Programação Orientada a Objetos para Redes de Computadores

Prof. Miguel Elias Mitre Campista

http://www.gta.ufrj.br/~miguel

PARTE 2

Programação em C++ - Funções

Protótipo de uma Função e Coerção de Argumentos

- Protótipo de função
  - Também chamado de declaração de função
  - Indica ao compilador:
    - O nome da função
    - O tipo de dados retornado à função
    - Os parâmetros que a função espera receber
      - O número de parâmetros
      - Os tipos de parâmetros
      - A ordem desses parâmetros

```
int soma(int a , int b);
```


Protótipo de uma Função e Coerção de Argumentos

- Assinatura de função (ou simplesmente assinatura)
  - Parte de um protótipo de função que inclui o nome da função e os respectivos tipos de parâmetros
    - Não especifica o tipo de retorno da função
  - As funções no mesmo escopo devem ter assinaturas exclusivas
    - O escopo de uma função é a região de um programa em que a função é conhecida e acessível

```
int soma(int a , int b);
            ≠
int soma(double a, double b);
```


Protótipo de uma Função e Coerção de Argumentos

- Assinatura de função (ou simplesmente assinatura)
  - Parte de um protótipo de função que inclui o nome da função e os respectivos tipos de parâmetros
    - Não especifica o tipo de retorno da função
  - As funções no mesmo escopo devem ter assinaturas exclusivas
    - O escopo de uma função é a região de um programa em que a função é conhecida e acessível

```
int soma(int a , int b);
            ≠
int soma(double a, double b);
```

Assinaturas diferentes

Protótipo de uma Função e Coerção de Argumentos

- Assinatura de função (ou simplesmente assinatura)
  - É um erro de compilação se duas funções do mesmo escopo tiverem a mesma assinatura, mas diferentes tipos de retorno

```
int soma(int a , int b);
            =
void soma(int a, int b);
```

Assinaturas iguais

Protótipo de uma Função e Coerção de Argumentos

- Coerção de argumentos
  - Forçar argumentos aos tipos apropriados especificados pelos parâmetros correspondentes
    - Por exemplo, chamar uma função com um argumento inteiro, mesmo que o protótipo da função especifique um argumento `double`
      - A função ainda assim continuará a funcionar corretamente

```
int a, b;
double soma(double a , double b);
```


Protótipo de uma Função e Coerção de Argumentos

- Regras de promoção C++
  - Indicam como converter de um tipo para outro sem perder dados
  - Aplicam-se a expressões (p.ex. uma equação) que contenham valores de dois ou mais tipos de dados
    - Expressões desse tipo são também referidas como expressões de tipo misto
    - Cada valor é promovido ao tipo "mais alto" na expressão
      - Uma versão temporária de cada valor é criada e utilizada para a expressão
        - » Os valores originais permanecem inalterados

Protótipo de uma Função e Coerção de Argumentos

- Regras de promoção C++
  - A promoção ocorre também quando o tipo de um argumento de função não corresponde ao tipo de parâmetro especificado
    - A promoção é como se o valor do argumento tivesse sido atribuído diretamente ao tipo do parâmetro
- A conversão de um valor em um tipo mais baixo
  - Pode provocar a perda de dados ou valores incorretos
  - Só pode ser executada explicitamente
    - Atribuindo o valor a uma variável ou tipo mais baixo (alguns compiladores emitirão um aviso nesse caso)

Protótipo de uma Função e Coerção de Argumentos

Tipos de dados
long double
double
float
unsigned long int (sinônimo de unsigned long)
long int (sinônimo de long)
unsigned int (sinônimo de unsigned)
int
unsigned short int (sinônimo de unsigned short)
short int (sinônimo de short)
unsigned char
char
bool

Tipo funda-mental mais alto

Protótipo de uma Função e Coerção de Argumentos

- Converter de um tipo de dados mais alto em um tipo mais baixo ou entre com sinal e sem sinal pode…
  - Corromper o valor dos dados, causando perda de informações
- Constitui erro de compilação:
  - Passar argumentos em uma chamada de função que não correspondem ao número e aos tipos de parâmetros declarados no protótipo da função correspondente
  - Passar um número correto de argumentos na chamada da função, mas passar argumentos que não possam ser implicitamente convertidos nos tipos esperados

Classes de Armazenamento

- Cada identificador tem diversos atributos
  - Nome, tipo, tamanho e valor
  - Além disso, classe de armazenamento, escopo e ligação
- C++ oferece cinco especificadores de classe de armazenamento:
  - `auto`, `register`, `extern`, `mutable` e `static`
- Classe de armazenamento do identificador
  - Determina o **período durante o qual esse identificador permanece na memória**
- Escopo do identificador
  - Determina em que **lugar o identificador pode ser referenciado** em um programa

Classes de Armazenamento

- Ligação do identificador
  - **Determina se um identificador é conhecido apenas no arquivo fonte em que é declarado ou nos múltiplos arquivos** que são compilados e depois ligados
- O especificador de classe de armazenamento do identificador ajuda a determinar a respectiva classe de armazenamento e ligação

Classes de Armazenamento

- Classe de armazenamento automática
  - Declarada com as palavras-chave `auto` e `register`
  - Variáveis automáticas
    - Criadas quando a execução do programa entra no bloco em que são definidas
    - Existem enquanto o bloco estiver ativo
    - São destruídas quando o programa sai do bloco
  - Variáveis locais e parâmetros

Classes de Armazenamento

- O armazenamento automático é um meio de economizar memória
  - Variáveis só existem na memória enquanto o bloco em que são definidas estiver executando
- O armazenamento automático é um exemplo do princípio do menor privilégio, que é fundamental para a engenharia de software
  - O código deve possuir somente a quantidade de privilégio e acesso que precisa para realizar a tarefa designada, não mais que isso

Classes de Armazenamento

- Especificador de classe de armazenamento `auto`
  - Declara explicitamente variáveis da classe de armazenamento automática
  - Variáveis locais são da classe de armazenamento automática por padrão
    - Portanto, a palavra-chave `auto` raramente é utilizada

Classes de Armazenamento

- Especificador de classe de armazenamento `register`
  - Dados na versão de linguagem de máquina de um programa normalmente são carregados em registradores para a execução de cálculos e outros tipos de processamento
    - O compilador tenta armazenar variáveis da classe de armazenamento automática em um registrador
  - Existe a possibilidade do compilador ignorar declarações `register`
    - Talvez não haja registradores suficientes para o compilador usar

Classes de Armazenamento

- O especificador de classe de armazenamento `register` pode ser colocado antes de uma declaração de variável automática
  - Sugere que o compilador mantenha a variável em um dos registradores de hardware de alta velocidade do computador, ao invés de na memória
  - Variáveis intensamente utilizadas como contadores podem ser mantidas em registradores de hardware
    - Elimina a sobrecarga de carregamento repetida das variáveis da memória nos registradores e de armazenamento dos resultados de volta na memória

Classes de Armazenamento

- É um erro de sintaxe utilizar múltiplos especificadores de classe de armazenamento para um único identificador
  - Ex.: `auto register int var;`
- Frequentemente, a palavra-chave `register` é desnecessária
  - Compiladores atuais reconhecem as variáveis utilizadas e podem decidir colocá-las em registradores sem precisar de uma declaração `register` do programador

