

Unidade V - Funções e Estrutura dos Programas

Disciplina Linguagens de Programação I
Bacharelado em Ciência da Computação da Uerj
Professores Guilherme Abelha e Gilson Costa

ANSI C

```
#include <stdio.h>
int main ()
{
    printf("Hello World!");
    return 0;
}
```

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 de software
 - 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

Palavras de Ordem da Modularização

- Racionalidade
- Custo do projeto
- Melhoria e controle de qualidade do software
- Reutilização de código
- Controle da complexidade
- Simplificação o teste do software
- Documentação
- Organização
- Divisão e conquista

Objetivos e Benefícios da Modularização

- Decompor o processamento em subpartes com complexidade limitada 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 a produtividade dos desenvolvedores
- Dividir, na fase de codificação, tarefas dentre inúmeros desenvolvedores

Objetivos e Benefícios da Modularização

- Promover o reuso dos módulos.
- Estruturar os algoritmos em partes fechadas.
- Favorecer a elaboração, a documentação e o teste.
- Tornar a manutenção mais simples e focalizada.
- Permitir o teste das partes módulos em separado
- Aumento da legibilidade
- Facilitar a manutenção e a extensão do software
- Implementação de bibliotecas reutilizáveis

Processo de Desenvolvimento

Ciclo Clássico do Software

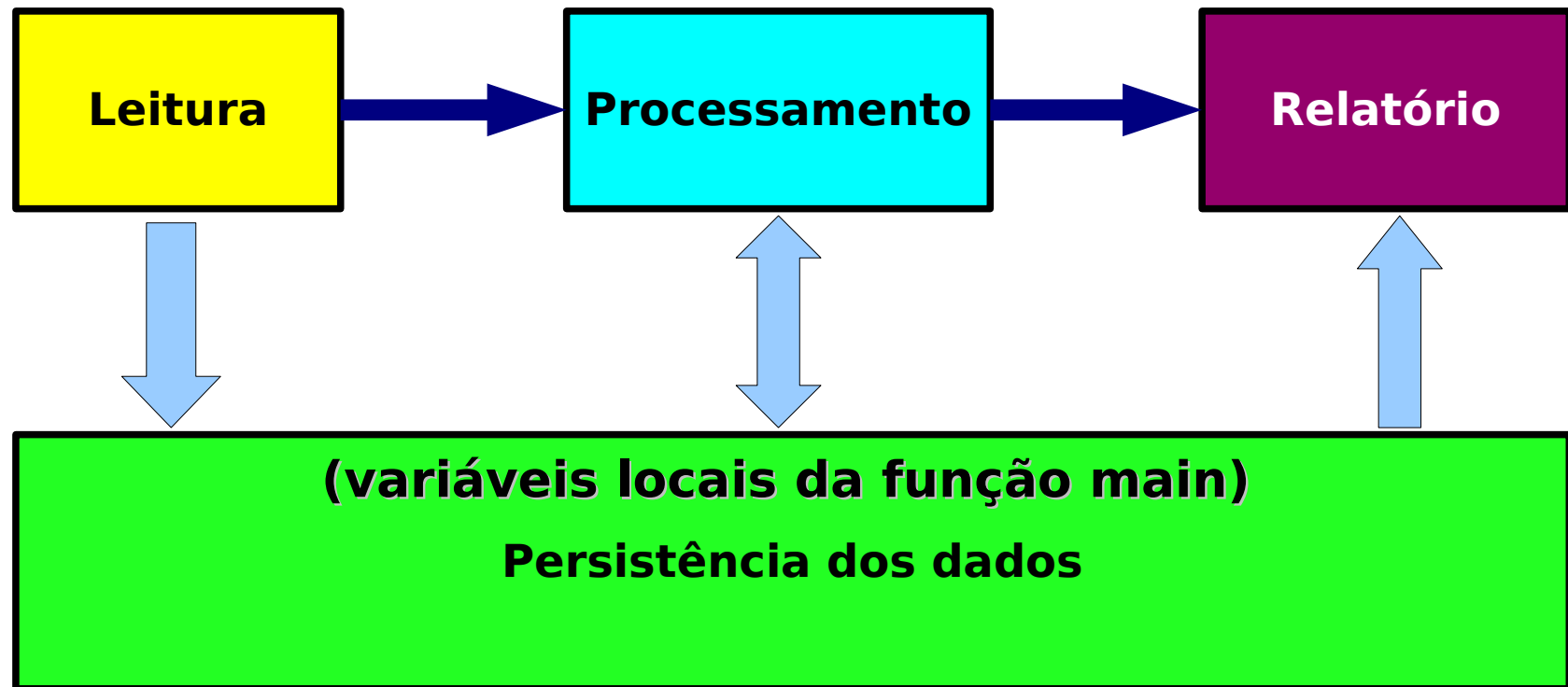
- **Planejamento** – elicitação dos requisitos
- **Projeto** – definição dos algoritmos, da estrutura geral e dos componentes principais
- **Implementação** – escrita e teste das rotinas
- **Teste** – teste *offline* do sistema
- **Implantação** – entrada do software em produção
- **Manutenção** – correção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

