

Linguagem de programação

Macros e Funções

Tópicos abordados

- o conceito de pré-processamento
- as duas principais diretivas utilizadas
- macros
- criação e o uso de funções em C

Pré-processamento

- Todo programa C, antes de chegar ao compilador é tratado por um módulo adicional denominado pré-processador.

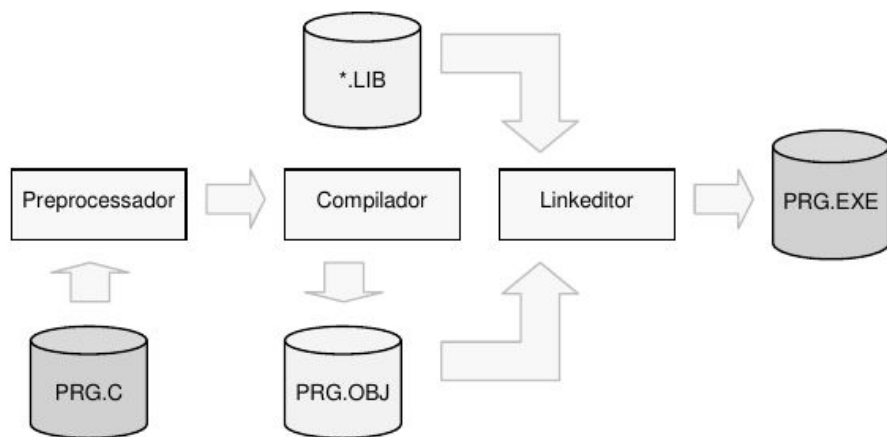


Figura 4.1 – O processo de geração de um executável

Pré-processamento

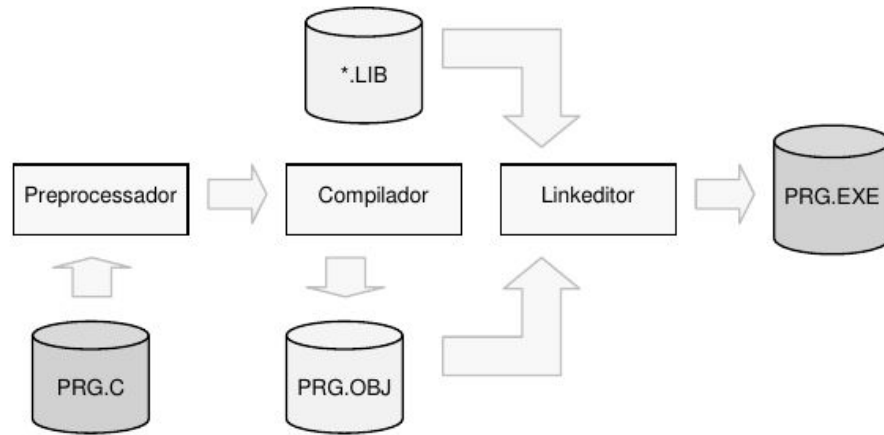


Figura 4.1 – O processo de geração de um executável

- o pré-processador realiza várias modificações no código antes que seja analisado pelo compilador
- Essas modificações são feitas por **diretivas**, que são embutidas no código-fonte
- As duas principais diretivas são:
 - `#include`
 - `#define`

A diretiva `#define`

- serve para definir constantes simbólicas que aumentam a legibilidade do código
- Associa um identificador a um texto da seguinte maneira: `#define`
`identificador texto`

O texto associado ao identificador pode ser inclusive uma palavra reservada

A diretiva `#define`

```
#include <stdio.h>
#define diga printf
#define oi "\nOlá, tudo bem?"
main() {
    diga(oi);
}
```

- Quando o programa passa pelo pré-processador, a diretiva `#define` é executada e, então, toda ocorrência da palavra `diga` é substituída por `printf` e toda ocorrência da palavra `oi` é substituída por `"\nOlá, tudo bem?"`.

