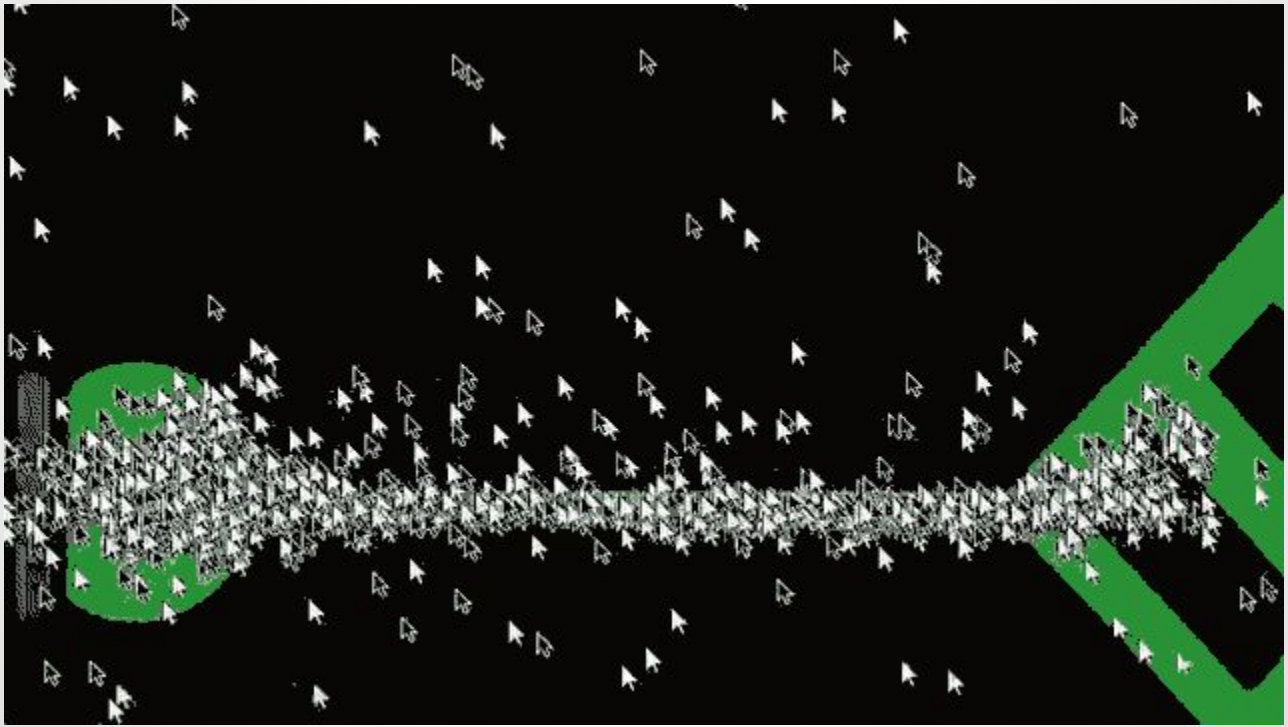
Ponteiros e Alocação Dinâmica

Aula 14

Às vezes, precisamos é apontar!

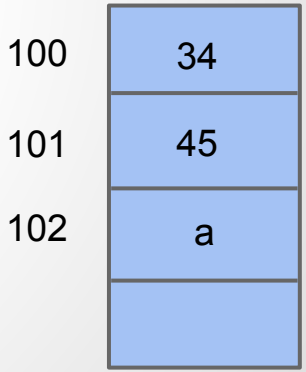


Sumário

- Variáveis e Memória
- Ponteiros
  - Conceito
  - Declaração
  - Uso
- Aplicações

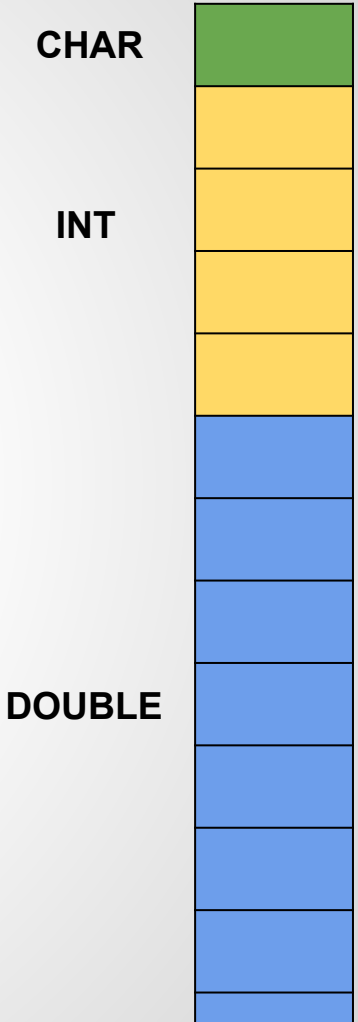
Memória e Variáveis

- Toda variável corresponde a um pedaço de memória
  - Guarda um valor
  - Tem um endereço