Ciclo do Software em LP1

- **Planejamento** – elicitação dos requisitos
- **Projeto** – definição dos algoritmos, da estrutura geral e dos componentes principais
- **Implementação** – escrita e teste das rotinas uma-a-uma
- **Teste** – teste *offline* da tarefa
- **Implantação** – obtenção da nota no AVA

Metodologia de Desenvolvimento

Estrutura Geral dos Programas da Disciplina



Ciclo do Software em LP1

- **Planejamento** – elicitação dos requisitos
- **Projeto** – definição dos algoritmos, da estrutura geral e dos componentes principais
- **Implementação** – escrita e teste das rotinas uma-a-uma
- **Teste** – teste *offline* da tarefa
- **Implantação** – obtenção da nota no AVA

Ciclo do Software em LP1: Planejamento

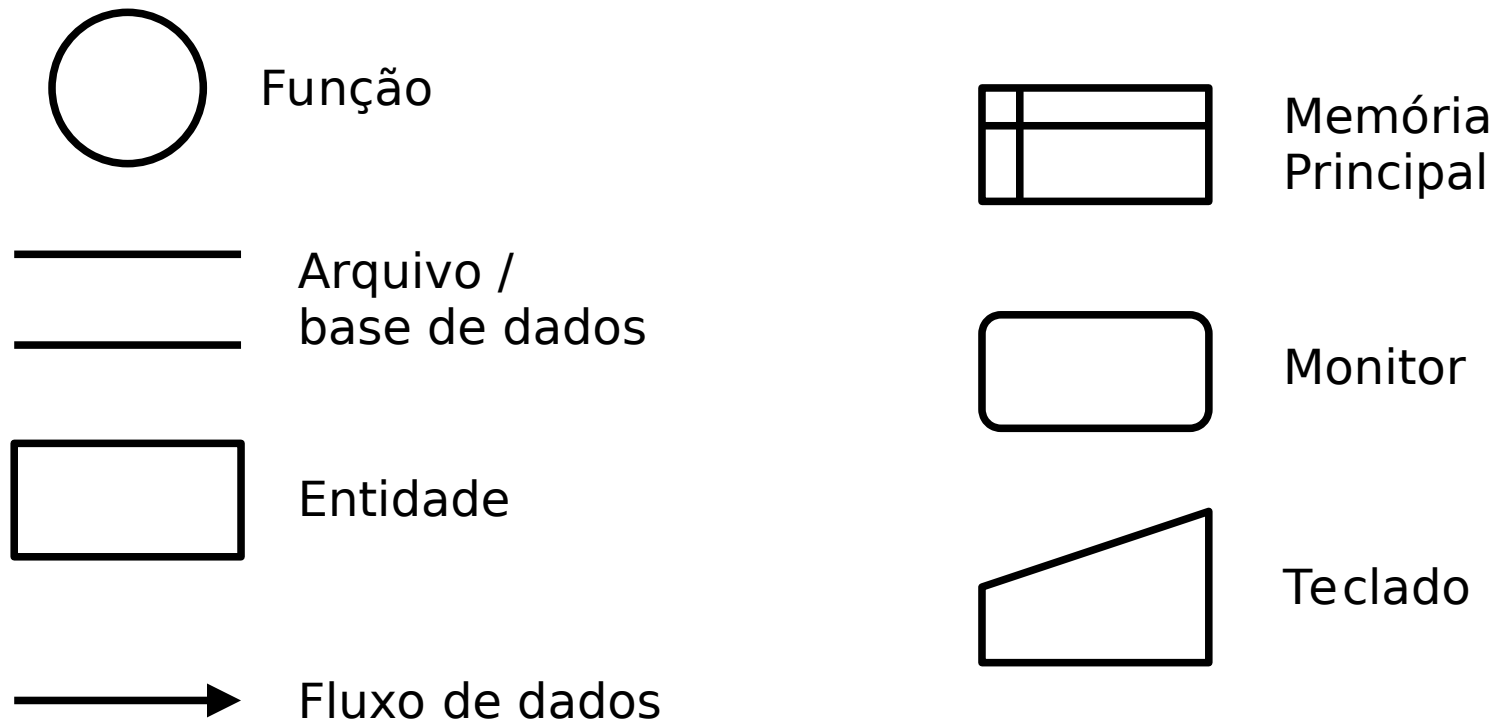
- **Organizar os requisitos:**
 - Ordenar os requisitos em ordem de precedência.
 - Quais podem ser implementados primeiro?
 - Manter ciclos de implementação de um requisito e um teste.
- **Requisitos funcionais:**
 - O que o software precisa fazer?
- **Requisitos não-funcionais:**
 - O que vai ser necessário de infraestrutura?
 - Quais estruturas e formatos de dados utilizar?

