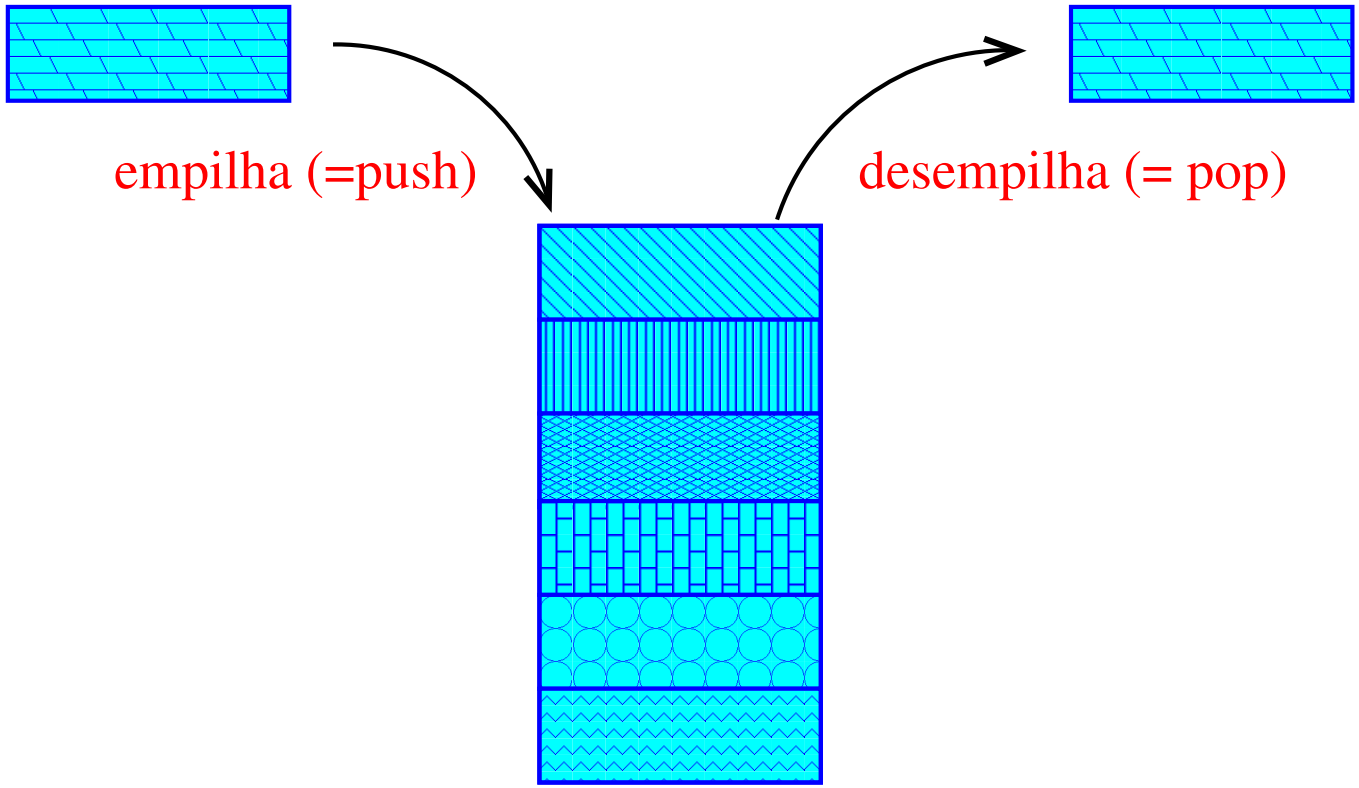
Melhores momentos

AULA 9

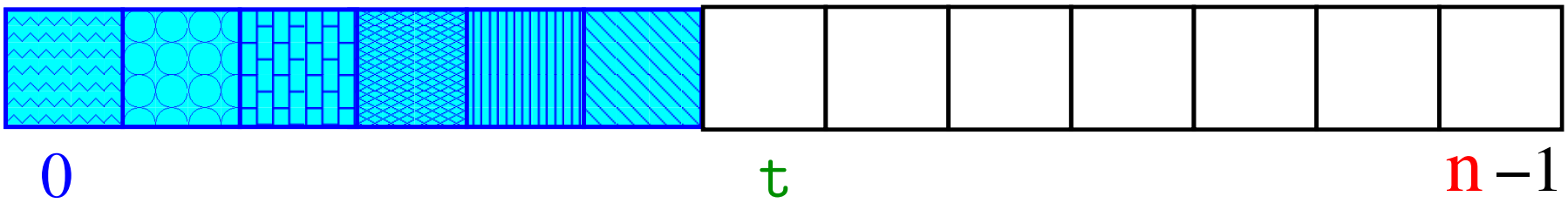
Pilhas

Uma **pilha** (=*stack*) é uma lista (=sequência) dinâmica em que todas as operações (inserções, remoções e consultas) são feitas em uma mesma extremidade chamada de **topo**.



Implementação em um vetor

A pilha será armazenada em um vetor `s[0..n-1]`.



O índice `t` indica o topo (=*top*) da pilha.

Esta é a primeira posição vaga da pilha.

A pilha está vazia se "`t == 0`".

A pilha está cheia se "`t == n`".

Interfaces

Uma **interface** (=*interface*) é uma fronteira entre a implementação de um biblioteca e o programa que usa a biblioteca.

Um **cliente** (=*client*) é um programa que chama alguma função da biblioteca.

Implementação

```
double sqrt(double x){
  [...]
return raiz;
}
       [...]
```

libm

Interface

```
double sqrt(double);
double sin(double);
double cos(double);
double pow(double,double);
       [...]
```

math.h

Cliente

```
#include <math.h>

       [...]
c = sqrt(a*a+b*b);
       [...]
```

prog.c

Interface item.h

```
/* item.h */

#ifndef HEADER_Item

#define HEADER_Item

typedef char Item;

#endif
```

Interface stack.h

```
/*
* stack.h
* INTERFACE: funcoes para manipular uma pilha
*/

#include "item.h"

void stackInit(int);
int stackEmpty();
void stackPush(Item);
Item stackPop();
Item stackTop();
void stackFree();
void stackDump();
```

Infixa para posfixa novamente

Recebe uma expressão infixa `inf` e
devolve a correspondente expressão posfixa.

```
char *infixaParaPosfixa(char *inf) {

   char *posf; /* expressao polonesa */
   int  n = strlen(inf);
   int  i;  /* percorre infixa */
   int  j;  /* percorre posfixa */
   char x;  /* item do topo da pilha */

   /*aloca area para expressao polonesa*/
   posf = mallocSafe((n+1)*sizeof(char));
   /* O '+1' eh para o '\0' */
```

cases '(' e ')'

```
stackInit(n) /* inicializa a pilha */

/* examina cada item da infixa */
for (i = j = 0; i < n; i++) {
   switch (inf[i]) {
      case '(':
         stackPush(inf[i]);
         break;

      case ')':
         while((x = stackPop()) != '(')
            posf[j++] = x;
         break;
```

cases '+', '-', '*' e '/'

```
case '+':
case '-':
   while (!stackEmpty()
      && (x = stackTop()) != '(')
      posf[j++] = stackPop();
   stackPush(inf[i]);
   break;
case '*':
case '/':
   while (!stackEmpty()
      && (x = stackTop()) != '('
      && x != '+' && x != '-')
      posf[j++] = stackPop();
   stackPush(inf[i]);
   break;
```

