

IMD0030

LINGUAGEM DE PROGRAMAÇÃO I

Aula 02 – Modularização e Compilação

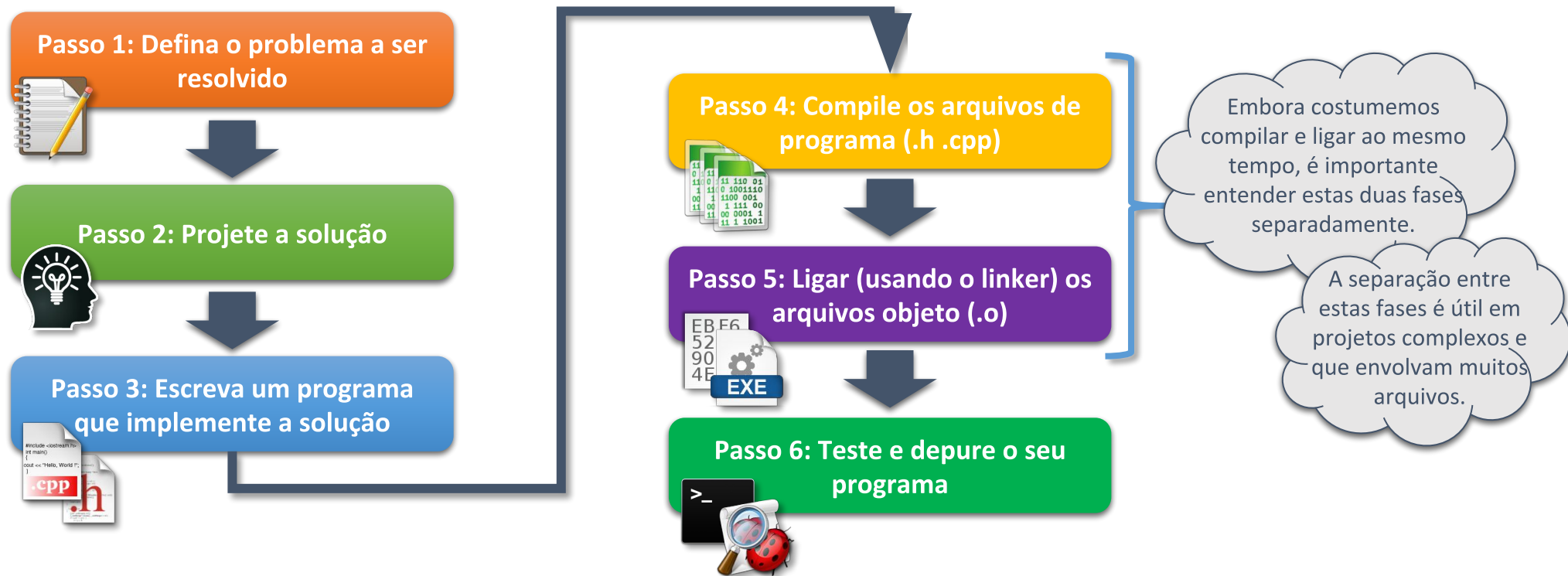
(material baseado nas notas de aula do Prof. Silvio Sampaio e Prof.
César Rennó-Costa)

Compilação

O compilador g++ e uso de Makefile

Introdução

- Antes de escrever e executar programas, é preciso entender em maior detalhe todo o processo de construção de um programa em C++



Introdução

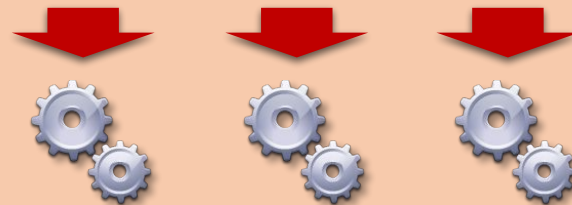
- Compilar X Ligar

Compilar: `g++ -c main.cpp func.cpp util.cpp`

Arquivos fonte (.cpp/.h)



Compilador

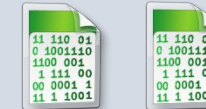


Arquivo objeto (.o)



Ligar (linker): `g++ -o prog main.o func.o util.o`

Arquivos objeto (.o)



Runtime Support



Biblioteca
(.lib, .a, .so)

Uma biblioteca (library) é uma coleção de arquivos objeto pré-compilados que permitem a reutilização de código entre diferentes projetos.

