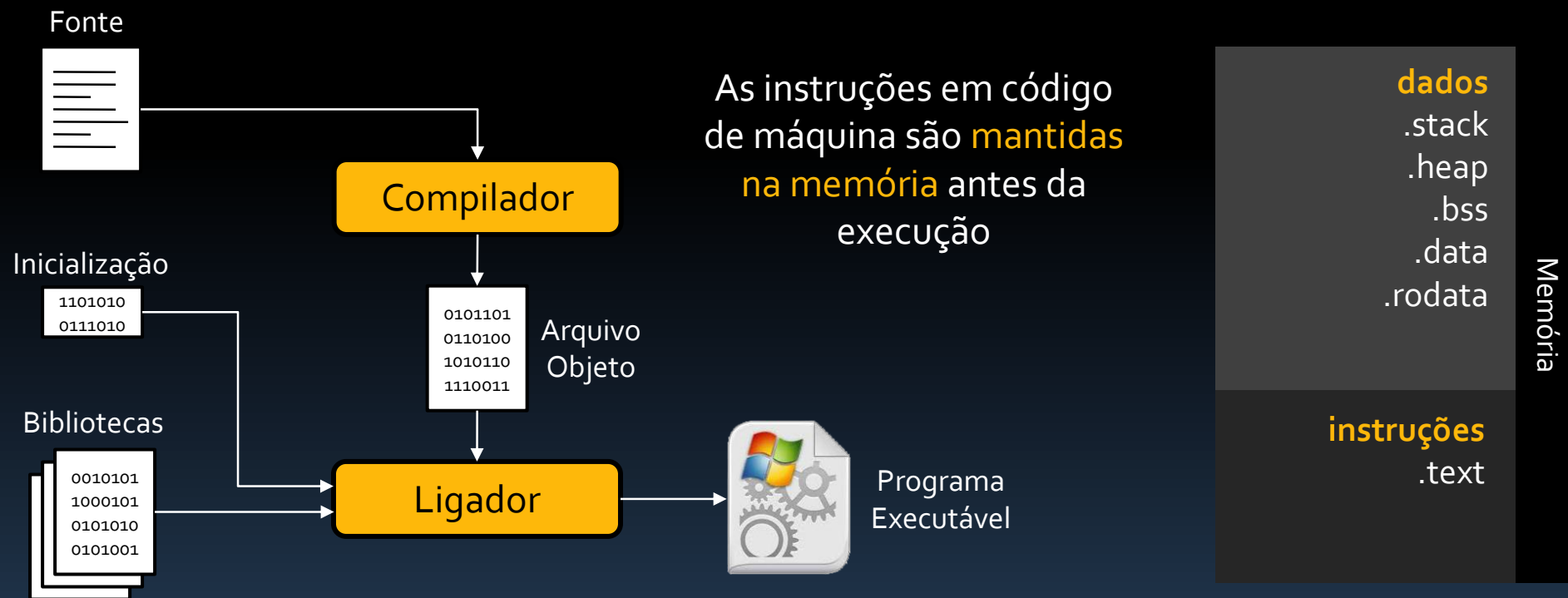


Programação de Computadores

# FUNÇÕES INLINE E PONTEIROS PARA FUNÇÕES

# Introdução

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



# Introdução

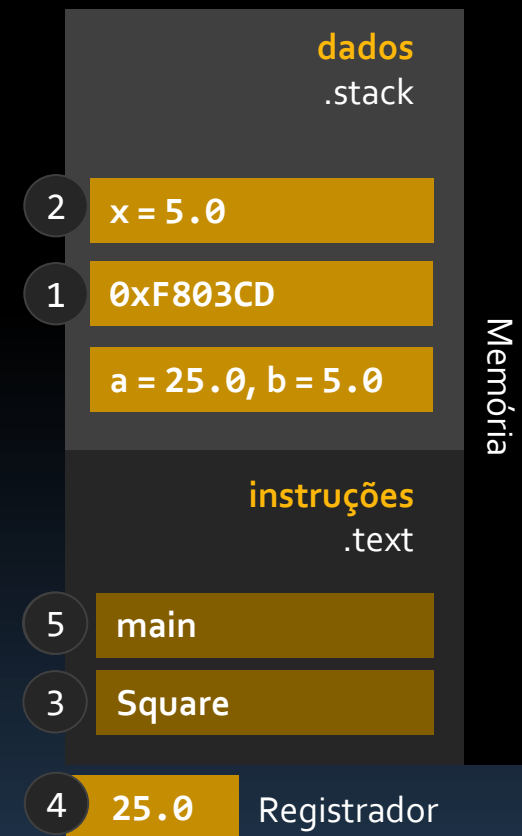
- A chamada de uma função implica na execução de várias tarefas:

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

double Square(double x);

int main()
{
    double a = 0.0;
    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 o corpo da função 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
```

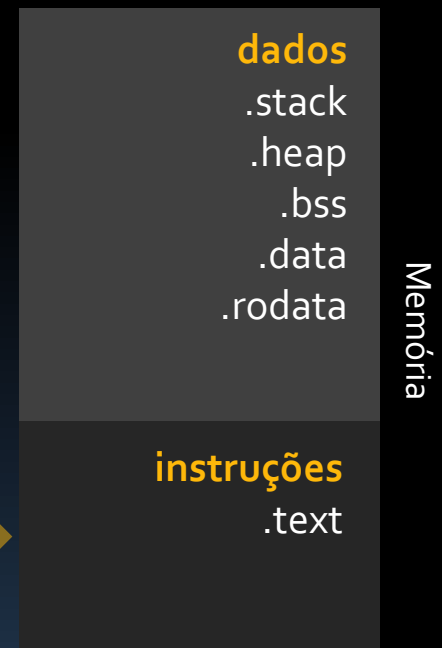
- Funções inline **se comportam como funções normais**:
  - Expressões são avaliadas antes da passagem de argumentos
  - Passagem é feita por valor (cópia)

# Funções Inline

- O **uso de funções inline** deve ser bem estudado:
  - Provocam o crescimento do código
    - O que implica em **maior uso de memória**

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

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



# Funções Inline

- O uso de funções inline deve ser bem estudado:
  - 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

```
inline double Square(double x) { return x * x; } ✓
```

# Funções Inline

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

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

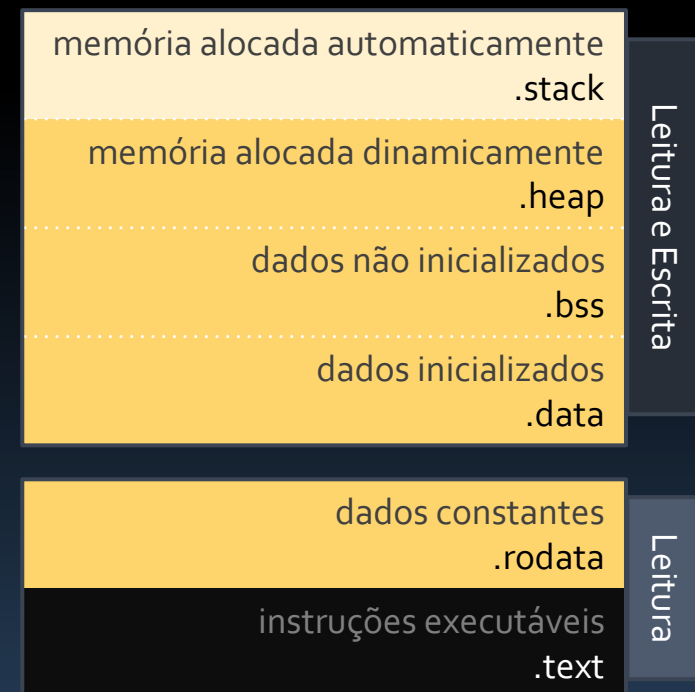
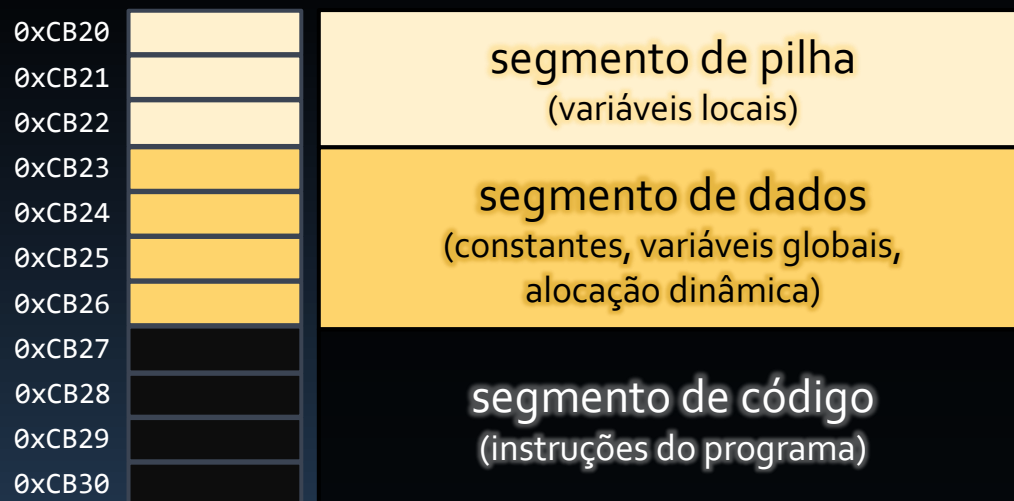
- Mas macros fazem apenas substituições de texto

