // aloca memória com new

--------------------

*p = 30;
cout << *p;

--------------------

delete p;           // libera memória
delete p;           // resultado indefinido
```

# Liberando Memória

- É fácil esquecer que a memória já foi liberada
  - Cuidado com múltiplos apontadores

```
int main()
{
    int * ptr = new int { 40 };
    int * aux = ptr;

    cout << *aux;
    delete aux;

    cout << *ptr; x
    delete ptr;   x
}
```



# Liberando Memória

- É fácil esquecer que a memória já foi liberada
  - Cuidado com memória compartilhada entre funções

```
int main()
{
    int * ptr = new int;
    *ptr = 40;

    Processar(ptr);
    Exibir(ptr);

    delete ptr;
}
```

```
void Processar(int * p)
{
    // esqueceu que outra função
    // fará uso da memória apontada
    delete p;
}

void Exibir(int * p)
{
    // p aponta para memória inválida
    cout << *p << endl; x
    delete p;           x
}
```

# Liberando Memória

- Não se pode liberar memória criada de forma automática, ou seja, pela declaração de variáveis

```
int val = 30;           // declaração de variável
delete val;  x          // inválido, val não é um ponteiro

--------------------

int val = 30;
int * ptval = &val;
...                     // dezenas de linhas depois
delete ptval;  x        // inválido, não foi alocado com new
```

# Alocação Dinâmica

- Por que utilizar memória alocada dinamicamente?

```
int * p = new int;      // aloca memória
*p = 30;                // atribuição de valor
cout << *p;             // exibição de valor
delete p;               // libera memória
```

```
int total;              // declaração de variável
total = 30;             // atribuição de valor
cout << total;          // exibição do valor
```

O propósito não é substituir a alocação automática

# Vetores Dinâmicos

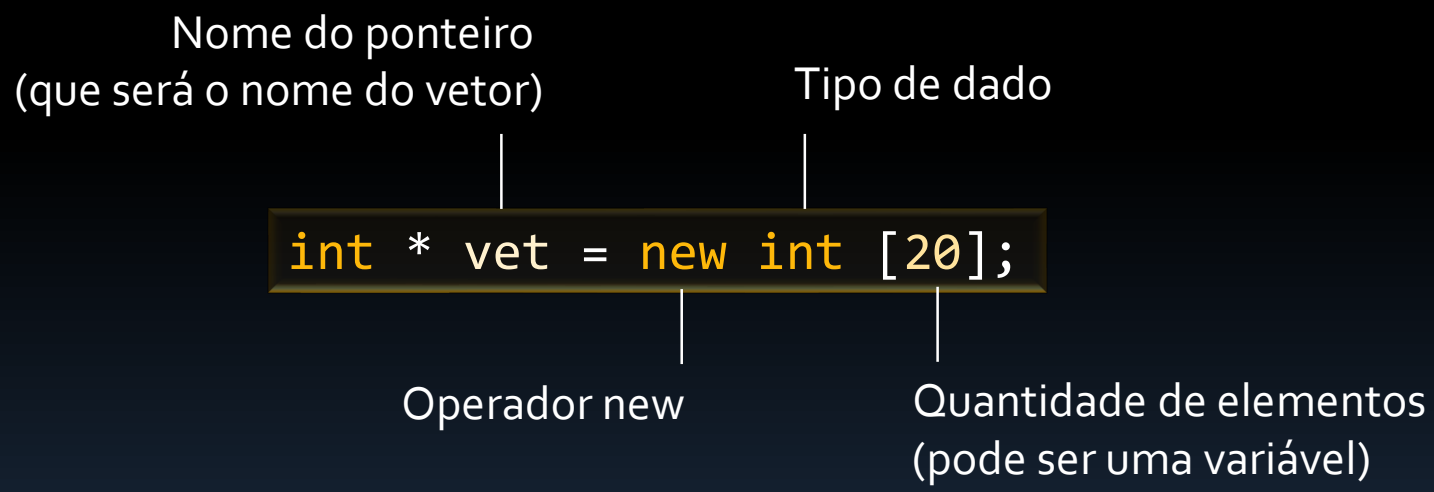
- O vetor tradicional é chamado de vetor estático
  - É preciso definir o tamanho do vetor na declaração (o tamanho precisa ser uma constante inteira)