Arquivo Executável



Compilar: `g++ -c main.cpp func.cpp util.cpp`

Ligar: `g++ -o prog main.o func.o util.o`

Compilar e ligar: `g++ -o prog main.cpp func.cpp util.cpp`

O compilador

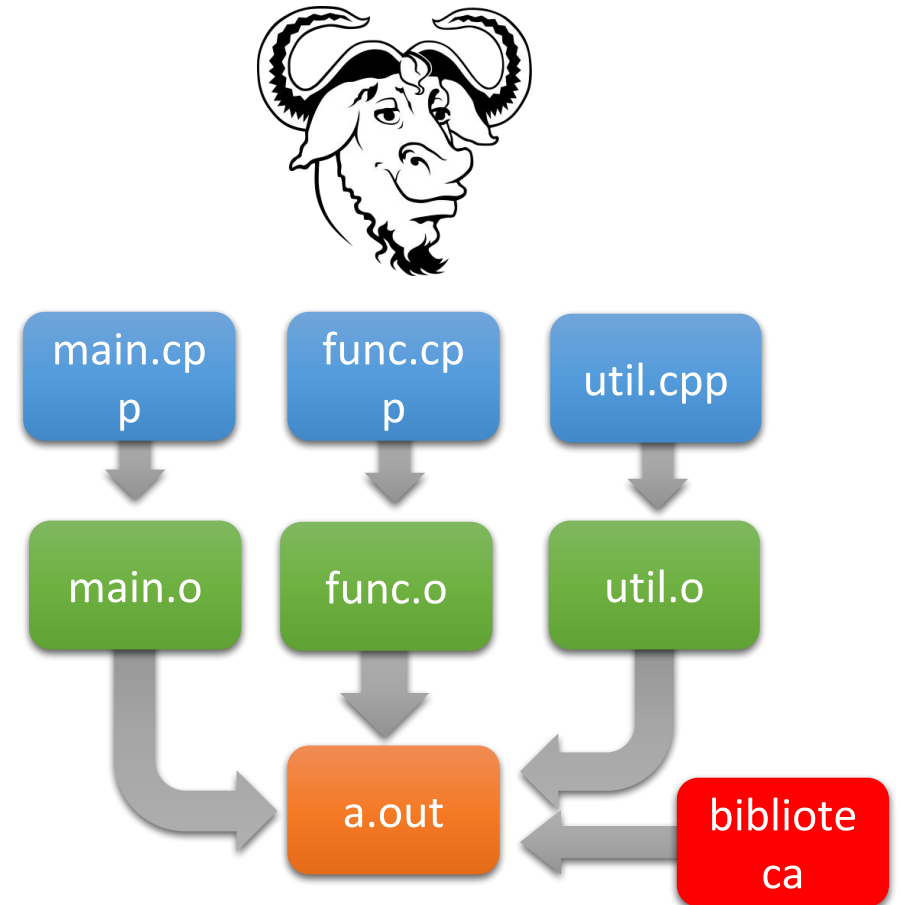
- Um **compilador** é um programa que, a partir de um código escrito em uma linguagem, o código fonte, cria um programa semanticamente equivalente porém escrito em outra linguagem, código objeto
 - Um compilador é um dos dois tipos mais gerais de tradutores, sendo que o segundo tipo que a ele deve ser comparado é um **interpretador**
 - Programas interpretados são geralmente mais lentos do que os compilados, mas são também geralmente mais flexíveis, já que podem interagir com o ambiente mais facilmente (frequentemente linguagens interpretadas são chamadas também de script)
 - Um interpretador, no momento da execução do programa, traduz cada instrução do programa e a executa em seguida
 - **C++ é uma linguagem compilada**
 - Normalmente, o código fonte é escrito em uma linguagem de programação de alto nível, com grande capacidade de abstração, e o código objeto é escrito em uma linguagem de baixo nível, como uma sequência de instruções a ser executada pelo processador
-

Opções do g++

```
g++ --help (return code: 0)
Usage: g++ [options] file...
Options:
  -pass-exit-codes      Exit with highest error code from a phase
  --help                Display this information
  --target-help          Display target specific command line options
  --help={common|optimizers|params|target|warnings|['^']}
                        {joined|separate|undocumented})[,...],
                        Display specific types of command line options
  (Use '-v --help' to display command line options of sub-processes)
  --version             Display compiler version information
  -dumpspecs            Display all of the built in spec strings
  -dumpversion          Display the version of the compiler
  -dumpmachine          Display the compiler's target processor
  -print-search-dirs    Display the directories in the compiler's search path
  -print-libgcc-file-name
                        Display the name of the compiler's companion library
  -print-file-name=<lib>
                        Display the full path to library <lib>
  -print-prog-name=<prog>
                        Display the full path to compiler component <prog>
  -print-multiarch      Display the target's normalized GNU triplet, used as
                        a component in the library path
  -print-multi-directory
                        Display the root directory for versions of libgcc
  -print-multi-lib      Display the mapping between command line options and
                        multiple library search directories
  -print-multi-os-directory
                        Display the relative path to OS libraries
  -print-sysroot        Display the target libraries directory
  -print-sysroot-headers-suffix
                        Display the sysroot suffix used to find headers
  -Wa,<options>         Pass comma-separated <options> on to the assembler
  -Wp,<options>         Pass comma-separated <options> on to the preprocessor
  -Wl,<options>         Pass comma-separated <options> on to the linker
  -Xassembler <arg>     Pass <arg> on to the assembler
  -Xpreprocessor <arg>  Pass <arg> on to the preprocessor
  -Xlinker <arg>        Pass <arg> on to the linker
  -save-temps           Do not delete intermediate files
  -save-temps=<arg>    Do not delete intermediate files
  -no-canonical-prefixes
                        Do not canonicalize paths when building relative
                        prefixes to other gcc components
  -pipe                Use pipes rather than intermediate files
  -time                Time the execution of each subprocess
  -specs=<file>         Override built-in specs with the contents of <file>
  -std=<standard>       Assume that the input sources are for <standard>
  --sysroot=<directory>
                        Use <directory> as the root directory for headers
                        and libraries
  -B <directory>       Add <directory> to the compiler's search paths
  -v                  Display the programs invoked by the compiler
  -###                Like -v but options quoted and commands not executed
  -E                  Preprocess only; do not compile, assemble or link
  -S                  Compile only; do not assemble or link
  -c                  Compile and assemble, but do not link
  -o <file>            Place the output into <file>
  -pie                 Create a position independent executable
  -shared              Create a shared library
  -x <language>        Specify the language of the following input files
                        Permissible languages include: c c++ assembler none
                        'none' means revert to the default behavior of
                        guessing the language based on the file's extension
```