Classes de Armazenamento

- Classe de armazenamento estática
  - Declarada com as palavras-chave `extern` e `static`
  - Variáveis da classe de armazenamento estática
    - Existem desde o início da execução do programa
    - São inicializadas assim que as declarações são encontradas
    - Duram enquanto o programa estiver executando
  - Funções da classe de armazenamento estática
    - O nome da função existe quando o programa começa a execução, assim como para todas as outras funções

Classes de Armazenamento

- Classe de armazenamento estática
  - Mesmo que as variáveis e os nomes de função existam desde o início da execução do programa
    - Não significa que esses identificadores podem ser utilizados durante todo o programa

Classes de Armazenamento

- Dois tipos de identificadores com classe de armazenamento estática
  - **Variáveis e funções globais**
    - Podem ser declaradas em arquivos diferentes e mesmo assim manter o escopo
      - Declaradas com o especificador de classe de armazenamento `extern`
  - **Variáveis locais**
    - Podem ser declaradas com o especificador de classe de armazenamento `static`

Classes de Armazenamento

- Variáveis globais
  - São criadas inserindo-se declarações fora da definição de qualquer classe ou função
  - Retêm seus valores enquanto o programa estiver executando
  - Podem ser referenciadas por qualquer função que siga suas declarações ou definições no arquivo de fonte

Classes de Armazenamento

- Variáveis globais
  - Podem provocar efeitos colaterais indesejáveis quando uma função que não precisa de acesso à variável a modifica acidental ou maliciosamente
    - Exceto por recursos verdadeiramente globais, como `cin` e `cout`, o uso de variáveis globais deve ser evitado
      - A não ser em certas situações em que haja requisitos de desempenho exclusivos

Classes de Armazenamento

externVariable.cpp

```
// declaração de g_nValue
int g_nValue = 5;
```

mainVariable.cpp

```
#include <iostream>

using namespace std;

int g_nValue;

int main () {
        cout << "g_nValue = " << g_nValue << endl;

        g_nValue = 7;

        cout << "g_nValue = " << g_nValue << endl;

        return 0;
}
```

O que acontece com esse programa se for compilado como:
`g++ -Wall externVariable.cpp mainVariable.cpp -o e`
?

Classes de Armazenamento

externVariable.cpp

```
// declaração de g_nValue
int g_nValue = 5;
```

mainVariable.cpp

```
#include <iostream>

using namespace std;

int g_nValue;

int main () {
        cout << "g_nValue = " << g_nValue << endl;

        g_nValue = 7;

        cout << "g_nValue = " << g_nValue << endl;

        return 0;
}
```

```
miguel@pegasus-linux:~$ g++ -Wall externVariable.cpp externMain.cpp -o e
/tmp/ccqoC3t3.o:(.bss+0x0): multiple definition of `g_nValue'
/tmp/ccopYI6V.o:(.data+0x0): first defined here
collect2: ld returned 1 exit status
```


Classes de Armazenamento

externVariable.cpp

```
// declaração de g_nValue
int g_nValue = 5;
```

mainVariable.cpp

```
// extern diz ao compilador que essa variável está
//declarada em outro lugar
#include <iostream>

using namespace std;

extern int g_nValue;

int main () {
        cout << "g_nValue = " << g_nValue << endl;

        g_nValue = 7;

        cout << "g_nValue = " << g_nValue << endl;

        return 0;
}
```

E agora se for compilado da mesma forma como:
`g++ -Wall externVariable.cpp mainVariable.cpp -o e`
?

Classes de Armazenamento

externVariable.cpp

```
// declaração de g_nValue
int g_nValue = 5;
```

mainVariable.cpp

```
// extern diz ao compilador que essa variável está
//declarada em outro lugar
#include <iostream>

using namespace std;

extern int g_nValue;

int main () {
        cout << "g_nValue = " << g_nValue << endl;

        g_nValue = 7;

        cout << "g_nValue = " << g_nValue << endl;

        return 0;
}
```

```
miguel@pegasus-linux:~$ g++ -Wall externVariable.cpp externMain.cpp -o e
miguel@pegasus-linux:~$ ./e
g_nValue = 5
g_nValue = 7
```


Classes de Armazenamento

- Variáveis locais declaradas com a palavra-chave **`static`**
  - São conhecidas apenas na função em que são declaradas
  - Mantêm seus valores quando a função retornar ao seu chamador
    - Na próxima vez em que a função for chamada, as variáveis locais `static` conterão os valores de quando a função completou pela última vez
  - São inicializadas em zero
    - Toda vez que as variáveis numéricas da classe de armazenamento estática não forem explicitamente inicializadas pelo programador

Regras de Escopo

- Escopo
  - Parte do programa em que um identificador pode ser utilizado
  - Quatro escopos para um identificador
    - Escopo de função
    - Escopo de arquivo
    - Escopo de bloco
    - Escopo de protótipo de função

Regras de Escopo

- Escopo de arquivo
  - Para um identificador declarado fora de qualquer função ou classe
    - Esse tipo de identificador é "conhecido" em todas as funções desde o momento em que é declarado até o fim do arquivo
  - Variáveis globais, definições de função e protótipos de função colocados fora de uma função, todos têm escopo de arquivo

Regras de Escopo

- Escopo de função
  - Os rótulos (identificadores seguidos por dois-pontos como `goto:`) são os únicos identificadores com escopo de função
    - Podem ser utilizados em qualquer lugar na função em que aparecem
    - Não podem ser referenciados fora do corpo da função
    - Os rótulos são detalhes de implementação que as funções ocultam entre si
    - Um rótulo inicia a sua "área de influência"

Regras de Escopo

- Escopo de bloco
  - Identificadores declarados dentro de um bloco têm escopo de bloco
    - O escopo de bloco inicia-se na declaração do identificador
    - O escopo de bloco finaliza na chave de fechamento direita (`}`) do bloco no qual o identificador é declarado
  - As variáveis locais e os parâmetros de função têm escopo de bloco
    - O corpo da função é o bloco dessas variáveis
  - Qualquer bloco pode conter declarações de variáveis

Regras de Escopo

- Escopo de bloco
  - Os identificadores em um bloco externo podem ser "ocultados" quando um bloco aninhado tiver um identificador local com o mesmo nome
  - As variáveis locais declaradas `static` continuam tendo escopo de bloco, mesmo que existam desde o momento em que o programa inicia a execução
    - A duração do armazenamento não influi no escopo de um identificador

Regras de Escopo

- Escopo de protótipo de função
  - Somente os **identificadores usados na lista de parâmetros de um protótipo de função** têm escopo de protótipo de função
  - Os nomes de parâmetro que aparecem em um protótipo de função são ignorados pelo compilador
    - Os identificadores utilizados em um protótipo de função podem ser reutilizados em qualquer lugar no programa, sem ambiguidades
    - Entretanto, em um único protótipo, um determinado identificador pode ser utilizado apenas uma vez

