

IMD0030

LINGUAGEM DE PROGRAMAÇÃO I

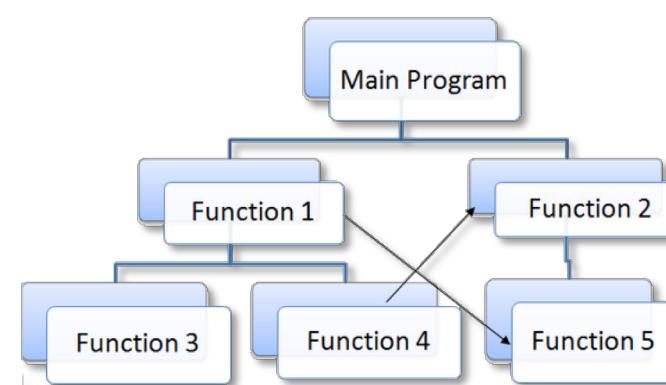
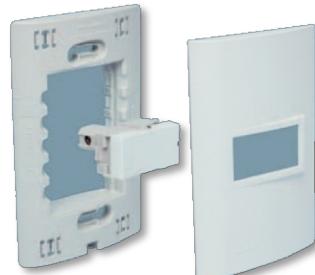
Modularização e Compilação

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 e guerreira 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;
```



```
switch(opcao) {
    float temp;
    cout << "Digite a temperatura: ";
    cin >> temp;

    float conv;
    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 opção: ";
    cin >> opcao;
```

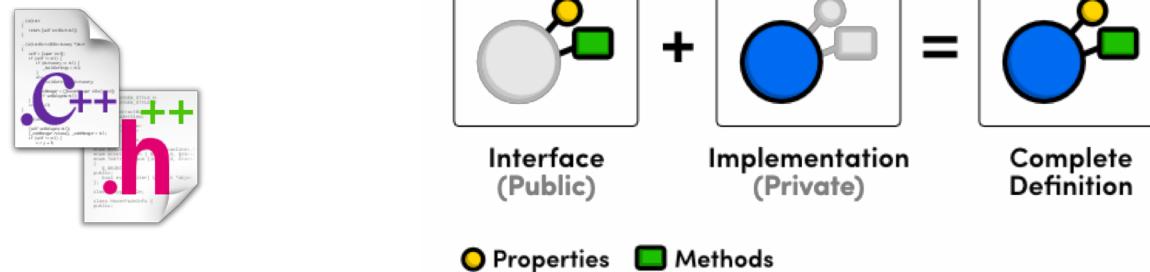
```
switch(opção) {
    float temp;
    cout << "Digite a temperatura: ";
    cin >> temp;

    float conv;
    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 << "Opção inválida" << 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 (1)

- 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 (2)

- 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

// Código do arquivo

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

Pré-processador (3)

- 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 portável** (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**

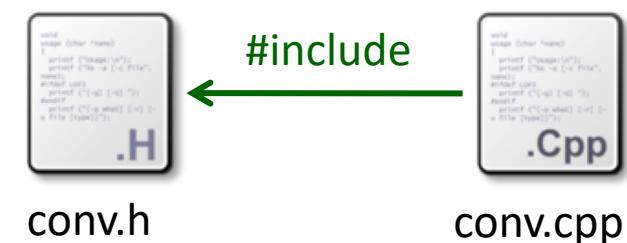
Voltando ao exemplo anterior...

```
#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



Arquivos de corpo em C++

- Ainda é possível tornar programas mais modulares **isolando o código** responsável pela função principal e testes da implementação
 - Permite a codificação de **diferentes programas de teste** utilizando as mesmas funções que foram previamente implementadas
- Isso pode ser feito criando-se um novo arquivo de corpo (.cpp) contendo **apenas a função principal**
 - Inclusão de arquivos de cabeçalho definidos pelo usuário
 - Inclusão de arquivos de cabeçalhos definidos na biblioteca padrão de C++
- **Também precisa ser compilado** juntamente com todos os outros existentes

Arquivos de corpo e método principal

- 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[])`

Voltando ao exemplo anterior...

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

#include "conv.h"

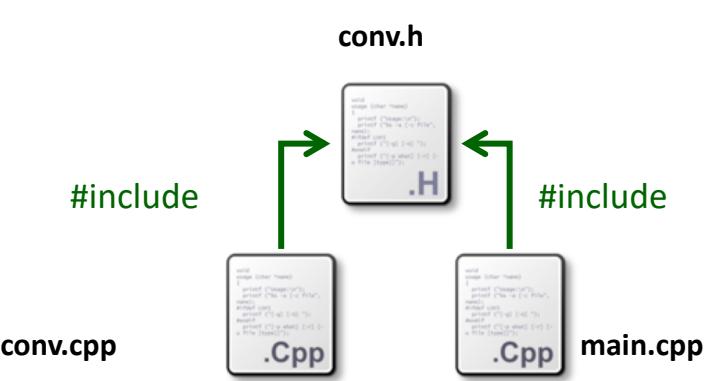
int main() {
    int opcao;

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

    switch(opcao) {
        float temp;
        cout << "Digite a temperatura: ";
        cin >> temp;
```

```
float conv;
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;
}
```

