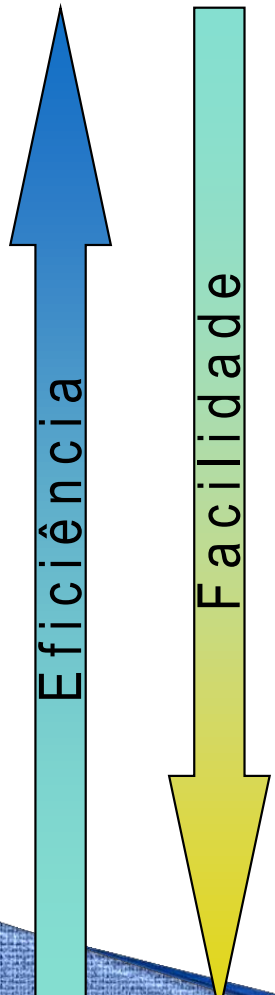
Unidade III: Classes e Objetos

- Revisão de conteúdo
  - Relacionados ao capítulo 2
    - O que significa o processo de compilação
    - Como criar projetos para efetuar compilação de vários módulos em separado
    - O que significa o processo de *"linkagem"* e como usar bibliotecas de funções em C
    - Qual a diferença entre declaração e definição e porque declarações são utilizadas
    - Qual a sintaxe utilizada para declarar funções em C
    - Como os *header files* são usados em C

Unidade III: Classes e Objetos
Revisão de conteúdo

- Linguagens de Programação: Classificação
  - Linguagem de máquina
    - Sequências de 0s e 1s
  - Linguagens simbólicas (abstratas)
    - Utilizam códigos mneumônicos
    - Baixo nível: Assembly
      - Mais voltadas para a máquina
      - Aprendizado demorado, codificação difícil
      - Rendimento máximo do computador
    - Médio nível: C, **C++**, Forth
      - Mais acessíveis ao computador
      - Modulares, flexíveis e portáteis
      - Aprendizado e programação não trivial
      - Rendimento otimizado do computador
    - Alto nível: Pascal, Fortran, SQL, Lisp, HTML, ASP, Java, C#, etc...
      - Voltadas para o programador, semelhantes à linguagem humana
      - Aprendizado e programação mais fáceis
      - Mais modulares, flexíveis e portáteis
      - Rendimento menor do computador



Unidade III: Classes e Objetos
Revisão de conteúdo

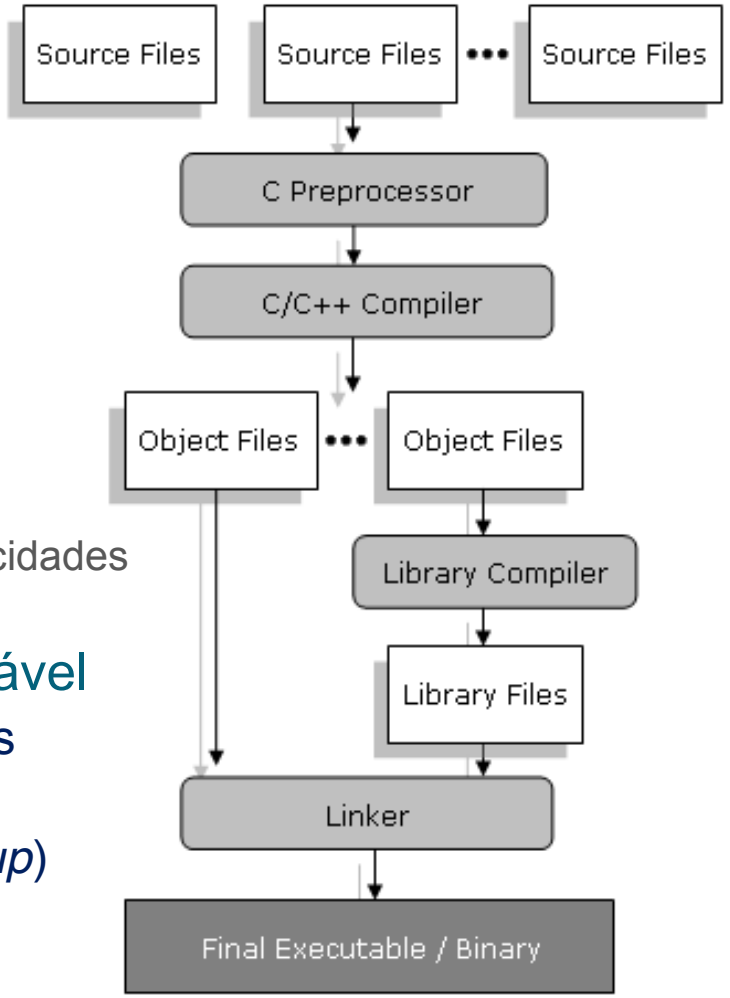
- A obtenção de código executável
  - Tradução de linguagem abstrata para linguagem binária
  - Principais formas
    - Interpretação: traduz o programa linha a linha
      - Analisa sintática e semanticamente o código: se ok, está pronto para executar
      - O programa vai sendo utilizado à medida em que vai sendo traduzido
        - Vantagens:
          - Transição entre escrever e executar é imediata
          - Código fonte está sempre disponível, facilitando identificar erros
          - Facilidade de interação e desenvolvimento rápido de programas
        - Desvantagens:
          - Execução é mais lenta do programa
          - Necessita sempre ser lido o código original inteiro para ser executado
          - Necessário traduzir e interpretar a cada execução do programa
          - Necessário retraduzir código repetido
          - Interpretador precisa estar na memória para executar o código

