EA871 – LAB. DE PROGRAMAÇÃO BÁSICA DE SISTEMAS DIGITAIS

EXPERIMENTO 1 – De Python para C

Profa. Wu Shin-Ting

OBJETIVO: Introdução à programação em C com base em Python.

ASSUNTOS: Diferenças e semelhanças entre programação em Python e em C.

O que você deve ser capaz ao final deste experimento?

Distinguir uma linguagem de programação interpretada e uma linguagem compilada.

Distinguir tipagem dinâmica e tipagem estática.

Ter uma noção dos recursos de gerenciamento de memória em C.

Entender a relevância da programação em C para microcontroladores.

Ter uma visão comparativa entre Python e C.

Gerar um código executável a partir de um código-fonte C no ambiente de desenvolvimento integrado (IDE) CodeWarrior.

Ter um primeiro contato com o ambiente IDE CodeWarrior.

INTRODUÇÃO

Na disciplina de Programação de micro- e mini-computadores foi introduzida a linguagem de programação Python. Nesta disciplina de Programação Básica de Sistemas Digitais precisamos nos inteirar com a linguagem C para programar os micro-controladores nos ambientes de desenvolvimento integrado IDE (*Integrated Development Environment*).

Linguagem interpretada e linguagem compilada

Python é uma **linguagem interpretada** de alto nível para propósito geral [1]. Ele foi desenvolvido por Guido van Rossum em 1989 com o objetivo de ter uma linguagem que apresenta uma sintaxe intuitiva, similar à linguagem natural em inglês, sem precisar se preocupar com a tipagem e o armazenamento de dados na memória como a linguagem C. A linguagem **C** é uma **linguagem compilada** de médio nível, também para propósito geral. Foi inventada por Dennis Ritchie em *Bell Laboratories* entre 1972--73 para programar sistemas operacionais que antes eram implementados com uma linguagem de baixo nível, o *assembly* [2]. 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.

Uma **linguagem compilada** é uma linguagem que requer que uma máquina converta, por uma cadeia de ferramentas (*toolchain*), os códigos de um programa, denominados **códigos-fonte**, em códigos binários da máquina, conhecidos por **códigos executáveis**, antes da sua execução. Uma **linguagem interpretada** é uma linguagem para a qual a máquina traduz, em tempo de execução, as suas instruções para as referências às funções pré-implementadas (*built-in functions*) e os valores dos seus argumentos. Por dispensar a 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*. 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. Assim, a linguagem interpretada tem sido a preferida para prototipagem e provas de conceito no desenvolvimento de um projeto.

Na Seção 2 (página 3) em [3] é apresentada de forma comparativa as estruturas básicas de um programa em Python e em C. As linhas de instrução em Python só contêm os comandos de execução para serem interpretados no tempo de execução, enquanto as linhas de instrução em C podem contem as diretivas do pré-processador (Figura 1) que são transformadas em comandos nativos de C antes de serem compilados junto com os comandos de instrução em códigos de máquina. Nas Tabelas 1 e 2 da Seção 3 em [3] são mostradas comparativamente os operadores e os comandos de Python e C. Uma breve explicação sobre diretivas é dada na Seção 3.1 (página 5) em [3].

Tipagem de dados

Em ambas as linguagens, as unidades de armazenamento em um espaço de memória são definidas em termos de endereços, medidos em *bytes*. Esses endereços representam unidades individuais de armazenamento, sendo os conteúdos ou valores armazenados nos endereços correspondentes. Os valores podem ocupar um ou vários *bytes*. Essas unidades são denominadas **variáveis** (Seção 4/página 11 em [3]). A linguagem C dispõe de uma série de recursos para um programador gerenciar o uso da memória de forma dedicada. Através das **declarações de variáveis**, programadores atribuem um nome a uma unidade de memória alocada e define o tipo de dado a ser armazenado nela.

Dizemos que a **tipagem** de dados em C é **estática**, porque t**odos os dados** utilizados num programa precisam ter seus espaços de memória alocados e seus tipos definidos antes do uso. Os tipos de dados básicos em C são: **char** (caractere), **int** (valor inteiro), **float** (valor em ponto flutuante), **double** (valor em ponto flutuante duplo) e **void** (ausência de tipo). O tipo **void *** é uma exceção e é reservado para declarar um ponteiro genérico que pode apontar para o endereço de memória de qualquer tipo. Para declarar o endereço de memória de um tipo de dado específico, utilizamos o operador de ponteiro para esse tipo. Por exemplo, para declarar que o conteúdo de uma variável **`c`** é o endereço de um valor inteiro, usamos a seguinte declaração:

```
char *c;
```

Nessa declaração, o asterisco (*) indica que o valor atribuído a `c` é interpretado como o endereço de um valor inteiro na memória. Portanto, `c` é também denominado **ponteiro** para um valor inteiro.

