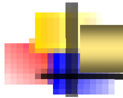
Macros e funções

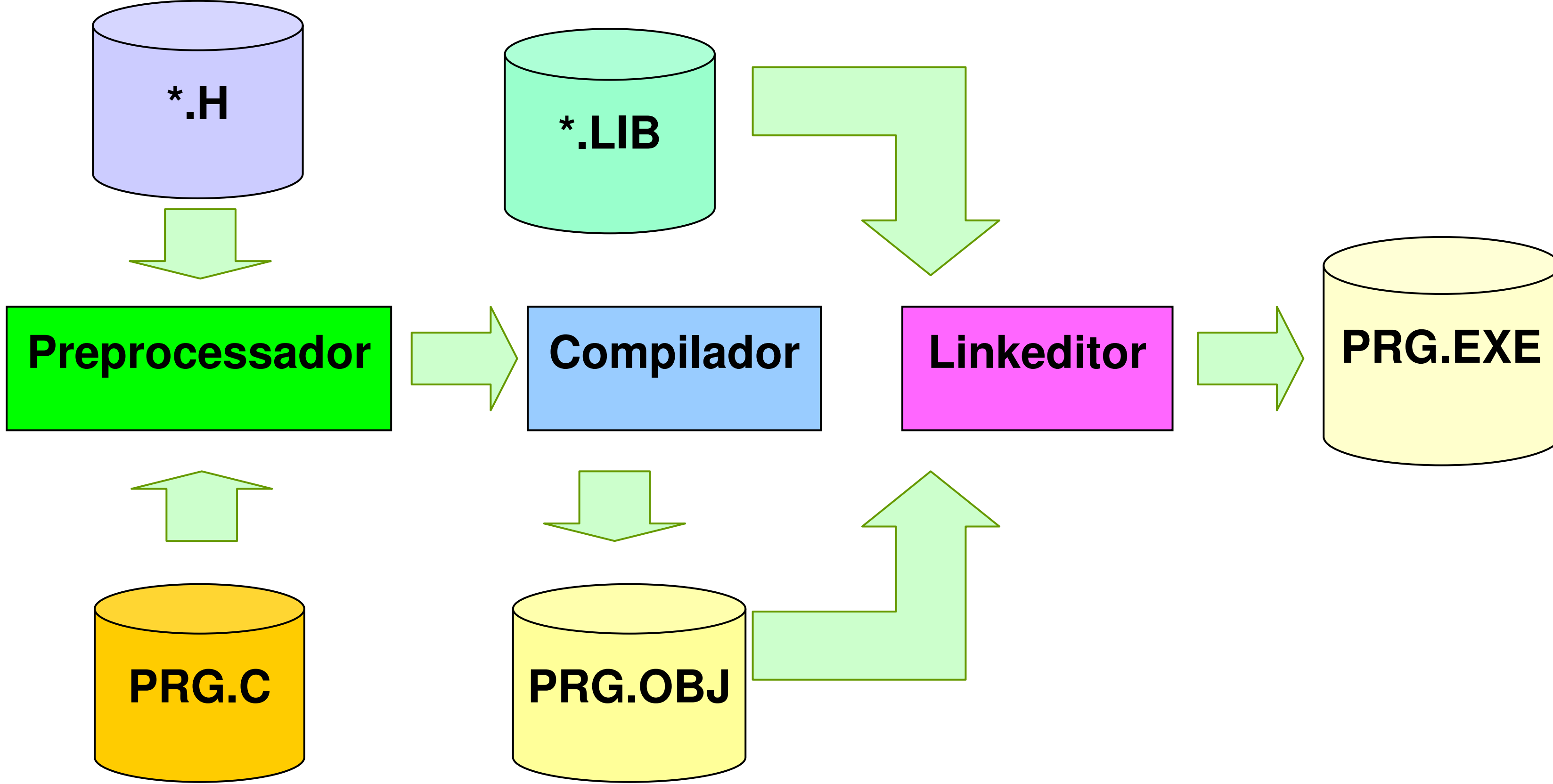


Prof. Dr. Silvio do Lago Pereira

Departamento de Tecnologia da Informação

Faculdade de Tecnologia de São Paulo

Fases na criação de um executável



Preprocessamento

arq.h

```
#define MSG "oi"
```

prog.c

```
#include <arq.h>
main() {
   puts(MSG);
}
```

processa #include

prog.c

```
#define MSG "oi"
main() {
   puts(MSG);
}
```

processa o #define

prog.c

```
main() {
   puts("oi");
}
```

Compilação e linkedição

prog.c

```
main() {
    puts("oi");
}
```

compilação

Supondo uma máquina hipotética!