Regras de Escopo

- Pode ser um erro de lógica utilizar o mesmo nome de um identificador em um bloco interno utilizado para um identificador em um bloco externo
  - O erro ocorre quando o programador quer que o identificador no bloco externo esteja ativo no bloco interno
- Evite nomes de variáveis que ocultem nomes de escopos externos
  - Isso pode ser conseguido evitando-se usar identificadores duplicados em um programa

```cpp
#include <iostream>

using namespace std;

// Protótipos de função
void useLocal();
void useStaticLocal();
void useGlobal();

int x = 1; // Variável global

int main() {
    int x = 5; // Variável local de main

    cout << "Variavel local x no escopo mais externo da main eh: "
         << x << endl;

    { // Inicio de novo escopo
        int x = 7;
        cout << "Variavel local x no escopo mais interno da main eh: "
             << x << endl;
    }

    cout << "Variavel local x no escopo mais externo da main eh: "
         << x << endl;
    cout << "******************************************" << endl;

    useLocal(); // useLocal tem uma variável local x
    useStaticLocal(); // useStaticLocal tem x estático local
    useGlobal(); // useGlobal usa x global
    useLocal(); // useLocal reinicializa o seu x local
    useStaticLocal(); // useStaticLocal retém o valor anterior do seu x
    useGlobal(); // useGlobal também retém o valor do seu x

    cout << "\nVariavel local x em main eh: " << x << endl;

    return 0;
}
```

```cpp
#include <iostream>

using namespace std;

// Protótipos de função
void useLocal();
void useStaticLocal();
void useGlobal();

int x = 1; // Variável global

int main() {
    int x = 5; // Variável local de main

    cout << "Variavel local x no escopo mais externo da main eh: "
         << x << endl;

    { // Inicio de novo escopo
        int x = 7;
        cout << "Variavel local x no escopo mais interno da main eh: "
             << x << endl;
    }

    cout << "Variavel local x no escopo mais externo da main eh: "
         << x << endl;
    cout << "******************************************" << endl;

    useLocal(); // useLocal tem uma variável local x
    useStaticLocal(); // useStaticLocal tem x estático local
    useGlobal(); // useGlobal usa x global
    useLocal(); // useLocal reinicializa o seu x local
    useStaticLocal(); // useStaticLocal retém o valor anterior do seu x
    useGlobal(); // useGlobal também retém o valor do seu x

    cout << "\nVariavel local x em main eh: " << x << endl;

    return 0;
}
```

Declaração de uma variável global fora de qualquer classe ou definição de função

```cpp
#include <iostream>

using namespace std;

// Protótipos de função
void useLocal();
void useStaticLocal();
void useGlobal();

int x = 1; // Variável global

int main() {
    int x = 5; // Variável local de main

    cout << "Variavel local x no escopo mais externo da main eh: "
         << x << endl;

    { // Inicio de novo escopo
        int x = 7;
        cout << "Variavel local x no escopo mais interno da main eh: "
             << x << endl;
    }

    cout << "Variavel local x no escopo mais externo da main eh: "
         << x << endl;
    cout << "******************************************" << endl;

    useLocal(); // useLocal tem uma variável local x
    useStaticLocal(); // useStaticLocal tem x estático local
    useGlobal(); // useGlobal usa x global
    useLocal(); // useLocal reinicializa o seu x local
    useStaticLocal(); // useStaticLocal retém o valor anterior do seu x
    useGlobal(); // useGlobal também retém o valor do seu x

    cout << "\nVariavel local x em main eh: " << x << endl;

    return 0;
}
```

Variável local **x** que oculta a variável global **x**

Variável local **x** em um bloco que oculta a variável local **x** no escopo externo

Primeiro Exemplo utilizando Funções em C++

```cpp
void useLocal() {
    int x = 25;
    cout << "\nVariavel local x eh: " << x << " ao entrar em useLocal" << endl;
    x++;
    cout << "Variavel local x eh: " << x << " ao sair de useLocal" << endl;
}

void useStaticLocal() {
    static int x = 50;
    cout << "\nVariavel local estatica x eh: " << x
         << " ao entrar em useStaticLocal" << endl;
    x++;
    cout << "Variavel local estatica x eh: " << x
         << " ao sair de useStaticLocal" << endl;
}

void useGlobal() {
    cout << "\nVariavel global x eh: " << x
         << " ao entrar em useGlobal" << endl;
    x *= 10;
    cout << "Variavel global x eh: " << x
         << " ao sair de useGlobal" << endl;
}
```


Primeiro Exemplo utilizando Funções em C++

```cpp
void useLocal() {
    int x = 25;
    cout << "\nVariavel local x eh: " << x << " ao entrar em useLocal" << endl;
    x++;
    cout << "Variavel local x eh: " << x << " ao sair de useLocal" << endl;
}

void useStaticLocal() {
    static int x = 50;
    cout << "\nVariavel local estatica x eh: " << x
         << " ao entrar em useStaticLocal" << endl;
    x++;
    cout << "Variavel local estatica x eh: " << x
         << " ao sair de useStaticLocal" << endl;
}

void useGlobal() {
    cout << "\nVariavel global x eh: " << x
         << " ao entrar em useGlobal" << endl;
    x *= 10;
    cout << "Variavel global x eh: " << x
         << " ao sair de useGlobal" << endl;
}
```

Variável local que é recriada e reinicializada toda vez que **useLocal** é chamada

Variável local **static** que inicializa apenas uma vez

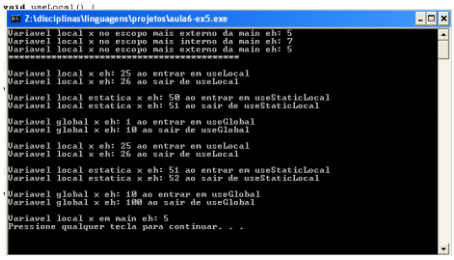
Primeiro Exemplo utilizando Funções em C++

```cpp
void useLocal() {
    int x = 25;
    cout << "\nVariavel local x eh: " << x
    x++;
    cout << "Variavel local x eh: " << x
}

void useStaticLocal() {
    static int x = 50;
    cout << "\nVariavel local estatica x eh: " << x
         << " ao entrar em useStaticLocal" << endl;
    x++;
    cout << "Variavel local estatica x eh: " << x
         << " ao sair de useStaticLocal" << endl;
}

void useGlobal() {
    cout << "\nVariavel global x eh: " << x
         << " ao entrar em useGlobal" << endl;
    x *= 10;
    cout << "Variavel global x eh: " << x
         << " ao sair de useGlobal" << endl;
}
```

A sentença refere-se à variável global **x** porque não existe nenhuma variável local denominada **x**

Primeiro Exemplo utilizando Funções em C++

```
void useLocal() {
```




```
}
```


Pilha de Chamadas de Função e Registros de Ativação

- Estrutura de dados
  - **Coleção de itens de dados relacionados**
- Estrutura de dados em pilha
  - Estrutura de dados último a entrar, primeiro a sair (*Last-In First-Out* - LIFO)
    - O último item colocado (inserido) na pilha é o primeiro item retirado da pilha

