**Processo de Compilação (C++**)

Do código C++ até ao executável e contexto de projetos

[**Exemplo de Código C++**](#_q0ubeiy1d3g3) **1**

[**Transformação passo-a-passo**](#_jeqcecvjq0ct) **2**

[**Geração de executáveis em contexto de projetos**](#_kiid6x9me205) **5**

Uma linguagem de programação é, em tudo, semelhante a uma linguagem escrita ou matemática. Assim sendo, os conceitos de sintaxe[[1]](#footnote-0) e de semântica[[2]](#footnote-1) são perfeitamente aplicáveis no contexto de linguagens de programação[[3]](#footnote-2).

Uma linguagem de programação permite criar um meio uniformizado para transmitirmos instruções a um computador. A linguagem de programação tem de ser, então, uma forma humanamente legível de dar «pistas» ao computador (através de um conjunto de termos específicos de uma linguagem) sobre a construção das várias instruções a serem executadas. Contudo, o computador não é capaz de interpretar esta mesma linguagem como os humanos e, por isso, será necessário converter esta linguagem para algo que o computador consiga perceber: sequências de bits e bytes.

Para que possamos transitar de uma linguagem de programação, como o C++, para a linguagem máquina (uma composição de sequências de bits, denominada de **código máquina**), precisamos de uma ferramenta que consiga converter as várias expressões que definimos na nossa linguagem de programação num conjunto de instruções devidamente codificadas em sequências de bits para que o processador do nosso computador as possa executar. A esta ferramenta dá-se o nome de **compilador**.

## Exemplo de Código C++

#include <iostream>

int main(void) {

std::cout << "Hello, World!" << std::endl;

return 0;

}

**Código 1** Código de exemplo “Hello, World!”.

Consideremos, para efeitos de demonstração, o código acima, relativo a um simples *Hello, World!* em C++. Este código está em C++, uma de muitas linguagens criadas para podermos descrever soluções de problemas a serem resolvidos por um computador. Se executarmos a compilação deste código, através do compilador GCC (ou outro compilador de C++), poderemos obter um executável que será capaz de imprimir no ecrã a mensagem “Hello, World!” (já que o código apresentado não tem erros).

No comando descrito de seguida (considerando que o $ é a tua linha de comandos[[4]](#footnote-3)) temos a instrução de compilação do ficheiro acima (aqui denominado de hello-world.cpp), em que passamos um parâmetro ao compilador g++[[5]](#footnote-4) que nos permitirá verificar cada um dos passos de compilação, gerando todos os ficheiros intermédios do processo, até ao final que é o **executável** pretendido.

$ g++ -save-temps hello-world.cpp -o hello-world

Executando este comando no terminal podemos verificar a geração de 5 ficheiros.

$ ls

hello-world hello-world.cpp hello-world.ii hello-world.o hello-world.s

## Transformação passo-a-passo

No início deste processo temos um ficheiro hello-world.cpp com o conteúdo do primeiro código exposto neste documento. Assim que o processo de compilação inicia, é realizada primeiramente a avaliação se o código em hello-world.cpp está completamente expandido ou não. Esta expansão verifica-se procurando por diretivas no código que indiquem que há **inclusões** por fazer.

De facto, no código do nosso *hello world* existe uma inclusão do ficheiro "cabeçalho" iostream. Assim sendo, através de um processo denominado de **pré-processamento**, o ficheiro é processado sendo gerada uma nova versão em que cada linha de inclusão de código é substituída pelo código completo correspondente ao que se pretende incluir: neste caso, dos cabeçalhos do iostream que fazem parte da biblioteca nativa do C++.

