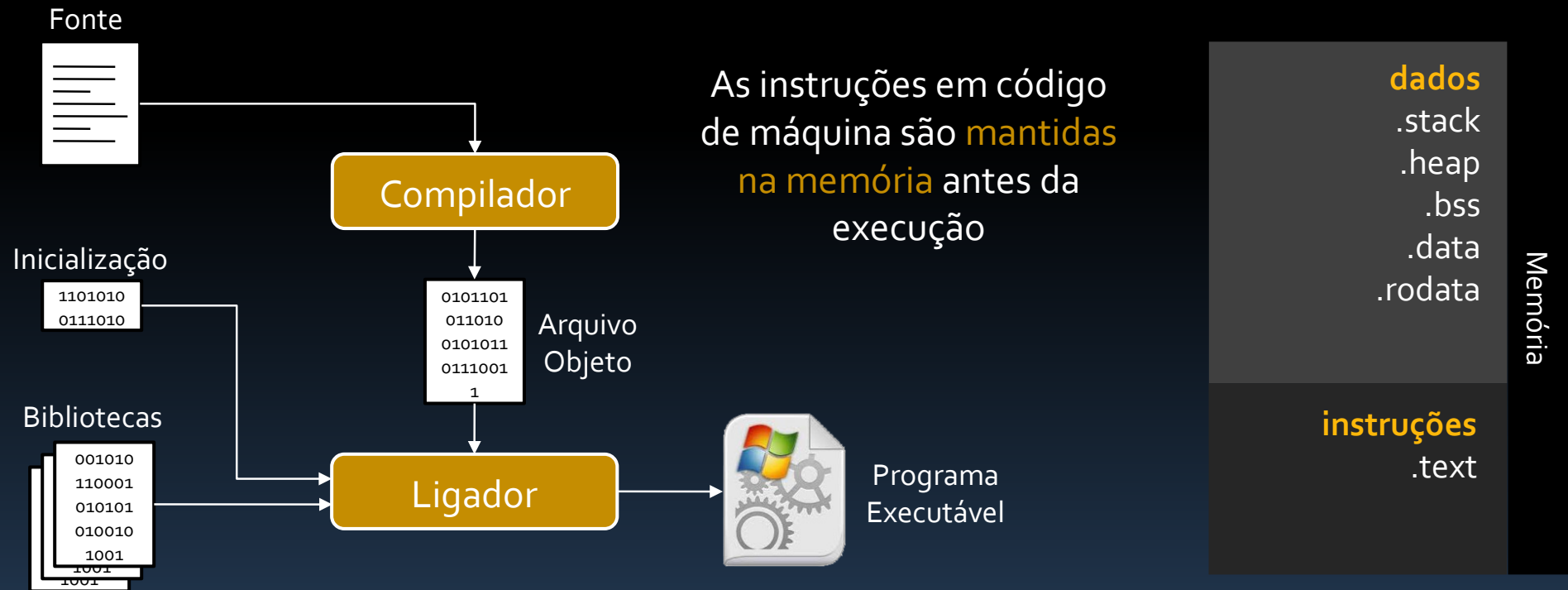


Programação de Computadores

FUNÇÕES INLINE E PONTEIROS PARA FUNÇÕES

Introdução

- Todo programa precisa ser **traduzido em código de máquina**



Introdução

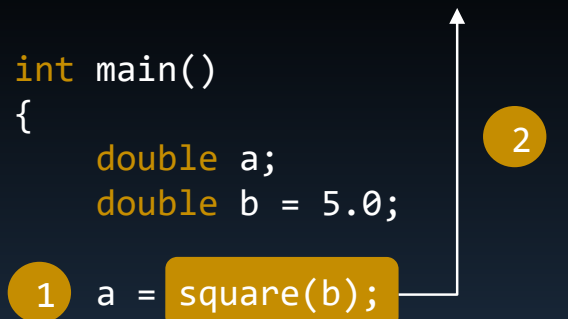
- A chamada de uma função implica na execução de várias tarefas:

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

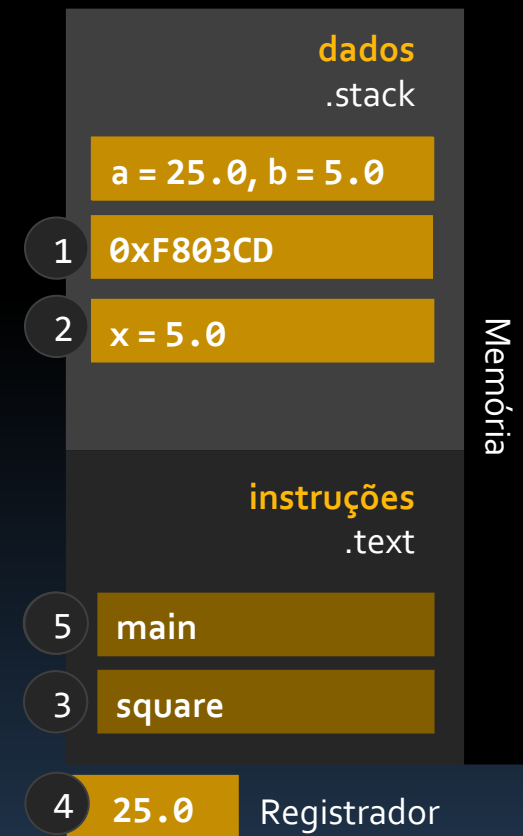
double square(double x);

int main()
{
    double a;
    double b = 5.0;

    1 a = square(b);
    cout << "a = " << endl;
    ...
}
```



- 1 Armazenar o endereço da próxima instrução
- 2 Copiar os argumentos da função para a pilha
- 3 Pular para o endereço de início da função e executá-la
- 4 Colocar o valor de retorno em um registrador
- 5 Pular para o endereço previamente armazenado



Introdução

- Esse processo de **chamada da função** possui um **custo** que cresce com o número de chamadas

```
int main()
{
    linha('-', 2); ➡
    ...
    linha('*', 4); ➡
    ...
    linha('=', 8); ➡
}
```

```
void linha(char ch, int n)
{
    for (int i=0; i < n; ++i)
        cout << ch;
}
```

Funções Inline

- Funções inline tornam o código mais rápido
 - O compilador incorpora as funções inline no código executável
 - A chamada é substituída pelo código da função

```
int main()
{
    linha('-', 2);
    ...
    linha('*', 4);
    ...
    linha('=', 8);
}
```



```
int main()
{
    for (int i=0; i < 2; ++i)
        cout << '-';
    ...
    for (int i=0; i < 4; ++i)
        cout << '*';
    ...
    for (int i=0; i < 8; ++i)
        cout << '=';
}
```

Funções Inline

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

inline double square(double x) { return x * x; }

int main()
{
    double a, b;
    double c = 13.0;

    a = square(5.0);
    b = square(4.5 + 7.5);
    cout << "a = " << a << ", b = " << b << endl;
    cout << "c = " << c;
    cout << ", c quadrado = " << square(c++) << endl;
    cout << "Agora c = " << c << endl;
}
```

Funções Inline

- A saída do programa é:

```
a = 25, b = 144  
c = 13, c quadrado = 169  
Agora c = 14
```

- Funções inline **se comportam como funções normais:**
 - Expressões são avaliadas antes da passagem de parâmetros
 - Passagem é feita por valor (cópia)

Funções Inline

- O uso de funções inline deve ser bem estudado:
 - Provocam o crescimento do código
 - O que implica em maior uso de memória
 - Não vale a pena usar para funções:
 - Complexas: o custo da chamada é insignificante comparado ao tempo de execução da função
 - Pouco usadas: o ganho do programa é pequeno
 - Se a função não couber em uma linha, ela provavelmente não é boa candidata a ser inline

Funções Inline

- Funções inline existem apenas no C++
 - A linguagem C usa macros para implementar uma funcionalidade semelhante

```
#define SQUARE(X) X*X
```