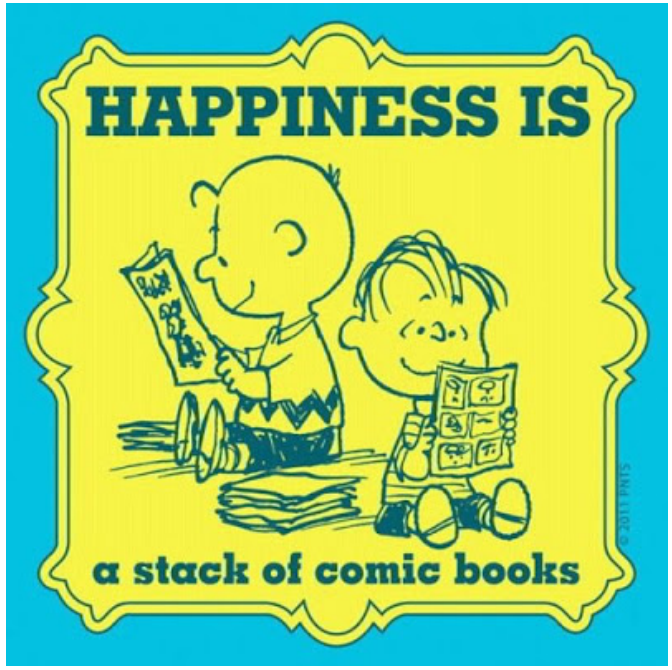
default e finalizações

```
        default:
            if(inf[i] != ' ')
                posf[j++] = inf[i];
        } /* fim switch */
    } /* fim for (i=j=0...)  */

    /* desempilha todos os operandos que restaram */
    while (!stackEmpty())
        posf[j++] = stackPop()
    posf[j] = '\0'; /* fim expr polonesa */
    stackFree();
    return posf;
} /* fim funcao */
```

AULA 10

Pilha implementada em um vetor



Fonte: http://powsley.blogspot.com.br/

PF 6.1, S 4.4

http://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html

Implementação stack.c

```
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"

/*
 * PILHA: implementacao em vetor
 */
static Item *s; /* pilha */
static int t;
/* t eh o indice do topo da pilha,
 * s[t] eh a 1a. posicao vaga da pilha
 */
```

Implementação stack.c

```
void stackInit(int n) {
   s = mallocSafe(n*sizeof(Item));
   t = 0;
}

int stackEmpty() {
   return t == 0;
}
```

Implementação stack.c

```
void stackPush(Item item) {
   s[t++] = item;
}

Item stackPop() {
   return s[--t];
}
```

Implementação stack.c

```
Item stackTop() {
   return s[t-1];
}

void stackFree() {
   free(s);
}
```

Implementação stack.c

```
void stackDump() {
   int k;

   fprintf(stdout,"pilha :  ");
   if (t == 0) fprintf(stdout, "vazia.");
   for (k = t-1; k >= 0; k--)
      fprintf(stdout, "%c ", s[k]);
   fprintf(stdout,"\n");
}
```

Compilação

cria o obj stack.o

```
> gcc -Wall -O2 -ansi -pedantic -Wno-unused-result -c stack.c
```

cria o obj polonesa.o

```
> gcc -Wall -O2 -ansi -pedantic -Wno-unused-result -c polonesa.c
```

cria o executável polonesa

```
> gcc stack.o polonesa.o -o polonesa
```

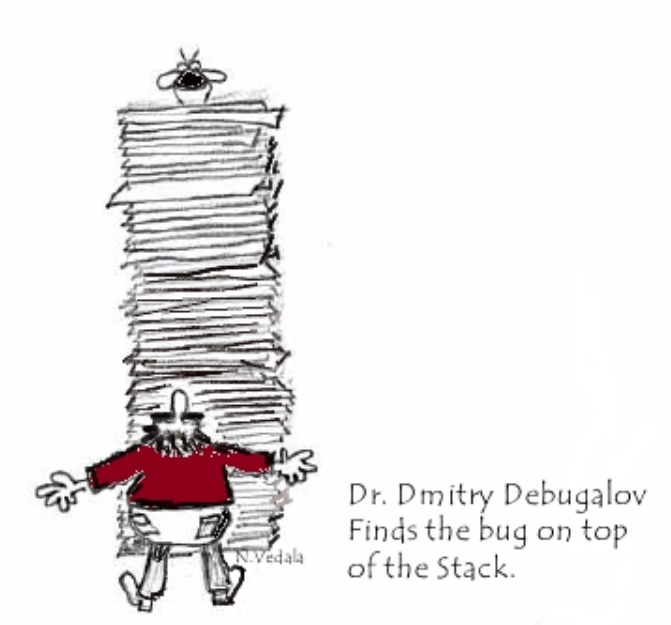
Makefile

```
polonesa:  polonesa.o stack.o
	gcc polonesa.o stack.o -o polonesa

polonesa.o:  polonesa.c stack.h item.h
	gcc -Wall -O2 -ansi -pedantic \
	-Wno-unused-result -c polonesa.c

stack.o:  stack.c stack.h item.h
	gcc -Wall -O2 -ansi -pedantic \
	-Wno-unused-result -c stack.c
```

Pilha implementada em lista encadeada



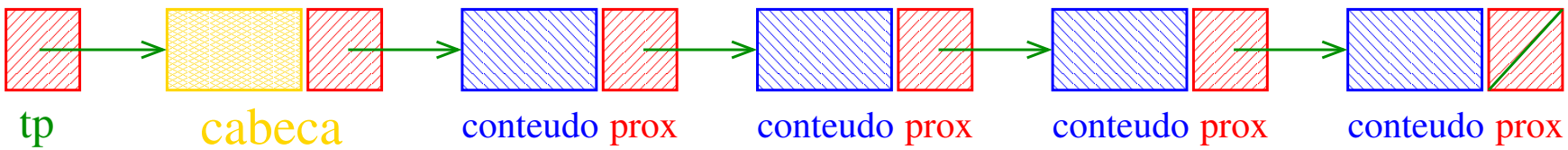
Fonte: http://www.dumpanalysis.org/

PF 6.3, S 4.4

http://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html

Pilha implementada em uma lista encadeada

A pilha será armazenada em uma **lista encadeada** com **cabeça**.



O ponteiro `tp` aponta para a `cabeca` da lista.

`tp->prox->conteudo` é o elemento do `topo` da pilha.

A pilha está vazia se "`tp->prox == NULL`".

A pilha está cheia se ... acabou a memória disponível.

Interface stack.h

```
/*
* stack.h
* INTERFACE: funcoes para manipular uma pilha
*/

#include "item.h"

void stackInit(int);
int stackEmpty();
void stackPush(Item);
Item stackPop();
Item stackTop();
void stackFree();
void stackDump();
```

Implementação stack.c

```
#include <stdlib.h>
#include <stdio.h>
#include "stack.h"
/*
 * PILHA: implementacao em lista encadeada
 */
typedef struct stackNode* Link;
struct stackNode{
   Item conteudo;
   Link prox;
};
static Link tp;
```

Implementação stack.c

```
void stackInit(int n) {
   tp = mallocSafe(sizeof *tp);
   tp->prox = NULL;
}

int stackEmpty() {
   return tp->prox == NULL;
}
```

Implementação stack.c

```
void stackPush(Item item) {
   Link nova = mallocSafe(sizeof *nova);

   nova->conteudo = item;
   nova->prox = tp->prox;
   tp->prox = nova;
}
```

Implementação stack.c

```
Item stackPop() {
   Link p = tp->prox;
   Item conteudo = p->conteudo;

   tp->prox = p->prox;
   free(p);
   return conteudo;
}
```

Implementação stack.c

```
Item stackTop() {
   return tp->prox->conteudo;
}
```

Implementação stack.c

```
void stackFree() {
   while (tp != NULL) {
      Link p = tp;
      tp = p->prox;
      free(p);
   }
}
```

Implementação stack.c

```
void stackDump() {
   Link p = tp->prox;

   fprintf(stdout,"pilha :  ");
   if (p == NULL) fprintf(stdout,"vazia.");
   while (p != NULL) {
      fprintf(stdout, "%c ", p->conteudo);
      p = p->prox;
   }
   fprintf(stdout,"\n");
}
```