prog.obj

```
0100 push ds:0000
0106 call cs:010A
0109 ret
```

linkedição

prog.exe

```
0100 push ds:0000
0106 call cs:0109
0109 ret
010A mov [sp], dx
010E mov 30, ah
0112 int 21
0115 ret
```

A diretiva #define

Substituição simples:

```
#define diga puts
#define oi   "\nOlá, tudo bem?"

main() {
   diga(oi);
}
```

```
main() {
   puts("\nOlá, tudo bem?");
}
```

Exercício 4.1

Inclua diretivas no programa, para que ele possa ser compilado.

```
#include <stdio.h>

programa
inicio
   diga("Olá!");
fim
```

```
#define programa main()
#define inicio {
#define fim }
#define diga printf
```

A diretiva #define

Substituição parametrizada (macro):

```
#define quad(n) n*n

main() {
   printf("%d", quad(2) );
}
```

```
main() {
   printf("%d", 2*2 );
}
```

Problemas com macros

```
#define quad(n) n*n

main() {
   printf("%d", quad(2+3) );
}
```

```
main() {
   printf("%d", 2+3*2+3 );
}
```

Problemas com macros

```
#define quad(n) (n)*(n)

main() {
   printf("%d", 100/quad(2+3) );
}
```

```
main() {
   printf("%d", 100/(2+3)*(2+3) );
}
```

Versão final da macro quad()

```
#define quad(n) ((n)*(n))
```

REGRA: utilize sempre uma expressão completamente parentetizada, ao definir uma macro!

A diretiva #include

`#include` inclui no código-fonte uma cópia do conteúdo do arquivo indicado.

Arquivos de inclusão padrão:

```
#include <...>
```

Arquivos de inclusão do usuário:

```
#include "..."
```

Definição e uso de funções

```
tipo nome(lista de parâmetros) {
    declarações;
    comandos;
}
```

Note que:

- o tipo de resposta pode ser **`void`**
- a lista de parâmetros pode ser **`void`**

Funções que não devolvem resposta

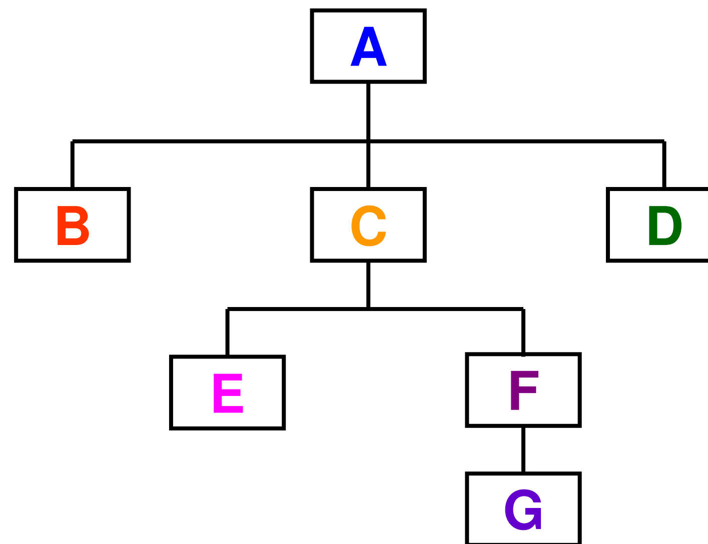
```
void alarme(void) {
   int f;
   for(f=100; f<=5000; f+=20) {
      sound(f);
      delay(6000);
   }
   nosound();
}
```

Declaração de funções: protótipos

```
#include <stdio.h>
#include <conio.h>
#define  SENHA 1234
void alarme(void); /* protótipo */
void main(void) {
   int s;
   printf("\nSenha: "); scanf("%d", &s);
   if( s != SENHA ) {
      printf("\nSenha inválida!");
      alarme();
   }
   else printf("\nSenha Ok!");
}
```

Convenção

- Por uma questão de conveniência, evitaremos o uso de protótipos.
- Isso é possível se cada função for definida antes de ser chamada.
- Podemos usar um DHF para escolher a melhor ordem de codificação.



