# Ambiente de Programação Linux

Levy G. S. Galvão

## 1 Etapas do processo de compilação

1. Pré-processamento (*preprocessing*)

Durante a etapa de pré-processamento, o pré-processador lida com suas diretivas, e.g. `#include`, `#define`, `#ifdef`, etc.

A exemplo da diretiva `#include`, esta é utilizada para importar funcionalidades de bibliotecas ou outros arquivos ao código do projeto, de forma que o pré-processador irá buscar estes arquivos e copiá-los ao código fonte do arquivo de chamada.

Ele também irá lidar com as macros geradas pela chamada do `#define` e realizará as substituições diretamente no código fonte em C/C++.

É durante esta etapa que os comentários são removidos do código fonte.

1. Compilação (*compilation*)

Durante a etapa de compilação, o código fonte em C/C++ é analisado de acordo com sua sintaze e semântica (agora sem diretivas do pré-processador) e o código na linguagem Assembly para a arquitetura do processador alvo é gerado.

Esse processo se baseia em organizar a linguagem em alto nível e distribuir o código de acordo com a arquitetura do conjunto de instruções (*instruction set architecture* ou ISA) do processador alvo.

2. Montagem (*assembly*)

Nesta etapa, o montador converterá o código Assembly gerado anteriormente, em um código puramente binário para ser interpretado pela máquina (arquivo objeto).

1. Vinculação (*linking*)

A etapa de vinculação é a última durante o processo de compilação. O vinculador é responsável por reunir todos os arquivos objeto de múltiplos módulos em um único arquivo.

Algumas bibliotecas já possuem seus arquivos objetos gerados, assim encessitam apenas ser vinculados juntamente aos demais, e.g. a função `printf()`.

No caso da vinculação estática, o vinculador copia todas as funções de bibliotecas que serão utilizadas para o arquivo executável final. Já na vinculação dinâmica, o código não é copiado, mas sim colocado o nome da biblioteca no arquivo binário.

## 2 Funções do gcc

Durante esta seção, serão usados os arquivos test no diretório `./options_test` para exemplificar as funções.

1. `-static`

Permite a vinculação de um programa estaticamente, não dependendo de bibliotecas dinâmicas. Mas para isso, se faz necessário que os arquivos `.a` da bilbioteca existam no sistema, e.g. possuir a versão esta´tica de bibliotecas C. Em termos de código, para se obter esse efeito, faz-se:

```
$ cd options_test/
$ gcc app.c -static
```

1. `-g`

Permite incluir informações de depuração padrão no binário. Em termos de código, para se obter esse efeito, faz-se:

```
$ cd options_test/
$ gcc -g app.c -o app
$ gdb app
(gdb) run
Starting program: /home/leavitt/SEII-LevyGabriel/Semana02/options_test/app
Hello World!.
[Inferior 1 (process 7258) exited normally]
(gdb) quit
```

1. `-pedantic`

Permite emitir todos os avisos que são requisitados pelo padrão ANSI/ISO C. Usando a função `-std=89`, algumas extensões do gcc são desativadas. Já com o `-pedantic`, ainda mais extensões são desativadas.

```
$ cd options_test/
$ gcc app.c -pedantic
```

1. `-Wall`

Emite todos os avisos úteis que o gcc pode oferecer.

```
$ cd options_test/
$ gcc -Wall app.c -o app
app.c: In function ‘main’:
app.c:5:12: warning: unused variable ‘j’ [-Wunused-variable]
    5 |     int i, j;
```

```
      |            ^
app.c:5:9: warning: unused variable ‘i’ [-Wunused-variable]
    5 |     int i, j;
      |
```

2. `-Os`

Realiza optimizações a respeito do tamanho do código, e.g. alinhamento de funções, pulos, laços etc.

```
$ cd options_test/
$ gcc -Os app.c -o app
```

3. `-O3`

Realiza maiores optimizações para o tamanho do código e em tempo de execução. Também realza todas as optimizações propostas por `-O2` e mais, e.g. permite os alinhamentos descritos anteriormente, como outros em termos de comandos em uma única linha etc.

```
$ cd options_test/
$ gcc -O3 app.c -o app
```