Referência Comparativa Rápida entre Python e C para Sistemas com Recursos Limitados

Wu Shin-Ting

30 de Abril de 2022

1 Introdução

Este documento é dirigido aos alunos que usaram Python na disciplina de Programação de micro- e mini-computadores e vão precisar se inteirar com a linguagem C para programar os micro-controladores em ambientes de desenvolvimento integrado (*Integrated Development Environment*) na disciplina de Programação Básica de Sistemas Digitais.

Python é uma linguagem **interpretada** de alto nível para propósito geral. Ele foi desenvolvido pelo Guido van Rossum em 1989 [1] com o objetivo de ter uma linguagem que apresenta uma **sintaxe intuitiva, similar à linguagem natural inglês,** sem precisar se preocupar com a tipagem e o armazenamento de dados na memória como a lingagem C. A **linguagem C** é uma linguagem **compilada** de médio nível, também para propósito geral. Foi inventada pelo Dennis Ritchie em *Bell Laboratories* entre 1972–73 [2] para programar sistemas operacionais que antes eram implementados com uma linguagem de baixo nível, o *assembly*. O sistema operacional Unix dos minicomputadores como DEC DPD7 foi integralmente implementado com linguagem C. A relação entre C e Python é maior do que imaginamos. Há muita equivalência entre as sintaxes das duas linguagens, a menos dos operadores relacionados com o uso da memória. Muitos módulos de Python são implementados com C ou C++. Por ser considerada uma camada que expõe a linguagem C/C++ de forma mais natural, muitos, como Carl Burch [3], acreditam que uma boa forma de introduzir um programador a C é começar com o Python.

Uma **linguagem compilada** é uma linguagem que requer que a máquina converte, por uma cadeia de ferramentas (*toolchain*), os códigos de um programa em códigos binários da máquina antes da sua execução. E uma **linguagem interpretada** é uma linguagem para a qual a máquina traduz, em tempo de execução, as suas instruções às referências das funções pré-implementadas (*built-in functions*) e aos valores dos seus argumentos. Por dispensar da interpretação dos códigos em funções pré-compiladas, o tempo de execução de um programa compilado é menor do que o tempo de execução de um programa interpretado. Um programa compilado apresenta um melhor **desempenho temporal**. Portanto, a linguagem compilada é ainda a preferida em aplicações relacionadas com o *hardware* quando o tempo é um fator crítico, como sistemas operacionais, *drivers* e *firmwares* [5]. Por outro lado, partindo da premissa de que todas as funções pré-implementadas estejam devidamente testadas, os erros dos programas interpretados se limitam aos erros detectados durante a interpretação das suas instruções no momento da execução.

Figura 1 mostra as ferramentas envolvidas na conversão de códigos em linguagem C armazenados em arquivos de extensão *.c* num arquivo executável de extensão *.elf*: **pré-processador** para traduzir as **diretivas** de C em arquivos de extensão *.i* contendo somente instruções puras de C; **compilador**

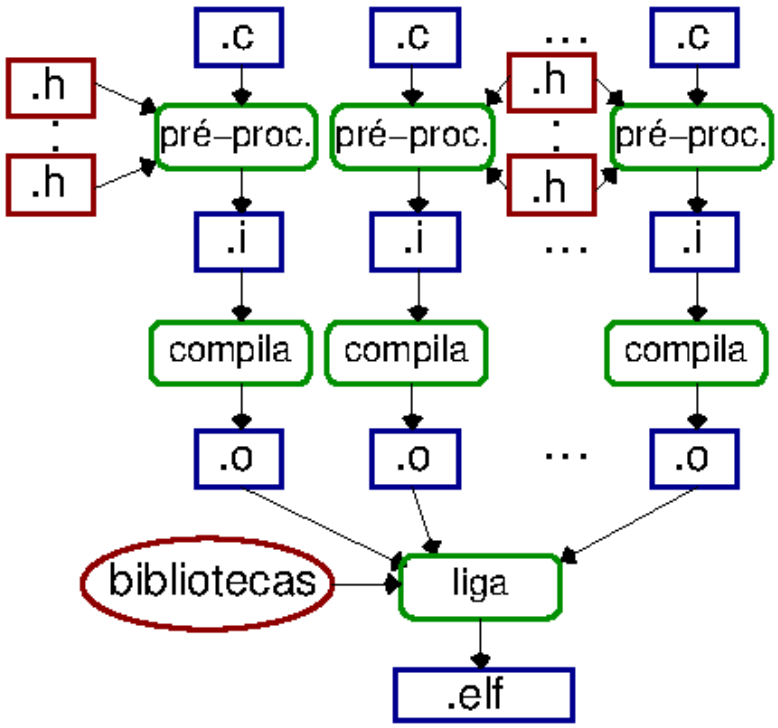

Figura 1: Cadeia de ferramentas para construção de um executável em C.

para traduzir as instruções puras em C em arquivos-objeto de extensão *.o* contendo códigos de máquina do processador-alvo, e **ligador** para juntar as instruções de diferentes arquivos e construir um arquivo executável de extensão *.elf*. Além dos erros durante a execução do programa, podem ocorrer erros em cada estágio de um *toolchain*. O **diagnóstico dos erros** em cada estágio nem sempre é uma tarefa simples. Assim, a linguagem interpretada tem sido a preferida para prototipagem e provas de conceito no desenvolvimento de um projeto.

A grande diferença entre C e Python está no **gerenciamento de memória**. Em ambas as linguagens, as unidades de armazenamento de um espaço de memória, em *bytes*, são abstraídas em endereços e conteúdos/valores dessas unidades. A linguagem C dispõe de uma série de recursos para o programador gerenciar o uso da memória **de forma dedicada**, enquanto Python abstrai essas unidades de armazenamento em objetos e dispõe funções pré-implementadas que alocam e desalocam tais objetos, de forma transparente para programadores. O fato do Python assumir a função de **coleta de lixo** (*garbage collector*) de unidades de memória simplifica a implementação de qualquer algoritmo, principalmente aquele que envolve relações mais complexas de dados. Por outro lado, visando a atender aplicações de **múltiplos propósitos**, as soluções consideradas otimizadas pelos seus desenvolvedors nem sempre são as melhores para uma tarefa específica. E o Python não oferece recursos adicionais para implementar alternativas, como alocar 1, ao invés de 4 *bytes*, para representar valores inteiros entre 0 a 255. Isso pode ser um problema para **dispositivos com recursos escassos** como microcontroladores.