Pilha de Chamadas de Função e Registros de Ativação

- Pilha de chamadas de função
  - Às vezes denominada pilha de execução do programa
  - Suporta o mecanismo de chamada/retorno de função
    - Sempre que uma função chama outra função, um quadro de pilhas (ou registro de ativação) é inserido na pilha
      - Mantém o endereço de retorno que a função chamada precisa para retornar à função chamadora
      - Contém variáveis automáticas: Parâmetros e variáveis locais que a função declara

Pilha de Chamadas de Função e Registros de Ativação

- Pilha de chamadas de função
  - Quando a função chamada retorna
    - O quadro de pilha de chamadas de função é retirado
    - O controle é transferido ao endereço de retorno no quadro de pilha removido
  - Se uma função fizer uma chamada a outra função
    - O quadro de pilha da nova chamada de função simplesmente é inserido na pilha de chamadas
    - O endereço de retorno à nova função chamadora será localizado na parte superior da pilha

Pilha de Chamadas de Função e Registros de Ativação

- Estouro de pilha
  - Erro que ocorre quando há mais chamadas de função do que as que podem ter seus registros de ativação armazenados na pilha de chamadas de função
    - Decorrência de restrições de memória

Segundo Exemplo utilizando Funções em C++

```cpp
/*
 * Aula 6 - Exemplo 6
 * Autor: Miguel Campista
*/
#include <iostream>

using namespace std;

int square(int ); // protótipo da função square

int main() {
    int x = 10;

    cout << "\t" << x << " elevado ao quadrado eh: " << square(x) << endl;


    return 0;
}

int square(int x) {
    return x * x;
}
```

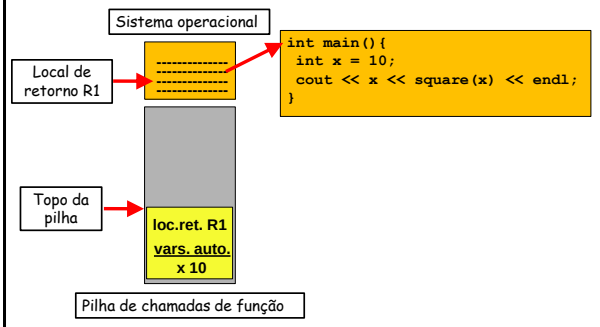
Segundo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 6
 * Autor: Miguel Campista
*/
#include <iostream>

using namespace std;

int square(int ); // protótipo da função square

int main() {
    int x = 10;

    cout << "\t" << x << " elevado ao quadrado eh: " << square(x) << endl;

    return 0;
}

int square(int x) {
    return x * x;
}
```

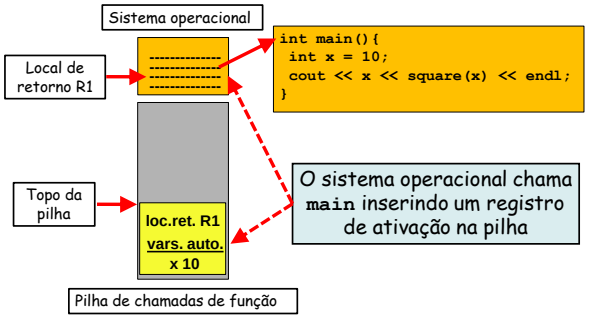
Chamada da função **square**

O que acontece com a pilha de chamadas de função?

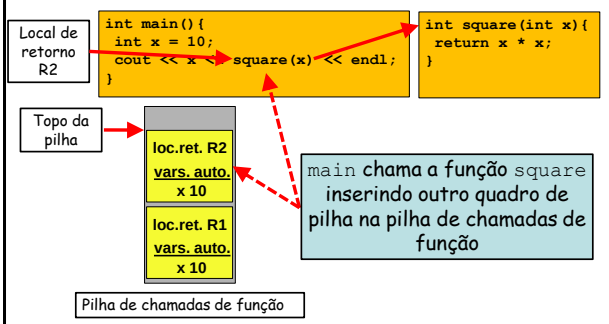
Pilha de Chamadas de Função e Registros de Ativação



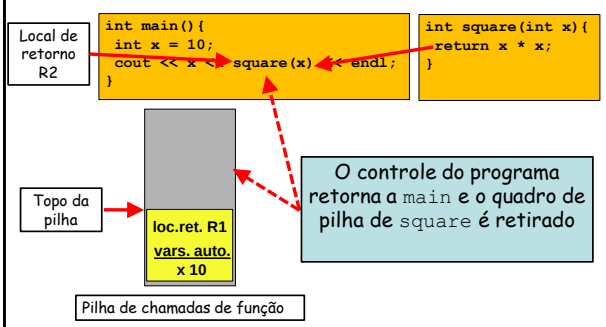
Pilha de Chamadas de Função e Registros de Ativação



Pilha de Chamadas de Função e Registros de Ativação



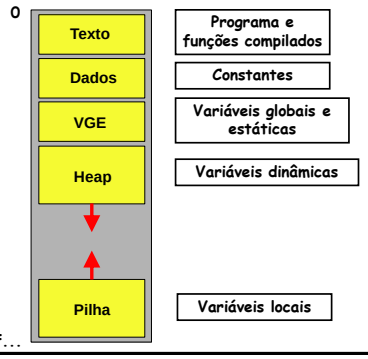
Pilha de Chamadas de Função e Registros de Ativação



Já que falamos de memória...

Como a memória é organizada?

Já que falamos de memória...



Funções Inline

- Reduzem o overhead de chamadas de função
  - Especialmente para funções pequenas
- Colocam o qualificador **`inline`** antes do tipo de retorno de uma função na definição de função
  - "Adverte" o compilador para que gere uma cópia do código da função em seu lugar (quando apropriado) para evitar uma chamada de função

Funções Inline

- Troca de funções **`inline`**
  - Múltiplas cópias do código da função são inseridas no programa (em geral tornando o programa maior)
- O compilador pode ignorar o qualificador **`inline`** e normalmente o faz para todas as funções
  - Exceto para as menores

Funções Inline

- Qualquer alteração em uma função **`inline`** pode exigir que os clientes da função sejam recompilados
  - Isso pode ser significativo em algumas situações de desenvolvimento e manutenção de programas
- O qualificador **`inline`** deve ser utilizado somente com funções pequenas
  - Funções **`inline`** podem reduzir o tempo de execução
    - Mas podem aumentar o tamanho do programa
- O qualificador **`const`** deve ser utilizado para indicar ao compilador que uma variável não pode ser alterada

Terceiro Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 7
 * Autor: Miguel Campista
*/

#include <iostream>

using namespace std;

inline double cube (const double side) {
        return side * side * side;
}

int main() {
    double sideValue;
    cout << "Entre com o tamanho do lado: ";
    cin >> sideValue;

    cout << "O volume do cube de lado " << sideValue
         << "eh: " << cube(sideValue) << endl;

    return 0;
}
```


Terceiro Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 7
 * Autor: Miguel Campista
*/

#include <iostream>

using namespace std;

inline double cube (const double side) {
        return side * side * side;
}

int main() {
    double sideValue;
    cout << "Entre com o tamanho do lado: ";
    cin >> sideValue;

    cout << "O volume do cube de lado " << sideValue
         << "eh: " << cube(sideValue) << endl;

    return 0;
}
```

O uso do qualificador **`inline`**

Terceiro Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 7
 * Autor: Miguel Campista
*/
```

```
shell>$ g++ exemplo.cpp –o ex7
shell>$ ./ex7
Entre com o tamanho do lado: 2
O volume do cubo de lado 2 eh: 8
shell>$
```

```
int main() {
    double sideValue;
    cout << "Entre com o tamanho do lado: ";
    cin >> sideValue;

    cout << "O volume do cube de lado " << sideValue
         << "eh: " << cube(sideValue) << endl;

    return 0;
}
```