O ficheiro final, com o nome hello-world.ii, estará então o código do ficheiro inicial hello-world.cpp, mas com a linha #include <iostream> substituída por uma larga porção de código, que corresponde ao ficheiro do iostream, como podemos ver abaixo:

1 # 1 "hello-world.cpp"

2 # 1 "<built-in>" 1

3 # 1 "<built-in>" 3

4 # 382 "<built-in>" 3

(...) (--- truncado ---)

15958 template<class \_Tp>

15959 inline

15960 shared\_ptr<\_Tp>::shared\_ptr(const shared\_ptr& \_\_r) throw()

15961 : \_\_ptr\_(\_\_r.\_\_ptr\_),

15962 \_\_cntrl\_(\_\_r.\_\_cntrl\_)

15963 {

15964 if (\_\_cntrl\_)

15965 \_\_cntrl\_->\_\_add\_shared();

15966 }

(...) (--- truncado ---)

42862

42863 int main(void) {

42864 std::cout << "Hello, World!" << std::endl;

42865 return 0;

42866 }

**Código 2** Código produto do pré-processamento de hello-world.cpp, em hello-world.ii, truncado em tamanho.

Do código pré-processado podemos então passar ao verdadeiro **compilador** no qual a primeira grande transformação é realizada, diminuindo em muito a legibilidade do código para o ser humano. Aqui, passando o nosso código já pré-processado para um código a que denominamos de **assembly**, pela primeira vez temos um código que terá sido traduzido para instruções do conjunto de instruções do processador da máquina onde estamos a compilar o nosso código. Esta transformação, que posteriormente em disciplinas de Arquiteturas de Computadores irás aprender com mais detalhe, é a visão mais próxima que uma pessoa pode ter em relação ao que será, efetivamente, executado no processador do computador. Gera-se, assim, o ficheiro hello-world.s, a partir do hello-world.ii, o qual podemos ver no Código 3 abaixo.

1 .section \_\_TEXT,\_\_text,regular,pure\_instructions

2 .build\_version macos, 11, 0 sdk\_version 11, 3

3 .globl \_main ## -- Begin function main

4 .p2align 4, 0x90

5 \_main: ## @main

6 .cfi\_startproc

7 ## %bb.0:

8 pushq %rbp

9 .cfi\_def\_cfa\_offset 16

(...) (--- truncado ---)

1568 movl %esi, -12(%rbp)

1569 movq -8(%rbp), %rdi

1570 movl 32(%rdi), %esi

1571 orl -12(%rbp), %esi

1572 callq \_\_ZNSt3\_\_18ios\_base5clearEj

1573 addq $16, %rsp

1574 popq %rbp

1575 retq

1576 .cfi\_endproc

1577 ## -- End function

1578 .section \_\_TEXT,\_\_cstring,cstring\_literals

1579 L\_.str: ## @.str

1580 .asciz "Hello, World!"

1581

1582 .subsections\_via\_symbols

**Código 3** Código produto da compilação de hello-world.ii, em hello-world.s, truncado em tamanho.

O código assembly não é tão legível quanto o C++ inicialmente produzido, dizendo-se assim que o código assembly é de mais baixo nível que o código C++ (quanto mais baixo for, mais próximo se diz estar do processador). O alto nível de abstração permite-nos descrever melhor a solução que temos em mente, permitindo-nos também ser mais meticulosos na transposição do nosso pensamento em código. Contudo, existe um custo que pode ser alto na sua implementação: a solução final poderá não ser otimizada. Isto acontece porque embora possamos ser meticulosos a implementar um código ajustado à nossa solução, a forma como nós pensamos não é propriamente certa que seja a melhor ou otimizada.

Por outro lado, se implementarmos a nossa solução através de uma linguagem de baixo nível, muito provavelmente teremos uma solução que mais se aproximará de uma solução ótima em termos de desempenho de tempo ou espaço. Contudo, também temos um outro custo associado: a implementação de uma solução com esta abordagem exige um cuidado acrescido, dado que com esta manipulação de baixo nível acresce a responsabilidade de não comprometer a integridade de dados preservados nos vários espaços de memória utilizados pelo programa.

Finda a compilação e tendo o assembly realizado, o processo de criação do executável ainda se distancia por dois últimos passos. O primeiro passo é converter o assembly para **linguagem máquina**, passando o ficheiro hello-world.s para um hello-world.o. O ficheiro hello-world.o é um ficheiro que já não é legível, mas que possui as várias instruções do assembly, como podemos ver no Código 4.

