Make

O utilitário `make` é uma ferramenta de automação de compilação amplamente utilizada em projetos de software, especialmente para programas escritos em C e C++. Ele é usado para gerenciar a construção e a compilação de projetos, facilitando a recompilação apenas das partes do projeto que foram modificadas, economizando tempo e esforço.

Conceitos Básicos

Makefile

O `make` utiliza um arquivo chamado `Makefile` (ou `makefile`) para determinar como compilar e ligar o projeto. O Makefile contém regras que descrevem como transformar um conjunto de arquivos fonte em um executável ou outro tipo de arquivo alvo.

Sintaxe do Makefile

Um Makefile é composto por regras que têm a seguinte sintaxe básica:

```
alvo: dependências
    comando
```

- **alvo (target):** O arquivo que você quer gerar (por exemplo, um executável ou um arquivo objeto).
- **dependências (dependencies):** Arquivos que são necessários para construir o alvo.
- **comando (command):** Comando(s) de shell que `make` executa para construir o alvo. Deve ser precedido por um tab.

Usando um Makefile

Para simplificar o processo de compilação, especialmente em projetos maiores, é uma boa prática usar um `Makefile`. O `Makefile` automatiza as etapas de compilação e ajuda a gerenciar dependências entre arquivos.

Exemplo de Makefile

Vamos utilizar o projeto da aula anterior e criar um **makefile** para ele. Crie um arquivo com nome "**Makefile**" (assim mesmo, iniciando por maiúscula) na raiz do projeto.

A estrutura do diretório é assim:

codigo/ ├── main.c ├── funcoes.c ├── funcoes.h └── Makefile

Crie um arquivo chamado `Makefile` no diretório do seu projeto com o seguinte conteúdo:

```
# Nome do executável
TARGET = teste

# Arquivos fonte
SRCS = main.c funcoes.c

# Arquivos objeto
OBJS = $(SRCS:.c=.o)

# Compilador e flags
CC = gcc
CFLAGS = -Wall -Wextra -std=c99

# Regras de compilação
all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS) $(TARGET)
```

Usando o Makefile

1. **Compilar o Projeto:**

   No terminal, navegue até o diretório do projeto e execute:

   ```
   make
   ```

Isso compilará todos os arquivos `.c` em arquivos objeto e, em seguida, ligará os arquivos objeto no executável `my_program`.

2. **Limpar Arquivos Compilados:**

   Para limpar os arquivos objeto e o executável, execute:

   ```
   make clean
   ```

Vamos detalhar o que cada uma das regras e variáveis neste Makefile significa e faz:

Variáveis

Compilador e Flags

```
CC = gcc
CFLAGS = -Wall -Wextra -std=c99
```

- **CC = gcc:** Define o compilador que será usado. Neste caso, `gcc` (GNU Compiler Collection).
- **CFLAGS = -Wall -Wextra -std=c99:** Define as flags (opções) que serão passadas ao compilador:
  - `-Wall`: Ativa uma grande quantidade de avisos úteis que ajudam a detectar possíveis erros.
  - `-Wextra`: Ativa avisos adicionais que não são cobertos por `-Wall`.
  - `-std=c99`: Define o padrão da linguagem C a ser usado, neste caso, o padrão C99.

Regras de Compilação

Regra Principal (default)

```
all: $(TARGET)
```

- **all:** É a regra padrão que será executada se você rodar `make` sem argumentos. Ela depende do alvo `$(TARGET)`.
- **$(TARGET):** É a variável que representa o nome do executável final. Se `TARGET = my_program`, esta regra será equivalente a `all: my_program`.

Regra para Criar o Executável

```
$(TARGET): $(OBJS)
    $(CC) $(OBJS) -o $(TARGET)
```

- **$(TARGET):** Esta regra especifica que o alvo `$(TARGET)` (por exemplo, `my_program`) depende dos arquivos objeto `$(OBJS)`.

- **$(CC) $(OBJS) -o $(TARGET):** O comando para criar o executável. Ele usa o compilador (`gcc`) para ligar os arquivos objeto (`$(OBJS)`) e gerar o executável (`$(TARGET)`).

