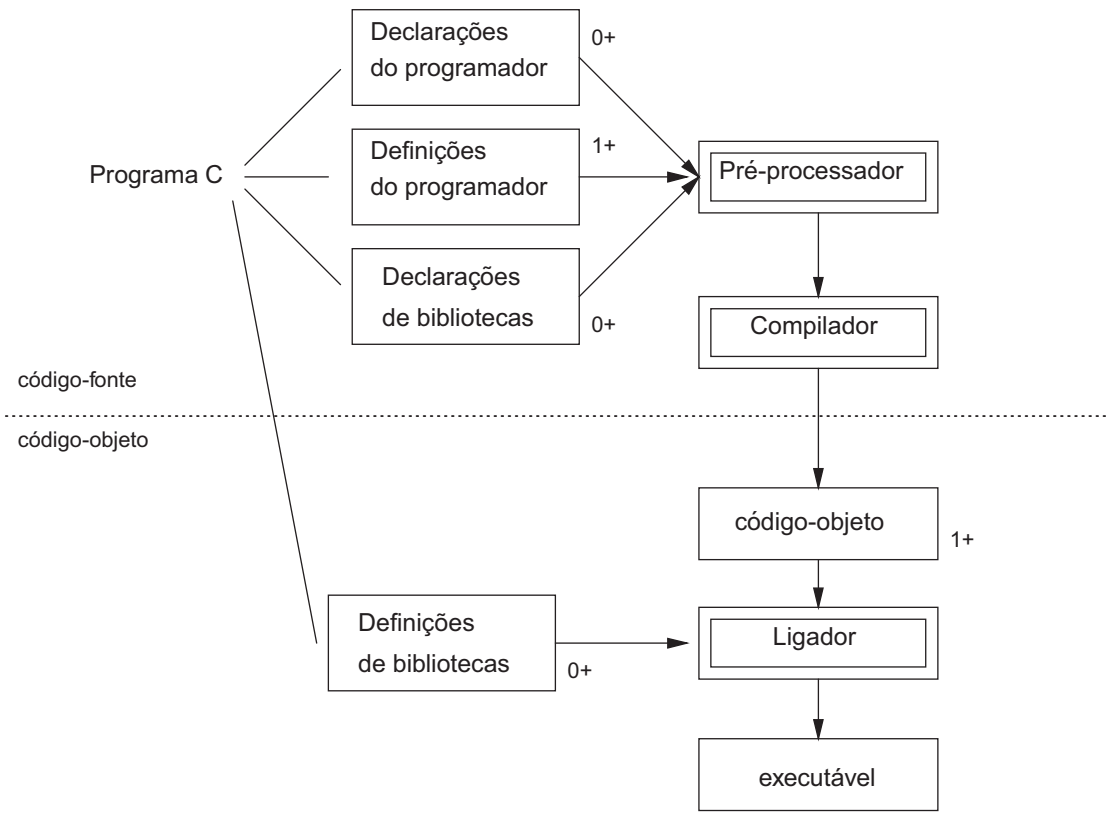
# Linguagem C

## *Pré-processador*

g.p. telles



# Compilação



# Diretivas do pré-processador

- Uma inha que começa com # é uma diretiva de pré-processador (pode ser precedida por espaços).
- Exemplos:

```
#include
#define
```

- A sintaxe é independente da linguagem. O pre-processador não sabe C.
- O efeito de uma diretiva começa onde ela aparece em um arquivo e continua até o fim do arquivo ou até que a diretiva seja desativada.



# #include

- `#include <arquivo>`
  `#include "arquivo"`
- O pré-processador substitui a linha por uma cópia do conteúdo do arquivo.
- O arquivo é procurado em diretórios que dependem do sistema.
- A forma com aspas inclui o diretório atual na busca.
- Não há restrição ao conteúdo do arquivo. Pode conter inclusive outras diretivas, que serão expandidas também.

# #define

- `#define identificador tokens`

- Uma definição feita com `#define` é chamada de constante simbólica ou macro.
- O pré-processador substitui cada ocorrência do `identificador` por `tokens`, exceto em constantes string.
- `Tokens` vai até o fim da linha e pode ser nulo.
- O caracter \ permite continuar a definição da macro nas linhas seguintes:

```
#define macro texto_1 \
              texto_2 \
               ...
```



# Exemplos

```
#include <stdio.h>

#define FIRST_CHAR 'a'
#define SECOND_CHAR 'b'
#define COMMA ','
#define STRING "um exemplo de macro."

void main() {
  putchar(FIRST_CHAR);
  putchar(SECOND_CHAR);
  putchar(COMMA);
  puts(STRING);
}
```

# #undef

- Faz com que a definição da macro seja desativada daquele ponto em diante.

```
#define pi 3.14
...
#undef pi
```



# Constantes simbólicas

- Constantes simbólicas melhoram a clareza, legibilidade e facilitam a manutenção.

```
#define MAX 100

int V[MAX], i;

for (i=1; i<MAX; i++)
  V[i] = 1;
```

# Constantes simbólicas

- Podem aumentar a velocidade de execução:

```
#define TAXA 0.3
...
imposto = preco * TAXA;
```

é mais eficiente que

```
float taxa = 0.3;
...
imposto = preco * taxa;
```

pois o não é preciso ler taxa da memória.



# Macros com parâmetros

- `#define id(id,...,id) tokens`
- Exemplo: `#define quad(x) ((x)*(x))`

`quad(7+z)` $\Longrightarrow$ `((7+z)*(7+z))`

`quad(quad(2))` $\Longrightarrow$ `((((2)*(2)))*(((2)*(2))))`

# Macros com parâmetros

- Macros são usadas no lugar de funções para evitar a chamada da função e gerar código mais eficiente.

```
#define min(x,y) (((x<y))?(x):(y))
```

- Uma definição de macro pode usar tanto macros quanto funções em seu corpo.

```
#define min(a,b,c,d) min(min(a,b),min(c,d))
```



# Macros com parâmetros

- Problema potencial:

```
#define quad(x) ((x)*(x))

int c=2;
quad(c++);
```

- Exemplo: `exm-macros-1.c`

# Syntatic sugar

- Macros podem ser usadas para alterar a sintaxe da linguagem, definindo comandos e operadores:

```
#define eq ==
#define elsif else if

if (a eq 0) {}
elsif (a eq 1) {}
else {}
```

- Esta prática não é recomendável, porque pode dificultar a manutenção.



# Compilação condicional

- Diretivas que fazem com que o pré-processador não envie trechos do programa para o compilador.

```
#if expressão-integral-constante
#elif expressão-integral-constante
#else
#endif

#ifdef identificador
#endif

#ifndef identificador
#endif

defined(constante)
```

# Compilação condicional

- Útil para
  - Setar variáveis que dependem da plataforma.
  - Ativar/desativar código de depuração.
  - Comentar trechos de código já comentados.
- Exemplo:

```
#ifdef UNIX
const int nice = 10;
#elif defined(MSDOS)
const int nice = 0;
#else
const int nice = 5;
#endif
```