Nesta nota de aula são apresentados alguns conceitos relevantes relacionados com o desenvolvimento dos projetos da disciplina EA871. Grande parte do material é baseado no documento de Carl Burch [3] e do *link* [4]. O material didático disponível pelo Projeto MAC Multimídia [6] contém a

implementação de uma série de algoritmos em C e em Python, que pode proporcionar uma boa visão comparativa entre as duas linguagens. Recomendo ainda a referência rápida à linguagem C para sistemas embarcados disponível em [7] para consulta rápida sobre linguagem C ao longo do curso.

2 Estrutura Básica

Tanto em Python quanto em C os programas são definidos por uma sequência de **linhas de instrução** estruturadas em diferentes níveis de **blocos de instruções**. Um bloco de instruções, que realiza uma tarefa específica e é identificado por um nome único e um conjunto de argumentos através dos quais transferimos os valores das suas variáveis, é denominado uma **função**.

Em Python, as linhas de instrução só contém os comandos de execução. Essas instruções são separadas por linhas e os blocos de instrução são diferenciados por espaços brancos, usualmente 4 espaços ou um *tab* em relação ao bloco de nível imediatamente superior. A definição de uma função é marcada com a palavra reservada *def* seguida do nome da função, da lista de argumentos entre os parênteses, e de dois pontos (":"), como demonstra o seguinte código-fonte em Python para cômputo do fatorial de um número inteiro não-negativo. Nesse módulo/arquivo, a definição da função *fatorial* é iniciada a partir da linha "def fatorial():" e vai até a linha que retorna ao nível de indentação da linha que contém a palavra reservada *def*. **A definição de uma função deve preceder sempre o seu uso**. Nesse exemplo, o uso acontece na última linha de instrução "fatorial ()" do programa quando se inicia a execução do programa [8] com a linha de comando em *prompt de comando, cmd*

```
C:\users\ea871> python fatorial.py
```

O interpretador desdobra a última linha na sequência de instruções da função *fatorial*.

```
#---fatorial.py
def fatorial():
    n = int(input("Digite o valor de n: "))
    fatorial = 1
    contador = 2
    while contador <= n:
        fatorial = fatorial*contador
        contador = contador + 1

    print("O valor de %d! eh =" %n, fat)

#-----
fatorial()
```

Em C, as linhas de instrução podem conter diretivas do pré-processador ou os comandos de execução. As linhas de instrução devem ser separadas por ";" e os blocos de instrução são delimitados pelo par de chaves "{" e "}". A definição de uma função é marcada por uma linha contendo tipo de dado de retorno, seguido do nome da função e da lista de argumentos entre os parênteses. A versão em C do algoritmo de cômputo do fatorial de um número inteiro não-negativo é apresentada a seguir. A definição da função *fatorial* é iniciada a partir da linha "int fatorial() {" e vai até "}" correspondente. Comparando com a versão em Python, temos uma função a mais no programa, a função

main que retorna um dado do tipo inteiro (*int*). Essa é uma função especial **presente em todos os programas em C**, pois ela serve como o ponto de entrada da execução de um programa. Temos ainda dois blocos adicionais de instruções no cabeçalho: um **bloco de diretivas** de pré-processamento e um **bloco de protótipos de funções** definidas no programa. Para o estágio de compilação de um programa em C, **é necessário que todas as variáveis e funções sejam declaradas antes do uso**. Por exemplo, dentro da função *fatorial*, as variáveis *n*, *contador* e *fatorial* são definidas como do tipo inteiro (*int*) antes do uso nas instruções que se seguem. Diferente de Python, não é necessário **definir** as funções antes do seu uso em C; basta **declará-las** como demonstra o código abaixo. Nesse código, a função *fatorial* é declarada no bloco de protótipo das funções antes do seu uso na função *main* e definida depois da definição da função *main*.

```
/*
 * fatorial.c
 */

/* Bloco de diretivas de pre-processamento */
#include <stdio.h>

/* Bloco de prototipos das funcoes */
int fatorial();

/* Definicao da funcao main */
int main (){
  fatorial();

  return 0;
}

/* Definicao da funcao fatorial */
int fatorial() {
  int n,        /* guarda o numero dado */
      contador,
      fatorial;

  printf("\n\tCalculo do fatorial de um numero\n");
  printf("\nDigite um inteiro nao-negativo: ");
  scanf("%d", &n);

  /* inicializacoes */
  fatorial = 1;
  contador = 2;

  while (contador <= n) {
    fatorial = fatorial * contador;
    contador = contador + 1;
  }
```

```
  printf("O valor de %d!: %d\n", n, fatorial);

  return 0;
}
```

Para executar o programa, é necessário *pré-processar* e *compilar* o código-fonte e *ligá*-lo com as funções declaradas no arquivo *stdio.h* da biblioteca-padrão de C. Inserindo a linha de comando de *GNU Compiler Collection, gcc,* no *prompt de comando, cmd,* em *Windows* [9][1]

```
C:\users\ea871> gcc -save-temps fatorial.c -o a.out
```

podemos ver todos os arquivos intermediários, *fatorial.i*, *fatorial.s* e *fatorial.o*, gerados entre o código-fonte *fatorial.c* e o código executável *a.out*. A primeira instrução a ser executada ao entrarmos com o código executável na linha de comando

```
C:\users\ea871> a.out
```

é a primeira linha de instrução da função *main*, que é a chamada da função *fatorial*. Essa chamada é desdobrada na sequência de instruções definida dentro de *fatorial ()*. Note que, se não quisermos a geração dos arquivos temporários, basta omitirmos a opção *-save-temps* na linha de comando

```
C:\users\ea871> gcc fatorial.c -o a.out
```

3 Linhas de Instrução

Nesta seção são apresentados comparativamente os elementos construtores das linhas de instruções em C e Python. Como Python foi desenvolvido com base em C, **há muita similaridade entre os comandos de execução dessas duas linguagens**.