Referências e Parâmetros de Referências

- Duas formas de passar argumentos a funções
  - Passagem por valor
    - Uma *cópia* do valor do argumento é passada à função chamada
    - As mudanças na cópia não afetam o valor original da variável no chamador
      - Isso evita efeitos colaterais acidentais das funções
  - Passagem por referência
    - Permite que a função chamada acesse e modifique diretamente dados do argumento do chamador

Referências e Parâmetros de Referências

- Duas formas de passar argumentos a funções
  - Passagem por valor
    - Uma *cópia* do valor do argumento é passada à função chamada
    - As mudanças na cópia não afetam o valor original da variável no chamador
      - Isso evita efeitos colaterais acidentais das funções
  - Passagem por referência
    - Permite que a função chamada acesse e modifique diretamente dados do argumento do chamador

Passagem por valor não é vantajosa se um item de dados passado for grande. Copiar esses dados pode exigir uma quantidade considerável de tempo de execução e memória!

Referências e Parâmetros de Referências

- Parâmetro de referência
  - Uma referência para seu argumento correspondente em uma chamada de função
  - **&** colocado após o tipo de parâmetro no protótipo de função e cabeçalho de função
    - Ex.: `int &count` em um cabeçalho de função
      - Pronuncia-se "`count` é uma referência a um `int`"
  - O nome do parâmetro no corpo da função chamada na verdade refere-se à variável original na função chamadora

Quarto Exemplo utilizando Funções em C++

```
#include <iostream>

using namespace std;

int squareByValue(int);
void squareByReference(int &);

int main() {
    int v = 3, r = 4;

    cout << "v = " << v << " antes da funcao squareByValue\n";
    cout << "Valor retornado pela funcao squareByValue: "
         << squareByValue(v) << endl;
    cout << "v = " << v << " depois da funcao squareByValue\n\n";

    cout << "r = " << r << " antes da funcao squareByReference\n";
    squareByReference(r);
    cout << "r = " << r << " depois da funcao squareByReference\n";


    return 0;
}

int squareByValue(int n) {
    return n *= n;
}
void squareByReference(int &n) {
    n *= n;
}
```


Quarto Exemplo utilizando Funções em C++

```
#include <iostream>

using namespace std;

int squareByValue(int);
void squareByReference(int &);

int main() {
    int v = 3, r = 4;

    cout << "v = " << v << " antes da funcao squareByValue\n";
    cout << "Valor retornado pela funcao squareByValue: "
         << squareByValue(v) << endl;
    cout << "v = " << v << " depois da funcao squareByValue\n\n";

    cout << "r = " << r << " antes da funcao squareByReference\n";
    squareByReference(r);
    cout << "r = " << r << " depois da funcao squareByReference\n";


    return 0;
}

int squareByValue(int n) {
    return n *= n;
}
void squareByReference(int &n) {
    n *= n;
}
```

Função com passagem de parâmetro por valor

Função com passagem de parâmetro por referência

As variáveis são sempre passadas através dos identificadores

Quarto Exemplo utilizando Funções em C++

```
#include <iostream>

using namespace std;

int squareByValue(int);
void squareByReference(int &);

int main() {
    int v = 3, r = 4;

    cout << "v = " << v << " antes da funcao squareByValue\n";
    cout << "Valor retornado pela funcao squareByValue: "
         << squareByValue(v) << endl;
    cout << "v = " << v << " depois da funcao squareByValue\n\n";

    cout << "r = " << r << " antes da funcao squareByReference\n";
    squareByReference(r);
    cout << "r = " << r << " depois da funcao squareByReference\n";

    return 0;
}

int squareByValue(int n) {
    return n *= n;
}
void squareByReference(int &n) {
    n *= n;
}
```

Recebe cópia de argumento

Recebe referência de argumento

Quarto Exemplo utilizando Funções em C++

```
#include <iostream>

using namespace std;

int squareByValue(int);
void squareByReference(int &);

int main() {
```

```
shell>$ g++ exemplo.cpp –o ex8
shell>$ ./ex8
v = 3 antes da funcao squareByValue
Valor retornado pela funcao squareByValue: 9
v = 3 depois da funcao squareByValue

r = 4 antes da funcao squareByReference
r = 16 depois da funcao squareByReference
shell>$
```

```
}

int squareByValue(int n) {
    return n *= n;
}
void squareByReference(int &n) {
    n *= n;
}
```


Referências e Parâmetros de Referências

- Parâmetros por referência podem ser inadvertidamente tratados como parâmetros por valor já que em ambos os casos eles são mencionados apenas pelo nome
- Para passar objetos grandes, utilize um parâmetro de referência constante a fim de simular a aparência e a segurança da passagem por valor e evitar o overhead de passar uma cópia do objeto grande

Referências e Parâmetros de Referências

- Referências
  - Podem ser também utilizadas por outras variáveis dentro de uma função
    - Todas as operações supostamente executadas na referência são na verdade executadas na variável original
    - Devem ser inicializadas em suas declarações
      - Não podem ser reatribuídas posteriormente

```
int count = 1;
int &cRef = count;
cRef++;
```

Incrementa `count` por meio da referência `cRef`

Quinto Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 9
 * Autor: Miguel Campista
 */

#include <iostream>

using namespace std;

int main() {
    int x = 3;
    int &y = x; // y é uma referência

    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7;
    cout << "\nx = " << x << endl << "y = " << y << endl;

    return 0;
}
```


Quinto Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 9
 * Autor: Miguel Campista
 */

#include <iostream>

using namespace std;

int main() {
    int x = 3;
    int &y = x; // y é uma referência

    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7;
    cout << "\nx = " << x << endl << "y = " << y << endl;

    return 0;
}
```

Criação de uma referência para **x**

Atribuição de um valor a **x** através da sua referência **y**

Quinto Exemplo utilizando Funções em C++

```
shell>$ g++ exemplo.cpp –o ex9
shell>$ ./ex9
x = 3
y = 3

x = 7
y = 7
shell>$
```

```
int main() {
    int x = 3;
    int &y = x; // y é uma referência

    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7;
    cout << "\nx = " << x << endl << "y = " << y << endl;

    return 0;
}
```


Quinto Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 9
 * Autor: Miguel Campista
 */