Compilação

cria o obj stack.o

```
> gcc -Wall -O2 -ansi -pedantic -Wno-unused-result -c stack.c
```

cria o obj polonesa.o

```
> gcc -Wall -O2 -ansi -pedantic -Wno-unused-result -c polonesa.c
```

cria o executável polonesa

```
> gcc -o polonesa stack.o polonesa.o
```

Makefile

```
polonesa:  polonesa.o stack.o
      gcc polonesa.o stack.o -o polonesa

polonesa.o:  polonesa.c stack.h item.h
      gcc -Wall -O2 -ansi -pedantic \
      -Wno-unused-result -c polonesa.c

stack.o:  stack.c stack.h item.h
      gcc -Wall -O2 -ansi -pedantic \
      -Wno-unused-result -c stack.c
```

Pilha de execução

Fonte: `http://meta.stackexchange.com/`

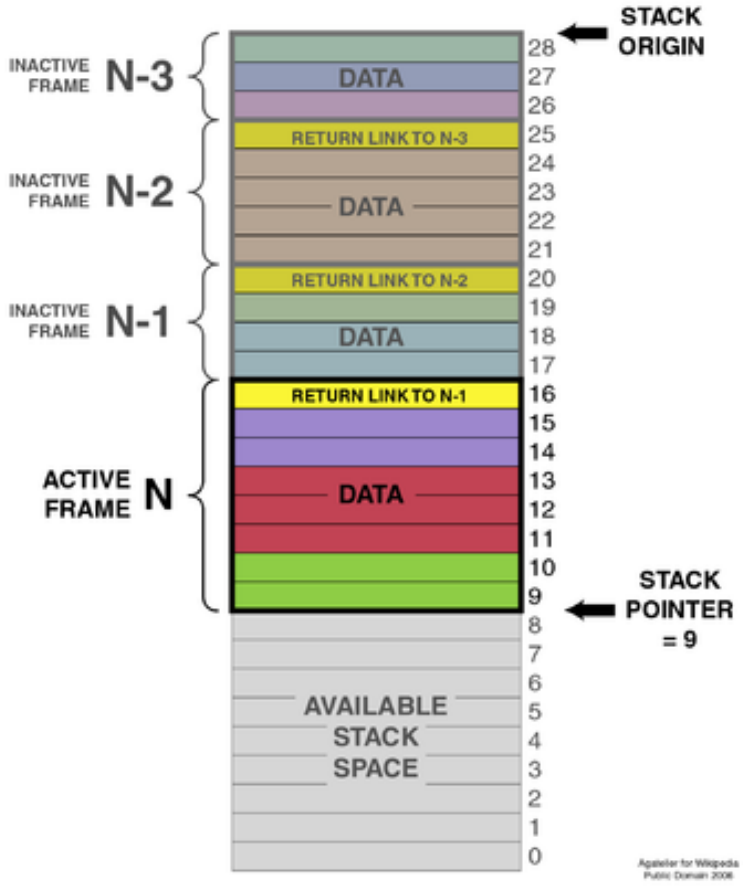
PF 6.5

http://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html

http://en.wikipedia.org/wiki/Call_stack

Pilha de execução

Para executar um programa, o computador utiliza uma **pilha de execução** (=*execution stack*=*call stack*).



Fonte: http://en.wikipedia.org/wiki/File:ProgramCallStack2.png

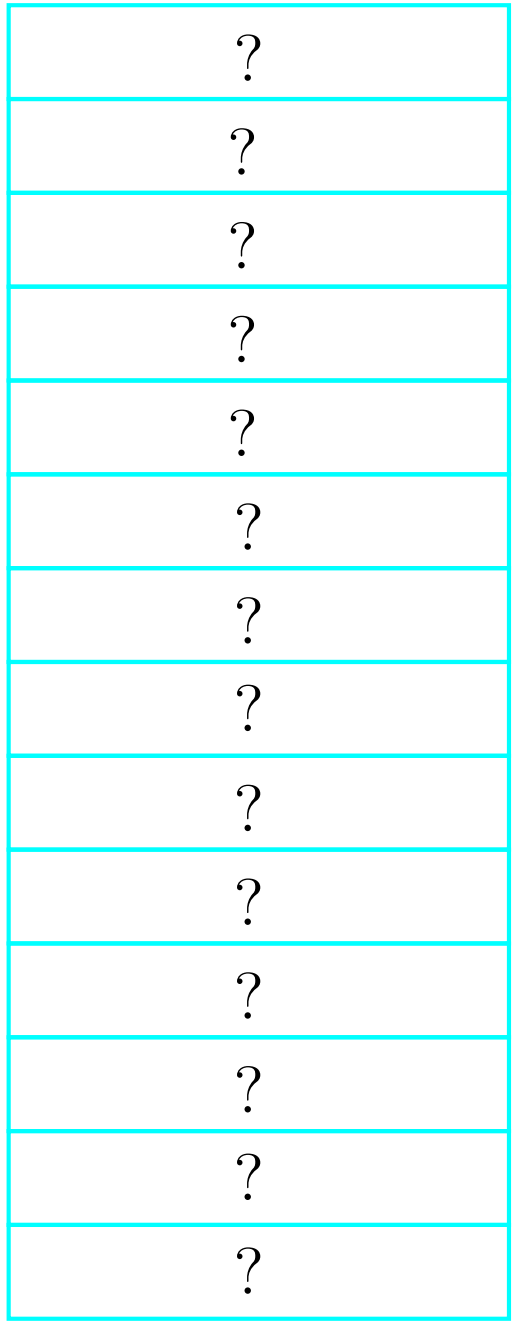
Pilha de execução

Uma pilha de execução é usada para armazenar:

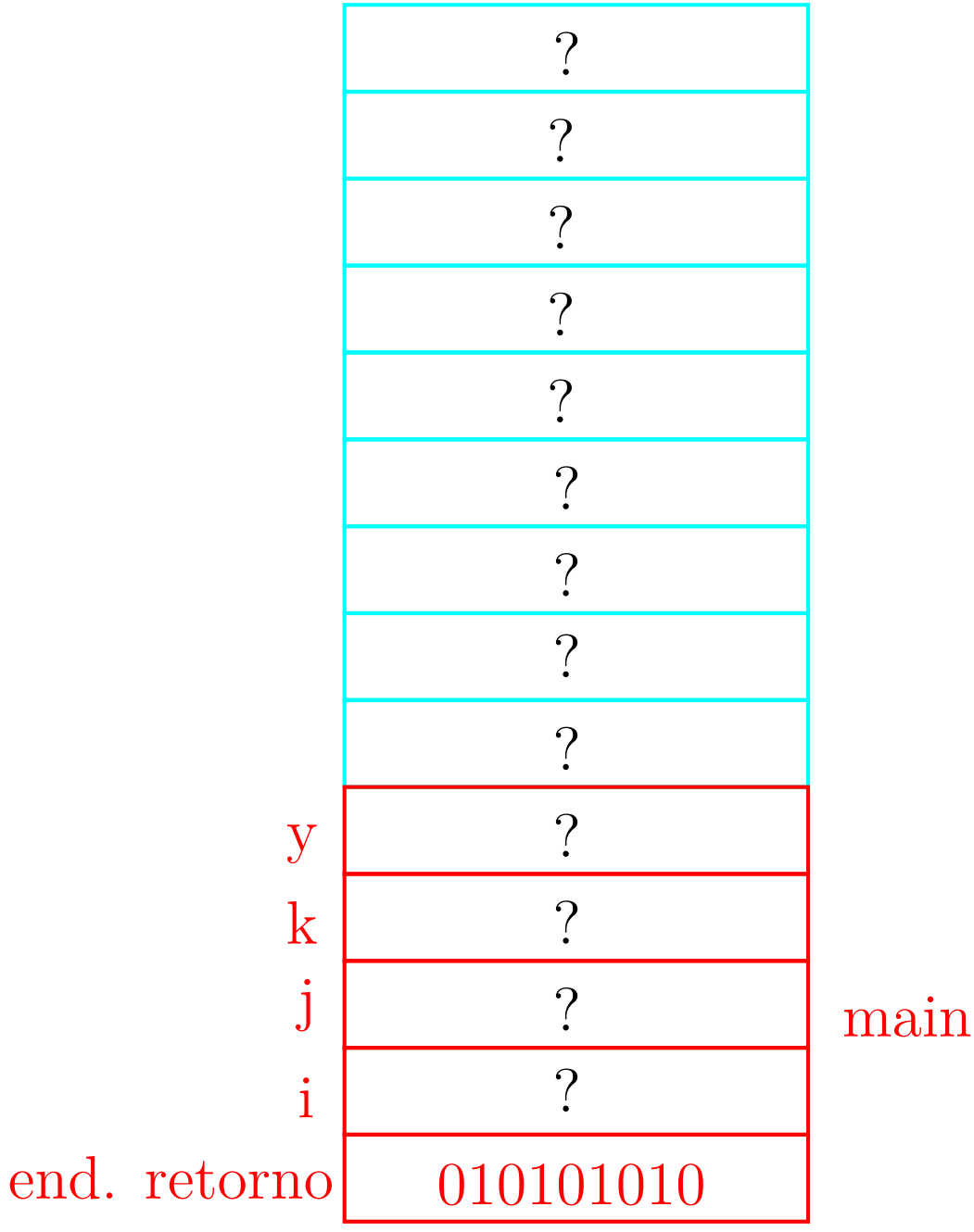
- variáveis locais das funções "ativas";
- parâmetros das funções "ativas";
- endereço de retorno para o ponto em que as funções foram chamadas;
- cálculo de expressões;