Exercício

Inclua diretivas `#define` no programa a seguir de modo que ele possa ser compilado corretamente:

```
#include <stdio.h>
programa
inicio
    diga("Olá!");
fim
```

Macros

- É possível substituições parametrizadas, ou seja **Macros**.
- Por ex.:
 - `#define quad(n) n*n`
- Ao passar pelo pré-processador, as ocorrências desse **macro** são substituídas pela expressão `n*n`, com o parâmetro *n* devidamente instanciado

Macros

Ocorrência	Instância
quad(x)	$x + x$
quad(2)	$2 * 2$
quad(f(x-3))	$f(x-3) * f(x-3)$
quad(x+4)	$x+4 * x+4$

Note que não pode haver espaço entre o identificador do macro e o parêntese que delimita a lista de parâmetros; caso haja, teremos um erro de substituição

Macros

- Nenhum cálculo é realizado durante o pré-processamento. Ocorre apenas uma simples substituição
 - Portanto, resultados inesperados podem surgir:

<i>ocorrência</i>	<i>esperado</i>	<i>instância</i>	<i>obtido</i>
<i>quad</i> (2+3)	25	2+3 * 2+3	11
100 / <i>quad</i> (2)	25	100 / 2 * 2	100

Macros

- Macro correto para calcular o quadrado de um número:
 - `#define` `quadrado(n)` `((n) * (n))` , a expressão de substituição deve ser completamente parentetizada

<i>ocorrência</i>	<i>instância</i>	<i>obtido</i>
<i>quad(2+3)</i>	<i>((2+3) * (2+3))</i>	25
<i>100 / quad(2)</i>	<i>100 / ((2) * (2))</i>	25

Exercício

Defina e teste os seguintes macros:

- `eh_maiuscula(c)`: informa se o caractere `c` é uma letra maiúscula
- `minuscula(c)`: converte a letra `c` para minúscula
 - dica: verificar a diferença entre uma letra minúscula e maiúscula
- `mod(a)`: retorna o módulo de `a`
 - dica: utilizar operador condicional
- `min(a,b)`: retorna o valor mínimo entre `a` e `b`
 - dica: utilizar operador condicional

Diretiva `#include`

- Faz com que uma cópia do arquivo cujo nome é dado entre `<` e `>` seja incluído no código-fonte
- Por ex. se criarmos o arquivo ***macros.h*** com os seguintes macros:

```
#define quad(n)    ( (n)*(n) )  
#define abs(n)    ( (n)<0 ? -(n) : (n) )  
#define max(x,y)  ( (x)>(y) ? (x) : (y) )
```

- não precisaremos digitá-las toda vez que precisarmos usá-las. Basta pedir para o pré-processador incluir uma cópia de macros.h no nosso código-fonte

Diretiva `#include`

- A convenção em C é que se use nome_arquivo.h
- `< >` é utilizado para inclusão de *headers* padrões do C
 - o pré-processador irá buscar o header em caminho padrão
- ao criar seus próprios headers, deve-se usar “”
 - pré-processador irá buscar o header no próprio local do código fonte

```
#include <stdio.h>
#include "macros.h"

void main() {
    int a, b;
    printf("\nDigite 2 números: ");
    scanf("%d %d", &a, &b);
    printf("\nO máximo é %d!", a);
}
```

Exercício

- Crie um cabeçalho (*header*) com o nome macros.h, que contenha todos os macros do exercício anterior. Em seguida faça um programa em C que inclua macros.h e teste todos os macros implementados.

Funções

Introdução

- **Funções** são usadas para modularizar programas
- Os programas C são normalmente escritos combinando **novas funções que você escreve com funções predefinidas disponíveis na biblioteca padrão C.**
- A biblioteca padrão C fornece uma rica coleção de funções para realizar **cálculos matemáticos comuns, manipulações de strings, manipulações de caracteres, entrada / saída e muitas outras operações úteis.**

Definição de função

- Até agora, sempre usamos uma função em todos os programas, a ***main()***
 - e outras como *scanf()* e *printf()*
- Mas podemos criar nossas próprias funções e utilizá-las da mesma maneira

Definição de função

- Para definir uma função básica, empregamos a seguinte forma básica
- O tipo deve ser void (vazio) se a função não tem valor de resposta;

```
tipo nome (parâmetro) {  
    declarações  
    parâmetros  
}
```

Definição de função

- Para que uma função seja reconhecida durante a compilação devemos declará-la ou defini-la antes de qualquer referência que é feita a ela no resto do programa.