O compilador g++

- O que acontece quando usamos o g++ para criar nosso programa?
 - Ex: **g++ main.cpp func.cpp util.cpp**
- Resposta: O compilador g++ cria o programa em duas fases
 - **Fase 1, Compilação:** Os arquivos fonte (.cpp) são compilados e geram os arquivos objeto (.o)
 - **Fase 2, Ligação (Linking):** Os arquivos objeto (.o) são ligados para criar um arquivo executável (em código de máquina). Nesta fase, códigos de bibliotecas são também ligados.
- Dica: Caso não seja indicado o nome do arquivo executável com o uso da opção **-o nome**, será criado o arquivo **a.out**
 - O exemplo **g++ -o prog main.cpp func.cpp util.cpp** cria um executável de nome **prog**
 - A ordem da opção não importa. Ex: **g++ main.cpp func.cpp util.cpp -o prog** tem o mesmo efeito do exemplo anterior.



O compilador g++

- Sintaxe geral: **g++ <option flags> <file list>**
 - **Option flags** são as opções usadas para alterar o comportamento padrão do compilador
 - Por exemplo, a forma mais simples de compilar seus arquivos fonte seria usar o comando **g++ *.cpp**
 - Isso geraria um comportamento padrão do compilador, entre outras coisas, geraria um executável com o nome **a.out**
 - Opções mais comuns para o compilador
 - **-c** : indica ao compilador para compilar os arquivos fonte (**.cpp**), mas não liga-los
 - A separação da compilação e ligação será útil na compilação de projetos usando **makefile**)
 - **-o <nome>** : especifica o **nome** do arquivo executável a ser criado a partir dos arquivos objeto (**.o**) já pré-compilados
 - **-g** : insere informações de depuração a serem usadas com depuradores compatíveis com o GDB (será visto mais à frente)
-

O compilador g++

- Opções mais comuns para o compilador (Cont.)
 - **-Wall** : diz ao compilador para indicar com um aviso (*warning*) qualquer instrução que possa levar a um erro
 - Por exemplo, ao habilitar esta opção, o compilador irá avisar sobre a instrução: **if (var = 5) { .. }**
 - A opção **-Wall** é uma combinação de um largo conjunto de opções de verificação do tipo **-W**, todas juntas. Tipicamente incluem:
 - variáveis declaradas, mas não utilizadas
 - variáveis possivelmente não inicializadas quando usadas pela primeira vez
 - padronização dos tipos de retorno
 - falta de colchetes ou parênteses em certos contextos que tornam uma instrução ambígua, etc.
-

O compilador g++

- Opções mais comuns para o compilador (Cont.)
 - **-I<dir>** (“i” maiúscula): adiciona o diretório **<dir>** na lista de diretórios para a busca de arquivos incluídos (através do uso da diretiva **#include**), ou seja, indica ao compilador uma fonte extra de arquivos cabeçalho (**.h**)
 - **-L<dir>** : adiciona o diretório **<dir>** na lista de diretórios para a busca de bibliotecas (**.a**)
 - **-l<libname>** (“L” minúscula) : faz com o que o compilador procure pela biblioteca indicada por **libname** no caso de nomes não resolvidos (*unresolved names*) durante a fase de ligação
 - **-ansi** : garante que o código compilado esteja em conformidade com o padrão ANSI C
 - **-pedantic** : torna a compilação ainda mais exigente no que diz respeito à obediência à padronização ANSI C
 - Para consultar ao conjunto completo de diretivas do compilador GNU gcc/g++, consulte:
 - <https://gcc.gnu.org/onlinedocs/>
-