3.1 Diretivas

As **diretivas** são comandos em códigos-fonte de linguagem compilada, como C. Elas dizem ao pré-processador as ações específicas a serem tomadas para construir um programa em C "puro", que usualmente é mais longo e mais difícil de entender. Através das diretivas, o pré-processador pode inserir o conteúdo de outros arquivos (*# include*), substituir expressões por tokens, denominados **macros** (*# define*), remover/adicionar trechos dos códigos (*# ifdef*, *# ifndef*), ou fazer inserção seletiva de trechos de códigos (*if ... elif ... else*). Para se distinguirem dos comandos de execução, todas diretivas em C são precedidas de "#". Vale chamar atenção que é o mesmo símbolo adotado pelo Python para indicar as linhas simples de comentários (Seção 3.4). Por ser uma linguagem interpretada, **não há diretivas no Python**.

Na fase de compilação, o compilador só verifica a sintaxe da chamada das funções (nome, quantidade e tipo de dados dos argumentos da função). Portanto, é muito comum agrupar em C o bloco de *protótipos*, mostrado no arquivo *fatorial.c* da Seção 2, num arquivo denominado **arquivo de cabeçalho** ou **arquivo-cabeçalho** de extensão ".h" (*header file*) e incluí-lo no arquivo de extensão

[1] No apêndice A há um roteiro de instalação de um ambiente de desenvolvimento minimalista em *Windows*.

".c", denominado código-fonte, usando a diretiva *# include*. Ao gerar o executável a partir do código-fonte com a opção *-save-temps* (Seção 2), podemos ver no arquivo de extensão ".i" o código-fonte extendido com os blocos de instrução dos arquivos incluídos.

Quando se fala em reuso, concisão, legibilidade e manutenção, passa a ser significativa a prática de dividir os códigos em

definições das funções em arquivos de extensão ".c", e

declarações ou protótipos das funções em arquivos de extensão ".h".

A inclusão de um arquivo-cabeçalho contendo somente as declarações das funções é suficiente para um compilador conferir se as chamadas às funções externas ao arquivo-fonte em compilação são feitas corretamente, mesmo que tais funções externas não tenham sido compiladas. Os arquivos-cabeçalho servem como uma *interface* às funções definidas nos arquivos de extensão ".c" correspondentes. Essas definições só são de fato necessárias na etapa de ligação.

Porém, inclusões indiscriminadas dos arquivos-cabeçalho em arquivos-fonte podem levar a múltiplas definições de uma mesma função e gerar conflitos em definições durante a compilação. Para proteger inclusões múltiplas de um mesmo arquivo-cabeçalho, são adicionadas algumas diretivas (do pré-processador) no arquivo-cabeçalho para que o pré-processador só inclua uma única vez um mesmo arquivo-cabeçalho na expansão do arquivo-fonte. Essas diretivas adicionais são conhecidas por *include guard*, *macro guard* ou *header guard*. Por exemplo, no arquivo-cabeçalo *fatorial.h* podemos proteger a inclusão múltipla da definição da função *int fatorial ()* na seguinte forma

```
#ifndef FATORIAL_H
#define FATORIAL_H

int fatorial();

#endif /* FATORIAL_H */
```

3.2 Operadores

Os **operadores** são símbolos especiais reservados numa linguagem de programação para especificar as diferentes operações sobre as variáveis. Podemos construir **expressões** a partir deles. São classificados em operadores aritméticos (exponenciação, multiplicação, divisão, adição e subtração), deslocamentos binários (para direita, para esquerda), relacionais (menor, menor ou igual, maior, maior ou igual), *bit*-a-*bit* ($\vee$, $\wedge$, $\neg$, $\oplus$), lógicos (OR, AND, NOT), e de atribuição (=). Na Tab. 1 são listados os operadores em Python e em C, de precedência mais alta para a mais baixa.

Note que **em C**

1. a prioridade do operador de negação (!) é mais alta do que em Python (*not*).
2. não temos o operador de exponenciação. Usa-se no lugar uma função pré-implementada *pow* da biblioteca padrão de C.
3. temos dois operadores de incremento (++) e decremento (- -) pós-fixos, que equivalem a, respectivamente, $(a++) \leftrightarrow a = a + 1$ e $(a--) \leftrightarrow a = a - 1$ em Python.

Tabela 1: Operadores em Python e C na ordem de maior para menor precedência

Operador	Python	C
Parênteses	()	()
In/decremento (pós-fixo)		++ - -
Exponenciação	**	
Lógicos unários		!
Aritméticos unários	+ -	+ -
Multiplicação	*	
Divisão	/ % //	/ %
Adição binária	+	+
Subtração binária	-	-
Deslocamentos	<< >>	<< >>
Relacionais	< <= > >=	< <= > >=
Igualdades	== !=	== !=
bit-a-*bit*	& | ^ ~	& | ^ ~
Lógicos	not and or	&& || !
Atribuição		=

4. a notação contraída de "=" com um operador binário *op* é utilizada. Uma expressão "a *op* = 5" equivale a "a = a *op* 5". Python também adota essa convenção.

5. "=" é um operador de atribuição, mas em Python, ele é interpretado como um comando de execuação. Portanto, em Python ele não pode ser incluído numa **expressão**, tal como "((a = b + c) & d)" em C.

6. não existe o operador de "divisão de inteiros" do Python (//) em que o resultado seja a parte inteira do quociente. O resultado da divisão pelo operador / é a parte inteira do quociente se os operandos forem números inteiros. Se um dos operandos for um valor decimal, todos os outros operandos são **convertidos implicitamente** para valores decimais e o operador é promovido automaticamente para divisão de valores decimais. Figura 2 ilustra a conversão implícita dos dados para que a Unidade Lógica-Aritmética (ULA) do processador possa tratar de forma uniforme os operandos de um operador (comando de execução). Na seção 4.1 são explicados os tipos de dados mostrados na Figura 2.