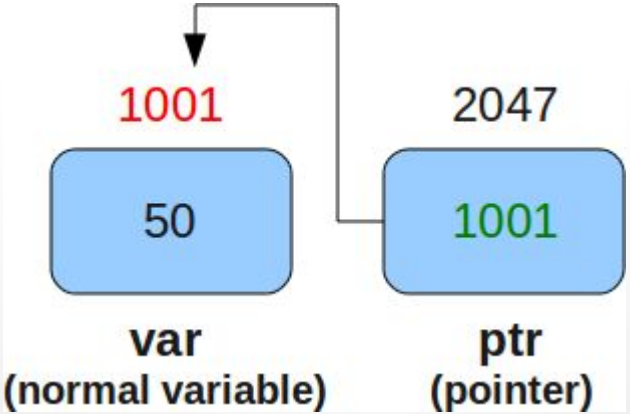
Memória e Variáveis

- Cada tipo de variável possui um tamanho diferente!



Ponteiros!

- Tipos especiais de objetos, que guardam o valor de um endereço na memória
  - Apontam para uma variável, por exemplo



Então...qual é a ideia?

- Um ponteiro em C é definido na seguinte forma:
  - `tipo de dado* nome_ponteiro;`

Ex:

```
int* ptr;
```

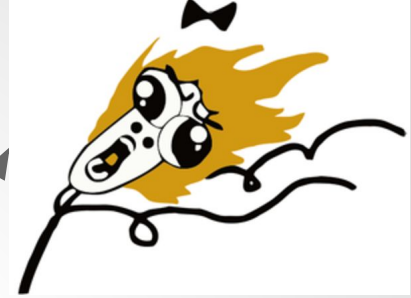
Cuidado com a inicialização!

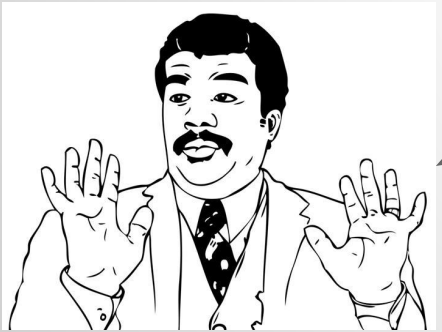
- Variáveis não inicializadas contém valores lixo.
- Ponteiros não inicializados iniciam o apocalipse.

Imagine o seguinte....

int *ptr; (aponta para??)

*ptr = 5;





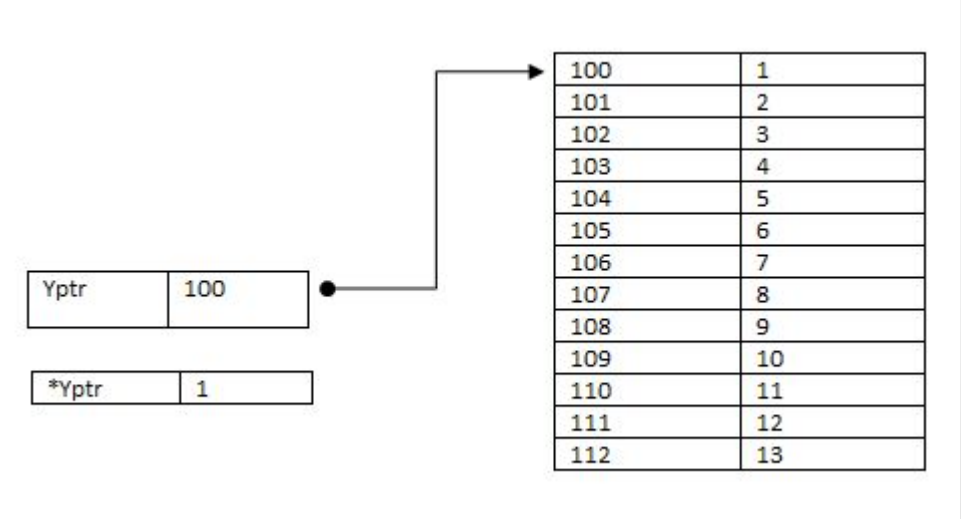
Lembra do operador &?