#include <iostream>

using namespace std;

int main() {
    int x = 3;
    int &y;

    cout << "x = " << x << endl << "y = " << y << endl;
    y = 7;
    cout << "\nx = " << x << endl << "y = " << y << endl;

    return 0;
}
```

O que acontece nesse caso?

Referências e Parâmetros de Referências

- Retornando uma referência de uma função
  - As funções podem retornar referências a variáveis
    - Isso só pode ser usado quando a variável cuja referência foi retornada é estática à função chamada
      - Retornar uma referência a uma variável automática causa problema pois essa variável deixa de existir depois que a função termina → Referência é perdida!

Argumentos Padrão

- Valor-padrão a ser passado a um parâmetro
  - Argumento passado comumente a um parâmetro de uma função
    - Chamada da função não especifica o argumento desse parâmetro
- Deve ser especificado na primeira ocorrência do nome da função
  - Em geral, o protótipo da função
- Deve(m) ser o(s) argumento(s) mais à direita na lista de parâmetros de uma função
  - Padronização necessária caso a função receba outros argumentos

Sexto Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 10
 * Autor: Miguel Campista
 */

#include <iostream>

int volume (int length = 1, int width = 1, int height = 1);

using namespace std;

int main() {
    cout << "O volume padrao eh: " << volume() << endl;
    cout << "\nO volume com comprimento 10 eh: "
         << volume(10) << endl;
    cout << "\nO volume com comprimento 10 e largura 5 eh: "
         << volume(10, 5) << endl;
    cout << "\nO volume com comprimento 10, largura 5 e altura 2 eh: "
         << volume(10, 5, 2) << endl;

   return 0;
}

int volume (int length, int width, int height) {
    return length * width * height;
}
```

Sexto Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 10
 * Autor: Miguel Campista
 */

#include <iostream>

int volume (int length = 1, int width = 1, int height = 1);

using namespace std;

int main() {
    cout << "O volume padrao eh: " << volume() << endl;
    cout << "\nO volume com comprimento 10 eh: "
         << volume(10) << endl;
    cout << "\nO volume com comprimento 10 e largura 5 eh: "
         << volume(10, 5) << endl;
    cout << "\nO volume com comprimento 10, largura 5 e altura 2 eh: "
         << volume(10, 5, 2) << endl;

   return 0;
}

int volume (int length, int width, int height) {
    return length * width * height;
}
```

Argumentos padrão

Função chamadora sem argumento

Sexto Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 10
 * Autor: Miguel Campista
*/

#include <iostream>

int volume (int length = 1, int width = 1, int height = 1);

using namespace std;

int main() {
    cout << "O volume padrao eh: " << volume() << endl;
    cout << "\nO volume com comprimento 10 eh: "
         << volume(10) << endl;
    cout << "\nO volume com comprimento 10 e largura 5 eh: "
         << volume(10, 5) << endl;
    cout << "\nO volume com comprimento 10, largura 5 e altura 2 eh: "
         << volume(10, 5, 2) << endl;

   return 0;
}

int volume (int length, int width, int height) {
    return length * width * height;
}
```

Argumentos padrão

Função chamadora com todos os argumentos

Sexto Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 10
 * Autor: Miguel Campista
*/

#include <iostream>

int volume (int length = 1, int width = 1, int height = 1);
```

```
shell>$ g++ exemplo.cpp –o ex10
shell>$ ./ex10
O volume padrão eh: 1

O volume com comprimento 10 eh: 10

O volume com comprimento 10 e largura 5 eh: 50

O volume com comprimento 10, largura 5 e altura 2 eh: 100
shell>$
```

```
int volume (int length, int width, int height) {
    return length * width * height;
}
```

Argumentos Padrão

- Utilizar argumentos-padrão pode simplificar a escrita de chamadas de função
  - Entretanto, pode ser mais claro especificar todos os argumentos explicitamente
- Se os valores-padrão de uma função mudam...
  - Todos os códigos-cliente que estiverem utilizando a função devem ser recompilados

Operador de Solução de Escopo Unário (: :)

- Usado para acessar uma variável global quando uma variável local com o mesmo nome estiver no escopo
  - Ex.: `cout << ::x;`
- Não pode ser usado para acessar uma variável local com o mesmo nome em um bloco externo
- Sempre utilizar o operador unário de resolução de escopo (::) para referenciar as variáveis globais torna os programas mais fáceis de ler e entender
  - Variáveis globais são explicitadas no código

Sétimo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 11
 * Autor: Miguel Campista
*/

#include <iostream>

using namespace std;

int x = 1; // Variável global

int main() {
    double x = 1.5;

    cout << "Valor double local eh: " << x << endl
         << "\nValor int global eh: " << ::x << endl;

    return 0;
}
```


Sétimo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 11
 * Autor: Miguel Campista
*/

#include <iostream>

using namespace std;

int x = 1; // Variável global

int main() {
    double x = 1.5;

    cout << "Valor double local eh: " << x << endl
         << "\nValor int global eh: " << ::x << endl;

    return 0;
}
```

Operador unário para definição de escopo

Sétimo Exemplo utilizando Funções em C++

```
shell>$ g++ exemplo.cpp –o ex11
shell>$ ./ex11
Valor double local eh: 1.5

Valor int global eh: 1
shell>$
```

```
using namespace std;

int x = 1; // Variável global

int main() {
    double x = 1.5;

    cout << "Valor double local eh: " << x << endl
         << "\nValor int global eh: " << ::x << endl;

    return 0;
}
```


Funções Sobrecarregadas

- As funções sobrecarregadas têm:
  - O mesmo nome e diferentes conjuntos de parâmetros
- O compilador seleciona a função apropriada
  - Baseado no nome, tipo e ordem dos argumentos na chamada de função
- A sobrecarga cria várias funções com o mesmo nome
  - Executam tarefas semelhantes, mas em tipos de dados diferentes
- Sobrecarregar funções que realizam tarefas intimamente relacionadas
  - Torna os programas mais legíveis e compreensíveis

Oitavo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 – Exemplo 12
 * Autor: Miguel Campista
*/

#include <iostream>

using namespace std;

int square(int x) {
    cout << "Quadrado do inteiro " << x << " eh: ";
    return x * x;
}

double square(double x) {
    cout << "Quadrado do double " << x << " eh: ";
    return x * x;
}

int main() {
    cout << square(2) << endl;
    cout << square(2.5) << endl;

    return 0;
}
```


Oitavo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 – Exemplo 12
 * Autor: Miguel Campista
*/

#include <iostream>

using namespace std;

int square(int x) {
    cout << "Quadrado do inteiro " << x << " eh: ";
    return x * x;
}

double square(double x) {
    cout << "Quadrado do double " << x << " eh: ";
    return x * x;
}

int main() {
    cout << square(2) << endl;
    cout << square(2.5) << endl;

    return 0;
}
```

Função **square** para **int**

Função **square** para **double**

Oitavo Exemplo utilizando Funções em C++

```
shell>$ g++ exemplo.cpp –o ex12
shell>$ ./ex12
Quadrado do inteiro 2 eh: 4
Quadrado do double 2.5 eh: 6.25
shell>$
```

```
int square(int x) {
    cout << "Quadrado do inteiro " << x << " eh: ";
    return x * x;
}

double square(double x) {
    cout << "Quadrado do double " << x << " eh: ";
    return x * x;
}

int main() {
    cout << square(2) << endl;
    cout << square(2.5) << endl;

    return 0;
}
```


Funções Sobrecarregadas

- Como o compilador diferencia as funções sobrecarregadas?
  - As funções sobrecarregadas são diferenciadas pela respectiva assinatura
  - Desfiguração de nome ou decoração de nome
    - O compilador codifica cada identificador de função com o número e o tipo de parâmetro para permitir a ligação segura para tipos
  - A ligação segura para tipos garante que:
    - A função sobrecarregada apropriada seja chamada
    - Os tipos de argumento correspondam aos tipos de parâmetro

Nono Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 13
 * Autor: Miguel Campista
*/

#include <iostream>

using namespace std;

int square (int x)
    return x * x;
}
double square (double x)
    return x * x;
}

int main() {
    return 0;
}
```