```
int vet[10];  // vetor de 10 inteiros
```

- Com o operador new é possível criar um vetor dinâmico
  - Seu tamanho pode ser definido a qualquer momento (o tamanho pode ser lido do usuário)

# Vetores Dinâmicos

- Declaração de um vetor dinâmico:




- O ponteiro recebe o endereço do primeiro elemento do vetor

# Vetores Dinâmicos

- O tamanho de um vetor dinâmico pode ser uma variável
  - O valor pode ser fornecido durante a execução do programa

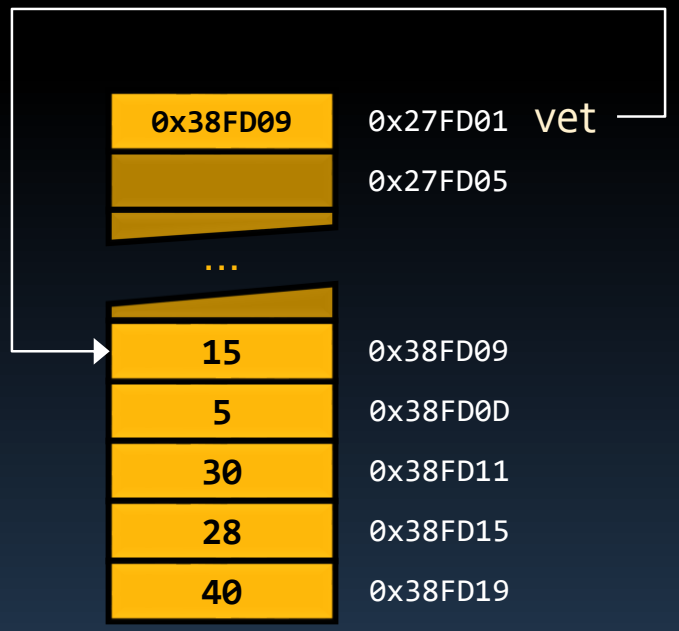
```
cout << "Digite o tamanho do vetor: ";
int tam;
cin >> tam;

int * vet = new int [tam];
```

  - Mas o vetor não muda de tamanho depois de criado

```
tam = tam + 1; // não muda o tamanho do vetor já criado
```

# Vetores Dinâmicos

- O ponteiro de um vetor dinâmico pode ser usado como se fosse um vetor



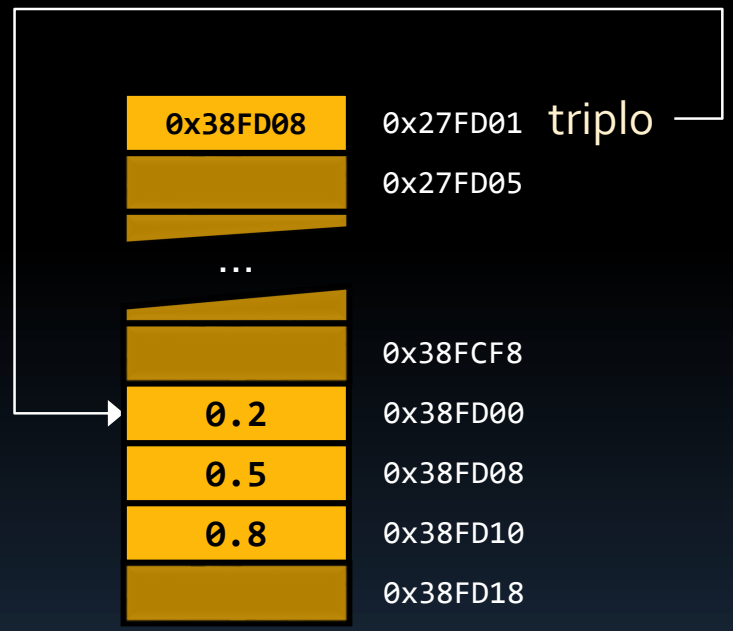