3.3 Comandos de Execução

Comandos de execução (*statements*) são blocos de instruções correspondentes a uma ação completa que um processador executa. Podemos distinguir os comandos de execução em: comandos simples, comandos compostos e comandos de controle (de fluxo). Um **comando simples** é uma linha de instrução contendo uma instrução completa, como calcular uma expressão e armazenar o resultado na memória. Um **comando composto** é um bloco de instruções. Como vimos na seção 2, em C ele é delimitado pelas chaves e em Python um bloco é um conjunto de linhas de instrução que tem o mesmo nível de indentação. Um **comando de controle** é um comando ou um bloco de comandos capaz de alterar o fluxo de controle do programa. Tab. 2 apresenta os comandos de controle mais encontrados para programar **estruturas de desvio, de repetição e de escolha**. Note que **em C**

Tabela 2: Comandos de Controle em Python e C

Comandos	Python	C
"Bloco vazio"	*pass*	{} ou *;*
Retorno ao fluxo anterior	*return* expressão	*return* expressão *;*
Desvio do fluxo para o fim do bloco do comando	*break*	*break;*
Desvio para próxima iteração instruções da iteração	*continue*	*continue;*
Escolha simples	*if* condição *:* bloco de instruções *else :* bloco de instruções	*if* (condição) { bloco de instruções } *else* { bloco de instruções }
Escolha múltipla if	*if* condição *:* bloco de instruções *elif :* bloco de instruções *else :* bloco de instruções	*if* (condição) { bloco de instruções } *else if* { bloco de instruções } *else* { bloco de instruções }
Escolha múltipla match/switch	*match* variável *:* *case* valor1 *:* bloco de instruções *case* valor2 *:* bloco de instruções *default:* bloco de instruções	*switch* (variável) { *case* valor1 *:* bloco de instruções *break;* *case* valor2 *:* bloco de instruções *break;* *default:* bloco de instruções }
Repetição For (exemplos na Seção 4.1)	*for* condição *:* bloco de instruções	*for* (condição) { bloco de instruções }
Repetição While (exemplos na Seção 2)	*while* condição *:* bloco de instruções	*while* (condição) { bloco de instruções }
Inserção de códigos em *assembly*		*asm* { mnemônicos e interface }

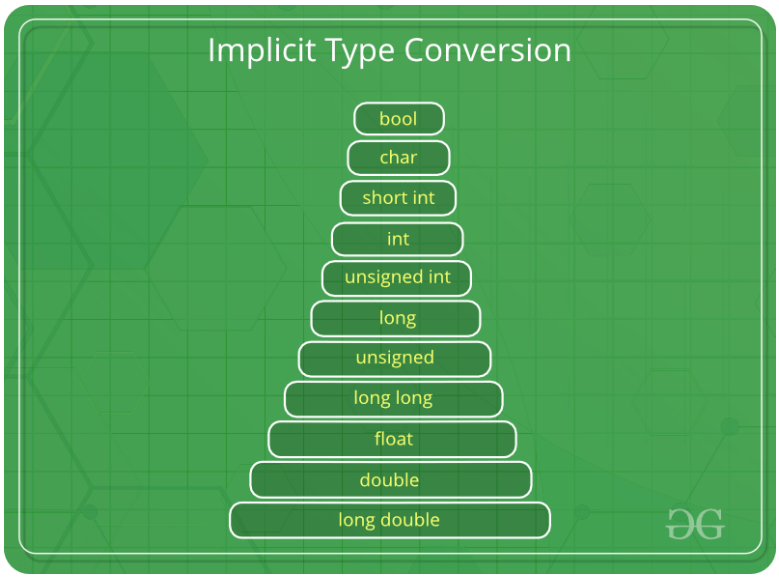

Figura 2: Conversão implícita de tipos de dados em C [10].

1. não há a palavra reservada ***elif*** para a implementação de escolhas múltiplas como em Python, porque o uso de chaves em C evita ambiguidades entre múltiplas escolhas

```
if (condição A) {
   bloco de instruções
} else if (condição B) {
   bloco de instruções
} else if (condição C) {
   bloco de instruções
} else {
   bloco de instruções
}
```

   e múltiplas alternativas simples

```
if (condição A) {
   bloco de instruções
} else {
   if (condição B) {
       bloco de instruções
   } else {
       if (condição C) {
          bloco de instruções
       } else {
          bloco de instruções
       }
   }
}
```

2. o comando de controle *for* percorre qualquer sequência de valores cujo valor inicial, valor final e passo são definidos no próprio comando. À medida que se avança no passo, o bloco de instruções dentro das chaves é executado repetidamente

   ```
   for (condição_inicial_do_iterador; condição_final_do_iterador; passo_do_iterador) {
    bloco de instruções
   }
   ```

   Em Python, o comando *for* é também um comando de repetição, mas ele percorre somente os elementos de um tipo de sequência, mais especificamente os tipos *string*, *list* e *tuple* (Seção 4.1)

   ```
   for iterador in sequence:
    bloco de instruções
   ```

   Os programas *string.py* e *string.c* apresentados na seção 4.1 ilustram o uso de *for* em Python e C, respectivamente.

3. não há uma palavra reservada *pass* como em Python. Porém, há várias formas para implementá-la usando as palavras-chave *for* ou *while* em C. Por exemplo, para o comando

   ```
   while True:
     pass
   ```

   temos as seguintes versões equivalentes em C:

   ```
   while (1) {}

   while (1);

   for (;;);

   for (;;) {}
   ```

   Esse comando é muito usado na implementação de um fluxo que fica aguardando por algum evento externo gerado pelos periféricos.

4. as opções (*case*) e a variável no comando de escolha *switch* e *match* devem ser do mesmo tipo de dados (Seção 4.1). A alternativa *default* é para tratar todos os casos não discriminados. Em C, o uso do comando *break* em cada alternativa é essencial para que o fluxo de controle seja desviado para o fim do bloco do comando quando finalizar a execução das instruções de um *case*; senão, serão executadas as instruções do próximo *case*. Em Python, é executado somente o bloco de instruções associado a cada *case*. O comando *match* foi introduzido na versão 3.10 do Python.