����X � x \_\_text\_\_TEXT�

x�X�\_\_gcc\_except\_tab\_\_TEXT�

X 7\_\_eh\_frame\_\_TEXTp��" �\\_\_cstring\_\_TEXT|

h2

"H�&

�H��H�5�1�H��]��UH��H�� H�}�H�u�H�E�H�E�H�E�H�E�H�}��H�}�H�u�H���H�� ]ÐUH��H��H�}�H�u�H�}��U�H��]�UH��H��H�}�H�E�H�E�H�}�H�H�@�HǾ

�H�}����H�}��H�E�H��]�DUH��H��H��H�}�H�u�H�M�H�u�H�}���H�}�����M���E����7H�u�H�}��H�E�H�E�H�}�H�H�@�H�����M���E�%��� �H�E�HE�H�E�H�E�H�

(...) (--- truncado ---)

���

��W�W ��� �0���x ���@p���y

�

AeL�\_��\_\_ZSt9terminate (...) truncado (...) GCC\_except\_table4GCC\_except\_table44\_\_\_gxx\_personality\_v0

**Código 4** Objeto após assemblagem de hello-world.s, em hello-world.o, ilegível e truncado em tamanho.

O segundo processo é usar a interface (podemos pensar como um encaixe) para outras bibliotecas externas ou outros objetos que possam ser usados na nossa solução. Podemos pensar nestas bibliotecas externas como soluções externas para pequenos subconjuntos da nossa solução, as quais nos podem ser úteis por já serem normalmente usadas para colmatar um determinado problema (o que nos garante maior segurança na sua escolha) e para evitar reinventar a roda, assim como nos objetos (ficheiros .o) como outras soluções provenientes de outros códigos C++ que já possamos ter construído noutros projetos, mas que possam ser utilizados no atual. Passamos assim para uma fase de **linkagem**, em que vários objetos são ligados para formar o **executável** final, o hello-world.

�������!H\_\_PAGEZERO(\_\_TEXT@@\_\_text\_\_TEXT�0�

�0�\_\_stubs\_\_TEXTt=�t�\_\_stub\_helper\_\_TEXT�=��=�\_\_gcc\_except\_tab\_\_TEXT�>��>\_\_cstring\_\_TEXTL?L?\_\_unwind\_info\_\_TEXT\?�\?�\_\_DATA\_CONST@@@@\_\_got\_\_DATA\_CONST@(@�\_\_DATA�@�@\_\_la\_symbol\_ptr\_\_DATA���\_\_data\_\_DATA���H\_\_LINKEDIT�@�8"��`h�x����p��K�8

P.68�1

/usr/lib/dylK?���3[���ݦZ�A2

(...) (--- truncado ---)

@:

P:X

�:�

�:�

�:W

;�

�;<2P<g�<� =�P=��>��>�?,?$ ��� <-�@<N��3t��1��01��0^r�0x���]��

HYq����(>F678./09:;<=>1ABC3DEFGI?2@HJ678./09:;<=>1ABC3DEFGI

\_\_ZNSt3\_\_111char\_traits (...) truncado (...) GCC\_except\_table39GCC\_except\_table44\_\_dyld\_private

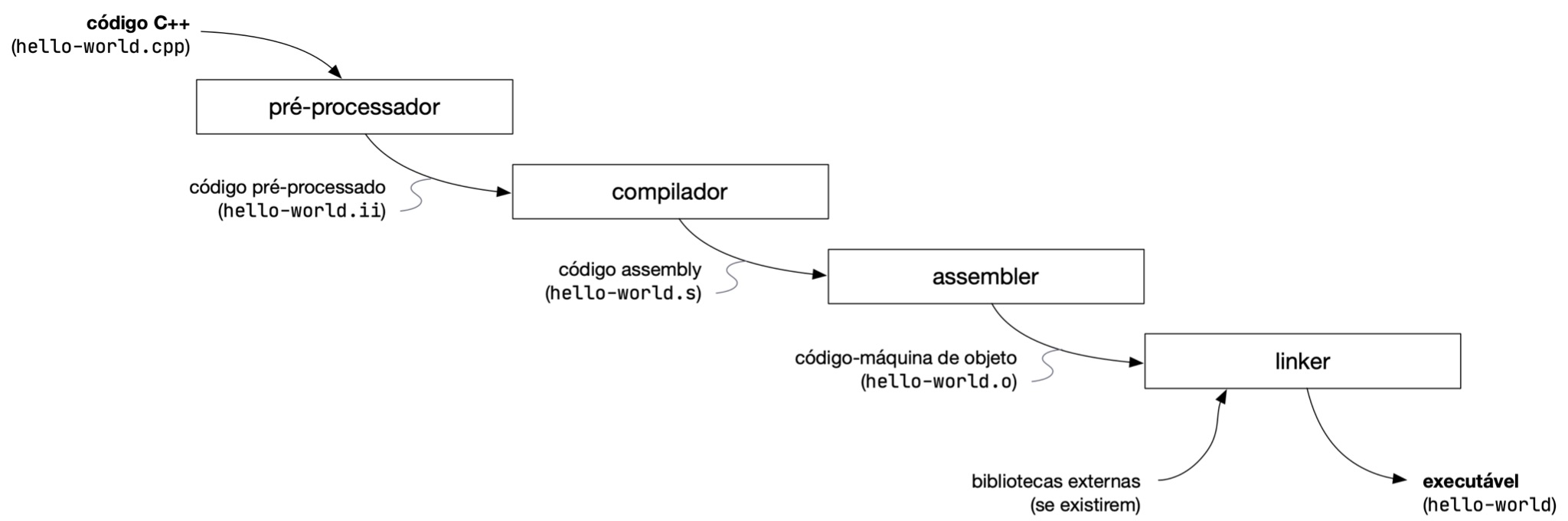
**Código 5** O ficheiro executável *hello-world*, ilegível para o ser humano.