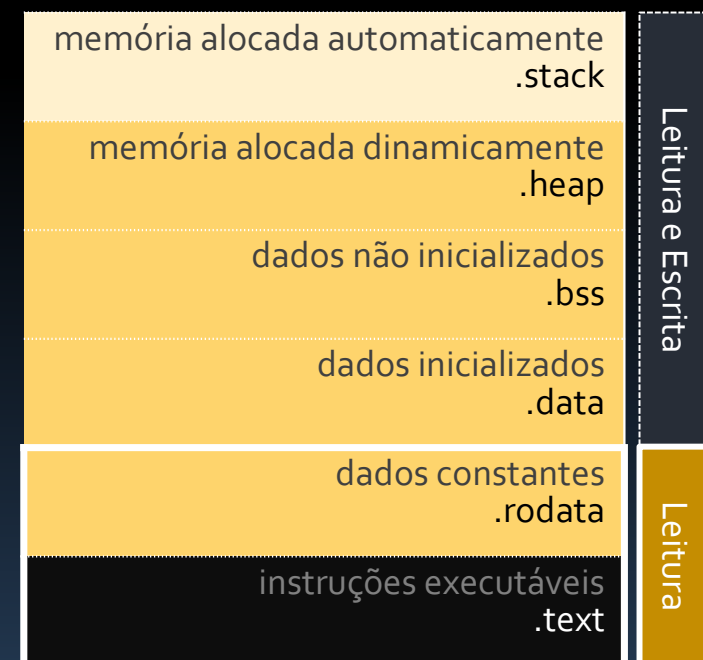
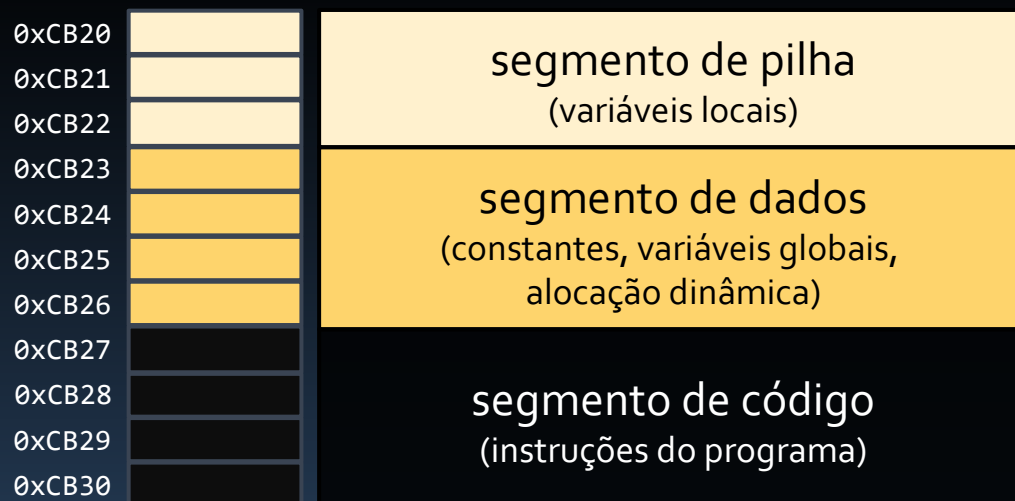
- Mas macros fazem apenas substituições de texto

```
a = SQUARE(5.0);           // a = 5.0*5.0; ✓  
b = SQUARE(4.5 + 7.5);     // b = 4.5 + 7.5*4.5 + 7.5; ✗  
c = SQUARE(x++);           // d = x++*x++; ✗
```

- Em C++ recomenda-se utilizar funções inline no lugar das antigas macros utilizadas na linguagem C