5. para microcontroladores é reservada a palavra ***asm*** para **inclusão direta de códigos em *assembly* no seu código-fonte** [7]. Diferente do Python que procura abstrair ao máximo o *hardware* sobre o qual um programa executa, C permite acesso direto ao *hardware*. Isso pode melhorar o tempo de resposta do sistema como também facilitar o desenvolvimento de *drivers* dos periféricos.

3.4 Comentários

Comentários são informações que adicionamos aos comandos da linguagem de programação para facilitar a leitura dos nossos códigos por outros programadores. Os comentários são ignorados pelos compiladores (C) e pelos interpretadores (Python). Tanto C quanto Python suportam comentários em linhas simples como em múltiplas linhas. Para demarcar uma linha única como texto de comentário, usa-se cerquinha ("#") em Python e barra dupla ("//") no início da linha. E quando o texto de comentário ocupa mais de uma linha, delimitamos o bloco de comentários por barra e asterisco ("/*") e asterisco e barra ("*/") em C. E em Python, por 3 aspas simples, ou então, 3 aspas duplas.

4 Variáveis

As **variáveis** são nomes/rótulos atribuídos a posições do espaço de memória alocadas para armazenar os dados. Em C e Python, o nome de uma variável só pode ser uma sequência de letras, dígitos e "_" iniciada com uma letra e não pode ser uma palavra reservada. Em Python, a definição de uma variável ocorre na interpretação do comando = quando é automaticamente instanciado um novo objeto de uma classe de **tipo de dado inferida pelo interpretador através do valor do dado**, como ilustra a definição automática das variáveis *n*, *fatorial* e *contador* como do tipo *int* no código *fatorial.py* na Seção 2. Em C, é necessário **definir explicitamente o tipo de dado** de uma variável, como mostra a linha de instrução "int n, contador, fatorial;" antes do seu uso no código *fatorial.c* na Seção 2. Portanto, dizemos que Python tem **tipagem dinâmica**, enquanto C tem **tipagem estática**.

4.1 Tipos de Dados

Os **tipos de dados** definem a forma de representação das variáveis e as operações válidas sobre elas. Em C, são definidos 4 tipos de dados básicos, 4 tipos de dados derivados a partir dos tipos básicos, 1 tipo de dado *enum* (formado por uma lista de constantes inteiros nomeados), e 1 tipo de dado *void* (sem valor e sem operação associada) (Figura 3). Através dos **tipos de dados básicos**, o programador especifica o tamanho do espaço de memória alocados para uma variável. Os tamanhos variam com o processador e com o compilador. Usualmente, para um compilador de 32 *bits* são alocados, respectivamente, 1 *byte*, 4 *bytes*, 4 *bytes* e 8 *bytes* para os tipos *char*, *int*, *float* e *double*. Na maioria dos programas o tipo *char* é usado na representação de códigos ASCII [12]. Operações inteiras são reservadas para os tipos *char* e *int*, enquanto as operações em ponto flutuante são aplicáveis aos tipos *float* e *double*. Com os quatro **tipos derivados**, um programador em C consegue organizar de forma rudimentar os dados de tipos básicos em listas/vetores/arranjos de elementos de um mesmo tipo básico ou derivado: *struct* (composição de elementos de tipos diferentes), *union* (um tipo dentre os tipos alternativos), e ponteiro (endereço de um tipo de dado).

Além das palavras reservadas para a definição dos tipos básicos, são disponíveis em C um conjunto de **qualificadores**, ou **modificadores**, de tipos que alteram, em relação aos tipos básicos, o tamanho, o sinal (Figura 4) e a forma (tipo) de modificação do conteúdo das variáveis definidas. Entre os **qualificadores de tamanho**, temos as palavras reservadas *short* (usualmente, metade do tamanho básico), *long* (dobro do tamanho básico) e *long long* (quádruplo do tamanho básico). Os **qualificadores de sinal** são *signed* (com sinal) e *unsigned* (sem sinal). O **qualificador de tipos** define se o valor da variável é constante, ou seja não modificável (*const*) ou se o valor não é modificável pelas instruções programadas (*volatile*). Esses qualificadores permitem otimizar a ocupação da memória conforme as demandas particulares de cada aplicação.

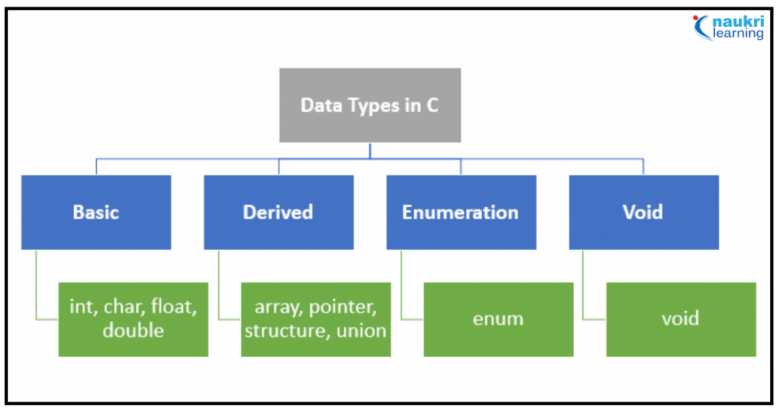

Figura 3: Tipos de dados em C [13].

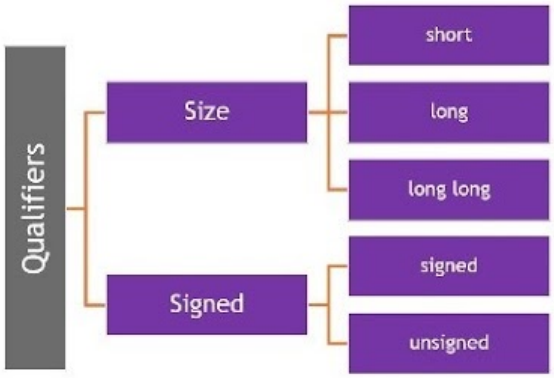

Figura 4: Qualificadores dos tipos de dados em C [14].