Ainda que ilegível, o executável final possui todas as características necessárias para se efetuar a sua execução na máquina onde foi compilado: não é legível para nós humanos, mas está antes em linguagem máquina. Assim sendo, para a sua execução, executamos o seguinte comando, que nos dará o output esperado no terminal.

$ ./hello-world

Hello, World!

Em suma, desde o código inicial em C++ até ao executável final, o processo de compilação possui várias fases de processamento, entre as quais o pré-processamento, a compilação, a assemblagem e a *linkagem*.



## Geração de executáveis em contexto de projetos

Até à disciplina de Programação os ficheiros que foram sendo produzidos eram únicos e respeitantes a exercícios únicos, cada um destes, visto como um projeto no contexto do IDE que foi usado. Mas com o aumento da complexidade das nossas soluções, como será o caso dos programas feitos em Programação por Objetos e outros futuros, a noção de projeto irá agora incorporar vários ficheiros relacionados, entre os quais ficheiros fonte (vulgarmente com nome .cpp) e ficheiros de cabeçalho (com nome .h).

Embora o processo de compilação seja igual, a forma como damos início ao mesmo torna-se diferente, dado que agora teremos de considerar não só um único ficheiro de entrada para o processo, mas sim um conjunto de ficheiros. Para darmos vários ficheiros como *input* de um processo de compilação podemos usar ferramentas como as *Makefiles*[[6]](#footnote-5). As *Makefiles* são sequências de instruções que definem que partes de um projeto necessita de ser recompilado. Olhando para o processo de compilação, existem dois pontos onde código poderá ser entregue para a criação de um executável, como são o ponto de entrada principal, onde se entrega o código em C++, e o *linker*, onde podemos entregar também quaisquer bibliotecas externas que possamos estar a usar no nosso projeto. As *Makefiles* permitem-nos não só entregar estes itens no momento certo, como identificá-los, e automaticamente compilar somente as dependências que foram alteradas desde a última compilação efetuada anteriormente.

A composição de um ficheiro *Makefile* é feita através de alvos, que são blocos de código com nome e que podem ser selecionados para serem executados. Abaixo podemos ver um pequeno exemplo de um código que mostra quatro alvos: dois feitos por nós, e um chamado *all* que permite que um conjunto de outros alvos sejam listados para serem executados.

all: one two

one:

touch ficheiro1

two:

touch ficheiro2

clean:

rm ficheiro1 ficheiro2

**Código 6** Código exemplo de *Makefile*, meramente académico. O comando touch num sistema Linux cria um ficheiro com o nome que tem como argumento, e o comando rm apaga os ficheiros cujos nomes são entregues em argumento.