São ainda disponíveis em C um conjunto de **qualificadores/modificadores** de acesso, do tamanho e do sinal desses tipos básicos (Seção 4.1/página 11 em [3]). Nos projetos de sistemas embarcados é muito comum usar os tipos de dados definidos no arquivo `stdint.h` pelo padrão ISO C99 [17], por especificarem explicitamente o tamanho e o sinal dos valores armazenados (Seção 4.1/página 12 em [3]). Por exemplo, para a seguinte declaração

```
uint16_t a;
```

o compilador aloca para a variável `a` um espaço de endereço de 16 *bits* (2 *bytes*) e gera instruções que interpreta o código binário armazenado neste espaço como um valor inteiro sem sinal.

A linguagem Python abstrai as unidades de armazenamento em objetos e oferece funções pré-implementadas que alocam e desalocam espaços de memória de maneira transparente para os programadores. Além de armazenar os valores propriamente ditos, um objeto no Python também inclui as funções que podem ser aplicadas a esses valores. Os tipos de dados das variáveis são automaticamente definidos no momento em que valores são atribuídos a elas, caracterizando a **tipagem dinâmica** no Python. Essa abordagem permite usar espaços de memória sem a necessidade de declarações explícitas de variáveis, proporcionando uma vantagem distintiva no Python.

Por outro lado, é importante observar que o Python não oferece recursos adicionais para implementar soluções alternativas, como alocar 1 *byte*, em vez de 4 *bytes*, para representar valores inteiros entre 0 e 255. Essa limitação pode se tornar um desafio em dispositivos com recursos escassos, como microcontroladores.

Escopo das Variáveis

Em termos de acessibilidade, tanto C quanto Python diferenciam as variáveis em locais e globais. As **variáveis locais** são apenas visíveis/acessíveis dentro do escopo de instruções em que elas são definidas, enquanto as **globais** são acessíveis por um escopo maior podendo abranger um arquivo ou um módulo de funções (Seção 4.2/página 15 em [3]). Diferente de Python, as variáveis locais e globais em C tem diferentes tempos de existência na memória. As variáveis locais são extintas da memória assim que se conclui a execução do escopo das instruções em que elas são válidas, enquanto as variáveis globais tem o seu endereço preservado até o final de execução de um programa.

Gerenciamento de memória

Uma grande diferença entre C e Python está no **gerenciamento de memória**. No Python, a coleta de lixo (*garbage collector*) é assumida para lidar com a gestão das unidades de memória, simplificando a implementação de algoritmos, especialmente aqueles envolvendo estruturas de dados mais complexas. Contudo, ao buscar atender a uma ampla variedade de aplicações, as soluções otimizadas pelos desenvolvedores do Python nem sempre são ideais para tarefas específicas.

Por outro lado, em C, o gerenciamento de memória é manual. Utilizando funções como `malloc`, `realloc`, `calloc` e `free` [9], os programadores alocam dinamicamente unidades de memória durante a execução do programa e liberam aquelas que não são mais necessárias para reutilização. No entanto, em aplicações embarcadas, onde a previsibilidade de resposta e o tempo de execução de instruções são fatores críticos, é recomendável evitar a alocação dinâmica de memória.

Passagem de parâmetros para as funções

O uso de funções desempenha um papel fundamental na prevenção da repetição de instruções ou códigos dentro de um programa, contribuindo para a modularização e o reuso eficiente do código. Muitas funções requerem dados de entrada e retornam dados de saída, e esses dados são transmitidos como argumentos ou parâmetros.

No Python, a passagem de parâmetros é feita **por referência**, ou seja, é transmitido o endereço de memória. Isso implica que, ao compartilharem uma função e a função que a chamou, as alterações nos valores das variáveis feitas na função são refletidas na função chamadora. Em contraste, em C, a passagem de parâmetros é **por valor**, ou seja, a função recebe uma cópia do valor passado. Portanto, as modificações no valor dentro da função não são refletidas na função que a chamou. Para atualizar o valor de uma variável da função chamadora dentro de uma função em C, é necessário passar o endereço da variável como "valor".