Melhor ordem de codificação: B, E, G, F, C, D, A

Funções que recebem parâmetros

```
void linha(int x, int y, int c) {
  int i;
  gotoxy(x,y);
  for(i=0; i<c; i++)
    putch(196);
}
```

Note que mesmo que todos os parâmetros sejam do mesmo tipo, é preciso declará-los separadamente!

Funções que devolvem resposta

```
float hip(float a, float b) {
   float h;
   h = sqrt( pow(a,2)+pow(b,2) );
   return h;
}
```

Note o comando `return` só pode devolver uma resposta, interrompendo a execução da rotina imediatamente!

Exercício 4.11

Codifique uma função que calcule a raiz quadrada r de x, usando o método de Newton:

1º chutamos uma valor inicial para r igual a x/2.

2º caso $|r^2-x|$ seja inferior a 0.001, r é a resposta.

3º caso contrário, aproximamos o valor de r tomando $r = (r^2+x) / 2r$ e voltamos ao 2º passo.

Observação

Antes de criar a função que calcula a raiz quadrada, crie funções para obter o quadrado e o módulo (valor absoluto) de um número.

Solução

```
double quad(double n) { return n*n; }

double abs(double n) { return n>0 ? n : -n; }

double raiz(double x) {
  double r = x/2;
  while( abs(quad(r)-x)>= 0.001 )
    r = (quad(r)+x)/(2*r);

  return r;
}

void main(void) {
   printf("\n %.1lf", raiz(81) );
}
```

Classes de armazenamento

- **automática: `auto`**
  - definida dentro de uma função
  - acessível somente à essa função
  - existe somente durante a execução dessa função
- **externa: `extern`**
  - definida fora de qualquer função do programa
  - acessível a partir do ponto da sua declaração
  - existe durante toda a execução do programa
- **estática: `static`**
  - visível apenas no local em que foi declarada
  - existe durante toda a execução do programa
- **registrador: `register`**
  - a variável é alocada num registrador da CPU
  - deve ter tipo (unsigned) char ou (unsigned) int

Exercício

Altere o programa a seguir para gerar uma seqüência crescente.

```
#include <stdio.h>

void seq(void) {
  auto int n=0;
  printf("%d, ", n++);
}

void main(void) {
  while( !kbhit() ) {
    seq();
    delay(7000);
  }
}
```

Números pseudo-aleatórios

```
int aleat(void) {
   static unsigned s = 1234;
   auto   unsigned n = s%100;
   s += s/10;
   return n;
}

void main(void) {
   while( !kbhit() ) {
      printf("\n%d", aleat() );
      delay(9000);
   }
}
```

```
⇒ 1234 +
   123
---------
  1357 +
   135
---------
  1492 +
   149
---------
  1641 +
   164
---------
  1805 +
   ...
```