Exemplo

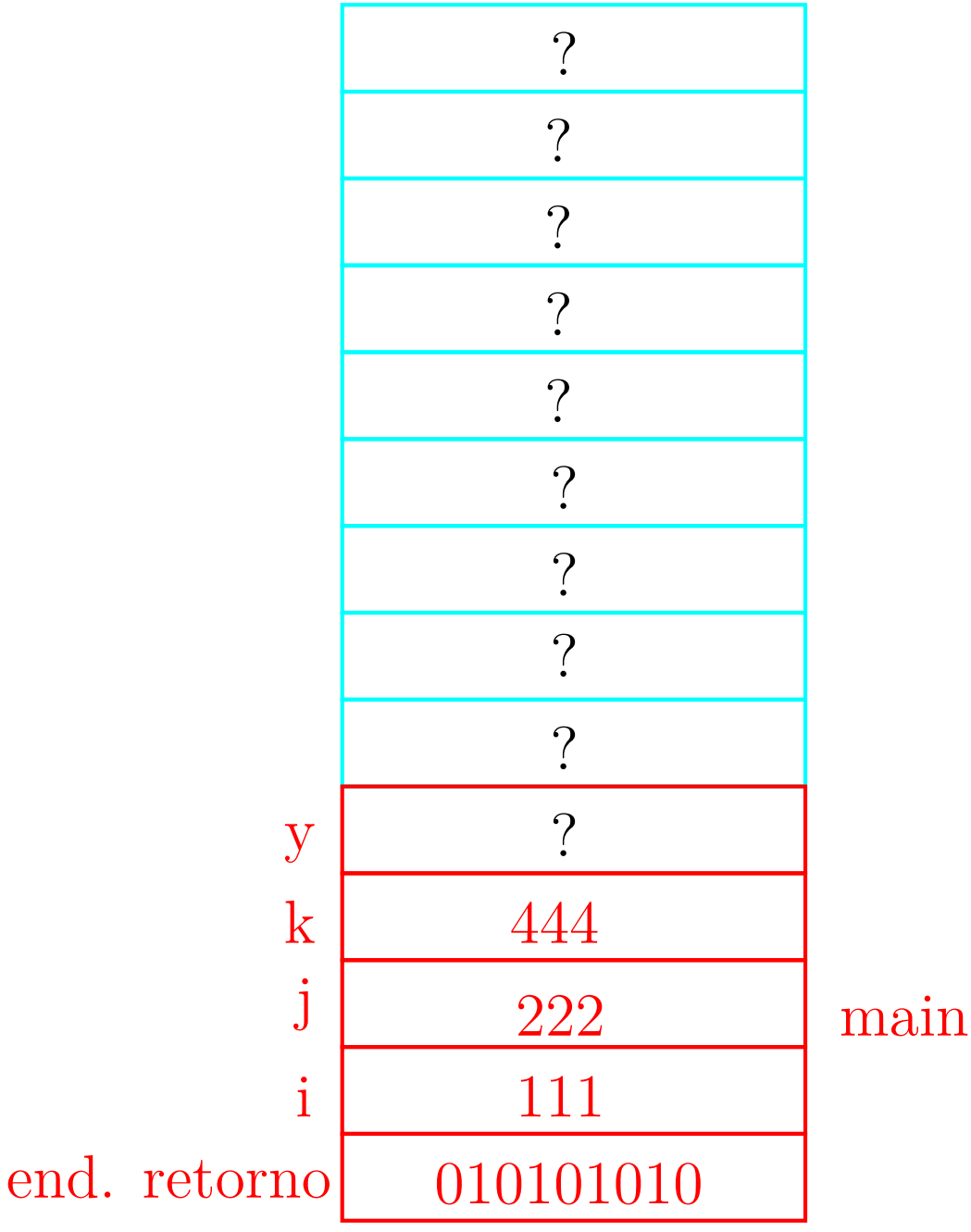
```
int main (void) {
   int i, j, k, y;
   i = 111; j = 222; k = 444;
   y = /*1*/ F (i, j, k) /*4*/;
   printf ("%d\n", y);
   return EXIT_SUCCESS;
}
int G (int a, int b) {
   int x;
   x = a + b;
   return x;
}
int F (int i, int j, int k) {
   int x;
   x = /*2*/ G (i, j) /*3*/;
   x = x + k;
   return x;
}
```



