

# Unidade VI - Funções e Estrutura dos Programas

Disciplina Linguagens de Programação I  
Bacharelado em Ciência da Computação da Uerj  
Professores Guilherme Mota e Leandro Marzulo

## ANSI C

```
#include <stdio.h>
main ()
{
    printf("Hello World!");
}
```

# Que assuntos serão abordados nesta unidade?

- Programação Estruturada
  - Divisão e conquista
  - Unidades Funcionais
  - Codificação
- Funções
  - Declaração, protótipo e implementação
  - Escopo de variáveis
  - Variáveis locais
  - Variáveis globais
  - Variáveis static
  - Recursão
- Projeto *Top Down*
  - Introdução à análise de sistemas
  - Exemplo
  - Decomposição do problema
  - Níveis de abstração
- Organização de programas em C
  - Código em diversos fontes
  - Comandos avançados do pré-processador

# Introdução

# Para quê servem?

- Decompor o processamento em partes pequenas de complexidade e objetivos bem definidos
- Retirar do programa principal todos os detalhes da lógica de baixo nível
- Permitir a decomposição de um programa ao longo de diversos arquivos fonte
- Aumentar a legibilidade e produtividade
- Dividir, na fase de codificação, tarefas entre inúmeros desenvolvedores

# Ciclo do Software

- **Planejamento** – elicitação dos requisitos
- **Projeto** – definição dos algoritmos, da estrutura geral e dos componentes principais
- **Implementação** – escrita e teste das rotinas
- **Teste** – teste *offline* do sistema
- **Implantação** – entrada do software em produção
- **Manutenção** – correção de erros e realização das modificações impostas por fatores externos
- **Extensão** – desenvolvimento de novas funcionalidades

# Projeto Top-Down

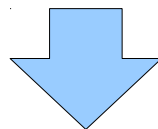
- Análise do sistema a partir do mais alto nível de abstração
- No início, abstrair todos os detalhes da linguagem de programação
- Começando com o uso da linguagem natural, por exemplo
- Refinamento sucessivo do problema
- A cada iteração aumenta o detalhamento da solução
- Culminando na escrita do código fonte

# Exemplo Top-Down

# Exemplo Projeto Top-Down

- Programa Processa Texto:
  - Dado um texto, retorna todas as linhas que contiverem a sequência de caracteres “ould”

```
Ah Love! could you and I with Fate conspire  
To grasp this sorry Scheme of Things entire,  
Would not we shatter it to bits -- and then  
Re-mould it nearer to the Heart's Desire!
```



```
Ah Love! could you and I with Fate conspire  
Would not we shatter it to bits -- and then  
Re-mould it nearer to the Heart's Desire!
```



# Exemplo Projeto Top-Down

- Programa Processa Texto:
  - Dado um texto, retorna todas as linhas que contiverem a sequência de caracteres “ould”

```
Enquanto houver nova linha a ser lida  
  Se Linha contiver "ould"  
    Imprima linha  
Fim
```

# Exemplo Projeto Top-Down

- Programa Processa Texto:
  - Dado um texto, retorna todas as linhas que contiverem a sequência de caracteres “ould”

```
Ler Linha
Enquanto (houver linha)
Início
    Se Linha contiver "ould"
    Então
        Imprima linha
    Ler linha
Fim
```

# Exemplo Projeto Top-Down

- Programa Processa Texto:
  - Dado um texto, retorna todas as linhas que contiverem a sequência de caracteres “ould”

```
LeLinha (Linha)
Enquanto (Linha > 0)
Início
    Se Localiza (Linha, "ould") >= 0
    Então
        Imprima (Linha)
        LeLinha (Linha)
Fim
```

# Exemplo Projeto Top-Down

- Programa Processa Texto:
  - Dado um texto, retorna todas as linhas que contiverem a sequência de caracteres “ould”

## **LeLinha:**

Ler e armazenar os caracteres obtidos do *stream* de entrada até que seja encontrado um '\n'

## **Localiza:**

Retorna a posição da primeira ocorrência de Str2 em Str1

# Declaração de funções em C

# Declaração de uma função

- Protótipo da função

```
Tipo_Retorno nome_func(Tipos_Args);
```

- Implementação da função

```
Tipo_Retorno nome_func(argumentos)
{
    Declarações e instruções;
    return expressão;
}
```