Exercício 4.15

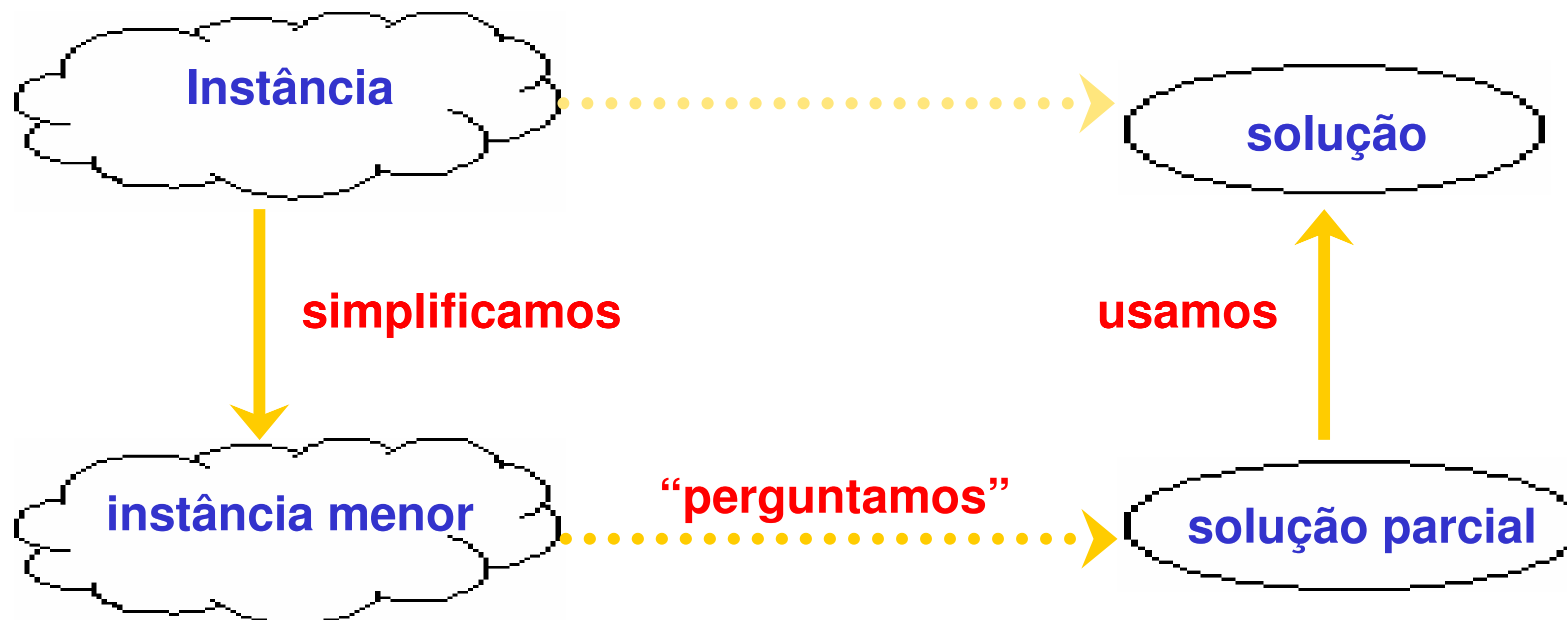
- Usando um valor de semente aleatório, tecnicamente a seqüência gerada será aleatória.
- Para gerar a semente, podemos usar a chamada `time(&t)`, onde `t` é do tipo `long`. Assim, obtemos em `t` o número de segundos que se passaram desde 01/01/1970.
- Altere a função `aleat()` para que o valor de `t` seja usado para gerar a semente.

Solução

```
#include <time.h>

int aleat(void) {
   static unsigned s = 0;
   auto   unsigned n;
   while( s==0 ) {
      long t;
      time(&t);
      s = t%10000;
   }

   n = s%100;
   s += s/10;
   return n;
}
```

Princípio de recursividade

- Permite obter a solução de um problema, a partir da solução de uma instância menor dele mesmo.
- Por hipótese, a solução da instância menor é conhecida.



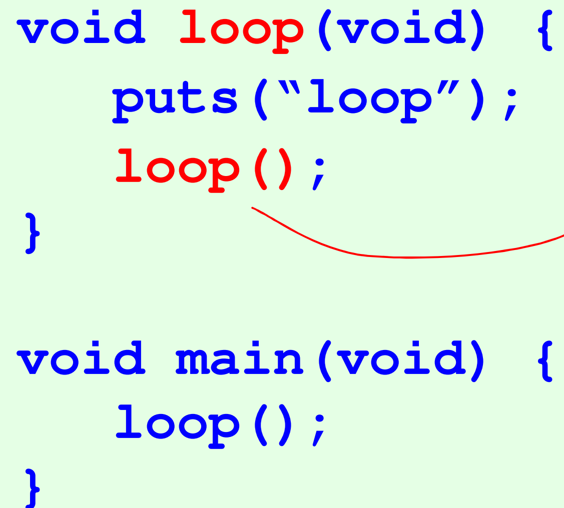
Definições recursivas

- **Uma definição recursiva é composta de:**
  - **Base: resolve diretamente o caso trivial (instância mais simples possível) do problema;**
  - **Passo: resolve recursivamente o caso geral (demais instâncias) do problema.**

Exemplo: potência com expoente natural

- **Problema:** x^n, sendo $x \in \mathbb{R}$ e $n \in \mathbb{N}$
- **Instâncias:** 2^{1000}, 10000^5, 123^{456}, ...
- **Ordenação:** $x^a \angle x^b \leftrightarrow a<b$
- **Caso trivial:** $x=0 \rightarrow x^n = 1$
- **Caso geral:** $x>0 \rightarrow x^n = ?$
  - como calcular recursivamente 2^5 ?
    - 2^4 é uma instância menor que 2^5
    - a solução dessa instância é 16 (conhecida por hipótese)
    - como $2^5 = 2 \times 2^4$, concluímos que $2^5 = 2 \times 16 = 32$

Recursividade em programação



IMPORTANTE!