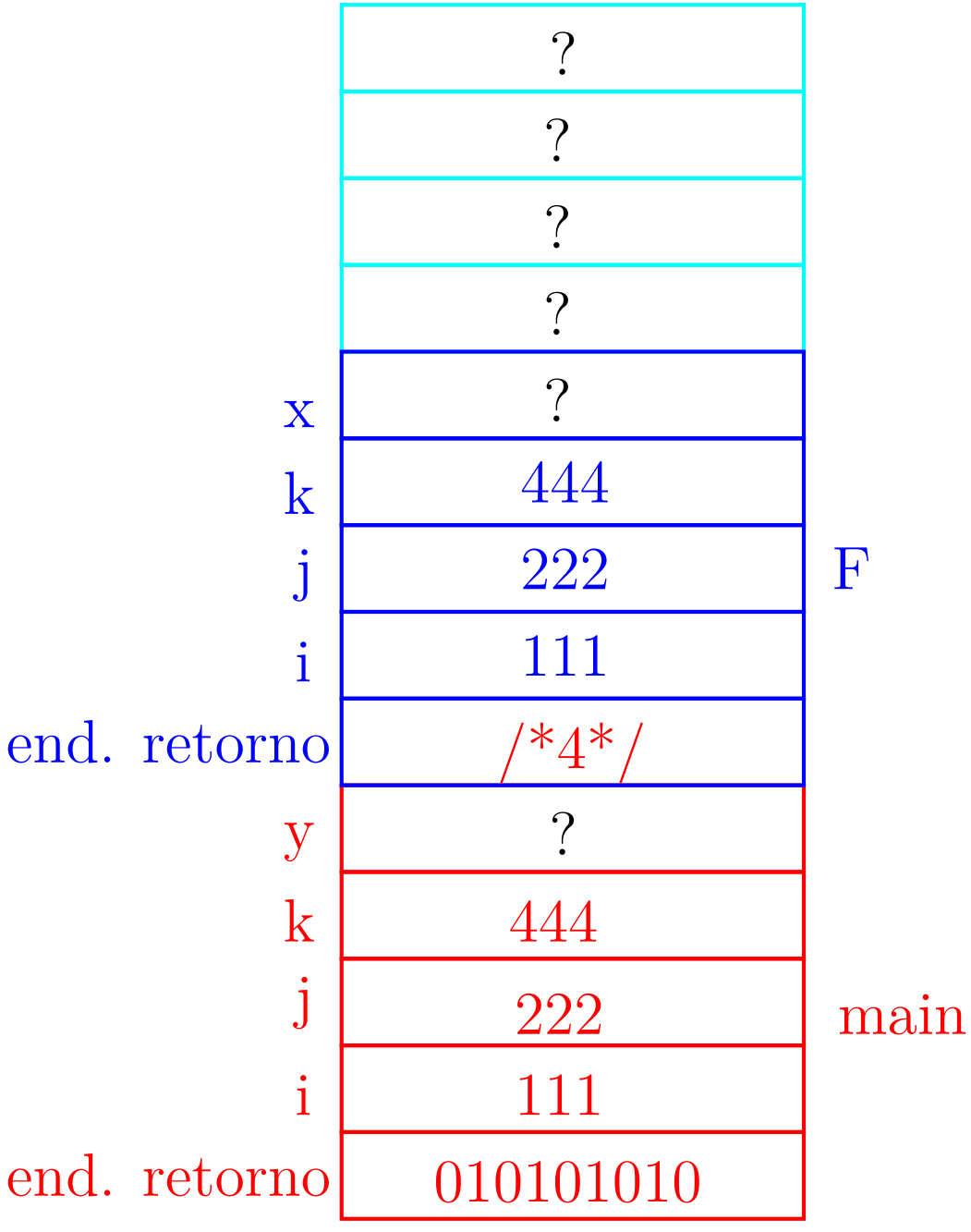
Saida



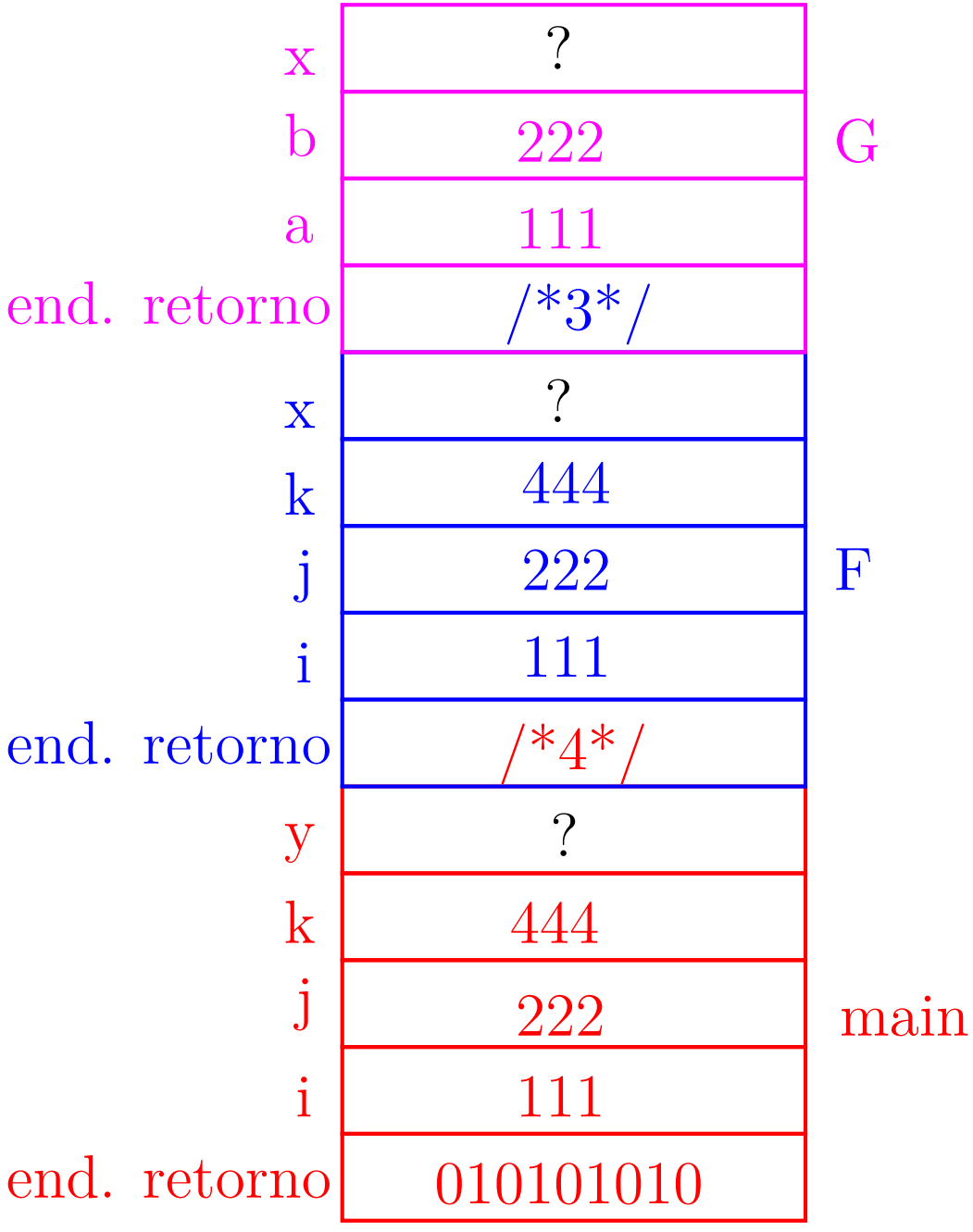
Saida



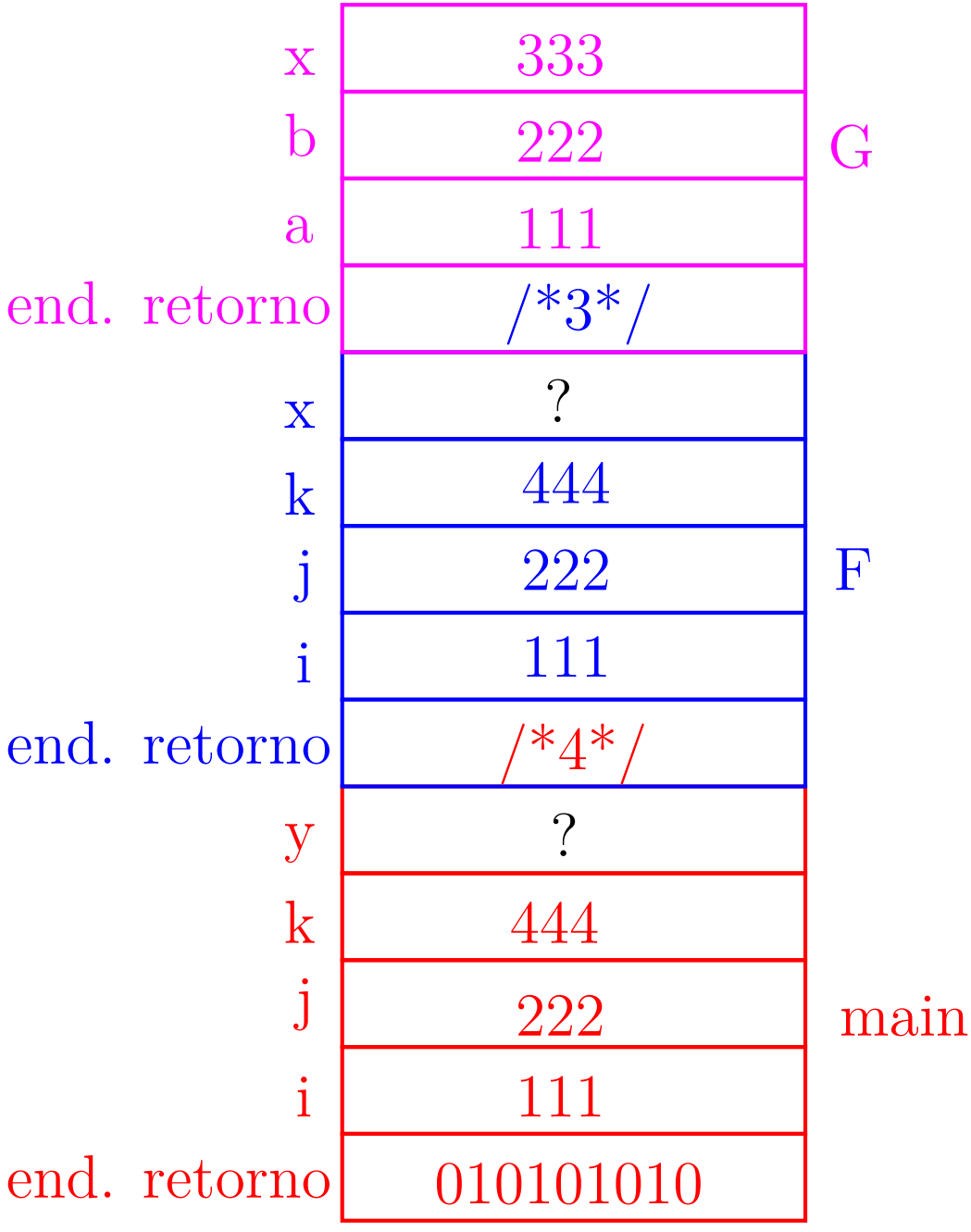
Saida


?
?
?
?
x
?
k
444
j
222
F
i
111
end. retorno
/*4*/
y
?
k
444
j
222