# **Código do projeto processa texto**

# Exemplo Projeto Top-Down

- Programa Processa Texto:
  - Dado um texto, retorna todas as linhas que contiverem a sequência de caracteres “ould”

```
main()
{
    char line[MAXLINE];
    int found = 0;
    while (getline(line, MAXLINE) > 0)
        if (strindex(line, "ould") >= 0)
        {
            printf("%s", line);
            found++;
        }
    printf("Found %d lines.\n", found);
}
```



# Exemplo Projeto Top-Down

## LeLinha:

Ler e armazenar os caracteres obtidos do *stream* de entrada até que seja encontrado um '`\n`'

```
int getline(char s[], int lim)
{
    int c, i;
    i = 0;
    while (--lim>0 && (c=getchar()) !=EOF && c!='\n')
        s[i++] = c;
    if (c == '\n')
        s[i++] = c;
    s[i] = '\0';
    return i;
}
```

# Exemplo Projeto Top-Down

## Localiza:

Retorna a posição da primeira ocorrência de t em s

```
int strindex(char s[], char t[])
{
    int i, j, k;
    for (i = 0; s[i] != '\0'; i++)
    {
        for (j=i, k=0; t[k] != '\0' && s[j] == t[k]; j++, k++)
            ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

# **Trabalho 4**

## **Projeto top-down**

## Trabalho 4: Projeto Top-Down

- Implemente um programa na linguagem C que processe a folha de pagamento de uma empresa com dez funcionários. Os funcionários trabalham por demanda. A cada mês o salário precisa ser recalculado em função das horas efetivamente trabalhadas.
- Há um arquivo texto sobre os funcionários com nome, matricula, endereço, cpf, cod\_banco, agência, conta, valor\_hora
- Todo mês é a gerência produz um arquivo com a matrícula e o número de horas trabalhadas
- A folha de pagamento gera um arquivo texto

# Política de Escopo de variáveis

# Funções e Regras de Escopo

- A priori todo identificador somente é conhecido no arquivo fonte atual e a partir de sua declaração
- **#include**: para que funções ou tipos sejam reconhecidos fora do arquivo em que foram declarados
- A vida útil de uma variável é definida pelo local de sua declaração
  - **escopo local**: funções ou blocos
  - **escopo global**: fora de funções ou blocos
- **extern**: permite que uma variável global seja reconhecida fora do arquivo em que foi declarada

# Variáveis Globais e extern

myExtern.h

```
#ifndef _MYEXTERN_H
#define _MYEXTERN_H
    int i = 0;
#endif
```

testExtern.c

```
#include <stdio.h>
#include "myExtern.h"
main()
{
    extern int i;
    i++;
    printf("\nO valor de i é: %d\n", i);
}
```

# Funções e Regras de Escopo

- **static:**

- **Variáveis globais e funções:** tem sua visibilidade limitada ao arquivo fonte em que foram declaradas
- **Variáveis locais:** somente a primeira chamada à função aloca a variável, as demais chamadas compartilham esta mesma posição de memória



# Variáveis Globais static

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

#define BUFSIZE 100

static char buf[BUFSIZE]; /* buffer for
ungetch and getch only */

static int bufp = 0;      /* next free position
in buf for ungetch and getch only */

int getch(void);          /* get a (possibly
pushed-back character */

void ungetch(int); /* push character back on
input */
```

# Variáveis Locais static

```
#include <stdio.h>
void test_LocVar_Static(void);

main ()
{
    test_LocVar_Static();
    test_LocVar_Static();
    test_LocVar_Static();
}

void test_LocVar_Static(void)
{
    static int nChamadas = 0;
    nChamadas++;
    printf("\nEsta é a chamada %d\n", nChamadas);
}
```

# **Código em diversos arquivos**

# Subdivisão do Código Fonte

main.c

```
#include <stdio.h>
#include <stdlib.h>
#define MAXOP 100
#include "calc.h"
main () {
    ...
}
```

calc.h

```
#define NUMBER '0'
void push (double);
double pop (void);
int getop(char[]);
int getch(void);
void ungetch(int);
```

getch.c

```
#include <stdio.h>
#include "calc.h"
#define BUFSIZE 100
char buf[BUFSIZE]
int bufp = 0;
int getch(double) {
    ...
}
void ungetch(int) {
    ...
}
```

getop.c

```
#include <stdio.h>
#include <ctype.h>
#include "calc.h"
int getop(char[]) {
    ...
}
```