Regra para Criar Arquivos Objeto

```
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
```

- **%.o: %.c:** Esta é uma regra de sufixo (ou regra implícita). Ela diz que qualquer arquivo `.o` depende do arquivo `.c` correspondente. O símbolo `%` é um curinga que pode corresponder a qualquer nome de arquivo.
- **$(CC) $(CFLAGS) -c $< -o $@:**
  - `$(CC)`: O compilador (`gcc`).
  - `$(CFLAGS)`: As flags de compilação.
  - `-c`: Compilar sem ligar (gera um arquivo objeto `.o`).
  - `$<`: O primeiro pré-requisito da regra (neste caso, o arquivo `.c` correspondente).
  - `-o $@`: Especifica o nome do arquivo de saída (neste caso, o arquivo `.o`).

Regra para Limpar Arquivos Gerados pela Compilação

```
clean:
    rm -f $(OBJS) $(TARGET)
```

- **clean:** Esta regra é usada para remover os arquivos gerados pela compilação.
- **rm -f $(OBJS) $(TARGET):** O comando para remover os arquivos objeto (`$(OBJS)`) e o executável (`$(TARGET)`).

Outras opções

Se você já baixou e compilou software da Internet em Linux, provavelmente já usou essa sequência de comandos:

```
./configure
make
sudo make install
```

Isso é quando o projeto foi criado com **autotools**.

O `./configure` é um script comumente usado em sistemas Unix/Linux como parte de um processo de configuração, compilação e instalação de software. Ele é geralmente utilizado em conjunto com `make` e `make install` para automatizar a construção de software a partir do código-fonte. Aqui está uma visão geral do que cada etapa faz:

./configure

O script `./configure` é usado para preparar o ambiente de construção para um programa. Ele faz uma série de verificações no sistema para garantir que todas as dependências necessárias e as ferramentas de compilação estejam presentes. Além disso, ele configura opções específicas de compilação com base no ambiente e nas preferências do usuário.

Principais Funções do `./configure`:

1. **Verificação de Dependências:**
    - Verifica se as bibliotecas e ferramentas necessárias estão disponíveis no sistema.
2. **Configuração do Ambiente:**
    - Determina detalhes específicos do sistema, como a localização de bibliotecas e headers.
3. **Personalização:**
    - Permite ao usuário especificar opções de configuração, como diretórios de instalação e habilitação/desabilitação de funcionalidades específicas.
4. **Geração de Makefile:**
    - Cria os Makefiles necessários para a compilação do projeto com base no ambiente e nas opções fornecidas.

Como Usar `./configure`:

No terminal, navegue até o diretório do projeto e execute:

```
./configure
```

Você também pode passar várias opções para personalizar a configuração:

```
./configure --prefix=/usr/local --enable-feature-x
```

- `--prefix=/usr/local`: Define o diretório de instalação.
- `--enable-feature-x`: Habilita uma funcionalidade específica do software.

make

Após a execução bem-sucedida do `./configure`, você usa `make` para compilar o projeto. O `make` lê os Makefiles gerados pelo `./configure` e executa as regras definidas para compilar o código-fonte.

Como Usar `make`:

No terminal, execute:

```

```

```
make
```

Isso compilará o código-fonte do projeto, gerando os binários necessários.

make install

Depois de compilar o projeto com `make`, você usa `make install` para instalar os binários e outros arquivos no sistema. Essa etapa geralmente copia os arquivos para diretórios apropriados, como `/usr/local/bin` para executáveis e `/usr/local/lib` para bibliotecas.

Como Usar `make install`:

No terminal, execute:

```
make install
```

Dependendo das permissões do sistema, você pode precisar usar `sudo` para instalar os arquivos em diretórios do sistema:

```
sudo make install
```

CMake

CMake é uma ferramenta de automação de compilação que gera arquivos de construção nativos (como Makefiles ou projetos de IDE) a partir de uma configuração de alto nível. É amplamente utilizada em projetos C e C++ devido à sua flexibilidade e suporte a múltiplas plataformas. CMake simplifica o processo de configuração, compilação e instalação de software.