Para executar um ficheiro *Makefile* apenas temos de indicar ao comando make o alvo que pretendermos executar. Olhando para o código 6 e fazendo make one executamos somente o comando touch ficheiro1. Por outro lado, se fizermos make two executamos o comando touch ficheiro2. O alvo *all*, se for executado com make all executa o comando do alvo one e do alvo two, ou seja, executa o touch ficheiro1 seguido do touch ficheiro2. Por fim, fazendo make clean podemos executar o comando rm ficheiro1 ficheiro2, que acaba por apagar os ficheiros que os alvos one e two criam.

Contudo, ainda assim, fazer um ficheiro *Makefile* poderá ser bastante exaustivo. Para colmatar este problema, existe um leque de ferramentas que nos permitem gerar estes ficheiros. Na disciplina de Programação por objetos usaremos a ferramenta CMake.

O CMake[[7]](#footnote-6) é uma ferramenta que nos permite descrever de forma algo simples várias tarefas de um processo de compilação através de ficheiros de configuração. Estes ficheiros de configuração, denominados de CMakeLists.txt, são ficheiros descritores do que o nosso projeto consiste e, de forma semelhante à *Makefile*, como é que as várias partes interagem entre si, e em que momentos do processo são intervenientes na geração de um resultado final.

Para efeitos de demonstração, consideremos um projeto em que temos a seguinte árvore de ficheiros:

**project-tutorial**

↳ student.h

↳ main.cpp

↳ CMakeLists.txt

Considerando que o nosso ficheiro main.cpp possui o student.h como uma das suas inclusões, podemos descrever o ficheiro CMakeLists.txt da seguinte forma básica:

**cmake\_minimum\_required**(VERSION 3.13) # Versão mínima para executar este manifesto.

**project**(project-tutorial) # Nome do projeto CMake.

**set**(CMAKE\_CXX\_STANDARD 17) # Marcar como versão do C++ a usar, a C++17.

**add\_executable**(programa\_de\_teste main.cpp) # Criar executável programa\_de\_teste, a partir do main.cpp.

**Código 7** Código de ficheiro CMakeLists.txt para um projeto simples.

Das quatro linhas do código 7, a última instrução (a add\_executable) permite-nos identificar não só que pretendemos que o processo de compilação termine com a criação de um executável, como também que identifiquemos quais são os ficheiros que são necessários para a sua criação (quais são as suas várias dependências).

Olhando para a figura atrás onde se expõem os vários passos de um processo de compilação, este passo de integração de ficheiros para a geração de um executável é representado pela entrada de código-fonte em C++. Note-se que se porventura houver vários ficheiros \*.cpp necessários para a criação de um executável, então a linha seria escrita da seguinte forma, em que dentro dos ficheiros \*.cpp só um pode ter a descrição da função main() (só pode haver um ponto de entrada para a execução final).

add\_executable(nome\_do\_executavel a.cpp b.cpp … z.cpp)

O outro local, revisitando a figura, em que podemos integrar novos itens ao processo, é na junção de bibliotecas externas (se estas existirem). Para o fazer com o CMake podemos usar a função target\_link\_libraries(), sendo que como argumento teremos de indicar as bibliotecas devidamente identificadas pelo CMake.

target\_link\_libraries(nome\_do\_executavel biblioteca1 biblioteca2 … bibliotecaN)

Este documento é só uma pequena referência para uma documentação maior, que poderá consultar para melhor se informar sobre os vários usos do CMake e outras ferramentas semelhantes. Abaixo listam-se algumas destas referências:

1. Tutorial oficial do CMake: <https://cmake.org/cmake/help/latest/guide/tutorial/index.html>
2. Tutorial rápido do CMake no CLion: <https://www.jetbrains.com/help/clion/quick-cmake-tutorial.html>

1. https://www.infopedia.pt/dicionarios/lingua-portuguesa/sintaxe [↑](#footnote-ref-0)
2. https://www.infopedia.pt/dicionarios/lingua-portuguesa/semantica [↑](#footnote-ref-1)
3. https://stackoverflow.com/questions/17930267/what-is-the-difference-between-syntax-and-semantics-in-programming-languages [↑](#footnote-ref-2)
4. https://ubuntu.com/tutorials/command-line-for-beginners [↑](#footnote-ref-3)
5. https://man7.org/linux/man-pages/man1/g++.1.html [↑](#footnote-ref-4)
6. https://makefiletutorial.com [↑](#footnote-ref-5)
7. http://cmake.org [↑](#footnote-ref-6)