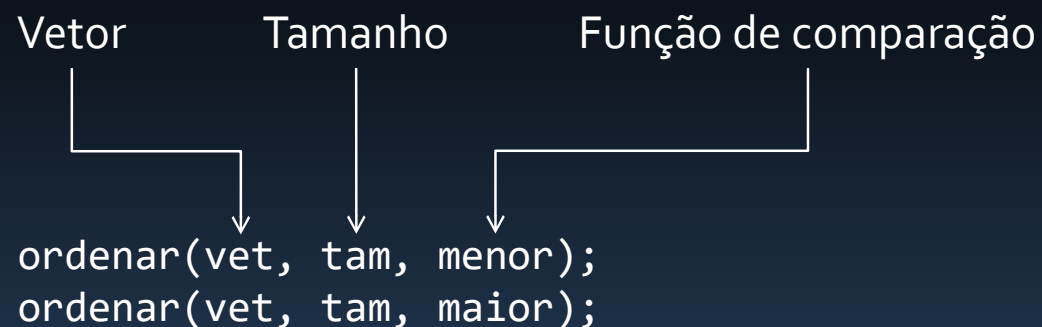
Ponteiros para Funções

- Assim como constantes e variáveis, **funções têm endereços**
 - O endereço de uma função é o **endereço inicial de memória do código** de máquina da função



Ponteiros para Funções

- Qual a **utilidade** do endereço de uma função?
 - Passar o endereço de uma função para outra
 - Permite que uma função chame outra
 - ▮ Isso já pode ser feito com uma chamada convencional!?
 - Permite **passar endereços de funções diferentes** em cada chamada



Ponteiros para Funções

- Para obter o endereço de uma função basta usar o seu nome sem parênteses

```
int pensar(void);      // protótipo da função pensar  
  
visualizar(pensar);    // passa o endereço de pensar()  
extrapolar(pensar()); // passa o valor de retorno de pensar()
```

- A função visualizar() recebe o endereço da função pensar()
- A função extrapolar() recebe o retorno da função pensar()

Ponteiros para Funções

- Um ponteiro sempre indica o **tipo de dado** apontado

```
int * ptr;          // ponteiro para inteiro
```

- Da mesma forma, um ponteiro para uma função precisa explicitar o **tipo da função**

- Tipo da assinatura (argumentos da função)
- Tipo de retorno

```
double chute(int);    // protótipo da função chute
```

```
double (*pf)(int);    // ponteiro para função tipo chute
```

Ponteiros para Funções

- Dica para criar um ponteiro para uma função:
Crie o protótipo e depois substitua o nome da função

```
double chute(int);      // protótipo da função chute  
double (*pf)(int);      // ponteiro para função tipo chute
```

- É necessário usar parênteses devido a precedência

```
double (*pf) (int);      // pf aponta para uma função  
                        // que retorna um valor double  
  
double *pf (int);        // pf é uma função que retorna  
                        // um ponteiro para double
```

Ponteiros para Funções

- Um ponteiro para uma função pode receber o endereço de uma função compatível

```
double chute(int);           // protótipo da função chute
double (*pf)(int);           // ponteiro para função tipo chute

pf = chute;                  // pf aponta para a função chute
```

- Atribuição incompatíveis são rejeitadas na compilação

```
double toque(double);        // protótipo da função toque
int passe(int);              // protótipo da função passe
double (*pf) (int);          // ponteiro para função
pf = toque; x                 // inválido - assinatura
pf = passe; x                 // inválido - tipo de retorno
```

Ponteiros para Funções

- Suponha que você queira construir uma função para **estimar o tempo** para escrever linhas de código

```
// protótipo da função estimar  
void estimar(int linhas, double (*pf)(int));
```

- A função **estimar recebe uma função** como segundo argumento

```
// chamada da função estimar  
estimar(50, chute);
```


Chamando Função com Ponteiro

- Para **invocar uma função através de um ponteiro** basta usar o ponteiro como nome da função

```
double chute(int);           // protótipo da função chute
double (*pf)(int);           // ponteiro para função
pf = chute;                   // pf aponta para chute
```

```
double x = chute(4);          // chamada com chute
double y = pf(4);              // chamada com o ponteiro pf
```

- Também é possível usar **(*pf)** como nome da função

```
double y = (*pf)(4);          // chamada com o ponteiro pf
```

Chamando Função com Ponteiro

- Como pode **pf** e **(*pf)** serem **equivalentes**?
 1. Como pf é um ponteiro para uma função, *pf é uma função, e deve-se usar (*pf)
 2. Como o nome de uma função é um ponteiro, um ponteiro para uma função deve agir como o nome dela, e deve-se usar pf
- **C++ considera as duas formas corretas** mesmo que elas sejam logicamente inconsistentes:
 - *pf resulta no endereço da função, ou seja, em pf