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

```

```
int main() {  
    int y;  
    for(y=1; y<=10; y++)  
        printf("%d ",  
raiz_quadrada(y));  
    return 0; }  

```

Definição de função

- No exemplo, a passagem de parâmetro é por **cópia**.
- Dentro da função é criada uma cópia de “x”, que inicia e termina seu ciclo de vida no contexto da função
- Nenhuma modificação em “x” surtirá efeito fora da função

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

```

```
int main() {  
    int y;  
    for(y=1; y<=10; y++)  
        printf("%d ",  
raiz_quadrada(y));  
    return 0; }  

```

Erros comuns

- Omitir o tipo-do-valor-de-retorno em uma definição de função causa um erro de sintaxe se o protótipo da função especificar um tipo de retorno diferente de `int`.
 - se nenhum retorno é definido, assume-se `int`
- Declarar parâmetros da função do mesmo tipo como **`float x, y`** em vez de **`float x, float y`**. A declaração de parâmetros **`float x, y`** tornaria na realidade **`y`** um parâmetro do tipo **`int`** porque **`int`** é o default.

Erros comuns

- Definir um parâmetro de função novamente como variável local dentro da função é um erro de sintaxe.

Protótipo de função

- ANSI C: conj. de padrões para a linguagem C, publicados pela American National Standards Institute
- Um dos recursos mais importantes do ANSI C é o protótipo de função
 - protótipo de função diz ao compilador o tipo do dado retornado pela função, o número de parâmetros que a função espera receber, os tipos dos parâmetros e a ordem na qual esses parâmetros são esperados.

Protótipo de função

- Um dos recursos mais importantes do ANSI C é o protótipo de função
 - protótipo de função diz ao compilador o tipo do dado retornado pela função, o número de parâmetros que a função espera receber, os tipos dos parâmetros e a ordem na qual esses parâmetros são esperados.
 - O compilador usa protótipos de funções para validar as chamadas de funções.

Protótipo de função

- compilador usa protótipos de funções para validar as chamadas de funções.
 - Em padrões anteriores do C, **essa validação não era feita**. Então **aconteciam vários erros de execução do programa**, que eram difíceis de serem detectados

Exemplo de protótipo:

- O protótipo do exemplo é:
 - `int raiz_quadrada(int);`
- Algumas vezes, os nomes dos parâmetros são incluídos nos protótipos de funções para fins de documentação. O compilador ignora esses nomes.

```
#include <stdio.h>

//protótipo de função
int raiz_quadrada(int);

int main() {
    int y;
    for(y=1; y<=10; y++)
        printf("%d ", raiz_quadrada(y));
    return 0;
}

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

Função *main*

- Observe que **main** tem um tipo de retorno **int**.
- O valor de retorno de **main** é usado para indicar se o programa foi executado corretamente.
- Em versões anteriores de C, colocamos explicitamente **return 0**;
- no final de main — **0** indica que um programa foi executado com sucesso.
- O padrão C indica que **main** implicitamente retorna **0** se você omitir a instrução anterior.

Função *main* (Cont.)

- Você pode retornar explicitamente valores diferentes de zero de *main* para indicar que **ocorreu um problema durante a execução do seu programa.**
- Para obter informações sobre como relatar uma falha de programa, consulte a documentação de seu ambiente de sistema operacional específico.

Exercício

- Crie uma função **double** *ajuste(double salario, double x)*, que receba o salário de um funcionário e retorne o salário ajustado (aumento). Se o salário for menor que R\$ 900, o reajuste será de $1,5 * x\%$ caso contrário será de apenas $x\%$

Exercício

- Crie uma função em C **void** *inverter*(int x), que mostra na tela o número x invertido.

Argumentos de linha de comando em C

O C permite o seus programas receberem argumentos por linha de comando