Em C, a distinção entre o valor de uma variável e o seu endereço é feita pelos operadores "**&**" (endereço) e "*" (valor-de). Quando o valor de uma variável é um endereço, é possível diferenciar entre o valor atribuído à variável (endereço) e o conteúdo do valor atribuído usando esses operadores (Seção 4.3/página 17 em [3]). Por exemplo, na seguinte declaração das variáveis `a` e `c`

uint16_t a;

```
char *c;
```

o conteúdo de `a` é um valor inteiro de 16 *bits* e `c` é um endereço ao tipo `char`. Para referenciar o endereço de `a` em C, usamos `&a`, e para expressar o conteúdo do endereço apontado por `c`, a convenção é `*c`.

Comunicação com o mundo externo

A comunicação entre uma máquina e o mundo externo, acompanhando as tarefas humanas, ocorre por meio das **funções de entrada e saída**. Na etapa de entrada, um programa recebe os dados essenciais para sua execução, transferindo-os dos periféricos de entrada para a memória. Na etapa de saída, o programa envia os resultados computados para os periféricos, realizando a transferência dos dados da memória para esses dispositivos. Normalmente, um usuário fornece dados via teclado, e o programa retorna os resultados na tela.

No Python, as funções básicas de entrada e saída, `input` e `print`, estão disponíveis por padrão [10]. Já em C, as funções básicas de entrada e saída, `scanf` e `printf`, fazem parte de uma biblioteca padrão, sendo necessário incluir o arquivo de cabeçalho `stdio.h` para utilizá-las [11]. Essas funções operam sobre sequências de caracteres conhecidas como "*streams*", que são gerenciadas pelo **sistema operacional**. Além das variáveis de entrada/saída, os formatos de como essas *streams* de caracteres devem ser interpretadas também são incluídos entre os argumentos dessas funções.

Conversão de códigos-fonte em códigos executáveis

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` com instruções puras de C; **compilador** 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.

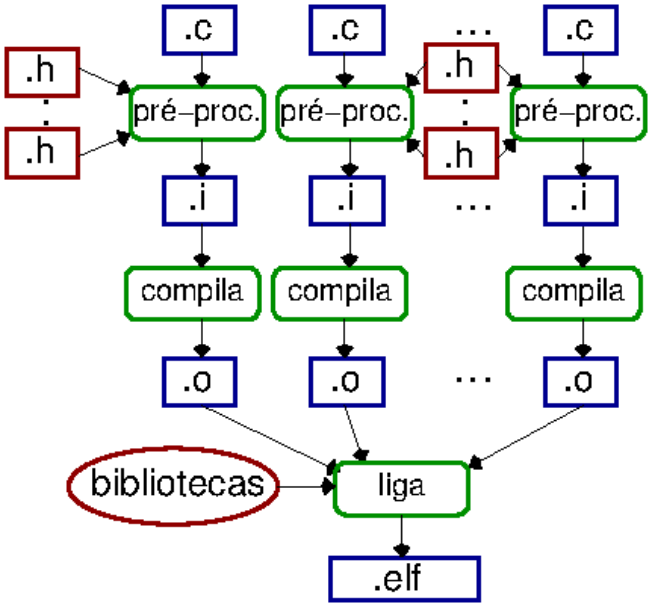

Figura 1: Processo de geração de um executável a partir de um código-fonte em C.

Nesta disciplina usaremos o **ambiente de desenvolvimento integrado (IDE) CodeWarrior**, que tem uma interface gráfica para assistir ao desenvolvimento de aplicativos embarcados nos microcontroladores NXP Kinetis.

Hardware, Firmware e Software

O conjunto de componentes físicos, circuitos integrados, dispositivos eletrônicos, placas, monitor, equipamentos periféricos etc., que forma um sistema digital (computador) é denominado ***hardware*** do sistema. O conjunto de instruções que controlam diretamente a operação de *hardware,* complementando as suas funções, é denominado ***firmware***. Exemplos de *firmware* são BIOS (*Basic Input/Output System*) e UEFI (*Unified Extensible Firmware Interface*). Os *drivers* são *firmwares* que controlam dispositivos específicos de *hardware*, como as placas de rede, vídeo e som, conectando-os com o **sistema operacional**. ***Software*** é um conjunto de instruções para um processador de um sistema digital controlar todos os *hardwares* reconhecidos por ele. O sistema operacional é um *software*. A maioria dos *softwares* com os quais estamos familiarizados, como navegador e editor de textos, é chamada de ***software* de aplicativo**. Ao contrário do *hardware*, o *software* é extremamente flexível, uma vez que permite ser continuamente atualizado e alterado sem que o sistema seja resetado.