- Para pegar o endereço de memória de uma variável, usamos o operador & (sim, o mesmo do scanf!!!).

```
int a;
int *ptr_a;
ptr_a = &a;
```

Referenciando e Derreferenciando

- Quando temos um ponteiro Yptr:
  - Yptr é o endereço de memória;
  - *Yptr é o valor que está no endereço;



Aritmética de ponteiros

- O que acontece quando eu faço uma operação com um ponteiro?

Ex:

```
ptr++;
ptr += 5;
```

Aritmética de Ponteiros

- Aplicação:
  - Estruturas de dados mutidimensionais
    - Vetores, Matrizes
    - Registros
  - Transferência de Dados

Vetores são ponteiros disfarçados!

Os vetores são armazenados de forma sequencial na memória.

Internamente, o C armazena um ponteiro para o início do vetor.

Quando fazemos vetor[5], o C faz ptr_vetor+5!

Outras aplicações

- Passagem de estruturas multidimensionais para funções
- Estruturas de Dados
- Referências para funções (wtf???)

Por que às vezes a memória vaza...

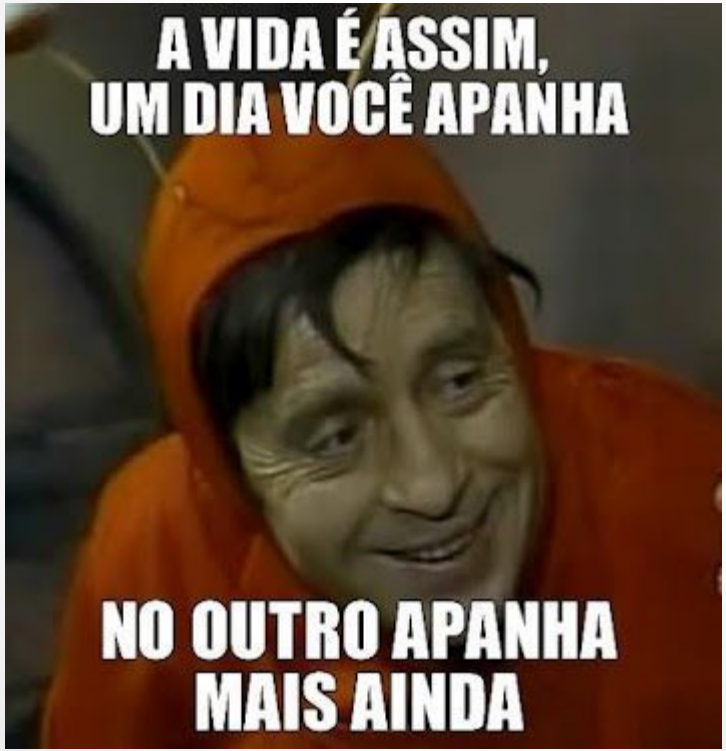
Cuidando da memória

- Existem duas estruturas de dados que implementam o acesso à memória no sistema operacional:
  - Stack (Pilha)
    - Rápida
    - Local
    - Limitada pelo Sistema Operacional
  - Heap
    - Lenta
    - Global
    - Limitada pela memória do seu PC

Alocando variáveis

- Variáveis declaradas dentro das funções são alocadas na pilha de memória
  - Eficiência
  - Gestão feita pelo SO
- Mas às vezes precisamos de mais do que a pilha pode nos oferecer:
  - Tamanho não conhecido
  - Tamanho variável
  - Tamanho muito grande

E agora quem poderá nos ajudar?