```
int main(int argc, char *argv[]) {}
```

ou

```
int main(int argc, char **argv) {}
```


Argumentos de linha de comando em C

- **argc (ARGument Count)** é `int` e armazena o número de argumentos de linha de comando passados pelo usuário, incluindo o nome do programa.
- **argv (ARGument Vector)** é um vetor de ponteiros de caractere listando todos os argumentos.
- **Argv[0]** é o nome do programa, depois disso até argv [argc-1] cada elemento é argumento de linha de comando.

Argumentos de linha de comando em C

```
int main(int argc, char** argv){  
    printf("Voce entrou com %d argumentos\n", argc);  
    for (int i = 0; i < argc; ++i)  
        printf("%s \t ", argv[i]);  
    return 0;  
}
```

./main aula de LP



Voce entrou com 4 argumentos
./main aula de LP

Gerando números aleatórios

- Para gerar valores aleatórios, podemos usar a função **rand()**
 - retorna valores entre 0 e **RAND_MAX** (constante definida em `<stdlib.h>`)
 - **RAND_MAX** representa o valor máximo de um inteiro de 2 bytes (16 bits)

Gerando números aleatórios

- Exemplo para produzir número aleatórios entre intervalos:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i;
    for(i=0;i<=20;i++){
        printf("%d ", 1+(rand()%6));
        if(i%5==0)
            printf("\n");
    }
    return 0; }
```

Gerando números aleatórios

- O código anterior gera a mesma sequência de n°s aleatórios, em cada execução
- Para gerar n°s aleatórios diferentes, devemos usar a função **SRAND(unsigned int)**
 - Atribui um valor inteiro sem sinal à função rand(), como forma de semente

Gerando números aleatórios

- Para gerar n°s aleatórios diferentes, devemos usar a função **SRAND(unsigned int)**
 - Atribui um valor inteiro sem sinal à função rand(), como forma de semente
 - Com sementes diferentes, a função rand() gera n°s diferentes
 - Semente recomendada: **time(NULL)**
 - retorna o tempo acumulado (segundos) desde à meia-noite de 1 de janeiro de 1970

Executando um exemplo com rand() e com srand()

- Faça dois programas que simulem um dado de 6 lados
 - O 1º programa deve usar apenas rand()
 - O 2º deve usar srand()

Exercício

- Em um dado de 10 lados, cada lado possui 10% de chance de ser sorteado. Crie um programa em C para simular um dado viciado de 10 lados, em que:
 - O lado 1 possui 30% de chance de ser sorteado
 - O lado 5 possui 2 vezes mais chances de ser sorteado que o lado 3

Classes de armazenamento

As classes de armazenamento são usadas para descrever os recursos de uma variável / função.

Esses recursos incluem basicamente o **escopo**, a **visibilidade** e o **tempo de vida** que nos ajudam a rastrear a existência de uma determinada variável durante o tempo de execução de um programa.

Classes de armazenamento

O C utiliza 4 classes de armazenamento:

- `auto`
- `extern`
- `static`
- `register`

auto

- é a classe de armazenamento padrão para **todas as variáveis declaradas dentro de uma função ou bloco**
- **Tempo de vida:** são conhecidas como *automáticas*, pois são automaticamente criadas com o início da execução da função/bloco, e automaticamente destruídas ao término da função/bloco
- **Escopo:** **local**, acessível apenas dentro da função/bloco em que foi declarada

auto (cont.)