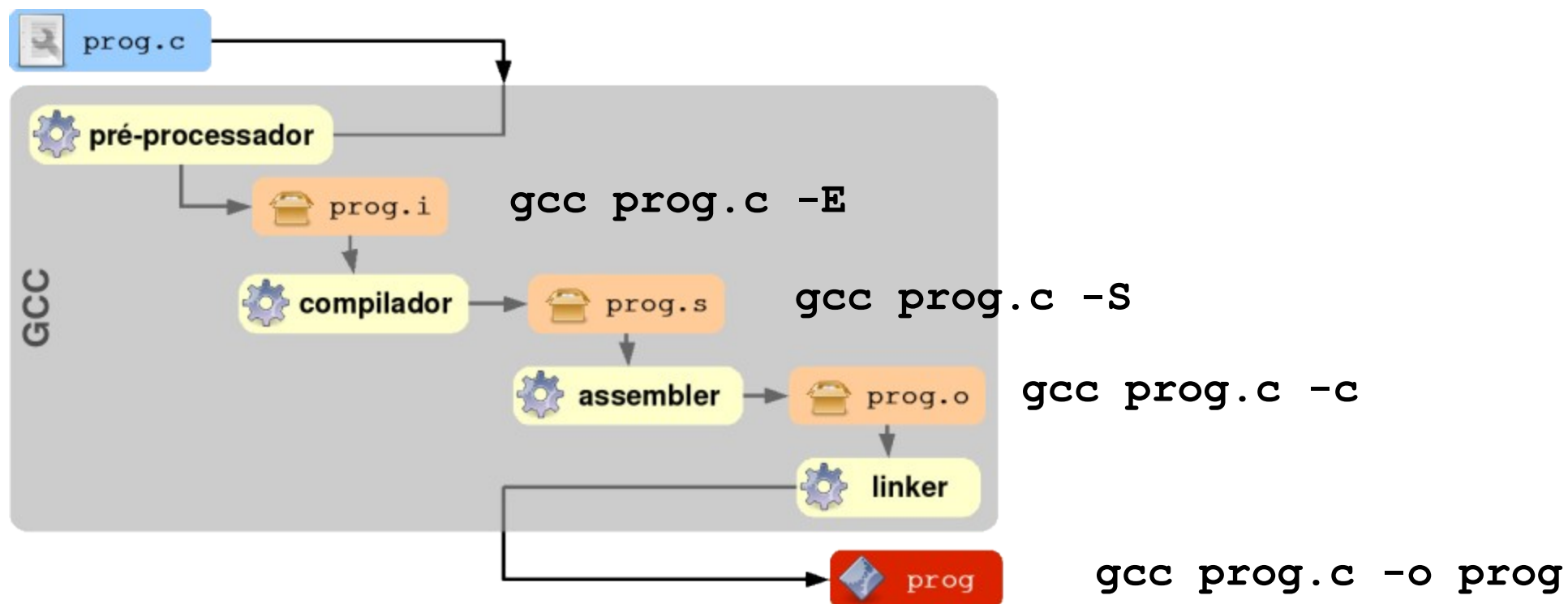
stack.c

```
#include <stdio.h>
#include "calc.h"
#define MAXVAL 100
int sp = 0;
double val[MAXVAL]
void push(double) {
    ...
}
double pop(void) {
    ...
}
```

# Pré-processador

# Compilação Passo a Passo

```
#include <stdio.h>
main(void)
{
    printf("Hello World!\n");
}
```



# Comandos de Pré-processador

- `#include`
- `#define`
- `#undef`
- `##`
- `#if`   `#ifdef`   `#ifndef`   `#endif`
- `#else`   `#elseif`   `#elif`

# #include

- Copia o conteúdo do arquivo especificado

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

```
#include "calc.h"
```



# #define

- Substituição de macro simples

```
#define macro substituto  
#define forever for(;;)
```

- Substituição de macros com argumentos

```
#define max(A,B) ((A)>(B) ? (A) : (B))
```

```
x = max (p+q, r+s) ;
```

```
x = ((p+q)>(r+s) ? (p+q) : (r+s)) ;
```

```
max (i++, j++) ; /* ERRO* /
```

```
#define square(x) x * x
```

```
square(z+1) /* ERRO* /
```

# #define

- Macro com argumentos

```
#define dprint(expr) printf(#expr " = %g\n", expr)
```

```
dprint (x/y);
```

```
printf ("x/y" " = %g\n", x/y);
```

```
printf ("x/y = %g\n", x/y);
```