```
int * vet = new int [5];

vet[0] = 15;
vet[1] = 5;
vet[2] = 30;
vet[3] = 28;
vet[4] = 40;

cout << vet[0];  // 15
cout << *vet;    // 15
```

# Vetores Dinâmicos

- Para liberar a memória de um vetor dinâmico é preciso usar delete com uma notação especial

# Vetores Dinâmicos

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

int main()
{
    // memória para três double's
    double * triplo = new double [3];
    triplo[0] = 0.2;
    triplo[1] = 0.5;
    triplo[2] = 0.8;

    cout << "triplo[1] = " << triplo[1] << endl;
    triplo = triplo + 1;   // incrementa o ponteiro
    cout << "Agora triplo[0] = " << triplo[0] << endl;
    cout << "Agora triplo[1] = " << triplo[1] << endl;
    triplo = triplo - 1;   // retorna ao inicio

    delete [] triplo;  // libera a memória
}
```




Triplo.cpp

# Vetores Dinâmicos

- Saída do programa:

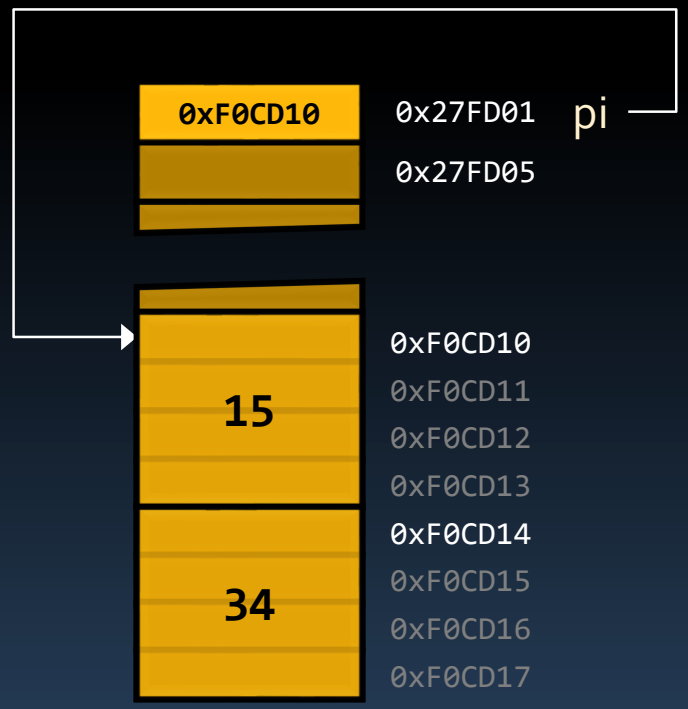
```
triplo[1] = 0.5
Agora triplo[0] = 0.5
Agora triplo[1] = 0.8
```

- Um ponteiro é uma variável e seu conteúdo pode ser modificado através de uma atribuição

```
triplo = triplo + 1;   // incrementa o ponteiro
triplo = triplo - 1;   // decrementa o ponteiro
```

# Aritmética com Ponteiros

- Endereços podem ser somados e subtraídos
  - O valor adicionado (ou subtraído) depende do tipo