Para completar, em C há uma palavra-chave ***typedef*** que permite ao programador definir um novo nome para um tipo de dado específico. Na programação de microcontroladores, a quantidade de *bits* alocados às variáveis pode ser um fator relevante na implementação de um programa. Muitos desenvolvedores preferem usar nomes de tipos de dados que explicitam o tamanho e o sinal da variável, como *int8_t* e *uint16_t*, definidos no arquivo-cabeçalho *stdint.h* [15]. Os seguintes comandos extraídos do arquivo ilustram o uso de *typedef* para criar os novos nomes. Para usar esses nomes no programa, basta incluir o arquivo-cabeçalho *stdint.h* no código-fonte.

```
typedef signed char int8_t;
typedef short int16_t;
typedef long int32_t;
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
```

Python, por sua vez, abstrai os dados em objetos de classes de tipos de dados que facilitam a organização de dados com relações não necessariamente lineares. Dos 5 tipos de dados (Figura 5),

somente os tipos de dados numéricos (*numeric*) e booleano (*bool*) se aproximam dos tipos de dados básicos em C. O tipo de dado booleano só assume um dos dois valores: *False* (0) e *True* ($\neq 0$). E os tipos de dados numéricos compreendem *int*, *float* e *complex*. Pelos valores atribuídos pelo programador, o interpretador aloca para o tipo de dado inferido o tamanho de espaço de memória pré-estabelecido. O programador não tem nenhum controle sobre adequações dessa alocação às demandas de uma atividade específica. Outros 3 grupos de tipos de dados, **tipos de sequência** (*str*, *list* e *tuple*), **conjunto** (*set*)

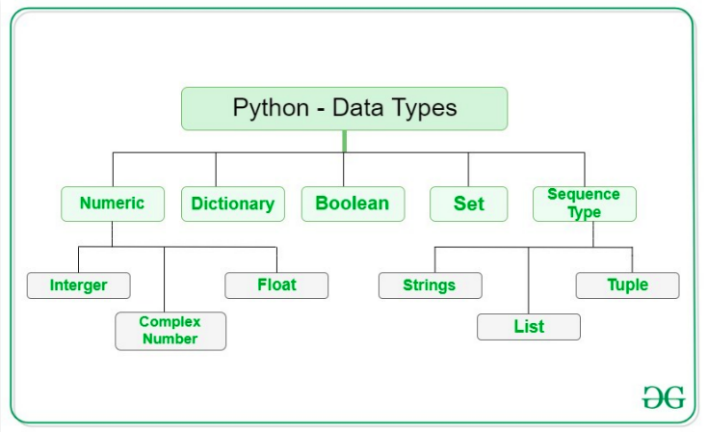

Figura 5: Tipos de dados em Python [16].

e **dicionário** (*dict*) são de fato **tipos de estruturas de dados** providos de funções que organizam, gerenciam e armazenam, de forma transparente, os dados que apresentam relações mais complexas.

O alto nível de abstração dos dados em diferentes estruturas torna Python uma linguagem atraente para múltiplos propósitos. Por outro lado, Python não oferece, como C, recursos que permitem um programador experiente fazer seu próprio gerenciamento de memória e desenvolver códigos dedicados e eficientes no uso da memória. Isso pode comprometer o desempenho de um sistema de recursos limitados [17]. Por exemplo, o tipo de sequência *str* (*String*) em Python é comparável com os **vetores de elementos do tipo *char*, com o valor 0 (NULL) como o último elemento,** junto com as funções de processamento de *string* disponíveis na biblioteca-padrão C. Porém, uma instância da classe do tipo de dado *str* é imutável em Python, enquanto temos acesso a cada caractere de uma *string* (um vetor de elementos do tipo *char*) em C através dos seus ponteiros (endereços). Isso pode resultar em implementações equivalentes, mas com grande diferença no desempenho.

Os dois códigos seguintes demonstram acessos aos elementos da variável *my_string* em Python e em C. Duas tarefas são executadas: (1) substituição da letra "r" pela letra "R" na variável, e (2) contagem da quanitade de "t" na variável. Em Python, a única forma para substituir a letra "r" foi remover o objeto *my_string* e criar um novo objeto *my_string* com o texto alterado. É uma alteração que envolve desalocação e alocação de espaços de memória. Quanto à contagem da letra "t", usamos a função pré-implementada *in* em Python de propósito geral para percorrer a sequência. Mesmo sendo uma versão otimizada sob o ponto de vista do seu desenvolvedor, não podemos afirmar se é a melhor solução para uma simples varredura de caracteres de uma *string*.

```
#---string.py
def string():
    my_string = "String em Python"
```

```
    print ("STRING1: " + my_string);

#--- Atribuicao nao permitida
#    my_string[2] = 'R'
    del my_string
    my_string = "StRing em Python"

    print ("STRING2: " + my_string);

    count = 0
    for letra in my_string:
        if (letra == 't'):
            count += 1

    print(str(count) + " letras t encontradas.")

#-----
string()
```

Em C, podemos definir uma variável ponteiro *tmp* para armazenar o endereço das letras da variável *my_string*, aplicar as operações aritméticas para computar os endereços absolutos de cada letra da variável a partir do seu endereço inicial *&my_string[0]*, e fazer acessos a qualquer um desses endereços. Por termos acesso aos endereços das variáveis, é possível implementar um procedimento mais eficiente para a troca de uma letra. Neste caso específico, basta fazer um acesso de escrita ao terceiro elemento da variável, cujo endereço relativo ao endereço inicial *&my_string[0]* é *&my_string[0]+2*, no lugar da desalocação e alocação de um espaço da memória implementada na versão em Python. Como não há uma função pré-implementada para busca de uma letra dentro de uma *string*, é implementada a varredura do endereço inicial até o último endereço da variável para testar, letra por letra, com "t" e fazer a contagem. Note o uso do operador unário "*" para acessar o conteúdo do ponteiro *tmp*. Embora mais trabalhosa a implementação dessa versão em C, temos uma noção melhor da quantidade de instruções executadas e do uso da memória.

```
/*
 * string.c
 */

/* Bloco de diretivas de pre-processamento */
#include <stdio.h>

/* Bloco de prototipos das funcoes */
int string();

/* Funcao main */
int main (){
  string();
```

```
  return 0;
}

/* Funcao string */
int string() {
  char my_string[]="String em C";
  char *tmp;
  int i=0, counter=0;

  printf("\tSTRIG1:  %s\n", my_string);
  tmp = &my_string[0]+2;       /* Computar endereco da 3a. letra */
  *tmp = 'R';                  /* Atribuir letra R */
  printf("\tSTRIG2:  %s\n", my_string);

  tmp = &my_string[0];         /* Atribuir endereco da 1a.letra */
  while (*tmp != '\0') {
     if (*tmp == 't')          /* Comparar o conteudo do endereco com t */
        counter += 1;
     tmp++;                    /* Avancar para prx letra */
  }

  printf ("\t%d letras t encontradas\n", counter);

  return 0;
}
```