Exemplo de Aplicação

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

double tom(int);
double pam(int);

void estimar(int linhas, double (*pf)(int));

int main()
{
    cout << "Quantas linhas de código você precisa? ";
    int code;
    cin >> code;
    cout << "Estimativa de Tom:\n";
    estimar(code, tom);
    cout << "Estimativa de Pam:\n";
    estimar(code, pam);
    return 0;
}
```

Exemplo de Aplicação

```
double tom(int lns)
{
    return 0.05 * lns;
}

double pam(int lns)
{
    return 0.03 * lns + 0.0004 * lns * lns;
}

void estimar(int linhas, double (*pf)(int))
{
    cout << linhas << " linhas levam ";
    cout << pf(linhas) << " hora(s)\n";
}
```

Exemplo de Aplicação

- Saída do Programa:

Quantas linhas de código você precisa? 100

Estimativa de Tom:

100 linhas levam 5 hora(s)

Estimativa de Pam:

100 linhas levam 7 hora(s)

- As funções tom() e pam() são compatíveis:

```
double tom(int);
```

```
double pam(int);
```

```
void estimar(int linhas, double (*pf)(int));
```

Mais Ponteiros para Funções

- As funções abaixo **compartilham** a mesma **assinatura e tipo de retorno**:

```
const double * f1(const double vet[], int n);  
const double * f2(const double [], int n);  
const double * f3(const double *, int n);
```

- Podemos usar o mesmo ponteiro para apontar para qualquer uma das funções

```
const double * (*pf)(const double vet[], int n);
```

Mais Ponteiros para Funções

- Um **ponteiro pode ser inicializado** para um endereço de uma função

```
const double * f1(const double vet[], int n);  
const double * f2(const double [], int n);  
const double * f3(const double *, int n);
```

```
// inicializa p1 para a função f1  
const double * (*p1)(const double vet[], int n) = f1;
```

- **auto** simplifica a inicialização

```
// inicializa p2 para a função f2  
auto p2 = f2;
```

Mais Ponteiros para Funções

- As chamadas das funções podem ser feitas assim:

```
const double * f1(const double vet[], int n);  
const double * (*p2)(const double vet[], int n) = f2;  
auto p3 = f3;
```

```
cout << f1(v,5)      << ": " << *f1(v,5)      << endl;  
cout << (*p2)(v,5)   << ": " << *(*p2)(v,5)   << endl;  
cout << p3(v,5)      << ": " << *p3(v,5)      << endl;
```

- `(*p2)(v,5)` e `p3(v,5)` são chamadas das funções `f2` e `f3` usando os ponteiros `p2` e `p3`
- Como o retorno da função é do tipo `double *` então ambos `*(*p2)(v,5)` e `*p3(v,5)` resultam em um valor `double`

Mais Ponteiros para Funções

- Um **vetor de ponteiros** poderia ser construído para trabalhar com as 3 funções

```
const double * f1(const double vet[], int n);  
const double * f2(const double [], int n);  
const double * f3(const double *, int n);  
  
const double * (*pv[3])(const double*,int n) = {f1,f2,f3};
```

- Cada elemento de pv é um ponteiro para função

```
const double * px = pv[0](av,5);  
const double * py = (*pv[1])(av,5);
```

Simplificando com typedef

- C++ fornece outras ferramentas, além do auto, para **simplificar declarações**

```
typedef double real;  
typedef unsigned short ushort;
```

- Esta técnica pode ser usada com ponteiros para funções

```
typedef const double * (*fpointer)(const double *, int);  
fpointer p1 = f1;  
fpointer func[3] = {f1, f2, f3};
```

Resumo

- Uma **função em C++** pode ser marcada como **inline**
 - Substitui a chamada da função pelo seu conteúdo
 - Ideal para funções **pequenas** que **se repetem** muito
 - Especialmente útil para o caminho crítico de um código
- **Ponteiros para funções** podem ser usados para mudar o comportamento da função sem ter que reescrevê-la
 - Muito usado na biblioteca STL do C++
 - Funções de ordenação, busca, mínimo, máximo, etc.