Ciclo do Software em LP1: Projeto

- **Elicitação dos requisitos:**
 - Sistematizar todas as demandas para o problema.
 - O que o software precisa fazer?
 - O que vai ser necessário de infraestrutura?
 - Quais estruturas e formatos de dados utilizar?
- **Diagrama de fluxo de dados**
- **Diagrama hierárquico**

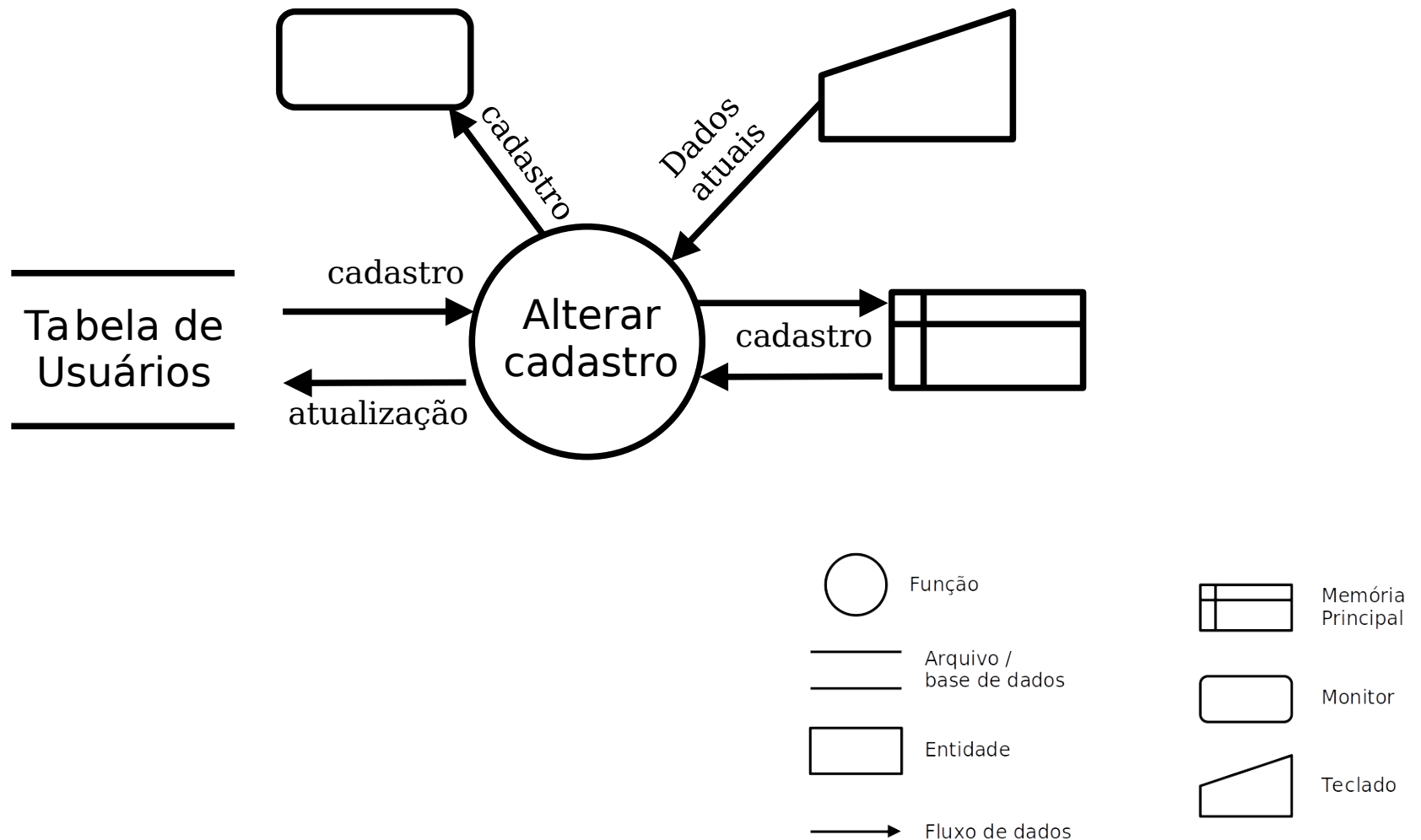
Ciclo do Software em LP1: Projeto

- **Diagrama de fluxo de dados**



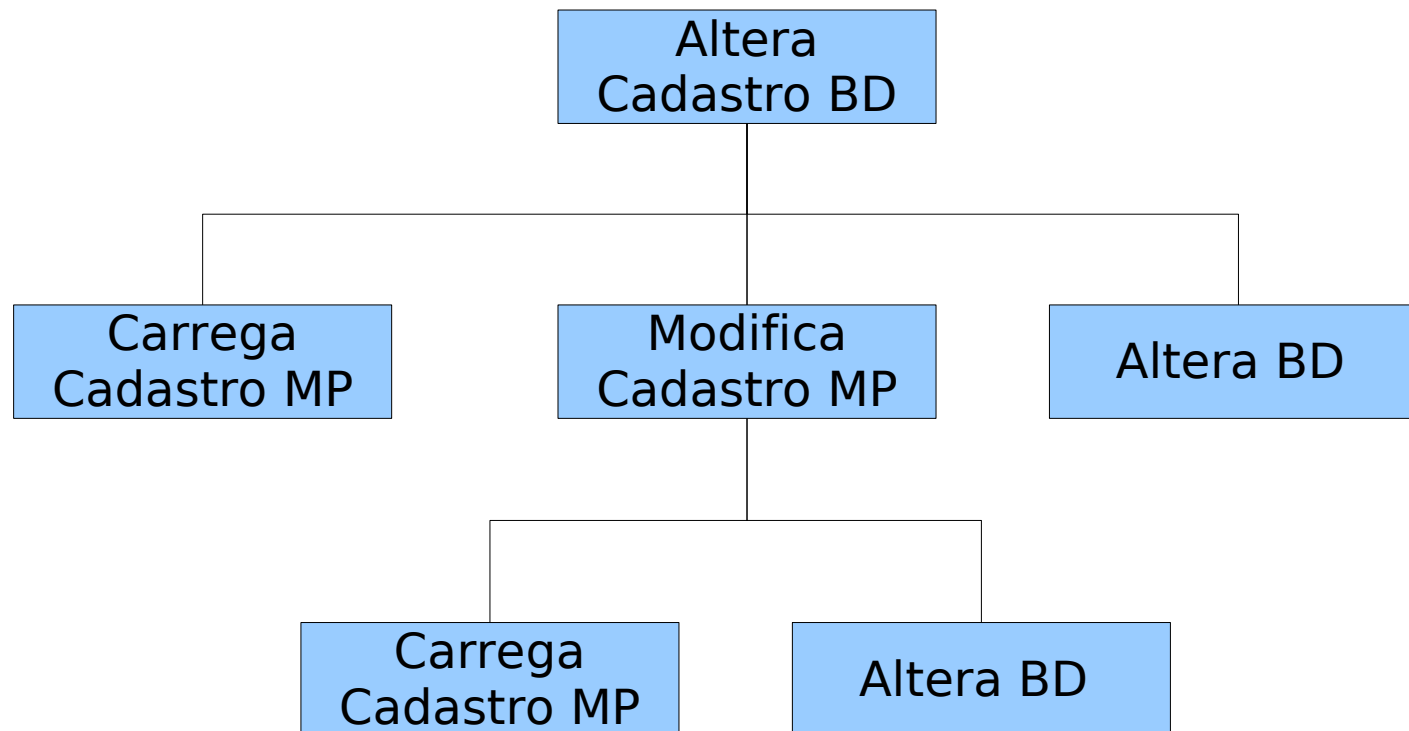
Ciclo do Software em LP1: Projeto