Recomendações para a disciplina

- Para os exercícios e projetos, **utilizem sempre** as diretivas **-Wall** e **tratem todos os warnings**
- Igualmente, recomendamos sempre o uso das diretivas **-ansi** e **-pedantic** para garantir que o seu programa está escrito de acordo com a padronização
- Recomendamos ainda o uso das diretivas **-g** e **-O0** para permitir o uso do depurador, em caso de erro
- Com isso, um exemplo de compilação em linha de comando seria:
 - **g++ -o programa -Wall -ansi -pedantic -O0 -g main.cpp**

Formato da mensagem de erro no g++

- Um exemplo de erro comum ao compilar programas com o g++ é o de comentário não finalizado
 - *arquivo.cpp:22:1: unterminated comment*
 - Cada parte da mensagem de erro é separada por “dois-pontos”
 - A primeira parte (*programa.cpp*) é o nome do arquivo fonte no qual o erro ocorreu
 - A segunda parte (**22**) indica o número da linha na qual o erro foi detectado
 - A terceira parte (**1**) indica o número da coluna na qual o erro foi detectado
 - Esta é a única parte que não está presente em todas as mensagens de erro
 - A quarta parte (*unterminated comment*) é uma mensagem resumida que descreve o que provavelmente gerou o erro
-

Formato da mensagem de erro no g++

- Mensagens de *Warning* no g++ seguem o mesmo formato básico de um erro, apesar de que um arquivo fonte apenas com *warnings* continuará a ser compilado
 - Exceto no caso do uso da diretiva **-Wall** que transforma *warnings* em erros
 - Vale ressaltar que quando o g++ indica o número da linha, isso não necessariamente indica a linha do erro, mas sim a linha na qual o erro foi detectado
 - Com isso, encontrar o ponto exato do erro pode exigir uma inspeção de outras partes do código
 - Geralmente, o erro está próximo à linha no qual foi detectado
-

Exemplo

```
1  #include <iostream>      //permite entrada e saída de dados na tela
2
3  int main(){
4
5      // lê um valor inteiro
6      int minhaVariavel;
7
8      std::cin >> minhaVariavel;
9
10     std::cout << "Leu o numero " << minhaVariavel << std::endl;
11
12     return 0;
13 }
14
```

g++ -o programa -Wall -ansi -pedantic -O0 -g main.cpp

Exemplo

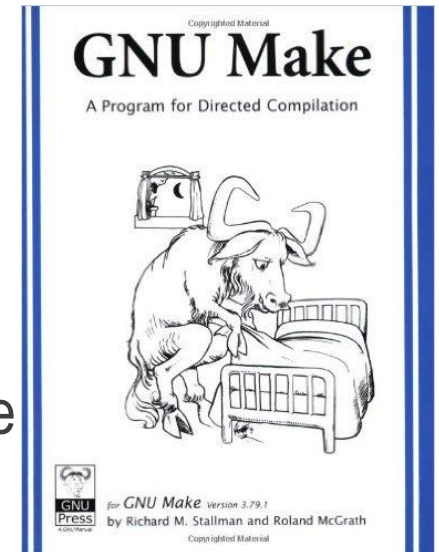
```
1  #include <iostream>    //permite entrada e saída de dados na tela
2
3  using std::cout;    //o programa utiliza cout
4  using std::cin;
5  using std::endl;
6
7  int main(){
8
9      // lê um valor inteiro
10     int minhaVariavel;
11
12     std::cin >> minhaVariavel;
13
14     cout << "Leu o numero " << minhaVariavel << endl;
15
16     return 0;
17 }
18
```

Relembrando

- Assim como em C, `main` é a primeira função (em C++ é chamado de método principal) a ser executada por qualquer programa em C++, mesmo que tenha outras funções escritas antes dela
 - Declaração
 - Mais simples:
`int main()` -- ou `int main(void)`
 - Completa, no caso de o programa receber argumentos via linha de comando:
`int main(int argc, char* argv[])`
 - `cin >> minhaVariavel`
 - `minhaVariavel = atoi(argv[1]);` // para float `atof`
-

Makefile

- A compilação de projetos acaba por se tornar uma tarefa difícil
 - Muitos arquivos que devem ser compilados
 - Repetição dos comandos de compilação para cada arquivo
 - A alteração em um único arquivo implica na recompilação de quais partes do projeto? Por garantia sempre compilamos tudo novamente?
- Esta dificuldade se agrava de acordo com a complexidade do projeto, que pode envolver centenas de arquivos