Unidade III: Classes e Objetos
Revisão de conteúdo

- A obtenção de código executável
  - Principais formas
    - Compilação: traduz o programa todo e passa a usá-lo desta forma
      - Produto final: um ou mais arquivos contendo código executável
        - Vantagens
          - Tendem a gerar código menor e mais eficiente
          - Alguns permitem geração de partes independentes e reusáveis
            - Programas muito grandes, que poderiam exceder limites, são construídos e testados por partes
            - Geração de bibliotecas: oculta complexidade e permite reúso
        - Desvantagens
          - Transição maior entre escrever e executar o código
          - Processo de compilação é mais trabalhoso
            - Necessita de arquivos auxiliares
            - Alguns usam mais memória para acelerar o processo

Unidade III: Classes e Objetos
Revisão de conteúdo

- Etapas do processo de geração de código executável
  - Pré-processador: substitui padrões
    - Facilitam a digitação e melhoram a legibilidade
    - Gera código pré-processado
  - Compilador: gera código objeto em 2 passos
    - *Parser*: quebra o código em unidades pequenas
      - Organiza componentes em uma árvore
      - Realiza verificação estática de tipos entre argumentos
      - Opcional: otimizador local
    - Gerador de código: percorre árvore e gera código
      - Em linguagem Assembly: pode ter otimizador p/ retirar duplicidades
      - Pode gerar direto em linguagem de máquina
  - Linkeditor (*Linker*): combina objs em módulo executável
    - Resolve referências a funções e variáveis entre módulos
      - Procura por arquivos especiais (bibliotecas padrão)
    - Gera módulo especial para inicializar o programa (*startup*)
      - Criar a pilha de execução e inicializar certas variáveis



Unidade III: Classes e Objetos
Revisão de conteúdo

- Principais conceitos relacionados
  - Declaração
    - Forma de dizer ao compilador o nome de funções e variáveis (externos) e suas características: não aloca espaço
      - "Esta função ou esta variável existe em algum lugar e é desta forma que ela deve ser"
        - Não é possível usar nomes repetidos em um mesmo contexto
  - Definição
    - Explicita o conteúdo da função ou variável: aloca espaço
      - "Coloque esta função / variável aqui"
      - Variável: verifica tamanho (tipo) e solicita reserva de espaço
      - Função: gera código correspondente, ocupando espaço de memória
  - Declaração x definição
    - São conceitos complementares
      - Principal diferença é o momento de alocação de espaço
      - Uma definição pode ser também uma declaração, caso não tenha ocorrido a declaração antes

Unidade III: Classes e Objetos
Revisão de conteúdo

- Declaração de funções em C e C++
  - *<extern> tipoRetorno nomeFunção (tipoArg1 a1, tipoArg2 a2, ...);*
    - Usar *extern* se for externa ao arquivo que a define
    - tipoRetorno deve ser definido (colocar *void* se não usa)
    - ";" ao final indica não ter a definição da função no momento
      - Se tiver a definição, aparecerá entre { e } e sem ";"
    - Os argumentos a1, a2, ... são opcionais na declaração
      - O tipo e sequência de argumentos devem ser fornecidos
        - *tipoRetorno nomeFunção ();*
          - Em C: função com qualquer número ou tipo de argumento
          - Em C++: função sem nenhum argumento
    - main() sempre retorna um tipo int

- Definição de funções em C e C++
  - Segue a declaração e contém o corpo
    - Coleção de comandos entre { e } (sem ";" ao final)
    - Se argumentos são usados no corpo, precisam ter seus nomes definidos na declaração
    - Se retornar valor, deve ser do tipo especificado em tipoRetorno

Unidade III: Classes e Objetos
Revisão de conteúdo

- Declaração x definição
  - Exemplos

```
//: C02:Declare.cpp
// Declaration & definition examples
extern int i;     // Declaration without definition
extern float f(float);    // Function declaration
float b;          // Declaration & definition
float f(float a) {        // Definition
    return a + 1.0;
}
int i;                    // Definition
int h(int x) {    // Declaration & definition
    return x + 1;
}
int main() {
    b = 1.0;
    i = 2;
    f(b);
    h(i);
}
```