main
i
111
end. retorno
010101010

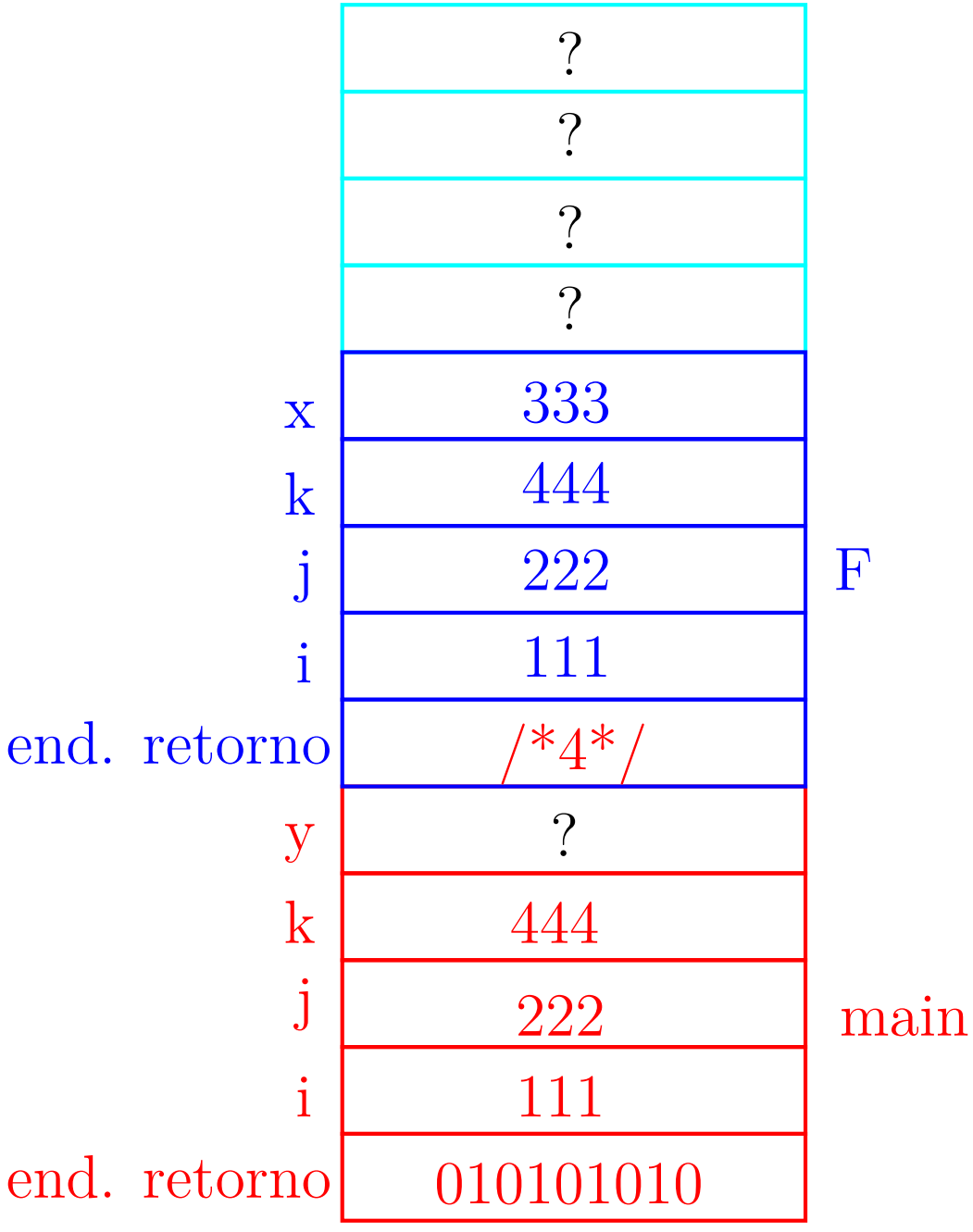
Saida


x
?
b
222
G
a
111
end. retorno
/*3*/
x
?
k
444
j
222
F
i
111
end. retorno
/*4*/
y
?
k
444
j
222
main
i
111
end. retorno
010101010