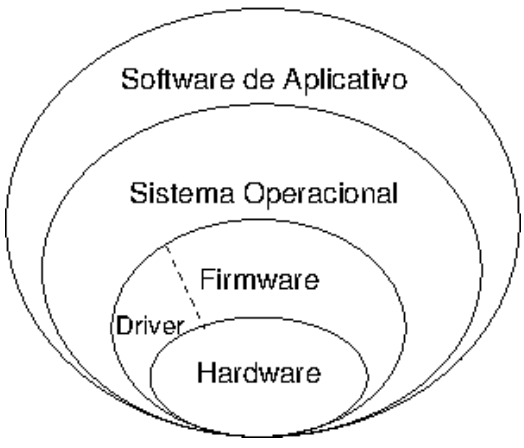

Figura 2: *Hardware*, *firmware*, *driver* e *software.*

O foco desta disciplina são os microcontroladores que operam sem a presença de um sistema operacional. Estaremos desenvolvendo programas diretamente no *hardware*, em uma abordagem conhecida como *bare-metal* (metal nu). Os programas resultantes são denominados ***bare-metal firmware***. Aprenderemos a configurar o *hardware* por meio de um programa em C e a programar o processador, também em C, para executar tarefas específicas no hardware previamente configurado.

Informações Adicionais

Convido fazer uma visita ao tutorial interativo *online* de C [4] e tentar fazer os exercícios propostos para ganhar familiaridade com C.

EXPERIMENTO

Neste experimento vamos nos familiarizar com a linguagem de programação C através dos conhecimentos prévios de Python. São usados o interpretador Python e o compilador C *online* [5] e o ambiente IDE CodeWarrior instalado nos computadores do LE-30.

1. Considere os programas em C e Python da solução do Problema 8 (projeto MAC Multimídia – inteiros [7]) e os programas em C [13] e Python [14] da solução do Problema 3 (projeto MAC Multimídia – reais [6]).

   1.a **Tipagem de Dados:** Identifique as variáveis em Python e os seus equivalentes em C na implementação da solução do Problema 8. Como você reconhece a tipagem de dados em cada linguagem por meio dos códigos?

   1.b **Escopo das Variáveis:** Dois escopos estão presentes nos programas desenvolvidos para resolver o Problema 3: o escopo do programa em si e o escopo dentro do laço de iteração utilizado para testar a pertinência. A qual desses escopos pertencem as variáveis utilizadas nos códigos implementados em C e Python para solucionar o Problema 3?

   1.c **Passagem de parâmetros para as funções:** A transferência do endereço de uma variável para uma função possibilita que a função leia ou modifique diretamente o valor da variável original na memória. Portanto, é comum passar endereços para funções, especialmente nas funções de entrada, para permitir operações de escrita nas posições de memória referenciadas. Por outro lado, nas funções de saída, a intenção é exibir os valores sem alterar o conteúdo da memória, sendo suficiente passar os próprios valores. Identifique as variáveis de entrada e saída na implementações da solução do Problema 3. Como você reconhece o tipo de passagem de variáveis adotado em cada linguagem por meio de códigos?

   1.d **Tipagem de Dados – Alocação explícita:** Nas implementações da solução do Problema 3, uma instrução de verificação do espaço de memória alocado para as variáveis é incluída por meio do operador `sizeof`. Em qual das duas linguagens, C ou Python, ocorre uma alocação de espaço maior para uma variável? Justifique.

   1.e **Leitura de um código em C:** Em [7] são apresentadas 2 soluções para o Problema 8 em Python e 3 soluções em C. Analise-as. Identifique e justifique qual das 3 implementações em C corresponde às implementações em Python.

2. **Tipagem de dados - Operadores *bit*-a-*bit*:** Para otimizar o uso da memória, operações *bit*-a-*bit* são aplicadas na compactação e na extração de informações nos registradores de um microcontrolador. O par de programas `bit_op.*` demonstra o uso de operadores *bit*-a-*bit* definidos em C [16] e Python [15]. No programa em C as variáveis a, b e c são declaradas como do tipo `uint8_t`, para o qual o compilador aloca um espaço de memória de 8 *bits* e interpreta todos os valores armazenados nesses 8 *bits* como um inteiro sem sinal. Analise de forma comparada os dois programas com diferentes valores de entrada para os operandos `a` e `b`.