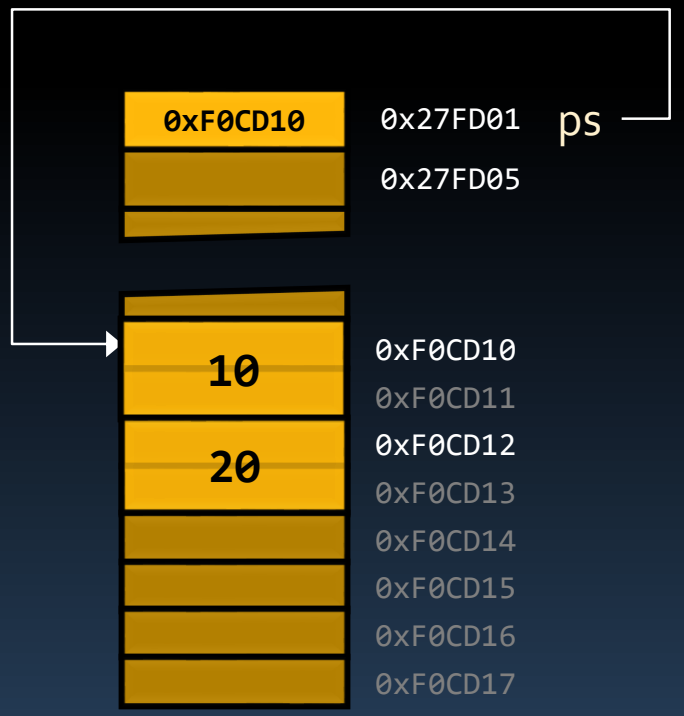

```
int * pi = new int [2] { 15, 34 };
// avança 4 bytes - sizeof(int)
pi = pi + 1;
```

```
short * ps = new short [2];
// avança 2 bytes - sizeof(short)
ps = ps + 1;
```

```
char * pc = new char [2];
// avança 1 byte - sizeof(char)
pc = pc + 1;
```

# Aritmética com Ponteiros

- Endereços podem ser somados e subtraídos
  - O valor adicionado (ou subtraído) depende do tipo



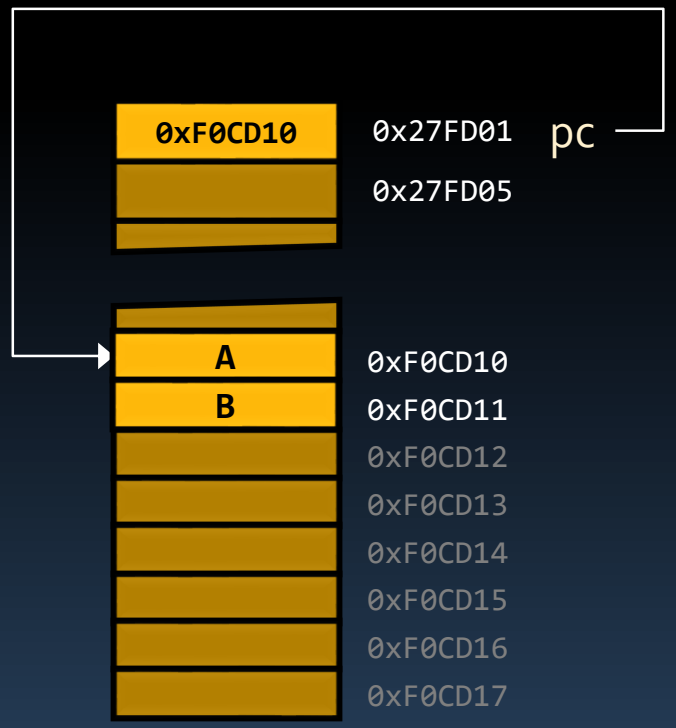

```
int * pi = new int [2];
// avança 4 bytes - sizeof(int)
pi = pi + 1;
```

```
short * ps = new short [2] { 10, 20 };
// avança 2 bytes - sizeof(short)
ps = ps + 1;
```

```
char * pc = new char [2];
// avança 1 byte - sizeof(char)
pc = pc + 1;
```

# Aritmética com Ponteiros

- Endereços podem ser somados e subtraídos
  - O valor adicionado (ou subtraído) depende do tipo




```
int * pi = new int [2];
// avança 4 bytes - sizeof(int)
pi = pi + 1;

short * ps = new short [2];
// avança 2 bytes - sizeof(short)
ps = ps + 1;

char * pc = new char [2] { 'A', 'B' };
// avança 1 byte - sizeof(char)
pc = pc + 1;
```