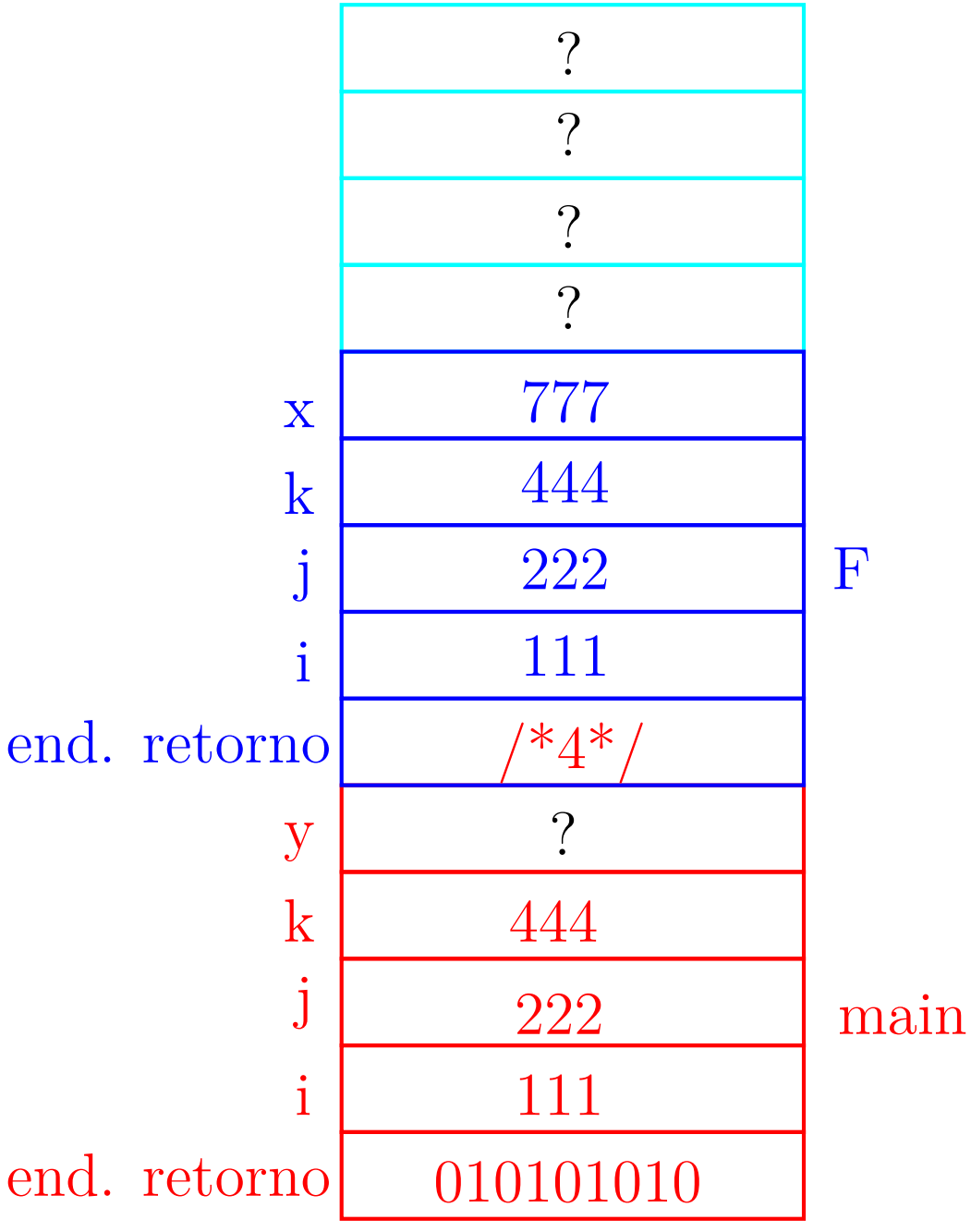
Saida


x
333
b
222
G
a
111
end. retorno
/*3*/
x
?
k
444
j
222
F
i
111
end. retorno
/*4*/
y
?
k
444
j
222
main
i
111
end. retorno
010101010

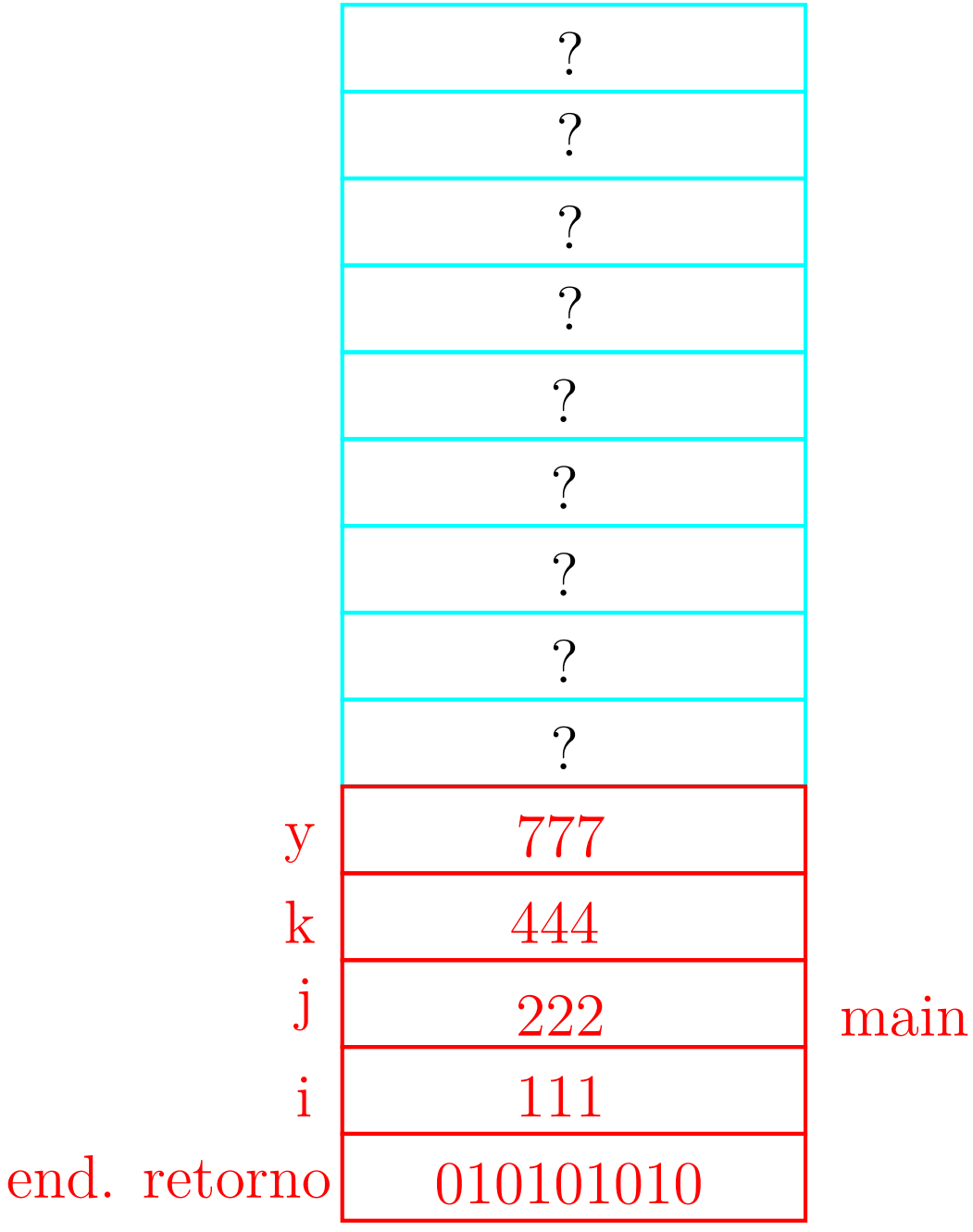
Saida


?
?
?
?
x
333
k
444
j
222
F
i
111
end. retorno
/*4*/
y
?
k
444
j
222
main
i
111
end. retorno
010101010

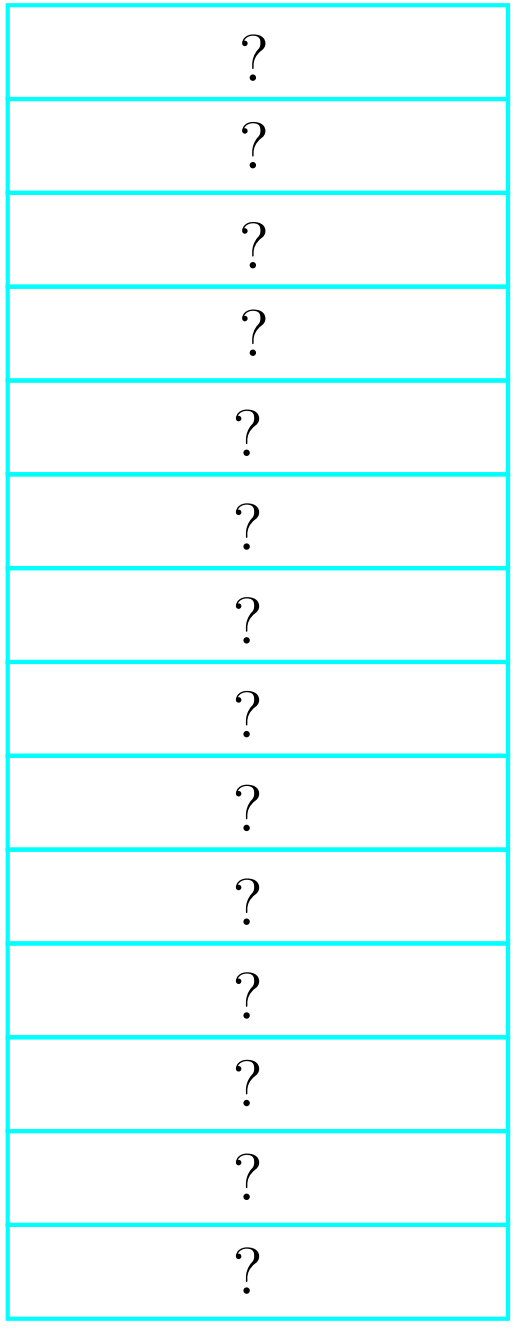
Saida

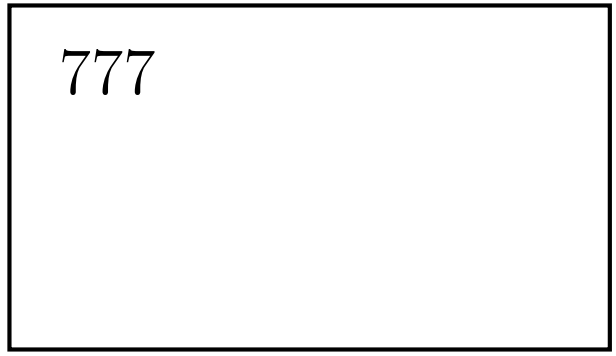

?
?
?
?
x
777
k
444
j
222
F
i
111
end. retorno
/*4*/
y
?
k
444
j
222
main
i
111
end. retorno
010101010