Makefile

- A título de exemplo, imagine um projeto com os seguintes arquivos: **ccountln.h**, **ccountln.cpp**, **fileops.h**, **fileops.cpp**, **process.h**, **process.cpp**, **parser.h**, **parser.cpp** e **main.cpp**
 - Para compilarmos manualmente este projeto, podemos seguir duas abordagens:
 - Compilar cada arquivo individualmente e ligar os arquivos objeto para criar o executável
 - `g++ -O0 -g -Wall -ansi -pedantic -c ccountln.cpp`
 - `g++ -O0 -g -Wall -ansi -pedantic -c parser.cpp`
 - `g++ -O0 -g -Wall -ansi -pedantic -c fileops.cpp`
 - `g++ -O0 -g -Wall -ansi -pedantic -c process.cpp`
 - `g++ -O0 -g -Wall -ansi -pedantic -c main.cpp`
 - `g++ ccountln.o parser.o fileops.o process.o main.o -o processos`
 - Compilar e ligar na mesma linha de comando
 - `g++ -O0 -g -Wall -ansi -pedantic -o processos ccountln.cpp parser.cpp fileops.cpp process.cpp main.cpp`
-

Makefile

- **Make** (responsável por processar o Makefile) é um utilitário GNU que determina quais partes de um projeto necessitam ser compilados ou recompilados, permitindo configurar os comandos para compilar e ligar o executável de forma automatizada (para todos os arquivos indicados)
 - Nos salva do tédio de repetir as linhas de comando ou de comandos gigantes do **g++**, além de economizar tempo
-

Makefile

- Um arquivo Makefile consiste de uma series de regras (*rules*), na seguinte forma:

```
alvo :    pre-requisitos ...  
          comando1  
          comando2  
          comando3  
          ...
```

- A regra explica como e quando gerar (ou regerar) o arquivo alvo
 - A simples chamada ao comando *make* executa, por padrão, a primeira regra
 - Para executar uma regra específica, é preciso chamar o comando “**make <alvo>**”
 - **Make exige um caracter <TAB> antes de cada comando (causa erros!)**
-


Makefile

- **“alvo”**: Usualmente o nome de um executável/binário ou arquivo objeto (.o) que deve ser gerado pelo compilador, mas pode indicar também uma ação a ser realizada (Ex: “clean”)
 - **“pre-requisitos”**: Uma lista dos arquivos necessários para criar o alvo
 - Se um destes arquivos tiver sido alterado, então o utilitário *make* irá reconstruir o alvo
 - Também chamado de “dependências”
 - **“comando”**: Uma ação a ser realizada
 - Usualmente, uma compilação ou ligação (usando o g++, por exemplo)
 - Pode ser algum comando do S.O. ou programa externo
 - No *make*, os comandos não executados e geram saídas como se estivessem a ser rodados a partir da linha de comando
-

Exemplo de um makefile

imd0030: main.cpp prime.h

g++ -o imd0030 main.cpp



apenas uma
regra

imd0030: main.o processos.o

g++ -o imd0030 main.o processos.o



série de regras

main.o: main.cpp util.h

g++ -c main.cpp

processos.o: processos.cpp prime.h

g++ -c processos.cpp

Exemplo de um makefile

#Makefile for "imd0030" C++ application

#Created by Silvio Sampaio 10/08/2016

Variável

PROG = imd0030

Executável

CC = g++

Compilador

CPPFLAGS = -O0 -g -Wall -pedantic -I/usr/imd0030/include

Diretivas de
compilação

LDFLAGS = -L/usr/imd0030/lib -lmylib

Diretivas para o
ligador (linker)

OBJS = main.o processos.o database.o util.o

Arquivos objeto definidos como pré-requisitos

\$(PROG) : \$(OBJS)

Alvo padrão

\$(CC) \$(LDFLAGS) -o \$(PROG) \$(OBJS)

Regra de
construção do
executável

(Cont.)

main.o :

\$(CC) \$(CPPFLAGS) -c main.cpp

Regra de
construção dos
arquivos objeto

processos.o : processos.h

\$(CC) \$(CPPFLAGS) -c processos.cpp

database.o : database.h

\$(CC) \$(CPPFLAGS) -c database.cpp

util.o : util.h

\$(CC) \$(CPPFLAGS) -c util.cpp

clean:

Alvo "clean"

rm -f core \$(PROG) \$(OBJS)

Regra de
limpeza dos
arquivos

Makefile

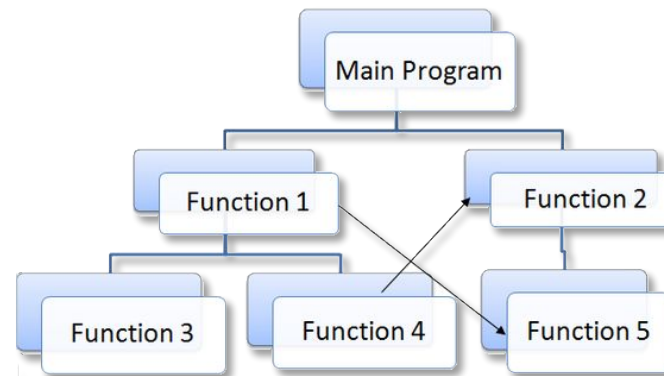
- Após a criação do arquivo *Makefile*, basta digitar o comando **make** no diretório contendo o *Makefile* para dar início ao processo automatizado de compilação e ligação
- O arquivo *Makefile* pode vir a ser grande e complexo (de acordo com o seu projeto), mas, uma vez pronto e funcional, basta o uso do comando **make** e pronto!
 - Voltaremos a este assunto para melhorar os nossos Makefiles (e, com isso, facilitar a compilação dos projetos)