# Dinâmico *versus* Estático

- Um ponteiro pode ser usado como um vetor

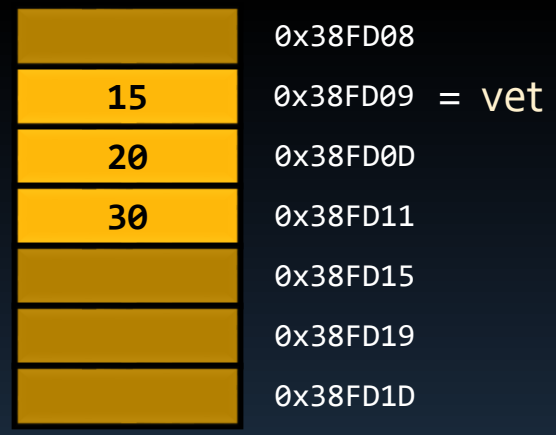
```
int * pvet = new int [10];

pvet[0] = 15;
cout << pvet[0];
pvet[1] = pvet[0] + 5;
```

- Um vetor pode ser usado como um ponteiro

```
int vet[10];

*vet = 15;                // vet[0] = 15;
cout << *(vet + 0);       // cout << vet[0];
*(vet + 1) = *vet + 5;    // vet[1] = vet[0] + 5;
```

# Dinâmico *versus* Estático

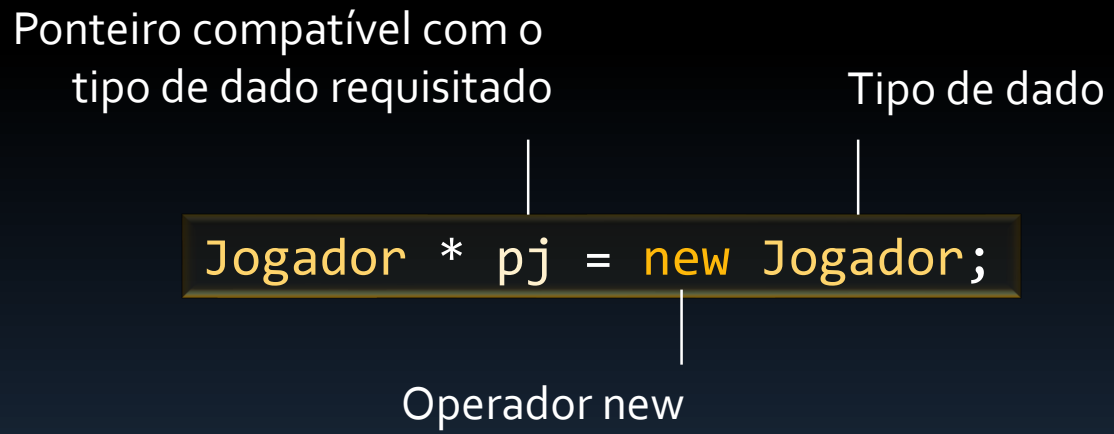
- O nome de um vetor estático é um endereço
  - Mas ele não pode ser alterado




```
int vet[3];

vet[0] = 15;          // *(vet+0) ou *vet
vet[1] = 20;          // *(vet+1)
*(vet+2) = 30;        // vet[2]

cout << vet[1];       // 20
cout << *(vet+1);     // 20

vet = vet + 1;    x   // inválido
```

# Registros Dinâmicos

- O operador new também aloca registros dinâmicos




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

# Registros Dinâmicos

- O operador membro (`.`) não pode ser usado com ponteiros
  - A linguagem oferece o operador (`->`)
    - Utiliza-se (`.`) com registros
    - Utiliza-se (`->`) com ponteiros

```
Jogador j;

cin >> j.nome;
cin >> j.salario;
cin >> j.altura;
```

```
Jogador * pj = new Jogador;

cin >> pj->nome;      // (*pj).nome
cin >> pj->salario;   // (*pj).salario
cin >> pj->altura;    // (*pj).altura
```

# Registros Dinâmicos

```
#include <iostream>
using namespace std;
struct Jogador
{
    char nome[20];
    float salario;
    unsigned gols;
};
int main()
{
     Jogador * pbeb = new Jogador;
     cout << "Digite nome, salário e gols do jogador: ";
     cin >> pbeb->nome >> pbeb->salario >> pbeb->gols;

     cout << "Contratação para o próximo ano:\n" << pbeb->nome
          << " por " << pbeb->salario << " Reais\n";

     delete pbeb;
}
```