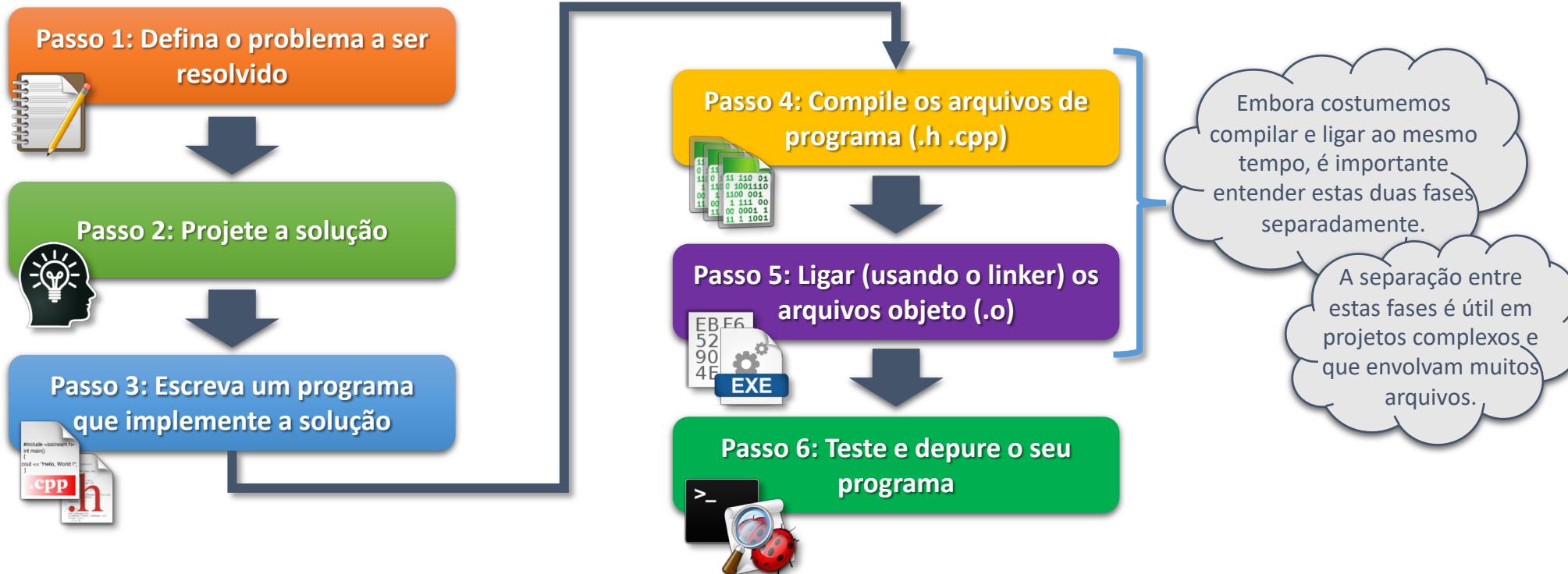


Compilação

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)

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`

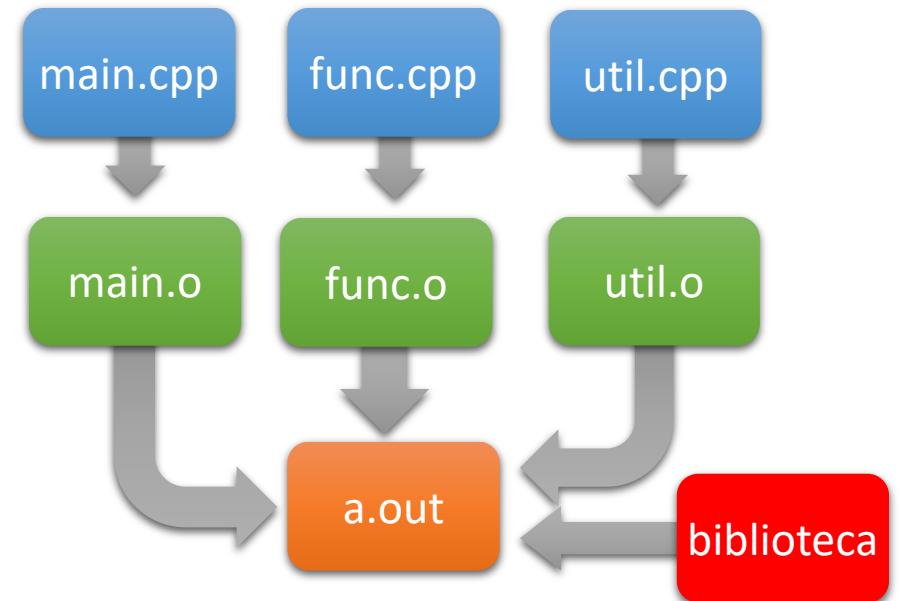
Uma biblioteca (library) é uma coleção de arquivos objeto pré-compilados que permitem a reutilização de código entre diferentes projetos.