Modularização

Dividir para conquistar

Modularização

- Estratégia para a construção de **software complexo** a partir de pequenas partes distintas, cada uma contendo responsabilidades específicas
 - **Dividir para conquistar:** dividir uma solução complexa em pequenas tarefas, cada uma sendo resolvida individualmente
- **Módulo:** conjunto de instruções em um programa que possui **responsabilidade bem definida** e é **o mais independente possível** em relação ao resto do programa



Por que modularizar?

Facilitar a vida do programador em termos de

- Organização do programa e das instruções que o compõem
 - O código torna-se mais fácil de gerenciar
 - Leitura do código produzido
 - O trabalho em equipe se torna mais fácil
 - Futura manutenção do código
 - Os módulos podem ser testados individualmente e de forma independente uns dos outros
 - Eventuais alterações são feitas em pontos específicos do programa, nos módulos
 - Eventual reutilização do código
 - Os módulos são independentes uns dos outros, então podem ser reusados de forma mais fácil em diferentes programas e/ou por outros programadores (inclusive na forma de bibliotecas)
-

Por que modularizar?

- **Código modular permite ser desenvolvido e testado uma só vez**, embora possa ser usado em várias partes de um programa
 - **Permite a criação de bibliotecas** que podem ser usadas em diversos programas e por diversos programadores
 - **Permite economizar memória**, dado que o módulo utilizado é armazenado uma única vez ainda que seja utilizado em diferentes partes do programa
 - **Permite ocultar código**, uma vez que apenas a estrutura do código fica disponível para outros programadores
-

Tipos de modularização

- **Modularização interna**

- Divisão do código contido em um arquivo em **múltiplas funções**
- Cada função executa um **conjunto de instruções bem definido**, representando uma tarefa específica dentro do programa

- **Modularização externa**

- Divisão do programa em **múltiplos arquivos**, cada um podendo conter um conjunto de funções e outros elementos (tipos, constantes, variáveis globais, etc.)
 - Programas mais complexos em C++ são tipicamente organizados na forma de **arquivos de cabeçalho e arquivos de corpo**
-

Modularização interna

- **Um programa não precisa ser composto por uma única, grande função principal (`main`)**
 - **Estratégia: dividir para conquistar**
 - Pensar na solução do problema e como ela pode ser dividida em partes menores
 - Implementar funções, cada uma realizando **uma** tarefa específica e bem definida
 - Implementar trechos de código que se repetem no programa como corpo de funções, chamadas em substituição a tais trechos que se encontravam repetidos
-

Um exemplo

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

int main() {
    int opcao;

    cout << "Conversor de temperatura" << endl;
    cout << "(1) Celsius -> Fahrenheit" << endl;
    cout << "(2) Fahrenheit -> Celsius" << endl;
    cout << "Digite sua opcao: ";
    cin >> opcao;

    float temp;
    cout << "Digite a temperatura: ";
    cin >> temp;
    float conv;
```

```
    switch(opcao) {
        case 1:
            conv = temp * 1.8 + 32;
            cout << temp << "°C = " << conv << "°F" << endl;
            break;
        case 2:
            conv = (temp - 32) / 1.8;
            cout << temp << "°F = " << conv << "°C" << endl;
            break;
        default:
            cout << "Opcao invalida" << endl;
    }

    return 0;
}
```

Modularizando em funções...

```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
```

```
float celsius2fahrenheit(float temp) {
    return temp * 1.8 + 32;
}
```

```
float fahrenheit2celsius(float temp) {
    return (temp - 32) / 1.8;
}
```

```
int main() {
    int opcao;
```

```
    cout << "Conversor de temperatura" << endl;
    cout << "(1) Celsius -> Fahrenheit" << endl;
    cout << "(2) Fahrenheit -> Celsius" << endl;
    cout << "Digite sua opcao: ";
    cin >> opcao;
```

```
    float temp;
    cout << "Digite a temperatura: ";
    cin >> temp;
    float conv;
```

```
    switch(opcao) {
```

```
        case 1:
```

```
            conv = celsius2fahrenheit(temp);
            cout << temp << "°C = " << conv << "°F" << endl;
            break;
```

```
        case 2:
```

```
            conv = fahrenheit2celsius(temp);
            cout << temp << "°F = " << conv << "°C" << endl;
            break;
```