```
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
      - Calma ai! Isso já pode ser feito com uma chamada comum!
      - 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 indica o **tipo de dado** apontado

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

- Um ponteiro para uma função indica o **tipo da função**

- Tipo da função consiste no:

- Tipo da assinatura (**dos parâmetros**)
- Tipo de retorno

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

# Ponteiros para Funções

- 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  
void Estimar(int linhas, double (*pf)(int));
```

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

```
// chamada da função  
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 semanticamente inconsistentes:
  - **\*pf** resulta no endereço da função, ou seja, em **pf**

# Exemplo de Aplicação

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

double Tom(int);
double Bia(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 Ana:\n";
    Estimar(code, Bia);
}
```

# Exemplo de Aplicação

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

double Bia(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 Bia:

100 linhas levam 7 hora(s)

- As funções Tom( ) e Bia( ) são compatíveis:

```
double Tom(int);
```

```
double Bia(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](v,5);  
const double * py = pv[1](v,5);  
const double * pz = pv[2](v,5);
```

# Simplificando com typedef

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

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

- Essa técnica pode ser usada com ponteiros para funções

```
typedef const double * (*fpointer)(const double *, int);  
fpointer p1 = F1;  
fpointer func[3] = {F1, F2, F3};
```

# Usos Avançados

- **Ponteiros para funções** podem ser usados para:
  - Eliminar o uso de condicionais

```
float Gerente(float v)
{ return 0.10 * v; }
```

```
float Assistente(float v)
{ return 0.05 * v; }
```

A função **Desconto** não  
precisar testar o tipo de  
funcionário

```
Desconto(1000, Gerente);    // aplica desconto de gerente
Desconto(1000, Assistente); // aplica desconto de assistente
```

# Usos Avançados

- **Ponteiros para funções** podem ser usados para:
  - Implementar menus

```
void Adicionar();  
void Excluir();  
void Listar();  
void Sair();
```

```
void (*menu[])(void) = { Adicionar, Excluir, Listar, Sair };
```

```
cout << " 1. Adicionar\n 2. Excluir\n 3. Listar\n 4. Sair\n";  
cout << " Digite sua opção: ";  
int escolha;  
cin >> escolha;  
menu[escolha-1]();
```

# 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
  - O compilador tem a palavra final
- **Ponteiros para funções** podem ser usados para mudar o comportamento da função sem ter que reescrevê-la
  - Muito usado na **biblioteca STL** do C++
    - Ordenação, busca, mínimo, máximo, etc.