- Para explicitar uma classe desse tipo, basta por **auto** antes do nome da variável
 - Porém, como essa classe é default, raramente é usada
- Recebem **um valor de lixo por padrão** sempre que são declarados.

extern

- nos diz que a variável **é definida em outro lugar e não dentro do mesmo bloco onde é usada**
- pode ser considerada uma variável global, inicializada com um valor válido onde é declarada para ser usada em outro lugar.
- **Escopo:** pode ser acessada **em qualquer função / bloco**
- **Tempo de vida:** **todo o programa**
- Para explicitar uma variável global, de forma mais segura, pode ser usada a palavra **extern**
 - mas não é necessário

extern

- O objetivo principal de usar variáveis externas é que elas **podem ser acessadas entre dois arquivos diferentes** que fazem parte de um grande programa
- a palavra-chave extern é usada para **estender a visibilidade de variáveis / funções**.
- Como as funções são visíveis em todo o programa por padrão, o uso de **extern** não é necessário em declarações ou definições de funções. **Seu uso é implícito.**

register

- Uma variável *register* é armazenada diretamente em um registrador da CPU (caso haja registrador livre)
 - *registrador: é a memória dentro da própria CPU que armazena n bits. É o tipo de memória mais rápida, mas também a mais cara*
 - Portanto, seu acesso é mais rápido.
- Apenas variáveis **char** e **int** podem ser dessa classe
- Recomendadas quando se usa um variável várias vezes
 - por ex.: um contador dentro de um loop
 - `register int contador = 1;`

register

- É raramente utilizada, porque os compiladores modernos já fazem essa otimização automaticamente
- A palavra-chave **register** só pode ser usada com variáveis de tempo de armazenamento automático.
- **Escopo:** Local, dentro do bloco/função.
- **Tempo de vida:** término da função/bloco

static

- uma variável estática tem o escopo de uma local e a duração de uma global
 - só são acessíveis ao seu respectivo bloco, mas existem durante toda a execução do programa
- **variáveis normais são sobrescritas sempre que 'redeclaradas'**, por exemplo em funções
 - **variáveis estáticas sempre mantêm o seu valor, mesmo em chamadas diferentes da mesma função**
- para uma variável ser estática, deve ser antecedita por ***static***

Exemplo: façam o seguinte teste

```
void teste() {  
    int a = 10;  
    static int sa = 10;  
    a += 5;  
    sa += 5;  
    printf("a = %d, sa = %d\n", a, sa);  
}  
  
int main() {  
    int i;  
    for (i = 0; i <= 3 ; ++i)  
        teste();  
    return 0;  
}
```

- Qual a saída mostrada após a execução do seguinte código?

Recursividade

Ideia geral

- É um termo usado para descrever o processo de repetição de um objeto, de um jeito similar ao que já foi mostrado
- Por ex.:
 - quando 2 espelhos são apontados um para o outro