Função **square** sobrecarregada

Nono Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 13
 * Autor: Miguel Campista
*/

#include <iostream>

using namespace std;

int square (int x)
    return x * x;
}
double square (double x)
    return x * x;
}

int main() {
    return 0;
}
```

Função **square** sobrecarregada

shell>$ g++ -S –o ex13 exemplo.cpp

```
.type  _GLOBAL__I__Z6squarei, @function
.type  _GLOBAL__I__Z6squared, @function
```


Funções Sobrecarregadas

- Uma função com argumentos padrão omitidos pode ser chamada de modo idêntico a outra função sobrecarregada
  - Isso constitui um erro de compilação!
    - Ex.: Uma função que não aceita explicitamente nenhum argumento e uma função de mesmo nome que contém todos os argumentos como padrão provoca um erro de compilação…

O compilador não consegue identificar qual função deve utilizar

Funções Sobrecarregadas

- Uma função com argumentos padrão omitidos pode ser chamada de modo idêntico a outra função sobrecarregada
  - Isso constitui um erro de compilação!
    - Ex.: Uma função que não aceita explicitamente nenhum argumento e uma função de mesmo nome que contém todos os argumentos como padrão provoca um erro de compilação…

```
int funcao(int a = 1, int b = 2);

double funcao(int x);
```


Templates de Funções

- Forma mais compacta e conveniente de sobrecarga
  - Lógica e operações de programação idênticas para cada tipo de dados

Templates de Funções

- Definição de template de função
  - É escrita por programadores uma única vez
  - Define toda a família de funções sobrecarregadas
  - Começa com a palavra-chave `template`
  - Contém uma lista de parâmetros template de parâmetros de tipo formal para a função template entre colchetes angulares (<>)
  - Parâmetros de tipo formal
    - Precedido pela palavra-chave `typename` ou `class`
    - São marcadores de lugar para tipos fundamentais ou tipos definidos pelo usuário

Templates de Funções

- Especializações de template de função
  - Geradas automaticamente pelo compilador para lidar com cada tipo de chamada para o template de função
  - Exemplo para o template de função **max** com o tipo de parâmetro **T** chamado com argumentos **int**
    - O compilador detecta uma invocação **max** no código do programa
    - **int** substitui **T** em toda a definição do template
    - Isso gera a especialização do template **max< int >**

Templates de Funções

- Não colocar a palavra-chave **class** ou **typename** antes de cada parâmetro de tipo formal de um template de função é um erro de sintaxe
  - Ex.: Escrever `< class S, T >`
    em vez de `< class S, class T >` é um erro

Templates com tipos diferentes de dados...

Décimo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 14
 * Arquivo template.h
 * Autor: Miguel Campista
*/

template <class T>
T maximo(T v1, T v2, T v3) {
      T vMax = v1;
      if (v2 > vMax)
            vMax = v2;
      if (v3 > vMax)
            vMax = v3;
      return vMax;
}
```


Décimo Exemplo utilizando Funções em C++

Usando o parâmetro de tipo formal T no lugar do tipo de dados

```
/*
 * Aula 6 - Exemplo 14
 * Arquivo template.h
 * Autor: Miguel Campista
*/

template <class T>
T maximo(T v1, T v2, T v3) {
      T vMax = v1;
      if (v2 > vMax)
            vMax = v2;
      if (v3 > vMax)
            vMax = v3;
      return vMax;
}
```


Décimo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 14
 * Arquivo principal
 * Autor: Miguel Campista
*/

#include <iostream>
#include "template.h"

using namespace std;

int main() {
    int i1, i2, i3;
    cout << "Entre com os valores de tres inteiros: " << endl;
    cin >> i1 >> i2 >> i3;
    cout << "O valor maximo eh: " << maximo(i1, i2, i3) << endl;

    double d1, d2, d3;
    cout << "\nEntre com os valores de tres doubles: " << endl;
    cin >> d1 >> d2 >> d3;
    cout << "O valor maximo eh: " << maximo(d1, d2, d3) << endl;

    char c1, c2, c3;
    cout << "\nEntre com os valores de tres chars: " << endl;
    cin >> c1 >> c2 >> c3;
    cout << "O valor maximo eh: " << maximo(c1, c2, c3) << endl;

    return 0;
}
```


Décimo Exemplo utilizando Funções em C++

Função **maximo** com argumentos **int**

```
/*
 * Aula 6 - Exemplo 14
 * Arquivo principal
 * Autor: Miguel Campista
*/

#include <iostream>
#include "template.h"

using namespace std;

int main() {
    int i1, i2, i3;
    cout << "Entre com os valores de tres inteiros: " << endl;
    cin >> i1 >> i2 >> i3;
    cout << "O valor maximo eh: " << maximo(i1, i2, i3) << endl;

    double d1, d2, d3;
    cout << "\nEntre com os valores de tres doubles: " << endl;
    cin >> d1 >> d2 >> d3;
    cout << "O valor maximo eh: " << maximo(d1, d2, d3) << endl;

    char c1, c2, c3;
    cout << "\nEntre com os valores de tres chars: " << endl;
    cin >> c1 >> c2 >> c3;
    cout << "O valor maximo eh: " << maximo(c1, c2, c3) << endl;

    return 0;
}
```

Décimo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 14
 * Arquivo principal
 * Autor: Miguel Campista
 */

#include <iostream>
#include "template.h"

using namespace std;

int main() {
    int i1, i2, i3;
    cout << "Entre com os valores de tres inteiros: " << endl;
    cin >> i1 >> i2 >> i3;
    cout << "O valor maximo eh: " << maximo(i1, i2, i3) << endl;

    double d1, d2, d3;
    cout << "\nEntre com os valores de tres doubles: " << endl;
    cin >> d1 >> d2 >> d3;
    cout << "O valor maximo eh: " << maximo(d1, d2, d3) << endl;

    char c1, c2, c3;
    cout << "\nEntre com os valores de tres chars: " << endl;
    cin >> c1 >> c2 >> c3;
    cout << "O valor maximo eh: " << maximo(c1, c2, c3) << endl;

    return 0;
}
```

Função **maximo** com argumentos **double**

Décimo Exemplo utilizando Funções em C++

```
/*
 * Aula 6 - Exemplo 14
 * Arquivo principal
 * Autor: Miguel Campista
 */

#include <iostream>
#include "template.h"

using namespace std;

int main() {
    int i1, i2, i3;
    cout << "Entre com os valores de tres inteiros: " << endl;
    cin >> i1 >> i2 >> i3;
    cout << "O valor maximo eh: " << maximo(i1, i2, i3) << endl;

    double d1, d2, d3;
    cout << "\nEntre com os valores de tres doubles: " << endl;
    cin >> d1 >> d2 >> d3;
    cout << "O valor maximo eh: " << maximo(d1, d2, d3) << endl;

    char c1, c2, c3;
    cout << "\nEntre com os valores de tres chars: " << endl;
    cin >> c1 >> c2 >> c3;
    cout << "O valor maximo eh: " << maximo(c1, c2, c3) << endl;

    return 0;
}
```