- Diagrama de fluxo de dados



Ciclo do Software em LP1: Projeto

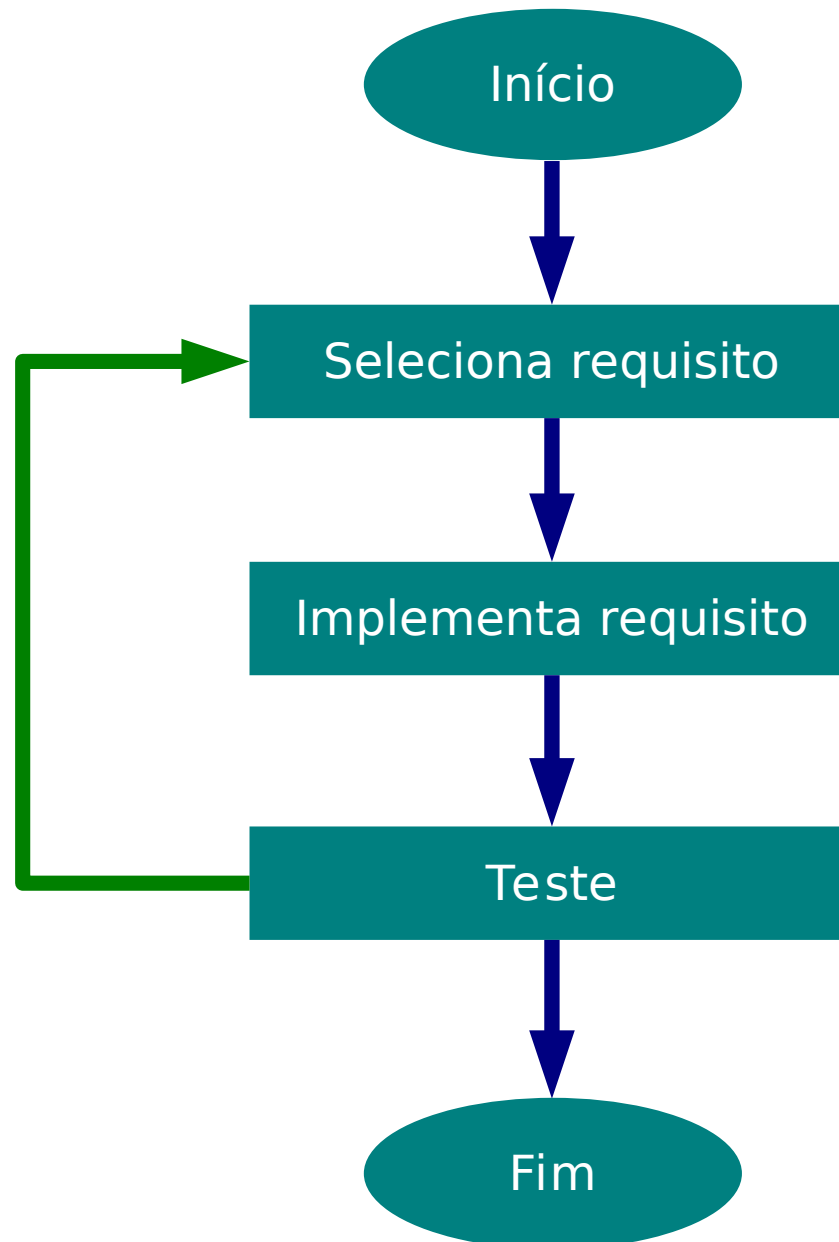
- **Diagrama hierárquico**



Ciclo do Software em LP1: Projeto Top-Down

- Análise do problema e dos subproblemas a partir do mais alto nível de abstração.
- No início do projeto, abstrair os detalhes.
- Obtenção do diagrama hierárquico.
- Aprofundamento da compreensão do problema.
- A cada iteração aumenta-se o detalhamento da solução e do diagrama hierárquico.
- Projeto da assinatura das funções.