   2.a Preencha em cada linha da tabela abaixo o nome de cada operador e os resultados obtidos para a = -20 e b = 100.

   2.b Explique as diferenças nos resultados das operações a|b, a^b, a<<2 e a>>2 em Python e C. Dica: Utilize a função `type ()` para descobrir o tipo de dado da variável `c` em Python.

   2.c Explique a diferença entre [~b] e ~b na implementação em C.

	Nome do Operador	Python	C
a & b			
a | b			
a ^ b			
~a			
~b			
a << 2			
a >> 2			

3. **Criação e execução de projetos sem funções de entrada/saída:** As funções de entrada (*scanf*) e saída (*printf*) são normalmente implementadas em cima de um sistema operacional. No entanto, nas bibliotecas-padrão C instaladas no ambiente IDE CodeWarrior, essas duas funções não estão disponíveis. Como resultado, ao utilizar os programas em C fornecidos nos itens 1 e 2, o ligador não encontra essas funções e não consegue gerar os executáveis correspondentes.

   Apesar dessa limitação, existe uma alternativa viável. Podemos inserir valores diretamente nos endereços das variáveis por meio da aba `Variables` na perspectiva `Debug` do IDE, aproveitando a interrupção do fluxo de controle do programa através de *breakpoints*.

   Quando o fluxo de controle é interrompido em uma linha de comando específica, todas as instruções anteriores foram executadas, e o processador aguarda a execução da linha de parada. Ao inserir valores em variáveis/endereços antes de continuar a execução, os valores inseridos serão considerados. Para ilustrar esse processo, consideremos a remoção de todas as funções de entrada e saída do seguinte programa `fatorial.c` [12] e a variação do valor da variável de entrada `n`, assim como ler o valor do resultado `res`, colocando na função `main` um *breakpoint*

na linha da chamada `fatorial (n, &res)` (antes da execução da função para inserir um valor em `n`) e um na linha seguinte (após a execução para ver `n!` em `res`).

```
/*
 * fatorial.c
 */

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

//#include "stdint.h"

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

/* Definicao da funcao main */
int main (){
  int n;        /* guarda o numero dado */
  int    res;

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

  fatorial(n, &res);

//  printf("O valor de %d!: %d\n", n, res);
  }

  return 0;
}

/* Definicao da funcao fatorial */
int fatorial(int n, int *fatorial) {
  int contador;

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

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

  return *fatorial;
}
```

Para criar um novo projeto no ambiente IDE CodeWarrior, siga as instruções na Seção 2.1, página 4, da referência [8]. No processo, selecione C:\ea871\<seu RA> como o seu *Workspace* e defina "fatorial" como o nome do projeto. Após concluir as etapas e chegar ao final (*Finish*), o projeto será aberto na Perspectiva C/C++, como mostra a Seção 2.2 na página 10 em [8]. Dentro do projejto, acesse a pasta `Sources` , abra o arquivo `main.c` , clicando no nome, e substitua todo o seu conteúdo pelo conteúdo do arquivo `fatorial.c`. Realize as alterações sugeridas na versão original de `fatorial.c`. Em seguida, prossiga com as instruções descritas na Seção 2.3, página 18, para gerar um executável. Após isso, siga as orientações da Seção 2.4, página 24, para transferir o executável gerado no PC para o microcontrolador no modo `Debug`. Assim que concluir a transferência, é chaveada para a Perspectiva `Debug`.

Após a geração do executável `fatorial`, identifique entre as linhas de mensagem mostradas na aba `Console` (Perspectiva C/C+) os arquivos de extensão .o e `.elf` (Figura 1). A ordem em que esses arquivos foram gerados é condizente com a sequência mostrada na Figura 1? Justifique.

4. **Tipagem de dados - Impacto do `tipo de dados` nos `resultados`:** O processador do nosso microcontrolador opera em 32 *bits*, realizando todas as operações lógico-aritméticas nesse tamanho. Quando os operandos são variáveis de tipos de dados de tamanho menor, eles são automaticamente estendidos para 32 *bits*. Para analisar o impacto da capacidade de armazenamento de um espaço alocado a uma variável e do seu tipo de dado nos resultados de uma operação aritmética usaremos a aba `Variables` da perspectiva `Debug`.