Option
  Compiler
    Code generation
      Test Stack Overflow

Cálculo da potência

$$x^n = \begin{cases} 1 & se \quad n = 0 \\ x.x^{n-1} & se \quad n > 0 \end{cases}$$

INTERESSANTE!

Execute passo-a-passo (F7) e use ^F3 para ver a pilha.

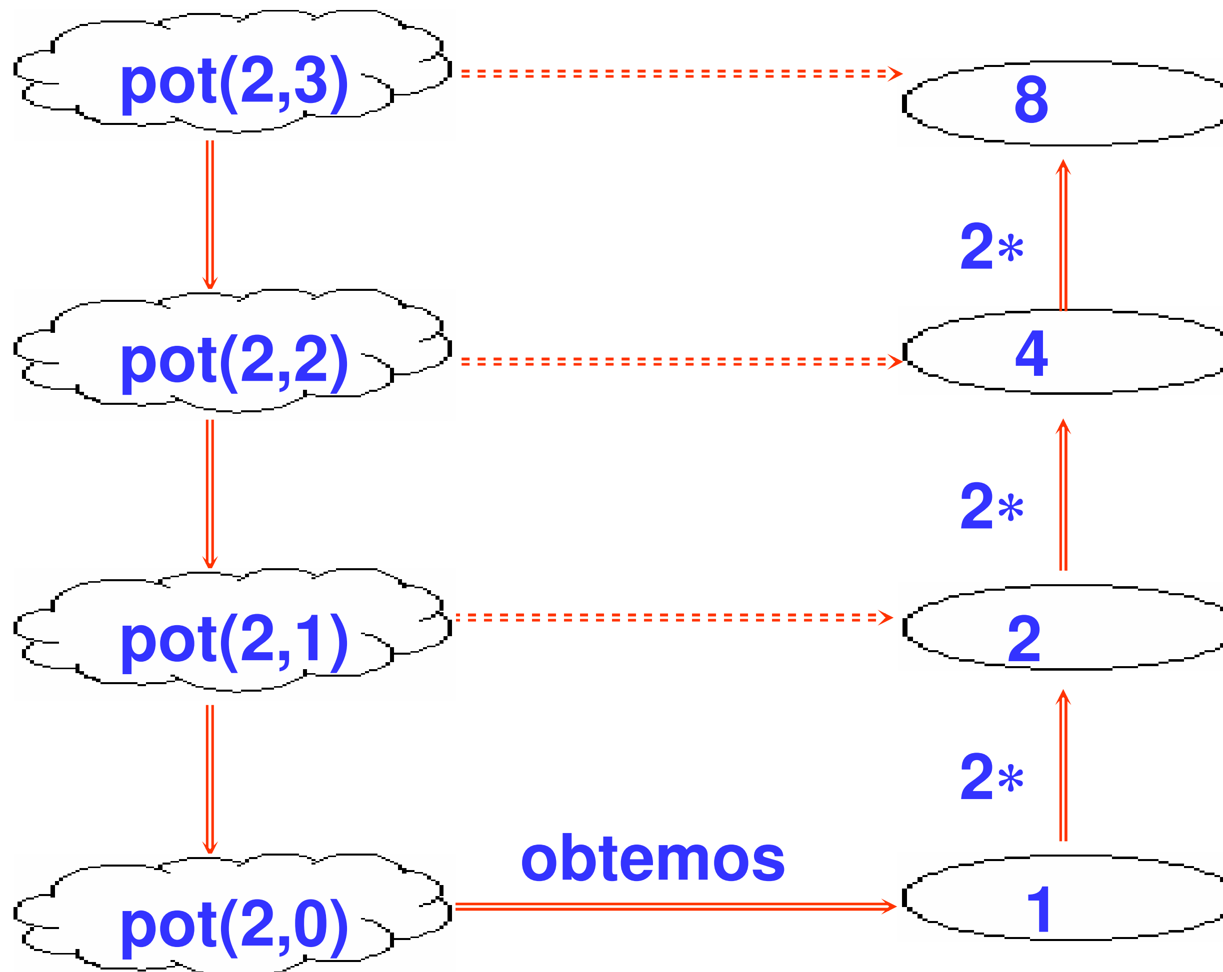
```
double pot(double x, unsigned n) {
   if( n==0 ) return 1;
   else       return x*pot(x,n-1);
}

void main(void) {
   p = pot(2,3);
   printf("%lf", p);
}
```