4.2 Escopo das Variáveis

O **escopo de uma variável** indica a sua acessibilidade dentro de um bloco de instruções. Em Python distinguem-se quatro escopos que seguem a prioridade de abrangência local (*Local*), aninhado (*Enclosing*), global (*Global*) e pré-definido (*Buit-in*) (Figura 6). O escopo **local** diz respeito ao bloco de instruções em que uma variável é definida. O escopo **aninhado** assegura a acessibilidade de uma variável definada num bloco de instruções para todos os blocos contidos nele. O escopo **global** refere ao acesso de uma variável por todo um módulo/arquivo em que ela é definida. E o escopo *Buit-in* é reservado para as palavras-chave do Python. Essas palavras tem a mesma interpretação em qualquer módulo programado em Python. Sendo Python uma linguagem de programação orientada a objetos, as variáveis de um módulo são objetos do módulo e **não há compartilhamento de um espaço de memória entre os módulos**.

Em C, distinguem-se também as variáveis locais e globais. As **variáveis locais** são as definidas dentro de uma função. Elas são acessíveis somente dentro da função. Porém, diferentes das variáveis locais do Python que são objetos de um módulo, as variáveis locais de C são automaticamente alocadas quando o fluxo de execução entra na função e desalocadas quando o fluxo deixa a função. O conceito de acessibilidade em C está relacionado também com o local de armazenamento das variáveis. Portanto, as palavras reservadas para a localidade são usualmente denominadas **classes de armazenamento**. Por padrão, as variáveis locais são da **classe *auto***. Há a opção de manter o endereço alocado a uma variável local até o final da execução de um programa. Basta adicionar ao tipo de dado da variável a

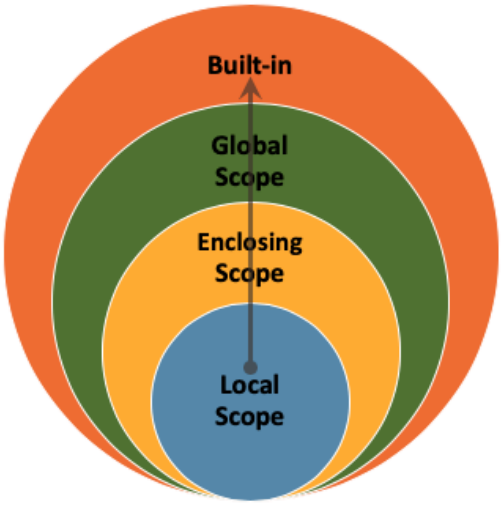

Figura 6: Ordem de abrangência do escopo de uma variável definida em Python: LEGB (extraído de [11]).

palavra reservada ***static***, como a variável *b* definida na função *myfunc()* no exemplo que se segue. É ainda possível especificar que uma variável local seja preferencialmente armazenada nos registradores do processador. Para isso, usa-se a palavra-chave ***register***. Em C, as variáveis definidas fora das funções são consideradas as **variáveis globais**, como a variável *a* no arquivo *main.c* apresentado no exemplo abaixo. Essa variável é desalocada somente no final da execução de um programa e pode ser compartilhada com outros arquivos desde que seja adicionada a palavra reservada da **classe de armazenamento *extern*** na declaração de *a* nos outros arquivos, como ilustra o arquivo *myfunc.c* do exemplo apresentado. Enfim, C dispõe de muitos recursos para um programador especificar o local de armazenamento das variáveis programadas. Como os tempos de resposta das memórias variam, esse controle direto pode impactar positivamente no desempenho do sistema [18].

```
main.c:
int a;
main() {
    a = 20;
    printf("main.a = %d\n", a);
    myfunc();
}

myfunc.c:
extern int a;
myfunc() {
    static char b='A';

    b += 1;
    printf("myfunc.a = %d\n", a);
}
```

4.3 Passagem de Parâmetros

Quando estruturamos os blocos de linhas de instruções em funções, podemos especificar tanto em Python quanto em C uma lista de parâmetros de passagem para a função chamada. Há duas formas de passagem de parâmetros, por valor e por referência. Dizemos que uma passagem é **por valor** se é passada para a função uma cópia dos valores, de forma que alterações nos valores dentro da função não sejam refletidas nos valores fora dela. Uma passagem é **por referência** quando são passadas para a função as referências das variáveis, tal que alterações nos valores dessas referências afetam os valores das variáveis que foram passadas para a função. Em C só se faz uma chamada por valor, enquanto em Python só por referência. Há, no entanto, um recurso em C que permite uma chamada cujo efeito seja similar a uma chamada por referência: passar como valor de um argumento o endereço de uma variável, ou seja um ponteiro (Seção 4.1). O operador unário "&" é usado para diferenciar o endereço de uma variável do valor dessa variável, e o operador unário "*" para diferenciar o conteúdo de um endereço do endereço. Ou seja, colocamos "&" na frente da variável que é passada para a função, se quisermos que as alteraçoes feitas nela dentro da função sejam retornadas. E, dentro da função, precisamos colocar "*" na frente do ponteiro passado para acessar o seu conteúdo.