Principais Características do CMake

1. **Portabilidade:** Suporta diversas plataformas e geradores (como Makefiles, projetos do Visual Studio, etc.).
2. **Flexibilidade:** Permite configurar e personalizar facilmente o processo de compilação.
3. **Configuração Modular:** Facilita a divisão do projeto em múltiplos módulos ou subprojetos.
4. **Detecção de Dependências:** Verifica automaticamente a presença de bibliotecas e outras dependências.

Estrutura Básica de um Projeto CMake

Um projeto CMake geralmente contém um ou mais arquivos `CMakeLists.txt` que definem as instruções de construção.

Exemplo Prático com CMake

Vamos criar um exemplo de projeto CMake com uma estrutura de diretórios similar ao exemplo anterior:

Estrutura do Projeto

```
my_project/
├── src/
│   ├── funcoes.c
│   ├── funcoes.h
│   └── main.c
└── CMakeLists.txt
```

Conteúdo dos Arquivos

1. `src/funcoes.c`

```c
#include <stdio.h>

int add(int a, int b) {
    return a + b;
}

void printMessage(const char *message) {
    printf("%s\n", message);
}
```

2. `src/funcoes.h`

```c
#ifndef FUNCOES_H
#define FUNCOES_H

int add(int a, int b);
void printMessage(const char *message);

#endif // FUNCOES_H
```

3. `src/main.c`

```c
#include <stdio.h>
#include "funcoes.h"

int main(void) {
    int result = add(5, 3);
    printf("Resultado: %d\n", result);

    printMessage("Olá, mundo!");
```

```
    return 0;
}
```

4. `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.10)

# Nome do projeto
project(my_project VERSION 1.0)

# Especifica os diretórios onde buscar por includes
include_directories(${PROJECT_SOURCE_DIR}/src)

# Adiciona o executável
add_executable(my_program src/main.c src/funcoes.c)
```

Passos para Configurar e Compilar o Projeto com CMake

1. **Instalar CMake:**

   Certifique-se de que o CMake está instalado no seu sistema. Em distribuições baseadas em Debian, você pode instalar com:

   ```
   sudo apt-get install cmake
   ```

2. **Criar um Diretório de Compilação:**

   É uma boa prática criar um diretório separado para a compilação, mantendo o diretório do código-fonte limpo.

   ```
   mkdir build
   cd build
   ```

3. **Configurar o Projeto com CMake:**

   No diretório de compilação (`build`), execute o comando CMake apontando para o diretório raiz do projeto (`..`):

   ```
   cmake ..
   ```

   Isso gerará os arquivos de construção (Makefiles, projetos de IDE, etc.) no diretório `build`.

4. **Compilar o Projeto:**

Depois de configurar o projeto com CMake, compile usando `make` ou a ferramenta de construção apropriada:

```
make
```

5. **Executar o Programa:**

   Após a compilação, o executável `my_program` estará disponível no diretório `build`. Você pode executá-lo diretamente:

```
./my_program
```

Resumindo

- **CMake:** Ferramenta de automação de compilação que gera arquivos de construção nativos a partir de configurações de alto nível.
- **Arquivo `CMakeLists.txt`:** Define as instruções de construção para o projeto.
- **Passos de Compilação com CMake:**
  1. Instalar o CMake.
  2. Criar um diretório de compilação.
  3. Configurar o projeto com `cmake ..`.
  4. Compilar o projeto com `make`.
  5. Executar o programa.

Vantagens do CMake

- **Portabilidade:** Suporta várias plataformas e ferramentas de construção.
- **Flexibilidade:** Facilita a configuração personalizada e modularização do projeto.
- **Integração com IDEs:** Suporta a geração de projetos para várias IDEs populares, facilitando o desenvolvimento.

Usar CMake pode simplificar significativamente o processo de construção de projetos C e C++, especialmente em ambientes de desenvolvimento complexos e multiplataforma.