Simulação por substituição

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

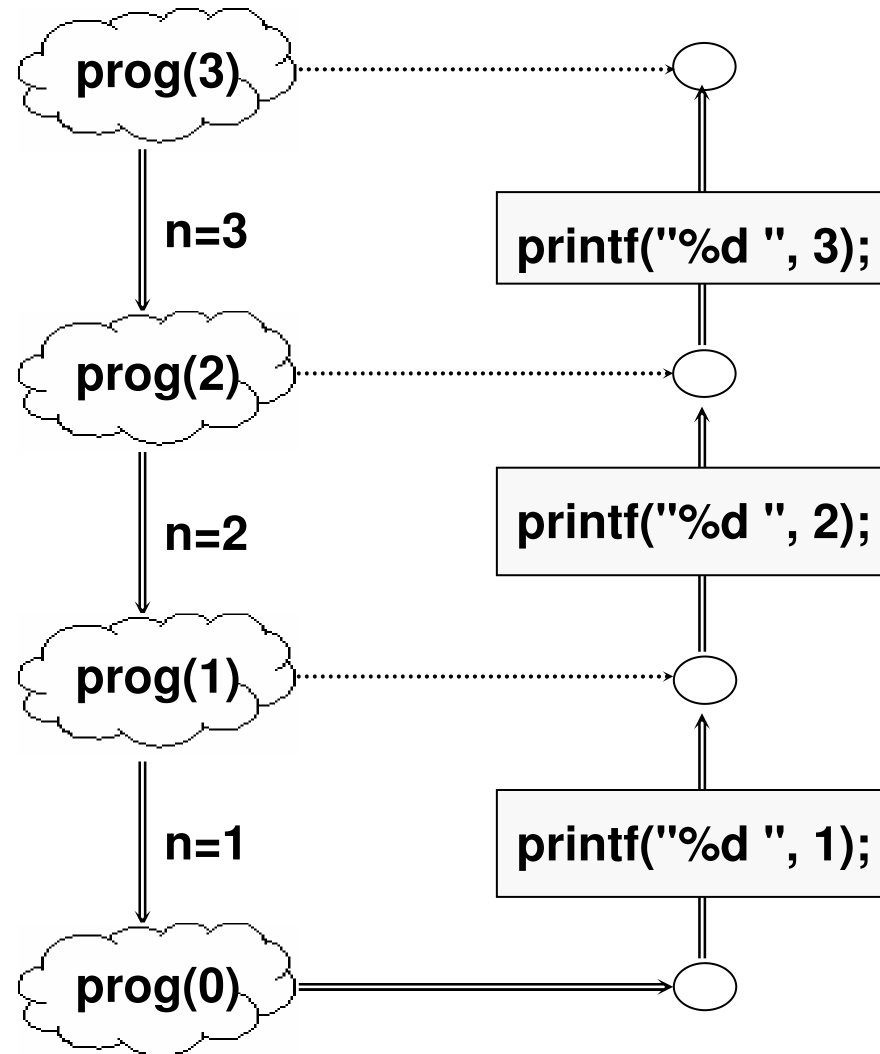
```
p = pot(2,3)
  = 2 * pot(2,2)
  = 2 * 2 * pot(2,1)
  = 2 * 2 * 2 * pot(2,0)
  = 2 * 2 * 2 * 1
  = 8
```

Simulação gráfica



Procedimentos recursivos

```c
void prog(int n) {
    if( n==0 ) return;
    prog(n-1);
    printf("%d ",n);
}

void main(void) {
    prog(3);
}
```



Exercício 4.18

a) codifique regr(n), que exibe contagem regressiva, a partir de n.