- Concatenação de argumentos em macros

```
#define paste(front, back) front ## back
```

```
paste(name, 1);
```

```
name1;
```

# #undef

- Apaga a definição de uma macro

```
#define max(A,B)  ( (A) > (B)  ?  (A) : (B) )
```

```
x = max (p+q, r+s) ;
```

```
#undef max
```

`#if #ifdef #ifndef #else #endif`

- Inclusão condicional no código fonte

```
#if !defined(HDR)
#define HDR
```

```
/* conteúdo condicionalmente
incluído */
```

```
#endif
```

`#if #ifdef #ifndef #else #endif`

- Inclusão condicional no código fonte

```
#ifndef HDR  
#define HDR
```

```
/* conteúdo condicionalmente  
incluído */
```

```
#endif
```

#if #ifdef #ifndef #else #endif

- Inclusão condicional no código fonte

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif
```

## Exercício

- Defina uma macro `swap(t,x,y)` que troque os valores dos argumentos `x` e `y` ambos do tipo `t`. (dica: utilize um bloco)
- Escreva uma função que calcule o fatorial de um número
- Escreva uma função que imprima o conteúdo de uma string de trás pra frente

# Recursão



# Recursão

- Serve para implementar equações recorrentes
- Pode gerar soluções e algoritmos mais legíveis do que os equivalentes não-recursivos
- Cada vez que uma função chama a si mesma há uma troca de contexto
  - A chamada em foco é empilhada e a nova chamada passa à execução
  - Quando uma chamada chega ao fim a função chamadora retorna ao foco, desempilhando o respectivo registro de ativação
- Uma função recorrente pode gerar *overheads* computacionalmente intratáveis
- Quando a recursão gerar chamadas repetidas, seu uso não é recomendado

# Exemplo: Fatorial

- Equação recorrente:

$$fatorial(n) = \begin{cases} 1 & \text{para } n=1 \\ n.fatorial(n-1) & \forall n>1 \end{cases}$$

- Código:

```
int fatorial(int n)
{
    if (i==1)
        return 1;
    else
        return n*fatorial (n-1) ;
}
```

# Recursividade de Cauda

- Caracterizada pela chamada recursiva ao final do código.

```
P(args)
{
    if (B)
    {
        S;
        P();
    }
}
```

- Algoritmos recursivos com esta característica são facilmente versionados em não recursivos

```
P(args)
{
    for (x=x0; B; x++)
    {
        S;
        P();
    }
}
```

# Fatorial Não Recursivo

```
int fatorial(int n)
{
    int fat=1;
    int i;
    for(i=1; i<=n; i++)
        fat *= i;
    return fat;
}
```

# Números de Fibonacci

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \text{ para } n \geq 2 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

# Números de Fibonacci Recursivo

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \text{ para } n \geq 2 \end{cases}$$

```
int fibonacci(int n)
{
    if (n < 2 && n >= 0)
        return n;
    else
        return fibonacci(n-1) + fibonacci(n-2);
}
```

# Números de Fibonacci Não-recursivo

```
int fibonacci(int n)
{
    int i=1, k, Fib=0;
    for(k=1; k<=n; k++)
    {
        Fib+=i;
        i=Fib-i;
    }
    return Fib;
}
```

# Comparação Fibonacci recursivo e não-recursivo

$$\begin{cases} f_0 = 0 \\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \text{ para } n \geq 2 \end{cases}$$

<i>n</i>	10	20	30	50	100
<i>recursivo</i>	8 ms	1 s	2 min	21 dias	10 <sup>9</sup> anos
<i>iterativo</i>	1/6 ms	1/3 ms	1/2 ms	3/4 ms	1,5 ms



# Exercício 1 - Análise das chamadas de funções recursivas

- (a) Modifique a função recursiva Fibonacci para que a mesma mostre a quantidade de chamadas recursivas. Use variáveis locais estáticas.
- (b) Use um vetor static para armazenar a quantidade de chamadas de Fibonacci para cada valor de  $n$

## Exercício 2 - Recursão em string

Faça uma função recursiva que imprima de traz para a frente o conteúdo de uma string.

## Exercício 3 - Conversão Arquivo Texto para Binário

Faça uma função recursiva imprima uma string equivalente ao valor octal do número inteiro fornecido como entrada para sua função.

# Trabalho - Flood fill

Faça um programa que mostre passo a passo a execução de uma versão recursiva do algoritmo de processamento de imagens flood fill.

**Fim**