Saida



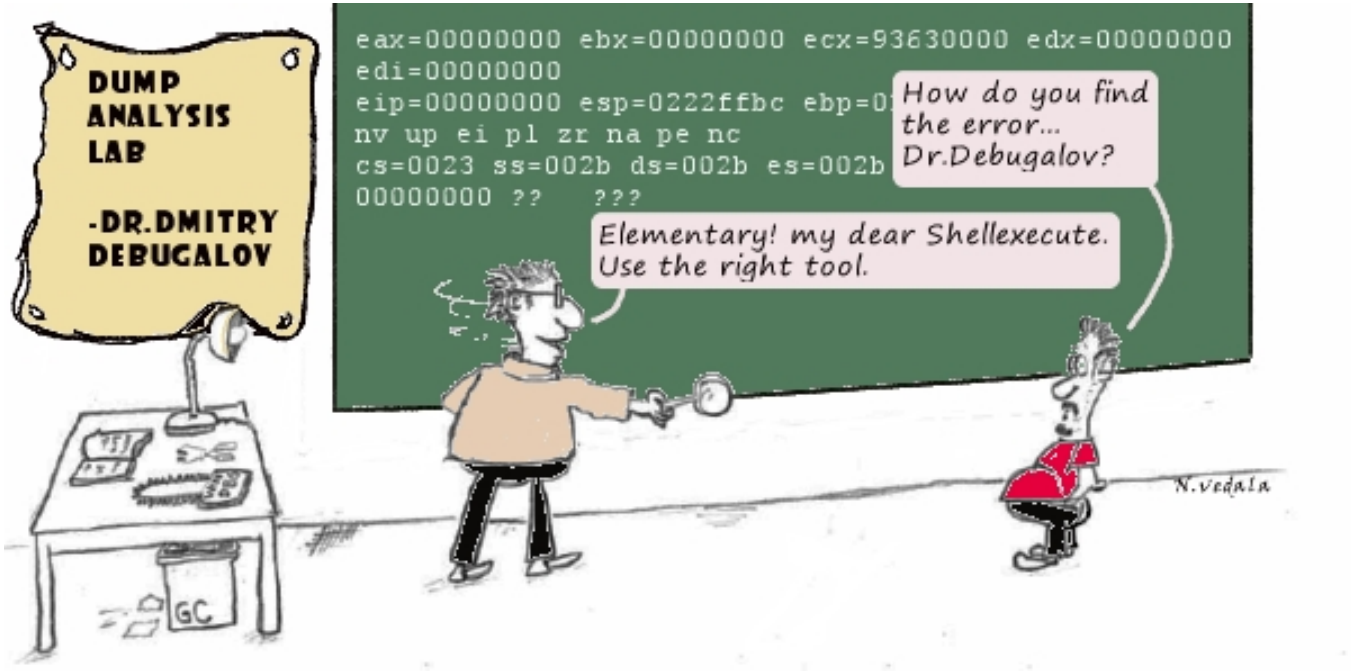
Saida





Saida

Leiaute da memória de um programa em C



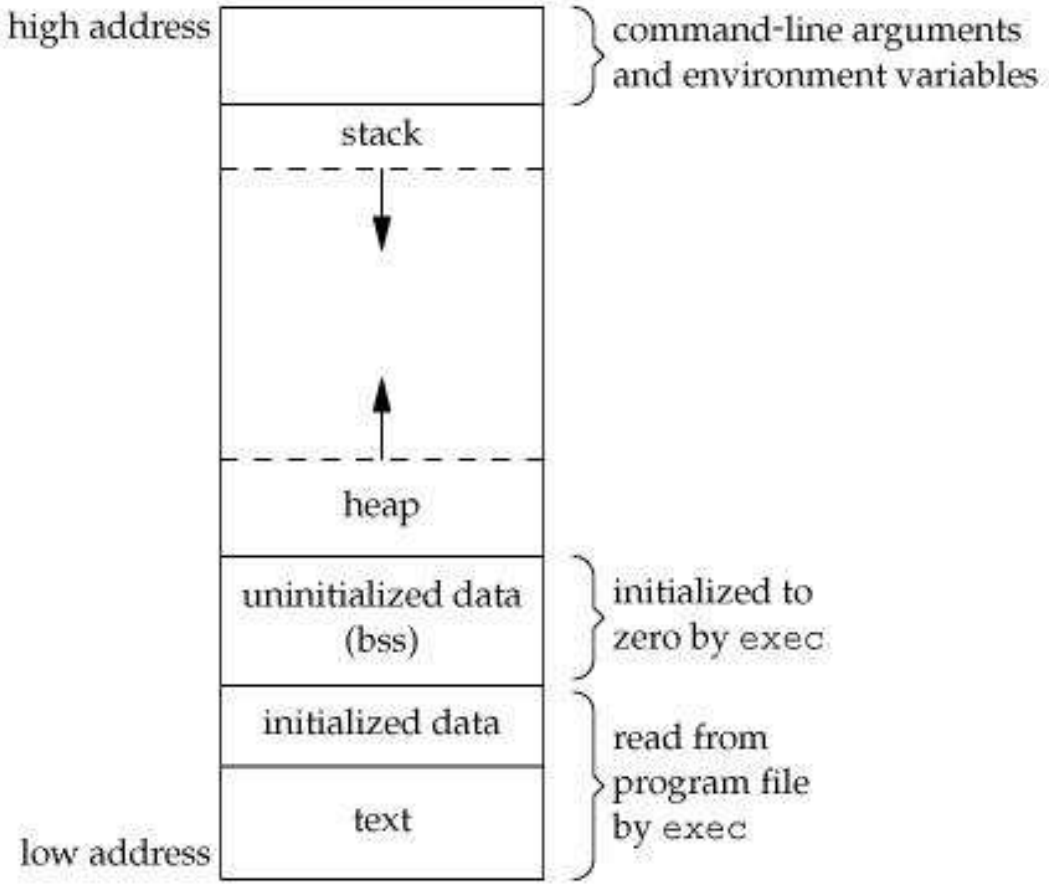
Fonte: http://meta.stackexchange.com/

http://www.geeksforgeeks.org/memory-layout-of-c-program/

http://cs-fundamentals.com/c-programming/

Leiaute da memória

Representação típica da memória utilizada por um programa.



Fonte: http://cs-fundamentals.com/c-programming/

Leiaute da memória

A memória utilizada por um programa divide-se em partes chamadas de segmentos (**memory segments**):

- *stack*: variáveis locais, parâmetros das funções, endereços de retorno;
- *heap*: para alocação de memória dinamicamente, administrada por `malloc()`, `free()` e seus amigos;
- BSS: variáveis estáticas (*static*) e globais não inicializadas (`static int i;`);
- *data*: variáveis estáticas (*static*) inicializadas (`static int i = 0;`);
- *text*: código do programa.

Size

```
meu_prompr> size polonesa
text    data    bss     dec     hex     filename
3392    316     56      3764    eb4     polonesa
```