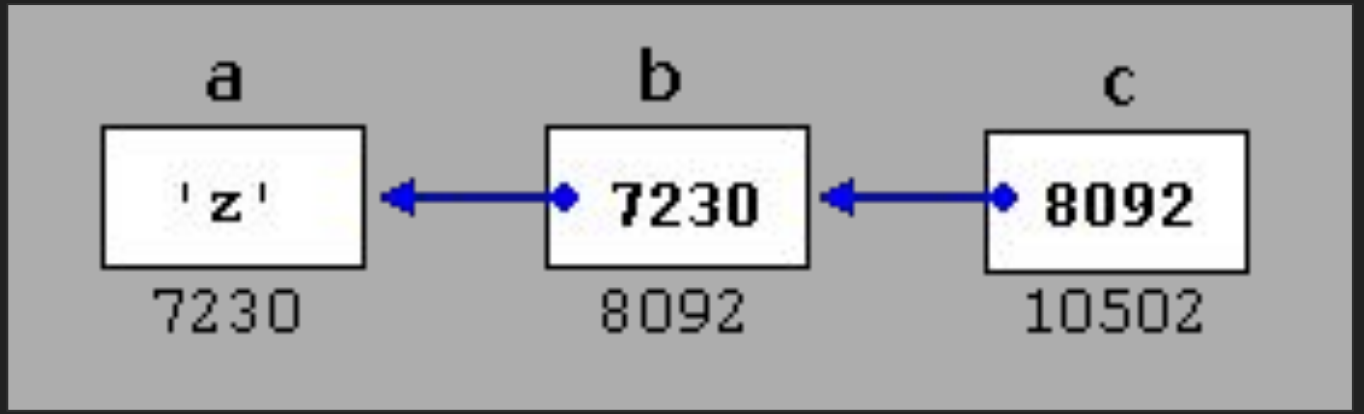
SCC0502 - ALGORITMOS E ESTRUTURAS DE DADOS I

Revisão de ICC, Makefile e .h

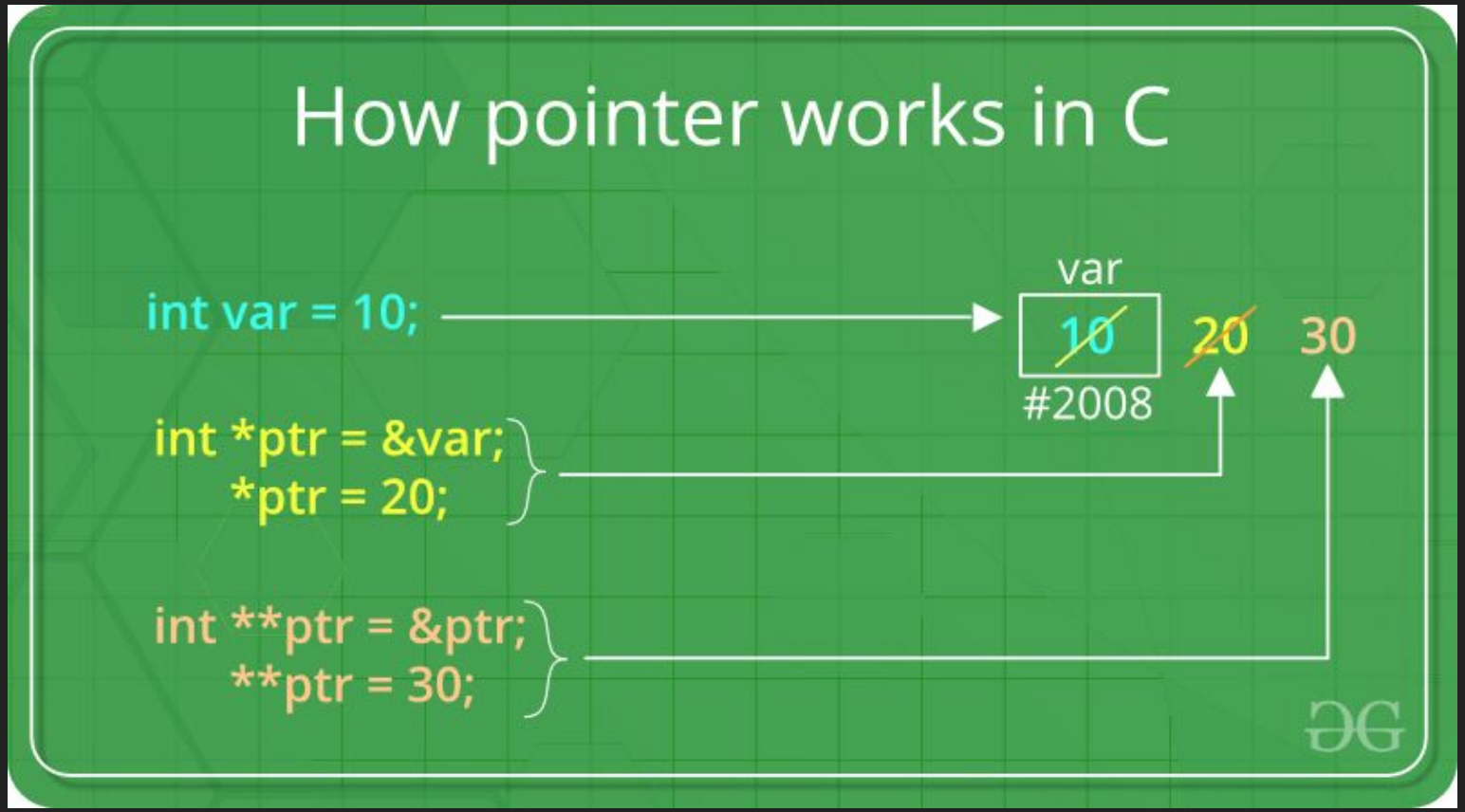
Prof.: Leonardo Tórtoro Pereira
leonardop@usp.br