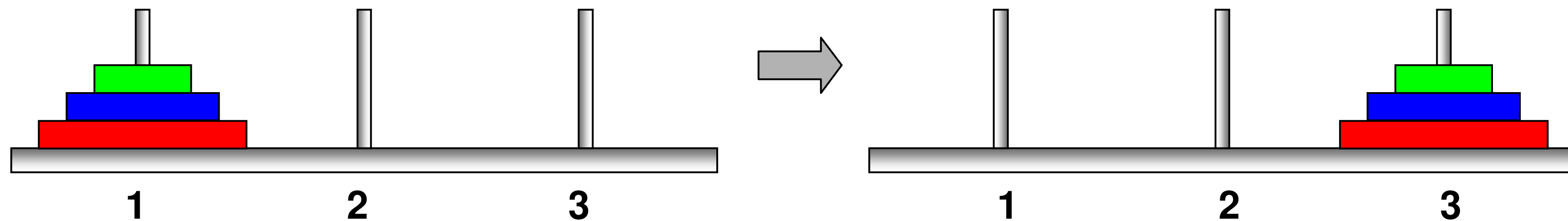
```
#include <stdio.h>
#include <conio.h>

void reg(int n) {
  if( n==0 ) return;
  printf("%d ",n);
  reg(n-1);
}

void main(void) {
  reg(3);
  getch();
}
```

Exercício 4.18

b) codifique bin(n), que exibe o natural n em binário.

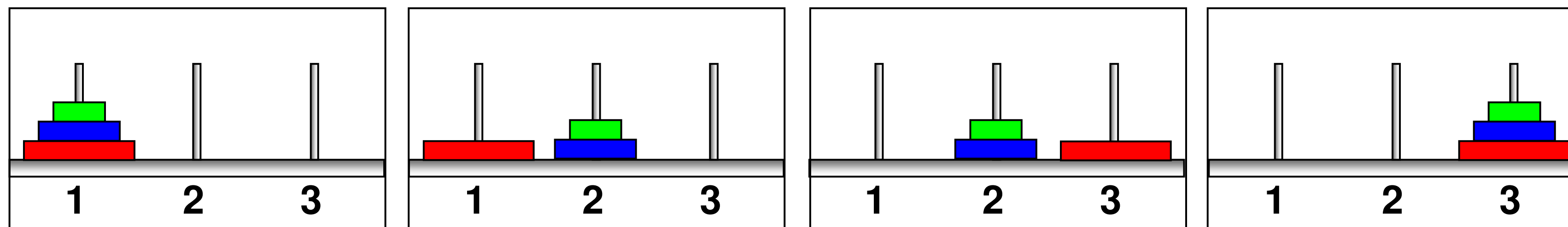
```
#include <stdio.h>
#include <conio.h>
void bin(unsigned n) {
  if( n<2 ) printf("%d",n);
  else {
    bin( n/2 );
    printf("%d", n%2);
  }
}
void main(void) {
  bin(13);
  getch();
}
```

Torres de Hanói: problema



- Restrições
  - Mover um disco de cada vez
  - Nunca colocar um disco sobre outro menor

Torres de Hanói: solução recursiva



- Se n=0 então
  - o problema está resolvido trivialmente
- Senão
  - mova recursivamente n-1 discos da torre 1 para a torre 2
  - mova diretamente o disco da torre 1 para a torre 3
  - mova recursivamente n-1 discos da torre 2 para a torre 3

Torres de Hanói: implementação em C

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

void hanoi(int n, int a, int b, int c) {
   if( n>0 ) {
      hanoi(n-1,a,c,b);
      printf("\n%d ==> %d",a,c);
      hanoi(n-1,b,a,c);
   }
}

void main(void) {
   hanoi(3,1,2,3);
   getch();
}
```

Qual o número mínimo de movimentos???

Resolva usando recursividade:

- Calcular o fatorial de n, sendo $n \in \mathbb{N}$.
- Calcular o termial de n, sendo $n \in \mathbb{N}$
- Determinar se um $n \in \mathbb{N}$ é par, sem usar resto da divisão.
- Calcular $m \div n$ $(m,n \in \mathbb{N})$, sem usar o operador de divisão.
- Calcular $m \times n$ $(m,n \in \mathbb{N})$, sem usar o operador de multiplicação.
- Calcular $H(n) = 1 + 1/2 + 1/3 + ... + 1/n$, sendo $n \in \mathbb{N}^*$.
- Calcular $m+n$ $(m,n \in \mathbb{N})$, usando apenas succ() e pred().

Fim