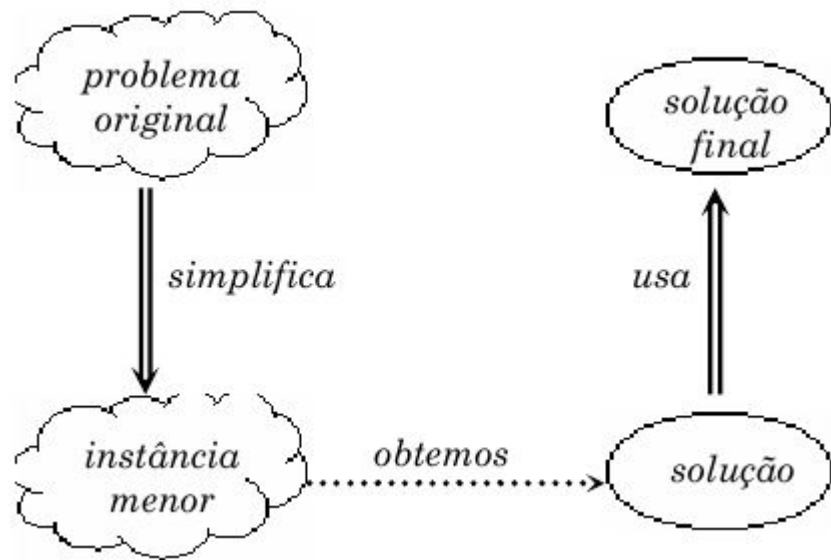
Recursividade na resolução de problemas

- É um princípio que nos permite obter a solução de um problema, por meio de versões menores de si mesmo
- Para aplicar esse princípio, devemos supor que sabemos a solução do problema menor
- Por ex.:
 - desejamos calcular a potência de 2^{11}
 - uma instância menor desse problema é 2^{10}
 - e para essa instância, “sabemos” a sua solução. 1024.
 - como $2 \times 2^{10} = 2^{11}$ então concluímos que $2^{11} = 2 \times 1024 = 2048$

Recursividade

Em resumo:

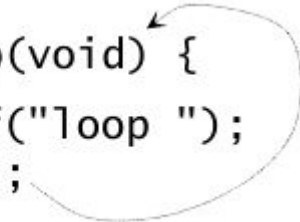
- 1) simplificamos o *problema original*, transformando-o numa *instância menor*;
- 2) *obtemos a solução* para essa instância menor, e a *usamos* para construir a *solução final*



Funções recursivas

- Em computação, o uso de recursividade se dá pelas funções recursivas, i.e. funções que chamam a si mesmas.

```
void loop(void) {  
    printf("loop ");  
    loop();  
}
```

A hand-drawn curved arrow originates from the `loop();` line within the function definition and points back to the opening curly brace of the `loop(void)` function, illustrating the recursive nature of the code.

- Aqui temos uma função recursiva descontrolada
- o programa mostra “loop ” por infinitas vezes

Para ser útil, uma função recursiva deve ter um ponto de parada, ou seja, deve ser capaz de interromper as chamadas recursivas e executar em tempo finito.

Funções recursivas

Ao definir uma função recursiva devemos identificar:

- 1º) **a base da recursão**, i.e. a instância mais simples do problema em questão
 - 2º) **o passo da recursão**, i.e. como simplificar o problema em questão.
- A base trata o caso mais simples, para o qual temos uma solução trivial, e o passo trata os casos mais difíceis, que requerem novas chamadas recursivas.

Voltando ao exemplo da potência

- No exemplo da potência, temos a seguinte definição recursiva:

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x.x^{n-1}, & \text{caso contrário} \end{cases}$$

- O caso base é aquele em que o expoente é 0. Qualquer n° elevado a 0 é 1
- Se diferente de 0, temos a função recursiva que garante chegarmos no caso base

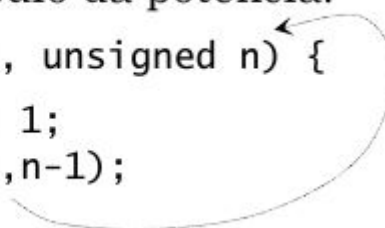
Voltando ao exemplo da potência

- Uma maneira de entender o funcionamento das funções recursivas é através de simulação por substituição

$$x^n = \begin{cases} 1 & \text{se } n = 0 \\ x.x^{n-1}, & \text{caso contrário} \end{cases}$$

Exemplo 4.11. Cálculo da potência.

```
double pot(double x, unsigned n) {  
    if( n=0 ) return 1;  
    return x * pot(x,n-1);  
}
```



Exemplo 4.12. Simulação por substituição.

```
p = pot(2,3)  
= 2 * pot(2,2)  
= 2 * 2 * pot(2,1)  
= 2 * 2 * 2 * pot(2,0)  
= 2 * 2 * 2 * 1  
= 8
```

Exercício

- Crie uma função recursiva que leia um inteiro n e mostre o n -ésimo termo da sequência fibonacci.

Exercício

- Crie uma função recursiva para ler um inteiro n e calcular $n!$.

Exercício

- Crie uma função recursiva que leia um inteiro n , e indique quantos dígitos possui n .
 - **obs.:** utilize uma *variável estática* como contador.