```
        default:
```

```
            cout << "Opcao invalida" << endl;
```

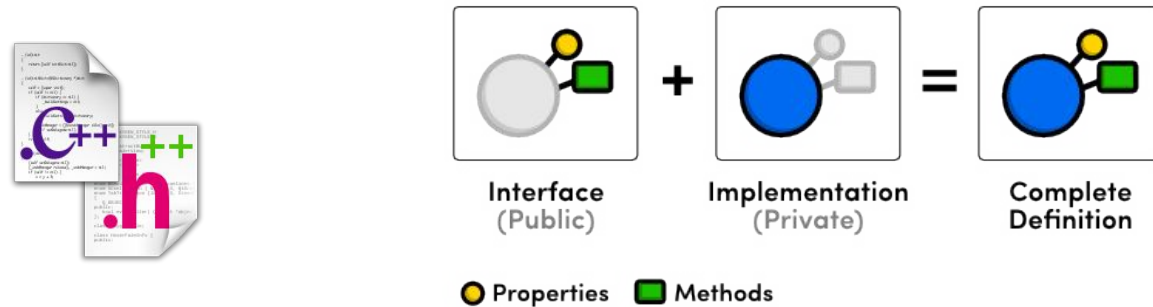
```
    }
```

```
    return 0;
```

```
}
```

Modularização externa

- Por convenção da linguagem C++, a organização do código de um programa pode ser feita da seguinte maneira:
 - **Arquivos de cabeçalho (.h)** contêm declarações de estruturas, tipos, variáveis globais, protótipos de funções, constantes, etc. e não podem conter a função principal do programa (`main`)
 - **Arquivos de corpo (.cpp)** implementam ou fazem chamadas ao que é definido nos arquivos de cabeçalho



Voltando ao exemplo anterior...

```
#ifndef CONV_H
#define CONV_H

// Conversao de temperatura em escala Celsius para Fahrenheit
float celsius2fahrenheit(float temp);

// Conversao de temperatura em escala Fahrenheit para Celsius
float fahrenheit2celsius(float temp);

#endif
```

O arquivo de cabeçalho `conv.h` contém os **protótipos das funções** que realizam as respectivas conversões de temperatura, uma para cada tipo



`conv.h`

Arquivos de cabeçalho em C++

- São incluídos no programa através da diretiva de pré-processamento `#include` seguida do nome do arquivo de cabeçalho
 - A inclusão **copia o conteúdo** de um arquivo em outro
 - A fim de **evitar duplicidade de cópias** do código a cada nova inclusão, os arquivos de cabeçalho precisam ser **identificados e protegidos contra múltiplas inclusões**, pois elas podem levar a **erros de redefinições**
 - A identificação é feita através da definição de um nome através da diretiva de compilação `#define`
 - A proteção contra múltiplas inclusões é implementada com a definição do bloco `#ifndef / #endif` (*if not defined / end if*) do pré-processador
-

Pré-processador

- O **pré-processador** é um programa que examina o código-fonte e executa certas modificações nele, baseado nas **diretivas de compilação**
 - As diretivas de compilação são comandos que não são compilados, sendo dirigidos ao pré-processador
 - O pré-processador é executado pelo compilador antes da compilação propriamente dita
 - Diretivas de compilação iniciam por um caractere # (*sharp* ou *hashtag*)
 - O comando `#include` é uma diretiva de compilação que diz ao pré-processador para copiar o conteúdo de um arquivo para outro, através de duas formas:
 - Quando usando arquivos das bibliotecas
 - `#include <iostream>` (biblioteca padrão de C++)
 - Quando usando arquivos do próprio usuário
 - `#include "conv.h"` (tipos e protótipos de sub-rotinas definidos pelo usuário)
-

Pré-processador

- O comando `#define` substitui palavras por valores, atuando como uma constante
 - Por exemplo, `#define PI 3.14159` permite substituir `PI` pelo valor `3.14159`
- Os comandos `#ifdef`, `#ifndef`, `#endif` permitem evitar que partes do código sejam inseridas no programa
 - Úteis nos arquivos de cabeçalho para evitar duplicidade
 - A diretiva de pré-processamento `#include` não verifica se um arquivo já foi incluído no programa, o que pode ser feito através da diretiva `#ifndef`

```
#ifndef NOME_DO_ARQUIVO_H    // Evita a redefinicao dos membros do arquivo
#define NOME_DO_ARQUIVO_H    // Inicio da definicao de NOME_DO_ARQUIVO_H