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

O compilador g++ (1)

- 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++ (2)

- 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++ (3)

- 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) { .. }**. Embora a instrução seja sintaticamente válida, não faz sentido, uma vez que a atribuição no teste fará com que a condição seja sempre verdadeira. Provavelmente o programador cometeu um engano e usou o operador de atribuição ao invés do de comparação. Certamente a instrução desejada era **if (var == 5) { .. }**. No comportamento padrão do compilador, a instrução “errada” não geraria nenhum aviso.
 - 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++ (4)

- 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 (Ex: string, STL Vector)
 - **-I<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** : usado em conjunto com a flag **-ansi**, torna a compilação ainda mais exigente no que diz respeito à obediência à padronização ANSI C, rejeitando todo código que não esteja em conformidade

O compilador g++ (5)

- Opções mais comuns para o compilador (Cont.)
 - **-O<level>** : informa ao compilador o nível de otimização a ser empregado no código compilado
 - O nível **level** varia de 0 a 3, ou seja **-O0** a **-O3**
 - O nível zero (**-O0**) indica “sem otimização” e deve ser usado em conjunto com a opção **-g** para depuração
 - As otimizações terão impacto na velocidade do processo de compilação e de execução do código compilado
 - **-std=c++11** : habilita o suporte ao conjunto de instruções da revisão C++11 do compilador GCC realizada em 2011
 - **C++11 não é um compilador, mas sim uma padronização ISO que é implementada pela maioria dos compiladores populares**
- Para consultar ao conjunto completo de diretivas do compilador GNU gcc/g++, consulte:
 - <https://gcc.gnu.org/onlinedocs/>

O compilador g++: Teste rápido



- Exemplos de uso do compilador g++. O que cada exemplo faz?

1. g++ -o teste *.cpp
2. g++ -Wall -g -O0 main.cpp
3. g++ -c -I/usr/imd0030/paulo/lab1/include main.cpp src1.cpp src2.cpp
4. g++ -o ordena -L/usr/imd0030/paulo -lplib *.o
5. g++ -std=c++11 -Wall -pedantic -g -O0 main.cpp src1.cpp src2.cpp -o teste.exe

Recomendações para a disciplina

- Para os exercícios e projetos, **recomendamos fortemente** que sempre utilizem as diretivas **-Wall** e que **tratem todos os warnings**
- Igualmente, recomendamos sempre o uso da diretiva **-std=c++11** para informar ao compilador que deve ser aplicada a revisão de 2011
- 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 -std=c++11 -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



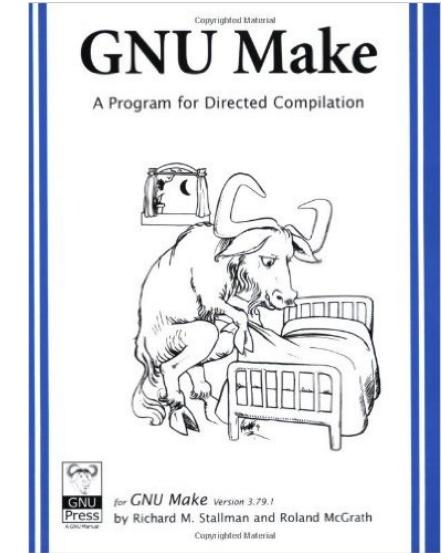
Formato da mensagem de erro no g++

- 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, (pra nossa felicidade) o erro está próximo à linha no qual foi detectado



Makefile (1)

- 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 que 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 (2)

- A título de exemplo, imagine um projeto que processa 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 -std=c++11 -c ccountln.cpp**
 - **g++ -O0 -g -Wall -std=c++11 -c parser.cpp**
 - **g++ -O0 -g -Wall -std=c++11 -c fileops.cpp**
 - **g++ -O0 -g -Wall -std=c++11 -c process.cpp**
 - **g++ -O0 -g -Wall -std=c++11 -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 -std=c++11 -o processos ccountln.cpp parser.cpp fileops.cpp process.cpp main.cpp**

Makefile (3)

- **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 (4)

- 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 (bobeira, mas causa erros!)**

Makefile (5)

- “**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

```
#Makefile for "imd0030" C++ application
```

```
#Created by Silvio Sampaio 10/08/2016
```

Variável

PROG = imd0030

Executável

CC = g++

Compilador

CPPFLAGS = -O0 -g -Wall -std=c++11 -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

Regra de construção do executável

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

(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 (6)

- 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)**

Exercício de Aprendizagem E1.1

- Números amigos são dois números que estão ligados um ao outro por uma propriedade especial:
 - Cada um deles é a soma dos divisores próprios do outro.
- Os divisores próprios de um número positivo N são todos os divisores inteiros positivos de N exceto o próprio N .