- Verificação da consistência da comunicação entre os módulos

Ciclo do Software em LP1: Implementação



Ciclo do Software em LP1: Implementação



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”

```
int 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);
    return 0;
}
```


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;
}
```

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"
int 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);

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

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


Exercício U5.1 - 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 a gerência produz um arquivo com a matrícula e o número de horas trabalhadas
- A folha de pagamento gera um arquivo texto

Código em diversos arquivos

Subdivisão do Código Fonte

progPag076.c

```
#include <stdio.h>
#include <stdlib.h>
#define MAXOP 100
#include "calc.h"
int 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[]){
    ...
}
```

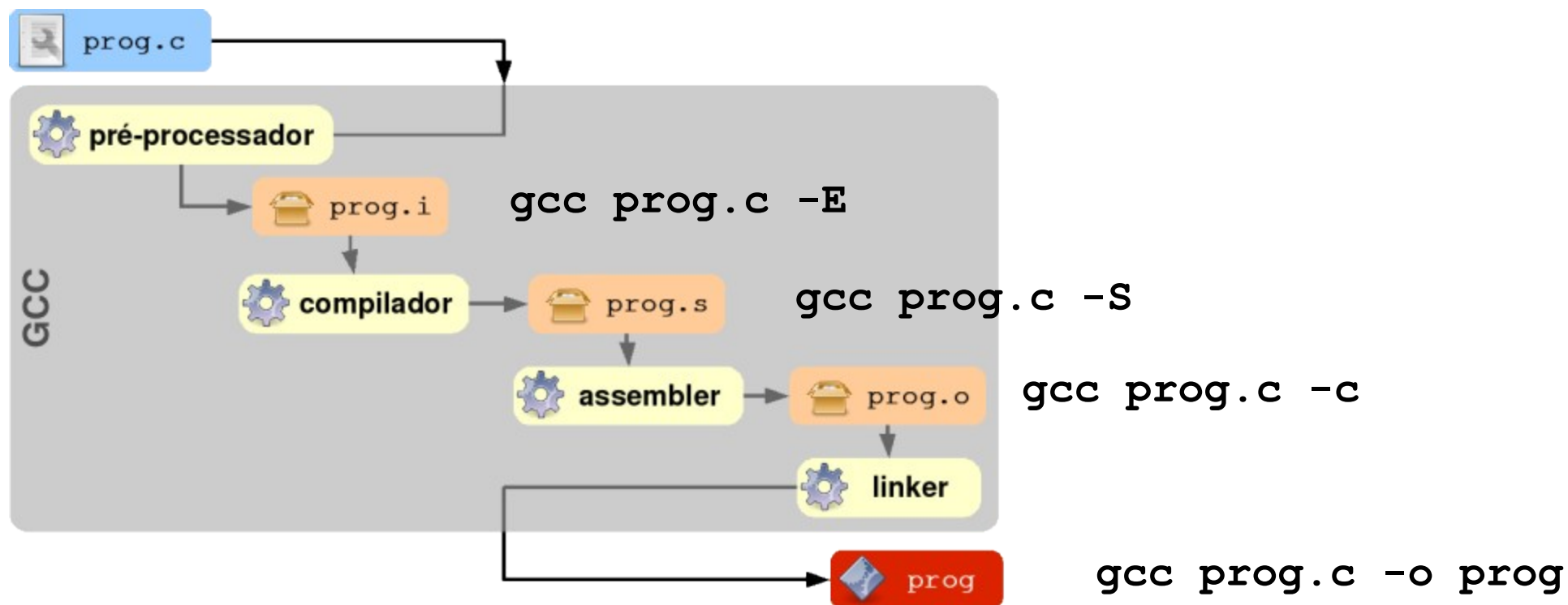
stack.c