As duas funções abaixo ilustram o uso de ponteiros para passar os parâmetros da função padrão da biblioteca C, *scanf*, e da função *soma* implementada. Através do segundo argumento de *scanf*, é retornado à função *main* o valor (inteiro) entrado pelo usuário, enquanto o terceiro argumento de *soma* retorna o resultado da soma dos dois operandos obtida na função. Observe o uso do operador unário "*" dentro da função *soma* para acessar o conteúdo de um ponteiro declarado como do tipo int (*int **).

```
#include <stdio.h>

void soma (int a, int b, int *c) {
    *c = a + b;
}

int main () {
    int a, b, c;

    printf ("Entre com o primeiro operando:");
    scanf ("%d", &a);
    printf ("Entre com o segundo operando:");
    scanf ("%d", &b);

    soma (a, b, &c);

    printf("%d + %d = %d\n", a, b, c);

    return 0;
}
```

5 Conclusões

Nesta nota de aulas foi dada uma pincelada sobre a diferença e a semelhança entre as duas lingaugens C e Python, com o foco no uso da linguagem C para programação dos microcontroladores. Mostramos que, mesmo que Python seja uma linguagem interpretada com tipagem dinâmica e C uma linguagem compilada com tipagem estática, elas tem muitos comandos de execução equivalentes. Seus fluxos de controle seguem lógicas similares. A grande diferença está no gerenciamento de memória e no acesso à camada próxima à circuitaria. Python, por ser uma linguagem projetada para fácil uso e multi-propósito, não permite que um programador chegue nos detalhes da máquina. C, por sua vez, é uma linguagem originalmente concebida para programação de sistemas operacionais. Ela demanda que um programador tenha um pouco da visão de *hardware*. São duas linguagens projetadas para dois extremos do eixo máquina–usuário. Ambas tem suas virtudes e atendem bem o seu propósito. Cabe a nós, os desenvolvedores, selecionar a que for mais conveniente para resolver cada um dos nossos problemas.

Referências

[1] Guido van Rossum. An Introduction to Python for Unix/C Programmers. *Proc. of the NLUUG najaarsconferentie. Dutch UNIX users group*, 1993.

[2] Brian W. Kernighan and Dennis Ritchie. C Programming Language. *Prentice Hall*, Primeira edição, 1978.

[3] Carl Burch. C for Python programmers. `http://www.cs.toronto.edu/~patitsas/cs190/c_for_python.html`, 2011.

[4] Rose-Hulman Institute of Technology. Python and C - Comparisons and Contrasts. `https://www.rose-hulman.edu/class/cs/csse120/Resources/C/Python_vs_C.html`.

[5] Janis Lesinskis. Janis Lesinskis' Blog: Backup of Quora content. `https://www.lesinskis.com/my-quora-content.html`, 2017.

[6] USP - IME. Projeto MAC Multimídia: Material Didático para disciplinas de Introdução à Computação. `https://www.ime.usp.br/~macmulti/exercicios/inteiros/index.html`, 2005.

[7] MIKROELEKTRONIKA SOFTWARE AND HARDWARE SOLUTIONS FOR THE EMBEDDED WORLD Quick Reference Guide for C language. `https://www.handsontec.com/pdf_files/AppNotes/C%20Syntax%20Ref.pdf`.

[8] Python Python Releases for Windows. `https://www.python.org/downloads/windows/`, 2022.

[9] GNU MinGW - Minimalist GNU for Windows. `https://sourceforge.net/projects/mingw/`, 2021.

[10] GeeksforGeeks Type Conversion in C. `https://www.geeksforgeeks.org/type-conversion-c/`, 2020

[11] Sejal Jaiswal. Scope of Variables in Python. `https://www.datacamp.com/community/tutorials/scope-of-variables-python`, 2020.

[12] Wikipedia ASCII Table `https://pt.wikipedia.org/wiki/Ficheiro:ASCII-Table.svg`.

[13] Deepali Learn Data Tpes in C Programming With Examples `https://www.naukri.com/learning/articles/data-types-in-c-programming-with-examples/`, 2022

[14] Jaiswal Pankaj Qualifiers or modifier in C Programming `https://cprogramtutorialfrombasics.blogspot.com/2021/02/qualifiers-or-modifier-in-c.html`, 2021

[15] The Open Group stdint.h `https://pubs.opengroup.org/onlinepubs/009696899/basedefs/stdint.h.html`, 2004

[16] GeeksforGeeks Python Data Types `https://www.geeksforgeeks.org/python-data-types/`, 2021

[17] Arwin Lashawn Python Memory Management: The Essential Guide `https://scoutapm.com/blog/python-memory-management`, 2020

[18] Ankit Pal Storage classes in C-types and examples `https://www.scaler.com/topics/c/storage-classes-in-c/`, 2021

A Instalação de MinGW, Python, Notepad++ em Windows

Um roteiro para instalação de um ambiente minimalista de desenvolvimento de aplicativos usando *toolchain* GNU (compiladores *gcc* e *g++*) e Python:

MinGW: acrônimo de *Minimalist GNU for Windows*. Contém uma coleção de ferramentas de construção de códigos executáveis a partir de códigos-fonte em C.

1. Baixar o instalador (`https://sourceforge.net/projects/mingw/`) em SourceForge
2. Executar o instalador mingw-get-setup.exe com a seguinte configuração minimalista: na opção Basic Setup marcar os pacotes mingw-developer-toolkit, mingw32-base, migw32-gcc-g++, msys-base.
3. Incluir C:\MinGW\bin no campo PATH de variáveis do ambiente (Start Menu > Computer > Properties > Advanced system settings > Environment Variables).

Python: Contém interpretador de Python configurado para usar o compilador de C de MinGW.

1. Baixar o instalador de uma versão até 3.4 em `https://www.python.org/downloads/windows/` (baixei 3.4.3).
2. Executar o instalador (no meu caso, python-3.4.3.exe).
3. Configurar mingw como compilador de C, criando o arquivo distutils.cfg com o seguinte conteúdo na subpasta \Lib\distutils dentro da pasta do Python instalado (no meu caso, C:\Python34):

```
[build]
  compiler=mingw32

[build_ext]
  compiler=mingw32
```

4. Incluir o caminho do executável de python (no meu caso, C:\Python34) no campo PATH de variáveis do ambiente (Start Menu > Computer > Properties > Advanced system settings > Environment Variables).

Notepad++: É um editor simples para códigos-fonte. Suporta diferentes codificações de fonte.

1. Baixar o instalador de notepad++ ((https://notepad-plus-plus.org/downloads).
2. Executar o instalador.