Unidade III: Classes e Objetos
Revisão de conteúdo

- Uso de *header files* em C e C++
  - Arquivos de cabeçalho
    - Contém declarações externas de uma biblioteca fornecida
    - Usar a diretiva #include <nomearq> ou “nomearq.h”
      - Indica ao processador para abrir o arquivo e incluir ali seu conteúdo
        - <nomearq>: indica para procurar no caminho especificado no ambiente ou linha de comando do compilador
        - “nomearq.h”: indica para procurar no caminho relativo à pasta atual
          - Se não achar, assume o caminho default (<>)
            - Extensões mais usadas: “.h”, “.hxx” e “.hpp”
        - Padrão atual de nome de arquivos
          - Pode ter mais de 8 caracteres e sem extensão
            - Ex: <iostream.h> ➔ <iostream>
          - As bibliotecas herdadas do C continuam com a extensão “.h” mas podem ser usadas também com um “c” antes
            - Ex: <stdio.h> ➔ <***c***stdio>

Unidade III: Classes e Objetos
Revisão de conteúdo

- Principais conceitos relacionados
  - *Namespace*
    - Necessidade
      - Programas grandes são divididos em partes, cada uma mantida por grupos / pessoas diferentes
        - Isso poderia levar à utilização de mesmo nome de variáveis
    - Aplicação
      - "Embrulha" espaços, possibilitando separar espaços distintos
        - Se houver nome idêntico, mas em outro *namespace*, não haverá colisão de nomes
    - Formato
      - *using namespace nome;* (o padrão do C++ é o std)
        - Esta diretiva deixa o *namespace* disponível para todo o arquivo
          - Evite colocar no *header*, pois o expõe de forma inadequada
            - Ex: # include <iostream> using namespace std; ➔ #include <iostream.h> (forma + antiga)

III.1. Implementando classes e objetos em C++

- Programa inicial

Notas:

- cout << (console output): saída padrão
  << concatena saídas
- cin >> (console input): entrada padrão
  >> concatena entradas
- endl: "\n" na saída padrão

```
Programa 1: Hello World em C++
// C02:Hello.cpp - Saying Hello with C++
#include <iostream> // Stream declarations

using namespace std;

int main() {

        int age = 8;

        cout << "Hello, World! I am "

                << age << " years old "

                        << endl;

}
```

III.1. Implementando classes e objetos em C++

- Exercícios
  1. Analise os programas a seguir, presentes em nosso livro texto, e que apresentam alguns aspectos específicos da entrada e saída em C++
     - Stream2.cpp - formatação da saída
     - Concat.cpp - concatenação de arranjos de caracteres
     - Numconv.cpp - leitura de entrada e formatação da saída
     - CallHello.cpp - uso da função *system()* para executar um programa de dentro de outro

III.1. Implementando classes e objetos em C++

- Stream2.cpp - formatação da saída

Notas:

- Manipuladores iostream: não imprimem nada, mas mudam o estado do stream de output (dec, oct, hex)
- Impressão de ponto flutuante definida pelo compilador
- char(27): cast de char() com valor ASCII (27) = "escape"

```
// C02:Stream2.cpp
// More streams features
#include <iostream>
using namespace std;
int main() {
// Specifying formats with manipulators:
     cout << "a number in decimal: "
          << dec << 15 << endl;
     cout << "in octal: " << oct << 15 << endl;
     cout << "in hex: " << hex << 15 << endl;
     cout << "a floating-point number: "
          << 3.14159 << endl;
     cout << "non-printing char (escape): "
          << char(27) << endl;
}
```

III.1. Implementando classes e objetos em C++

- Concat.cpp - concatenação de arranjos de caracteres

Notas:

- C++ é uma linguagem de formato livre e permite que a sentença continue na linha seguinte
- O ";" termina uma sentença

```
// C02:Concat.cpp
// Character array Concatenation
#include <iostream>
using namespace std;
int main() {
cout << "This is far too long to put on a "
     "single line but it can be broken up with "
     "no ill effects\nas long as there is no "
     "punctuation separating adjacent character "
     "arrays.\n";
}
```

III.1. Implementando classes e objetos em C++

- Numconv.cpp - leitura de entrada e formatação da saída

Notas:

- cin e cout pertencem às classes istream e ostream do C++ e podem ser redirecionados para nova entrada e saída padrão
- Manipuladores iostream: não imprimem nada, mas mudam o estado do stream de output (oct, hex)

```
// C02:Numconv.cpp
// Converts decimal to octal and hex
#include <iostream>
using namespace std;
int main() {
     int number;
     cout << "Enter a decimal number: ";
     cin >> number;
     cout << "value in octal = 0"
          << oct << number << endl;
     cout << "value in hex = 0x"
          << hex << number << endl;
}
```