RegDin.cpp

# Registros Dinâmicos

- Saída do programa:

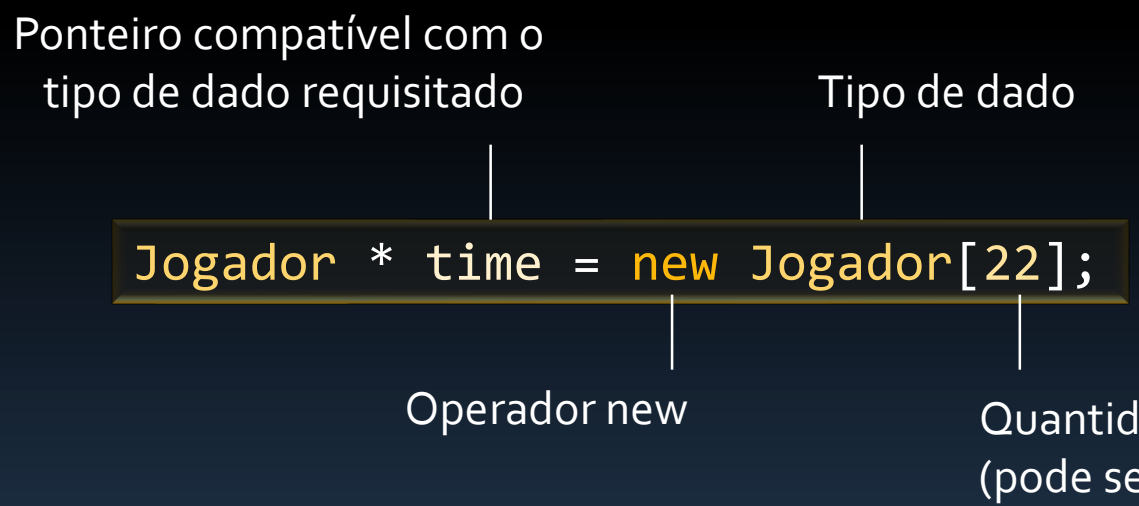
```
Digite nome, salário e gols do jogador:
Bebeto 200000 600
Contratação para o próximo ano:
Bebeto por 200000 Reais
```

- Atribuição aos campos de um registro dinâmico:

```
Jogador * prom = new Jogador;

strcpy(prom->nome, "Romario");
prom->salario = 300'000;
prom->gols = 800;
```

# Vetores Dinâmicos

- O operador new também pode ser usado para criar vetores dinâmicos de registros




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

# Vetores Dinâmicos

- O operador (.) deve ser usado com registros

```
cout << time[0].nome;       // nome do primeiro jogador
cout << time[1].salario;    // salário do segundo jogador
cout << time[21].altura;    // altura do último jogador
```

- O operador (->) deve ser usado com ponteiros

```
cout << time->nome;          // nome do primeiro jogador
cout << (time+1)->salario;   // salário do segundo jogador
cout << (time+21)->altura;   // altura do último jogador
```

# Vetores Dinâmicos

```cpp
#include <iostream>
using namespace std;
struct Jogador
{
    char nome[20];
    float salario;
    unsigned gols;
};
int main()
{
    Jogador * time = new Jogador[22];

    cout << "Digite o nome, salário e gols de dois jogadores:\n";
    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 << "Custo da aquisição: R$" << time[0].salario + time[1].salario << "!\n";

    delete [] time;
}
```

VetDin.cpp

# Vetores Dinâmicos

- Saída do programa:

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

Custo da aquisição: R$500000!
```

- O delete de vetores dinâmicos deve usar colchetes

```
delete [] time; // delete de vetor
```

# Resumo

- Ponteiros são variáveis que armazenam endereços de memória

- A sua principal função é guardar o endereço de memória alocada dinamicamente com o operador new
  - Permite alocar memória durante a execução
  - Permite criar registros dinâmicos
  - Permite criar vetores dinâmicos