```
#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>
int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```



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

```
#include <stdio.h>
#ifdef __unix__
    #define OS "UNIX"
#elif defined(_win32)
    #define OS "WINDOWS"
#elif defined(__APPLE__)
    #define OS "MacOS"
#endif

int main()
{
    printf(OS "\n");
}
```

Maiores detalhes em:

http://nadeausoftware.com/articles/2012/01/c_c_tip_how_use_compiler_predefined_macros_detect_operating_system

Exercício de Aula - Macro de pré-processador

- Defina uma macro `swap (t, x, y)` que troque os valores dos argumentos `x` e `y` ambos do tipo `t`.
(dica: utilize um bloco)

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 (n==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;  
        Q;  
    }  
}
```

Fatorial Não Recursivo

```
int fatorial(int n) {  
    int fat;  
    for(fat=1; n>0; n--)  
        fat *= n;  
    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);  
}
```

Número de Chamadas

f_N	Número de Chamadas														
	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}	f_{14}
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
3	1	2	1	1	0	0	0	0	0	0	0	0	0	0	0
4	2	3	2	1	1	0	0	0	0	0	0	0	0	0	0
5	3	5	3	2	1	1	0	0	0	0	0	0	0	0	0
6	5	8	5	3	2	1	1	0	0	0	0	0	0	0	0
7	8	13	8	5	3	2	1	1	0	0	0	0	0	0	0
8	13	21	13	8	5	3	2	1	1	0	0	0	0	0	0
9	21	34	21	13	8	5	3	2	1	1	0	0	0	0	0
10	34	55	34	21	13	8	5	3	2	1	1	0	0	0	0
11	55	89	55	34	21	13	8	5	3	2	1	1	0	0	0
12	89	144	89	55	34	21	13	8	5	3	2	1	1	0	0
13	144	233	144	89	55	34	21	13	8	5	3	2	1	1	0
14	233	377	233	144	89	55	34	21	13	8	5	3	2	1	1

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

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

Idade do universo:
13,7x10⁹ anos

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

fib₄₅ recursivo realiza 3.672.623.805 de chamadas

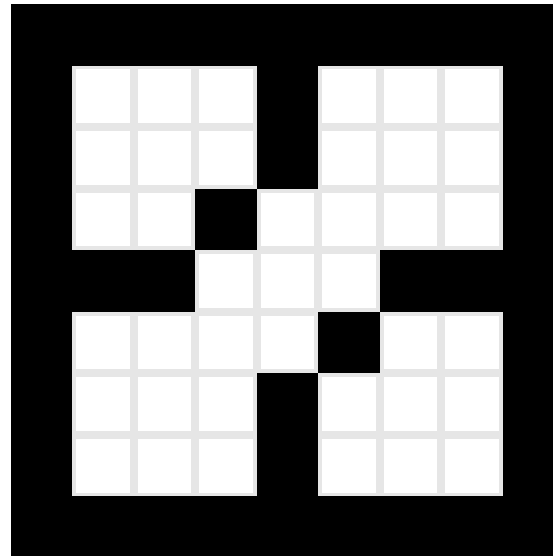
Análise das chamadas de funções recursivas

- (a) Modifique a função recursiva Fibonacci para que a mesma mostre a quantidade de chamadas recursivas. Use variáveis locais estáticas.
- (b) Use um array (no escopo da função main) para armazenar a quantidade de chamadas de Fibonacci para cada valor de n

Exercícios

Exercício U5.2 - Flood fill

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



Exercício U5.3 - Produto Escalar

Faça uma função recursiva que calcule o produto escalar entre dois vetores.

Exercício 5.4 - Conversão Inteiro para Octal

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

Fim