Alocação na Heap

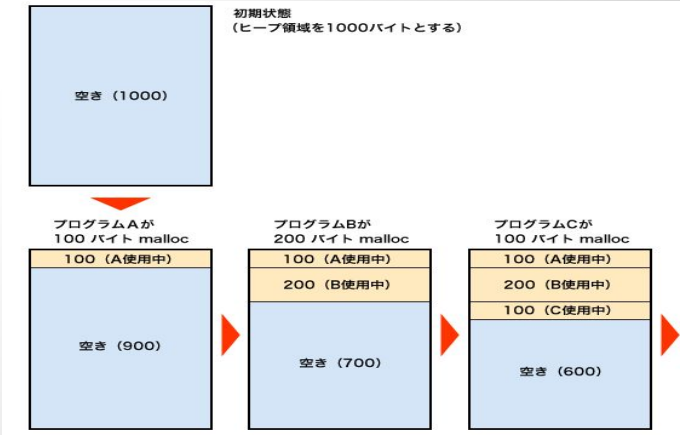
- Para evitar essas limitações, alocamos as variáveis na heap de memória
    - Variáveis Globais
    - Variáveis alocadas dinamicamente

- Cuidado!!! Fácil de perder o controle.

- Solução recomendada: alocação dinâmica!

Alocação dinâmica

- Em C, podemos definir em tempo de execução qual será o tamanho das nossas variáveis
- Uso de funções da biblioteca stdlib.h
  - malloc
  - calloc
  - realloc
  - free



Como funciona?

- O programa inicia apenas com uma declaração de um ponteiro para a variável (já que ainda não se sabe o tamanho dela).
- As funções vão pegar o ponteiro e tentar encontrar um espaço na memória do tamanho que o programador requisitar
  - Se existir, a aplicação retorna um ponteiro com o endereço inicial do bloco de memória alocado
  - Se não, retorna NULL

Função malloc

- void *malloc(size_t size)
- Recebe um argumento (tamanho em bytes) para alocar,
- Retorna um ponteiro com o endereço
    - Como ela pode ser alocada para vários tipos de dados diferentes, o ponteiro para void será convertido com casting para o tipo adequado implicitamente
    - Não inicializa o vetor (o mesmo que apenas declarar o vetor).

```
int* vetorino_sales;
vetorino_sales = malloc(200*sizeof(int)); //Aloca um vetor com duzentos inteiros
```

Função calloc

- void *calloc(size_t nmemb, size_t size);
- Similar ao malloc, porém com dois parâmetros:
  - a quantidade de elementos (ou posições) que serão alocados,
  - o tamanho do tipo de dado
- Inicializa os bytes alocados com o valor 0!

```
int* vetorino_sales;
vetorino_sales = calloc(200, sizeof(int)) //Mesma alocação anterior
```

Função realloc

- void *realloc(void *ptr, size_t size);

- Função que altera o tamanho de uma estrutura.
  - Tenta alocar o novo tamanho na memória, liberando o espaço anterior.

```
int* vetorino_sales = malloc(10*sizeof(int));
//Muda o tamanho do vetor de 10 para 20
vetorino_sales = realloc(vetorino_sales, 20*sizeof(int));
```

Função free

- void free(void *ptr);
- Libera uma área de memória, apontada pelo ponteiro ptr
    - Ponteiros retornados contém lixo
    - Não se deve usar free e realloc em ponteiros que não foram criados a partir das funções de alocação!

```
int* vetorino_sales = calloc(30, sizeof(int));
free(vetorino_sales); //E ele está livre!!!!!
```

Vazamento de memória

- Por que precisamos liberar a memória que alocamos no heap?
  - Porque ela acaba!
  - O SO se responsabiliza apenas pelo Stack
- Vazamento de memória == valores lixos ocupando memória.
  - Se a memória não é liberada, os programas não conseguem mais guardar informações lá.

Dúvidas??

Treinando

```
int main( )
{
   int i = 2;
   int j = i * i;
   int *k = &i;
   int m = *k * *k;
   *k = j * *k * m;

   printf( "%i %i %i %i", i, j, *k, m );
   return 0;
}
```

Treinando

```
int main( void )
{
   int  x = 5;
   int *y = &x;
   int z = *y;
   printf( "%i, %i, %i\n", x, *y, z );

   x = 7;
   printf( "%i, %i, %i\n", x, *y, z );

   *y = 2;
   printf( "%i, %i, %i\n", x, *y, z );
   return 0;
}
```

Treinando

```
int main( void )
{
   int x, y = 0, *p = NULL;
   p = &y;
   x = *p;
   x = 4;
   (*p)++;
   --x;
   (*p) += x;

   printf( "%i %i %i", x, y, *p );
   return 0;
}
```