// Codigo do arquivo

#endif                      // Fim da definicao de NOME_DO_ARQUIVO_H
```

Pré-processador

- As principais diretivas de compilação são:
 - `#include` (inclusão)
 - `#define` (definição), `#undef` (remoção de definição)
 - `#ifdef` (*if defined*), `#ifndef` (*if not defined*)
 - `#if`, `#else`, `#elif` (*if / else / else-if*)
 - `#endif` (*end if*)
 - O domínio de diretivas de compilação permite a escrita de um código:
 - **mais rápido**
 - **mais legível**
 - **mais dinâmico**
 - **mais portátil** (multiarquitetura)
-

Arquivos de cabeçalho em C++

- Quando usamos uma biblioteca, **não deve ser preciso ler suas milhares de linha de código** para saber o que ela é capaz de fazer
- Solução: os arquivos de cabeçalho devem conter toda a descrição de **como usar** as funcionalidades da biblioteca sem se preocupar como elas foram implementadas
 - Os arquivos de cabeçalho tornam-se **pequenos, simples e explicativos**

Arquivos de corpo em C++

- Implementam ou fazem chamadas ao que é definido nos arquivos de cabeçalho
 - Mudanças na implementação definida no arquivo de corpo podem ser feitas sem necessariamente impactar os arquivos de cabeçalho
 - Necessário fazer a inclusão do(s) arquivo(s) de cabeçalho por meio da diretiva `#include`
 - `#include <iostream>` (biblioteca padrão de C++)
 - `#include "conv.h"` (tipos e protótipos de sub-rotinas definidos pelo usuário)
 - A inclusão de arquivos de cabeçalho em outros arquivos é efetuada somente uma vez no programa devido à verificação do identificador feita no próprio arquivo de cabeçalho
 - **Precisam ser compilados**
-

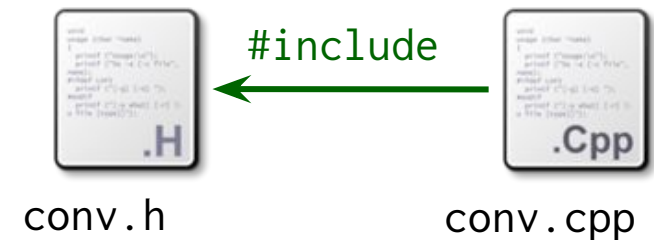
Voltando ao exemplo anterior...

```
#include "conv.h"

// Conversao de temperatura em escala Celsius para Fahrenheit
float celsius2fahrenheit(float temp) {
    return temp * 1.8 + 32;
}

// Conversao de temperatura em escala Fahrenheit para Celsius
float fahrenheit2celsius(float temp) {
    return (temp - 32) / 1.8;
}
```

O arquivo de corpo `conv.cpp` inclui o arquivo de cabeçalho `conv.h` e contém a implementação das funções que realizam as respectivas conversões de temperatura



Voltando ao exemplo anterior...

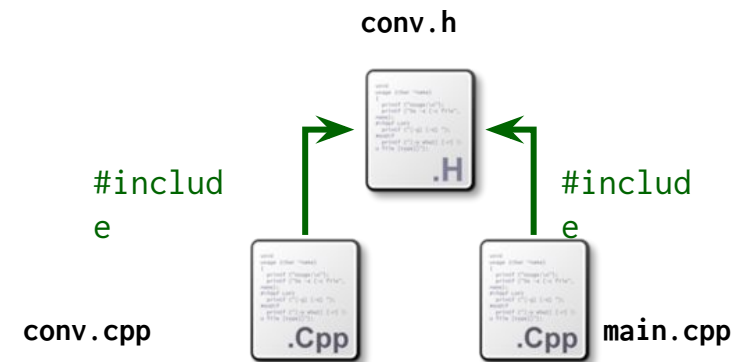
```
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include "conv.h"

int main() {
    int opcao;
    cout << "Conversor de temperatura" << endl;
    cout << "(1) Celsius -> Fahrenheit" << endl;
    cout << "(2) Fahrenheit -> Celsius" << endl;
    cout << "Digite sua opcao: ";
    cin >> opcao;
    float temp;
    cout << "Digite a temperatura: ";
    cin >> temp;
    float conv;
```

```
    switch(opcao) {
        case 1:
            conv = celsius2fahrenheit(temp);
            cout << temp << "°C = " << conv << "°F" << endl;
            break;
        case 2:
            conv = fahrenheit2celsius(temp);
            cout << temp << "°F = " << conv << "°C" << endl;
            break;
        default:
            cout << "Opcao invalida" << endl;
    }

    return 0;
}
```



Exercício de Aprendizagem E1.1

1. Escreva um programa que calcula a média aritmética de 3 números inteiros fornecidos pelo usuário.
 2. Modifique o programa (1) para que a entrada seja fornecida pela entrada padrão
 3. Escreva um programa que calcule a média de um conjunto de números. O programa deve receber pela entrada padrão o número de elementos do conjunto e os elementos do conjunto. A saída deve ser impressa na tela.
 4. Escreva um programa que calcule variância de um conjunto de números. O programa deve receber pela entrada padrão o número de elementos do conjunto e os elementos do conjunto. O programa deve ter pelo menos uma função definida em um arquivo de cabeçalho. A saída deve ser impressa na tela.
-

Aula 02

- Compilação
- Modularização

Próxima aula

- Depuração
-