   Para realizar essa análise de forma abrangente, faremos **modificações consistentes** no tipo de dado da variável `res` na função `main` de `fatorial`. As alterações incluirão os tipos **uint8_t** (inteiro sem sinal de 8 *bits*), **uint16_t** (inteiro sem sinal de 16 *bits*), **uint32_t** (inteiro sem sinal de 32 *bits*), **int8_t** (inteiro com sinal de 8 *bits*), **int16_t** (inteiro com sinal de 16 *bits*) e **int32_t** (inteiro com sinal de 32 *bits*). A consistência requer que as variáveis `res`, `fatorial` e a função `fatorial` destacadas no código do item 3 sejam declaradas com o mesmo tipo de dados**.**

   Como C é uma **linguagem compilada, deve-se refazer "*Build Project*" para cada modificação no código-fonte** (Seção 2.3/página 18 em [8])**.** Anote os resultados armazenados no endereço `res`, **na base hexadecimal e decimal**, para n=5, 6, 7, 8, 9, 10, 11, 12, 13. Siga o procedimento descrito na Seção 2.5.1, página 28, em [8] para setar um *breakpoint* na linha "`fatorial (n, &res)`". Use a aba `Variables` para modificar o valor da variável `n`, como instruído na Seção 2.5.3, página 31, em [8], antes de avançar (`Step Over`) para a linha seguinte. Ao avançar para a próxima linha, note que o conteúdo de `res` será automaticamente atualizado com o valor do fatorial de `n` atualizado.

   4.a Com base nos **resultados representados em decimal**, qual é o maior valor `n` para o qual os resultados gerados são corretos em cada tipo de dado (coluna)?

   4.b Com base no valor do *bit* mais significativo dos **resultados representados em hexadecimal**, explique por quê os tipos de dados sem sinal (`uint8_t, uint16_t` e `uint32_t`) conseguem cobrir mais resultados corretos de `n!` do que os tipos de dados com sinal (`int8_t`, `int16_t` e `int32_t`)?

   4.c Explique sucintamente como os tipos de dados impactam nos resultados gerados embora o *hardware* seja o mesmo para todas as operações executadas.

	uint8_t	uint16_t	uint32_t	int8_t	int16_t	int32_t
n=5						
n=6						
n=7						
n=8						
n=9						
n=10						
n=11						
n=12						
n=13						

RELATÓRIO

O relatório deve ser devidamente identificado, contendo a identificação do instituto e da disciplina, o experimento realizado, o nome e RA do aluno. Para este experimento, elabore um documento, no **formato pdf**, contendo as respostas dos itens 1 a 4 do roteiro. Suba-o no sistema *Moodle*.

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] Wu Shin Ting. Referência Comparativa Rápida entre Python e C para Sistemas com Recursos Limitados.
https://www.dca.fee.unicamp.br/cursos/EA871/references/complementos_ea871/Python_C.pdf
[4] learn-c.org. Interactive C tutorial. https://www.learn-c.org/
[5] Compilador de C *online*. https://www.onlinegdb.com/online_c_compiler
[6] USP – IME. Projeto MAC Multimídia: reais
https://www.ime.usp.br/~macmulti/exercicios/reais/index.html
[7] USP – IME. Projeto MAC Multimídia: inteiros.
https://www.ime.usp.br/~macmulti/exercicios/inteiros/index.html
[8] Ambiente de Desenvolvimento – *Software*
https://www.dca.fee.unicamp.br/cursos/EA871/references/apostila_C/AmbienteDesenvolvimentoSoftware_V1.pdf
[9] Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
[10] Python: Input/Output Functions
https://www.decodejava.com/python-input-output.htm
[11] C Input and Output - printf()/scanf(), and more
https://www.studytonight.com/c/c-input-output-function.php
[12] fatorial.c
https://www.dca.fee.unicamp.br/cursos/EA871/1s2022/codes/fatorial.c
[13] pertence.c

https://www.dca.fee.unicamp.br/cursos/EA871/1s2024/codes/pertence.c
[14] pertence.py
https://www.dca.fee.unicamp.br/cursos/EA871/1s2024/codes/pertence.py
[15] bit_op.py
https://www.dca.fee.unicamp.br/cursos/EA871/1s2024/codes/bit_op.py
[16] bit_op.c
https://www.dca.fee.unicamp.br/cursos/EA871/1s2024/codes/bit_op.c
[17] stdint.h
https://pubs.opengroup.org/onlinepubs/009695399/basedefs/stdint.h.html

Revisado em 28/01/2024
Revisado em 07/08/2022
Elaborado em 23/2/2022