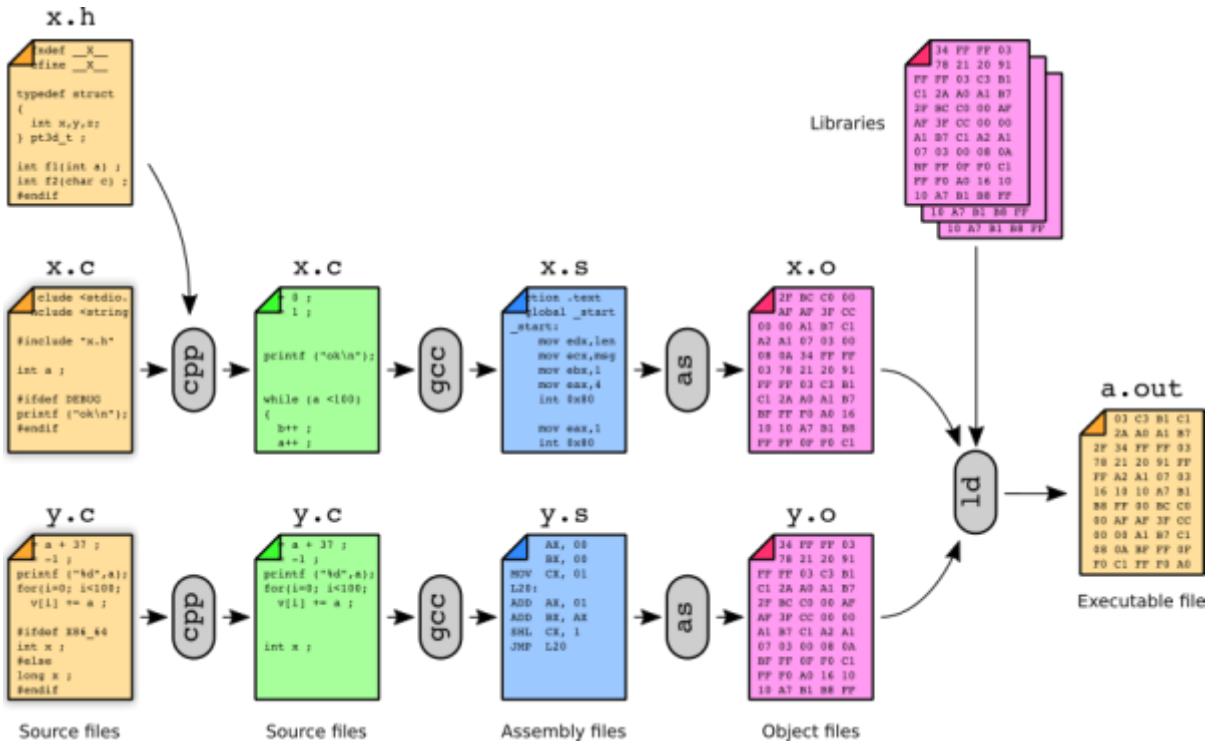
# O preprocessador C

Video desta aula

A transformação de um programa C em um arquivo executável é um processo complexo, com várias etapas, sendo as mais importantes:

- **preprocessamento**: tratamento das diretivas do preprocessador (`#include`, etc)
- **compilação**: conversão de C para *assembly*
- **tradução**: conversão de *assembly* para código de máquina (binário)
- **ligação**: junção dos arquivos-objeto e bibliotecas em código de máquina para formar o arquivo executável

Ele está ilustrado na figura a seguir, onde `cpp` corresponde ao preprocessamento, `gcc` à compilação, `as` à tradução e `ld` à ligação.




## Preprocessamento

O preprocessador C (CPP - *C PreProcessor*) é uma ferramenta de **substituição de texto** invocada automaticamente pelo compilador C/C++ no início do processo de compilação. Apesar de não fazer parte formal da sintaxe da linguagem C, seu uso é praticamente indispensável para a estruturação de programas C, mesmo os mais simples.

Todos os comandos do preprocessador começam com o símbolo # no início de uma linha (podem haver espaços e tabs antes).

A saída do preprocessador C, que será enviada ao compilador propriamente dito, pode ser obtida através do flags `-E`:

```
gcc -E arquivo.c
```

# Inclusão de arquivos

O preprocessador é frequentemente usado para incluir arquivos externos em um código-fonte.

Considere este exemplo:

escreva.h

```
#ifndef __ESCREVA__
#define __ESCREVA__

void escreva (char *msg) ;

#endif
```

escreva.c

```
#include <stdio.h>

void escreva (char *msg)
{
  printf ("%s", msg) ;
}
```

hello.c

```
#include "escreva.h"

int main ()
{
  escreva ("Hello, world!\n") ;
  return (0) ;
}
```

Neste exemplo, o preprocessador irá **substituir** cada linha `#include …` pelo conteúdo do respectivo arquivo, gerando um único arquivo temporário que será entregue ao compilador C.

Há uma diferença importante entre as duas formas de inclusão:

- `#include <…>`: o arquivo indicado será buscado nos diretórios default do compilador, geralmente `/usr/include/*` nos sistemas Unix.
- `#include "…"`: o arquivo indicado será buscado primeiro no diretório corrente (onde está o arquivo que está sendo compilado), e depois nos diretórios default do compilador.