Função **maximo** com argumentos **char**

Décimo Exemplo utilizando Funções em C++

```
shell>$ g++ exemplo.cpp –o ex14
shell>$ ./ex14
Entre com os valores de tres inteiros:
1 2 3
O valor maximo eh: 3

Entre com os valores de tres doubles:
1.2 2.3 3.4
O valor maximo eh: 3.4

Entre com os valores de tres chars:
a b c
O valor maximo eh: c
shell>$
```

```
    char c1, c2, c3;
    cout << "\nEntre com os valores de tres chars: " << endl;
    cin >> c1 >> c2 >> c3;
    cout << "O valor maximo eh: " << maximo(c1, c2, c3) << endl;

    return 0;
}
```


Exemplo: Máquina para Testes de Multiplicação

- Escreva um programa em C++ para tomar a tabuada de alunos de primário
  - Cada acerto e erro gera uma mensagem aleatória de incentivo
  - Após 10 rodadas, se o desempenho tiver sido abaixo do mínimo o programa termina e avisa ao usuário o motivo

?

Exemplo: Máquina para Testes de Multiplicação

```
/*
 * Aula 6 - Exemplo 17
 * Arquivo Principal
 * Autor: Miguel Campista
 */
#include <iostream>
#include <iomanip>
#include "machine.h"

using namespace std;

int main() {
    int exit;
    string name;

    cout << "Entre com o seu nome: ";
    getline(cin, name);

    Machine machine(name);

    while (1) {

        exit = machine.multiplicationTests();

        if(exit == 1) {
            cout << "\n\nTchau!\n\n";
            break;
        }
```



```
        if (machine.getRuns() % 10) == 0) {

            if (machine.getPerf() <= 0.75) {
                cout << name << ", seu desempenho na rodada "
                     << machine.getRound()
                     << "foi abaixo do esperado: "
                     << machine.getPerf() << endl;
                cout << "\nPor favor, ";
                cout << "procure o seu professor para ajuda extra."
                     << endl;
                break;
            } else {
                string cont;
                cout << name << ", seu desempenho na rodada "
                     << machine.getRound()
                     << " foi acima do esperado: "
                     << setprecision(2) << fixed
                     << 100*machine.getPerf() << "%." << endl;
                cout << "\nParabens!!!\n" << endl;

                cin.ignore(100, '\n');

                cout << "Deseja continuar? (S/N)" << endl;
                getline(cin, cont);

                if (cont.compare("S") == 0 || cont.compare("SIM") == 0) {
                    machine.setRound();
                    cout << "\n== Proxima rodada: Rodada "
                         << machine.getRound() << " ==\n" << endl;
                } else if (cont.compare("N") == 0 || cont.compare("NAO") == 0) {
                    cout << "\n\nTchau!\n\n";
                    break;
                } else {
                    cout << "\nResposta desconhecida\nTerminando."
                         << endl;
                    break;
                }
            }

        }
    }

    return 0;
}
```

```
/*
 * Aula 6 - Exemplo 17
 * Arquivo machine.h
 * Autor: Miguel Campista
*/
#include <string>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "machine-template.h"

using namespace std;

class Machine {
	public:
		// Construtor
		Machine(string);
		// Função para lançar novo desafio matemático
		int multiplicationTests();
		// Função para obter o desempenho nas últimas 10 rodadas
		double getPerf();
		// Função para obter o número de rodadas
		int getRuns();
		// Função para obter o número conjunto de 10 rodadas
		int getRound();
		// Função para ajustar o número conjunto de 10 rodadas
		void setRound();
	private:
		// Variáveis privadas
		enum Performance { SUCCESS, FAILURE };
		int successes, failures, runs, round;
		string nameStudent;
		// Função para checar se a resposta está correta
		void checkResp(int, int, int);
		// Funções para uma respostas após o desafio
		void correctAnswer();
		void wrongAnswer();
		// Função para contabilizar sucessos e falhas
		void setPerf(Performance);

};
```

Exemplo: Máquina para Testes de Multiplicação

```
/*
 * Aula 6 - Exemplo 17
 * Arquivo machine.cpp
 * Autor: Miguel Campista
*/
#include "machine.h"

Machine::Machine(string name) {
	successes = 0;
	failures = 0;
	runs = 0;
	round = 1;
	srand(time(0));
	nameStudent = name;
	cout << "\nAprendizado de matematica\n\n******* "
		<< nameStudent << ", Bem-vindo! *******\n\n"
		<< "Digite -1 para sair.\n" << endl;
}

int Machine::multiplicationTests() {
	int a = rand() % 10;
	int b = rand() % 10;
	int resp, perf;

	cout << "Quanto eh " << a << " x " << b << " ?\nResposta: ";
	cin >> resp;

	if(resp == -1)
		return 1;
	else
		checkResp(a, b, resp);

	return 0;
}
```

Exemplo: Máquina para Testes de Multiplicação

```
void Machine::checkResp(int a, int b, int resp) {
	Performance perf;
	if(multiply(a, b) == resp) {
		perf = SUCCESS;
		correctAnswer();
	} else {
		perf = FAILURE;
		wrongAnswer();
	}
	setPerf(perf);
}

void Machine::correctAnswer() {
	int randAnswer = rand() % 4;

	switch (randAnswer) {
		case 0:
			cout << "\nMuito bem!\n" << endl;
			break;
		case 1:
			cout << "\nExcelente!\n" << endl;
			break;
		case 2:
			cout << "\nBom trabalho!\n" << endl;
			break;
		case 3:
			cout << "\nContinue com o bom trabalho!\n" << endl;
			break;
	}
}
```



```
void Machine::wrongAnswer() {
	int randAnswer = rand() % 4;

	switch (randAnswer) {
		case 0:
			cout << "\nNum, por favor, tente novamente.\n" << endl;
			break;
		case 1:
			cout << "\nErrado. Tente mais uma vez.\n" << endl;
			break;
		case 2:
			cout << "\nNao desista!\n" << endl;
			break;
		case 3:
			cout << "\nNao. Continue tentando.\n" << endl;
			break;
	}
}

void Machine::setPerf(Performance perf) {
	if(perf == SUCCESS)
		successes++;
	else
		failures++;
	runs++;
}

double Machine::getPerf() {
	return static_cast <double> (successes) / runs;
}

int Machine::getRuns() {
	return runs;
}

int Machine::getRound() {
	return round;
}
void Machine::setRound() {
	round++;
	runs = 0;
	successes = 0;
	failures = 0;
}
```

Exemplo: Máquina para Testes de Multiplicação

```
/*
 * Aula 6 - Exemplo 17
 * Arquivo machine-template.h
 * Autor: Miguel Campista
*/

template <class T>
T multiply (T a, T b) {
	return a * b;
}
```


Leitura Recomendada

- Capítulo 6 do livro
  - Deitel, "*C++ How to Program*", 5th edition, Editora Prentice Hall, 2005