Um exemplo conhecido de números amigos são 284 e 220:

220: $1+2+4+5+10+11+20+22+44+55+110=284$

284: $1+2+4+71+142=220$

Exercício de Aprendizagem E1.1

- Um pouco de história:
 - A descoberta deste par de números é atribuída a Pitágoras.
 - Houve uma aura mística em torno deste par de números, e estes representaram papel importante na magia, feitiçaria, na astrologia e na determinação de horóscopos.
 - Outros números amigos foram descobertos com o passar do tempo.
 - Pierre Fermat anunciou em 1636 um novo par de números amigos formado por 17296 e 18416, mas na verdade tratou-se de uma redescoberta pois o árabe al-Banna (1256 - 1321) já havia encontrado este par de números no fim do século XIII.
 - Leonardo Euler, matemático suíço, estudou sistematicamente os números amigos e descobriu em 1747 uma lista de trinta pares, e ampliada por ele mais tarde para mais de sessenta pares.
 - Todos os números amigos inferiores a um bilhão já foram encontrados

(Fonte: <http://www.matematica.br/historia/namigos.html>).

Exercício de Aprendizagem E1.1

- Implemente um programa em C que encontre todos os pares de números inteiros amigos até 70000. Modularize seu código com o uso de funções.

Exemplo de saída:

(220, 284)

(1184, 1210)

(2620, 2924)

(5020, 5564)

(6232, 6368)

Exercício de Aprendizagem E1.2

- Ronnilsson é um aluno de ITP+PTP e acaba de aprender sobre recursividade. Porém, Ronnilsson ainda tem muitas dúvidas sobre como implementar suas funções de maneira recursiva.
- Atualmente Ronnilsson está travado em sua lista de exercícios com um problema que pede para implementar um programa que lê um inteiro X e retorna o maior número ímpar anterior ao valor do fatorial de X.
- Por exemplo, se $X = 5$, temos que fatorial de $X = 120$, logo o maior número ímpar anterior a X é 113.

Exercício de Aprendizagem E1.2

- Seguindo a sugestão do professor, Ronnilsson modularizou o programa em um conjunto de funções **fatorial()**, **primo()**, **primalidade()** e **menorPrimoAnterior()**.
- Ronnilsson sabe que algumas destas funções devem ser implementadas de forma recursiva, mas não sabe bem como.
- Ele já iniciou o seu código, baseado na função principal **main()** que o professor passou em aula para testar a implementação das funções.
- Mostre que você entende de recursividade e ajude Ronnilsson a completar o seu código de maneira correta, respeitando os comentários feitos por Ronnilsson.

Exercício de Aprendizagem E1.2

```
#include <iostream>
#include <cmath>

int fatorial(const int n) {
    // Calcula recursivamente o fatorial de n
    // Insira o código aqui
}

int primalidade(const int n, int i) {
    // Testa recursivamente os divisores ímpares de i até a raiz quadrada do N
    // Insira o código aqui
}
```

Exercício de Aprendizagem E1.2

```
int primo(const int n) {
    // Testa se é divisível por 2 executa a função primalidade() para testar os
    // divisores ímpares de 3 até a raiz quadrada do N
    if (n%2==0)
        return 0;
    return primalidade(n,3);
}
```

```
int maior_primo_anterior (int valor) {
    // Busca recursiva pelo maior valor primo anterior a valor
    // Insira o código aqui
}
```

Exercício de Aprendizagem E1.2

```
int main(int argc, char const *argv[])
{
    int x = 5;
    // Imprime: X =5 (5! = 120) => 113
    std::cout << "X = " << x << "(" << x << "!" = " << fatorial(x) << ")" => "
        << maior_primo_anterior(fatorial(x)) << std::endl;

    x = 3;
    // Imprime: X =3 (3! = 6) => 5
    std::cout << "X = " << x << "(" << x << "!" = "
        << fatorial(x) << ")" => " << maior_primo_anterior(fatorial(x)) << std::endl;
```

Exercício de Aprendizagem E1.2

```
x = 9;  
// Imprime: X = 9 (9! = 362880) => 362867  
std::cout << "X = " << x << "(" << x << "!" = "  
    << fatorial(x) << ") => " << maior_primo_anterior(fatorial(x)) << std::endl;  
  
return 0;  
}
```

Alguma Questão?

