gthc.c low_level.rs

```
GREEN TEAM HACKER CLUB {
    [Programação Low Level]

        < Conteúdo: Apresentação e C >

}
```

Tabela de ‘Conteúdo’ {

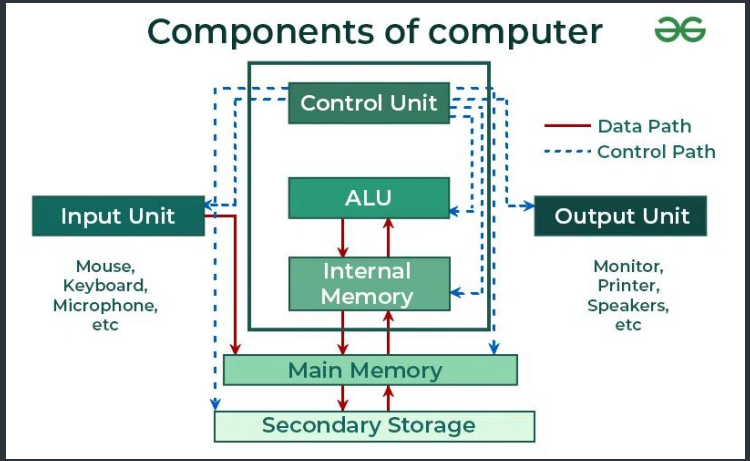
}

```
01 {
    [Programação Low Level]
        < Definição >
}
```


O que é Programar? < /1 > {

Quando escrevemos um código, sabemos o que realmente está sendo executado pela máquina?
Quando escrevemos um algoritmo ou pedimos para executar uma função na memória, quais são as instruções realizadas?

}



Bits, Bytes e Memória < /1 > {

Sabemos que o computador opera utilizando bits (0 e 1) que são agrupados em bytes. Esses bytes são armazenados em endereços de memória específicos para que posteriormente o computador interpretar os dados e realizar cálculos, chamar funções, executar arquivos. A CPU é responsável por recuperar os dados da memória e realizar todas as operações em binário para garantir o funcionamento desejado do computador.

Posteriormente podemos abordar mais detalhadamente a arquitetura de um computador

}

Linguagens de Baixo Nível {

Então, o que são as linguagens de baixo nível?
São as linguagens de programação que se aproximam do código de máquina e do hardware do computador, realizando chamadas de instruções para conseguir realizar todo o tipo de tarefas.

}

Exemplos {

Assembly ARM - 64 bits, Assembly AMD/Intel - 64 bits, Assembly 8086,

C,C++, C#,Rust

}

Vantagens: Controle do código{

Quando você chama um função em Python, você sabe o que realmente está acontecendo por baixo dos panos? Quais são os parâmetros, onde as variáveis vão parar na memória?

}

Vantagens: Velocidade e Eficiência{

Por conta da manipulação de memória, compilação e execução de binários, os códigos de baixo níveis são mais rápidos e podem rodar em uma variedade de dispositivos

}

Desvantagens: Riscos{

Uma das principais questões relacionadas ao uso de linguagens de baixo nível são os riscos relacionados ao utilizá-las de maneira incorreta ou insegura!

}

STACK OVERFLOW e BUFFER OVERFLOW!

```
02 {
    [C]
        < Linguagem C e Ponteiros >
}
```


Origens < /1 > {

C é uma linguagem de programação criada por Dennis Ritchie nos Laboratórios Bell em 1972.

É uma linguagem muito popular, apesar de ser antiga. O principal motivo de sua popularidade é porque é uma linguagem fundamental no campo da ciência da computação.

C é fortemente associado ao UNIX, pois foi desenvolvido para escrever o sistema operacional UNIX.

}

Execução e GCC< /1.1 > {

Ao programar um programa em C é possível utilizar VSCode (útil por conta das extensões), CLion, Vim, ou até um ambiente online.

Entretanto, para garantir que o programa seja executado, é necessário que o código seja compilado pelo GCC, onde o código será convertido em um executável!

}

C< /2 > {

C é uma linguagem processual imperativa, que suporta programação estruturada, escopo de variável lexical e recursão, com um sistema de tipo estático. Foi projetada para ser compilada para fornecer acesso de baixo nível à memória e construções de linguagem que mapeiam eficientemente as instruções de máquina, tudo com. suporte mínimo de tempo de execução Apesar de seus recursos de baixo nível, a linguagem foi projetada para encorajar a programação multiplataforma. Um programa C compatível com os padrões, escrito com a portabilidade em mente, pode ser compilado para uma ampla variedade de plataformas de computador e sistemas operacionais com poucas alterações. seu código-fonte.

}

Sintaxe< /3 > {

Variáveis:

- char: Armazena um caractere*
- int: Armazena um número inteiro
- float: Armazena um número real com uma certa precisão
- double: Armazena um número real com uma precisão maior que o float
- void: Tipo vazio
- structs*

}

Ponteiros< /4 > {

Ponteiros são extremamente úteis e uma ferramenta muito poderosa em C, com eles é possível implementar diversos tipos de dados (Strings, Arrays e Matrizes) e também estruturas de dados (Listas, Pilhas, Árvores).

Por isso é fundamental entender como eles funcionam e quais são as restrições ao utilizá-los

}

Ponteiros: Heap e Stack< /4 > {

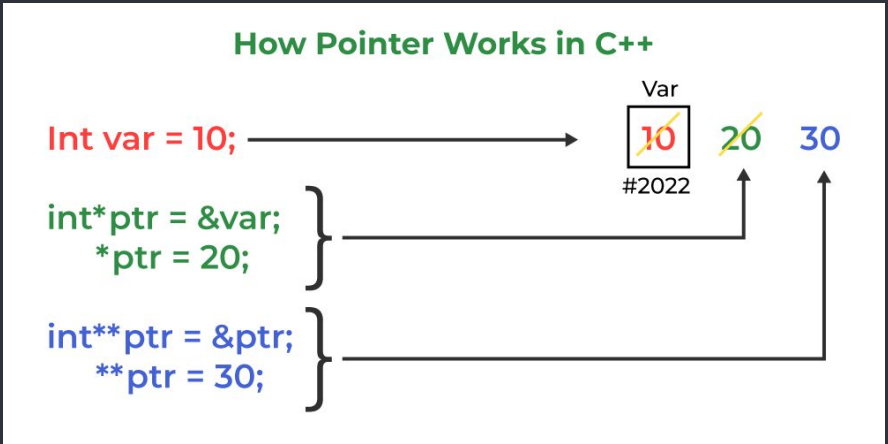
Quando declaramos uma variável como:
`int x;`

A variável é declarada na pilha de execução (STACK), mas o que fazer quando precisamos de grandes variáveis ou quando precisamos acessar a memória em diferentes escopos?

Nesse caso, usa-se um ponteiro para a HEAP:
`int* x;`

}

Ponteiros: Heap e Stack< /4 > {



}

Próximo Encontro {

< Memória: HEAP e STACK >

}

Referências {

<Low Level Programming Book - https://evalandaply.neocities.org/books/lowlevelprogramming.pdf >

}

```
Obrigado; {

    ‘Dúvidas?’

        luccas.h.cortes@hotmail.com
        https://github.com/Cortesz/
        https://github.com/greenteamhc/greenteamhc.low_level

}
```