III.1. Implementando classes e objetos em C++

- CallHello.cpp - uso da função *system()* para executar um programa de dentro de outro

Notas:

- <cstdlib> define system();
- O argumento de *system* deve ser o comando que seria colocado no *prompt* do sistema operacional, contendo seus próprios argumentos (pode ser montado em tempo de execução)
- O comando é executado e o controle retorna para a instrução seguinte em main()
- Mostra também como é simples utilizar as funções da biblioteca do C no C++ (basta incluir o *header* e chamar a função)

```
// C02:CallHello.cpp
// Call another program
#include <cstdlib> // Declare "system()"
using namespace std;
int main() {
     system("Hello");
}
```

Exercícios

2. Escreva um programa em C++ que leia um conjunto de 4 valores `i`, `a`, `b`, `c`, onde `i` é um valor inteiro e positivo e `a`, `b`, `c`, são quaisquer valores reais e os escreva:
   - Se `i=1` escrever os três valores `a`, `b`, `c` em ordem crescente.
   - Se `i=2` escrever os três valores `a`, `b`, `c` em ordem decrescente.
   - Se `i=3` escrever os três valores `a`, `b`, `c` de forma que o maior entre `a`, `b`, `c` fique dentre os dois.

III.1. Implementando classes e objetos em C++

III.1.1. Utilizando objetos de classes existentes na biblioteca padrão

- Exemplo: funcionalidade básica das strings em C++

```
// C02:HelloStrings.cpp
// The basics of the Standard C++ string class
#include <string>
#include <iostream>
using namespace std;
int main() {          //HelloStrings.cpp
   string s1, s2;         // Empty strings
   string s3 = "Hello, World.";      // Initialized
   string s4("I am");          // Also initialized
   s2 = "years old";          // Assigning to a string
   s1 = s3 + " " + s4;   // Combining strings
   s1 += " 8 ";          // Appending to a string
   cout << s1 + s2 + "!" << endl;
}
```

III.1.1. Utilizando objetos de classes existentes na biblioteca padrão

- Exemplo: Lendo e escrevendo em arquivos

```
// C02:Scopy.cpp
// Copy one file to another, a line at a time
#include <string>
#include <fstream>  // ifstream and ofstream
using namespace std;

int main() {
  ifstream in("Scopy.cpp");      // Open for reading
  ofstream out("Scopy2.cpp");    // Open for writing
  string s;
  while(getline(in, s))          // Discards newline (\n) char
        out << s << "\n";        // ... must add it back
}
```

Nota:

- getline () retorna *true* quando lê com sucesso e *false* ao final do arquivo

III.1.1. Utilizando objetos de classes existentes na biblioteca padrão

- Lendo de um arquivo e armazenando todas as linhas numa string

```
// C02:FillString.cpp
// Read an entire file into a single string
#include <string>
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ifstream in("FillString.cpp");
    string s, line;
    while(getline(in, line))
        s += line + "\n";
    cout << s;
}
```

III.1.1. Utilizando objetos de classes existentes na biblioteca padrão

- Criando vetores com a biblioteca vector<T> da biblioteca padrão
  - Classe *template*, que permite a criação de vetores de tipos diferentes. Especificamos o tipo no momento da declaração do vetor
  - Podemos adicionar novos elementos ao vetor com *push_back*()

```
// C02:Fillvector.cpp
// Copy an entire file into a vector of string
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
int main() {
   vector<string> v;     // vetor de strings com 0 elementos !
   ifstream in("Fillvector.cpp");
   string line;
   while(getline(in, line))
      v.push_back(line); // Add the line to the end of vector
   // Add line numbers and print
   for(int i = 0; i < v.size(); i++)
      cout << i << ": " << v[i] <<  endl;
}
```

Nota:

- push_back() acrescenta item ao final do vetor

III.1.1. Utilizando objetos de classes existentes na biblioteca padrão

- Pode-se criar um vetor de qualquer tipo
  - Exemplo: uso de um vector<int>

```
// C02:Intvector.cpp
// Creating a vector that holds integers
#include <iostream>
#include <vector>
using namespace std;
int main() {
   vector<int> v;    // Vetor com 0 elementos
   for(int i = 0; i < 10; i++)
      v.push_back(i);     // Assign i to v[i]
   for(int i = 0; i < v.size(); i++)
     cout << v[i] << ", " ;
   cout << endl;
   for(int i = 0; i < v.size(); i++)
      v[i] = v[i] * 10;  // Assignment
   for(int i = 0; i < v.size(); i++)
      cout << v[i] << ", "; cout << endl;
}
```