# Definição e uso de constantes

O preprocessador é frequentemente usado para a definição de constantes, através do comando `#define`:

vetor.c

```c
#define VETSIZE 64

int main ()
{
  int vetor[VETSIZE] ;

  for (i = 0; i < VETSIZE; i++)
    vetor[i] = i ;
}
```

Após a definição, todas as ocorrências da string VETSIZE no arquivo serão substituídas pelo valor 64, **antes da compilação**, resultando no seguinte código-fonte:

```c
int vetor[64] ;

for (i = 0; i < 64; i++)
  vetor[i] = 0 ;
```

Para evitar confusões entre variáveis da linguagem C e constantes do preprocessador, convencionou-se definir as constantes em MAIÚSCULAS.

## Constantes predefinidas

Algumas constantes são definidas previamente pelo sistema:

- __DATE__: data atual (formato “MMM DD YYYY”)
- __TIME__: horário atual (formato “HH:MM:SS”)
- __FILE__: nome do arquivo corrente.
- __LINE__: número da linha corrente do código-fonte.
- __func__: nome da função corrente.

Um exemplo de uso:

data.c

```c
#include <stdio.h>

int main ()
{
  printf ("Este código foi compilado em %s\n", __DATE__) ;
}
```

Além destas, muitas outras constantes podem estar disponíveis, dependendo da plataforma:

- Macros predefinidas no preprocessador GNU
- Pre-defined C/C++ Compiler Macros

# Compilação condicional

Uma constante pode ser definida sem um valor específico, Neste caso ela funciona como como um *flag* verdadeiro/falso, que pode ser testado pelo preprocessador através de comandos específicos:

debug.c

```
#include <stdio.h>

#define VETSIZE 64

int main ()
{
  int i, vetor[VETSIZE] ;

  for (i = 0; i < VETSIZE; i++)
  {
    vetor[i] = i ;
    #ifdef DEBUG
    printf ("Valor de i: %d\n", i) ;
    #endif
  }
}
```

No exemplo acima, a linha do `printf` só estará presente no código enviado ao compilador se a constante DEBUG estiver definida.

Constantes podem ser definidas no código-fonte usando o comando `#define` (como nos exemplos acima), mas também podem ser definidas na linha de comando, ao invocar o compilador:

```
gcc -DDEBUG debug.c
```

Um uso frequente da compilação condicional é a construção de *include guards*, ou seja código para evitar múltiplas inclusões do mesmo arquivo:

headers.h

```
#ifndef _THIS_HEADER_FILE_
#define _THIS_HEADER_FILE_
...
#endif
```

Da mesma forma, pode-se evitar redefinir constantes que já estejam definidas:

```
#ifndef NULL
#define NULL (void *) 0
#endif
```

Além do `ifdef`, existem outros operadores condicionais, como o `if` - `elif` - `else`:

```
#if DEBUG_LEVEL > 5
  // print all debug messages
```

```
  ...
#elif DEBUG_LEVEL > 3
  // print relevant debug messages
  ...
#elif DEBUG_LEVEL > 1
  // print prioritary debug messages
  ...
#else
  // print no debug messages
#endif
```

O operador `defined` permite testar se uma macro está definida:

```
#if defined (__arm__)         // macro definida em sistemas ARM
  #warning "Generating code for ARM processor."
  // code for ARM processors
  ...
#elif defined (__i386__)      // idem, x86
  #warning "Generating code for x86 processor."
  // code for Intel 32-bit processors
  ...
#else
  // abort compilation
  #error "Unknown architecture, aborting."
#endif
```

Observe o uso das diretivas `#warning` e `#error` no programa acima.

## Macros com parâmetros

O preprocessador é usado com frequência para construir macros, que são funções simples com parâmetros:

macro1.c

```
#include <stdio.h>

#define SQUARE(x) x*x

int main ()
{
  printf ("O quadrado de 5 é %d\n", SQUARE(5)) ;
}
```

O código acima, ao ser tratado pelo preprocessador, será transformado em:

```
printf ("O quadrado de 5 é %d\n", 5*5) ;
```

Observe que a macro SQUARE não computou o resultado de 5*5, apenas fez a substituição do parâmetro 5 pela expressão que ela define (5*5).

Tome **muito** cuidado ao definir macros com parâmetros, pois a substituição de texto pode levar a expressões erradas!

O exemplo abaixo apresenta um erro dessa natureza:

```
#define SQUARE(x) x*x
...
printf ("O quadrado de 2+3 é %d\n", SQUARE(2+3)) ;
```

O preprocessador transformará essa expressão em:

```
printf ("O quadrado de 2+3 é %d\n", 2+3*2+3) ;
```

O resultado da expressão deveria ser 25 (o quadrado de 2+3) mas será 11, por causa da precedência entre os operadores aritméticos (que o preprocessador não trata).

Para evitar esse erro, a macro deve ser declarada usando **parênteses**:

macro2.c

```
#include <stdio.h>

#define SQUARE(x) (x)*(x)

int main ()
{
  printf ("O quadrado de 5 é %d\n", SQUARE(2+3)) ;
}
```

Que resulta na expressão correta:

```
printf ("O quadrado de 2+3 é %d\n", (2+3)*(2+3)) ;
```

## Operações avançadas

O preprocessador C tem operações avançadas que vão muito além do escopo desta breve introdução. Para uma visão mais completa e profunda consulte este documento: The C Preprocessor.