ICC

Ponteiros?



Fonte: http://www.cplusplus.com/doc/tutorial/pointers/



Fonte:
https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/

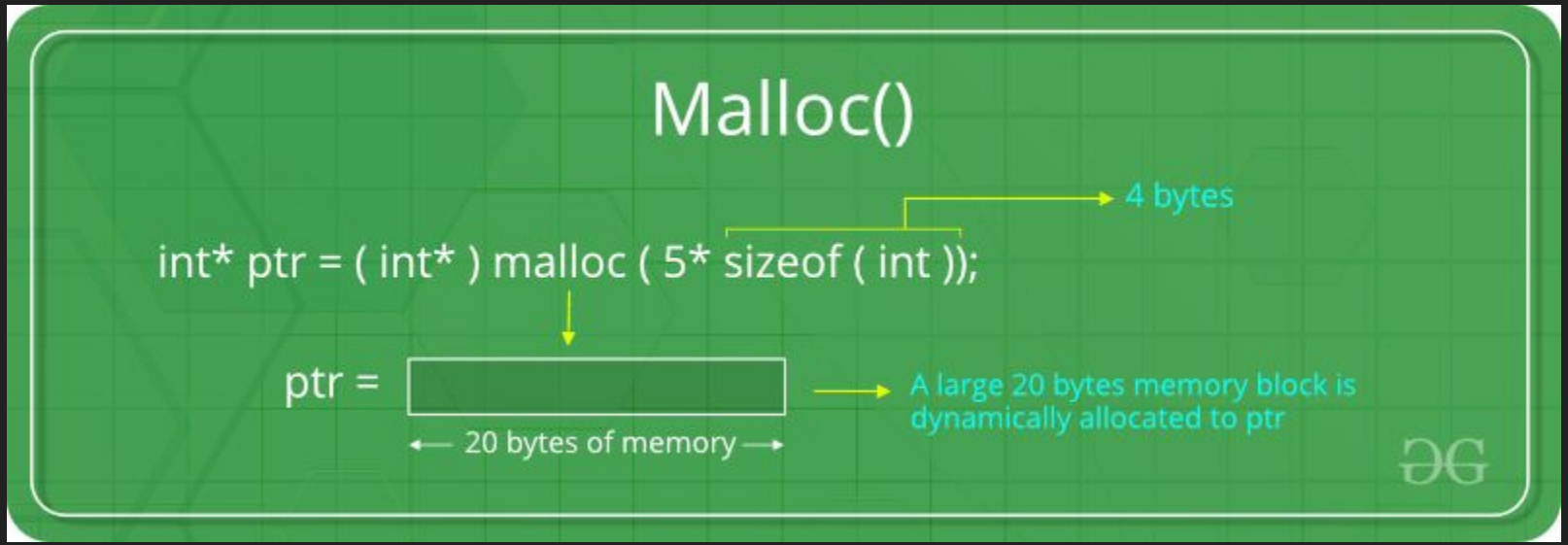
Alocação Dinâmica

Alocação dinâmica [4]

- Ponteiros também são usados como um receptáculo para regiões de memória alocadas dinamicamente
- Isso é muito útil quando desejamos usar vetores, strings, matrizes, etc. de tamanhos variados, ou dos quais não sabemos o tamanho a tempo de compilação
- Para tal, temos 4 funções da *stdlib*
  - *malloc(), calloc(), free()* e *realloc()*

Alocação dinâmica [4]

- *malloc()* vem de "*memory allocation*", e serve para alocar um único bloco de memória com o tamanho especificado
- Retorna um ponteiro de tipo *void* que pode receber *cast* para um ponteiro de qualquer forma
- `ptr = (cast-type*) malloc(byte-size)`
- É costume usar o *sizeof()* para indicar o tamanho do tipo a ser alocado, e multiplicá-lo pela quantidade de unidades daquele tipo que deseja-se alocar



Fonte:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Ponteiros [4]

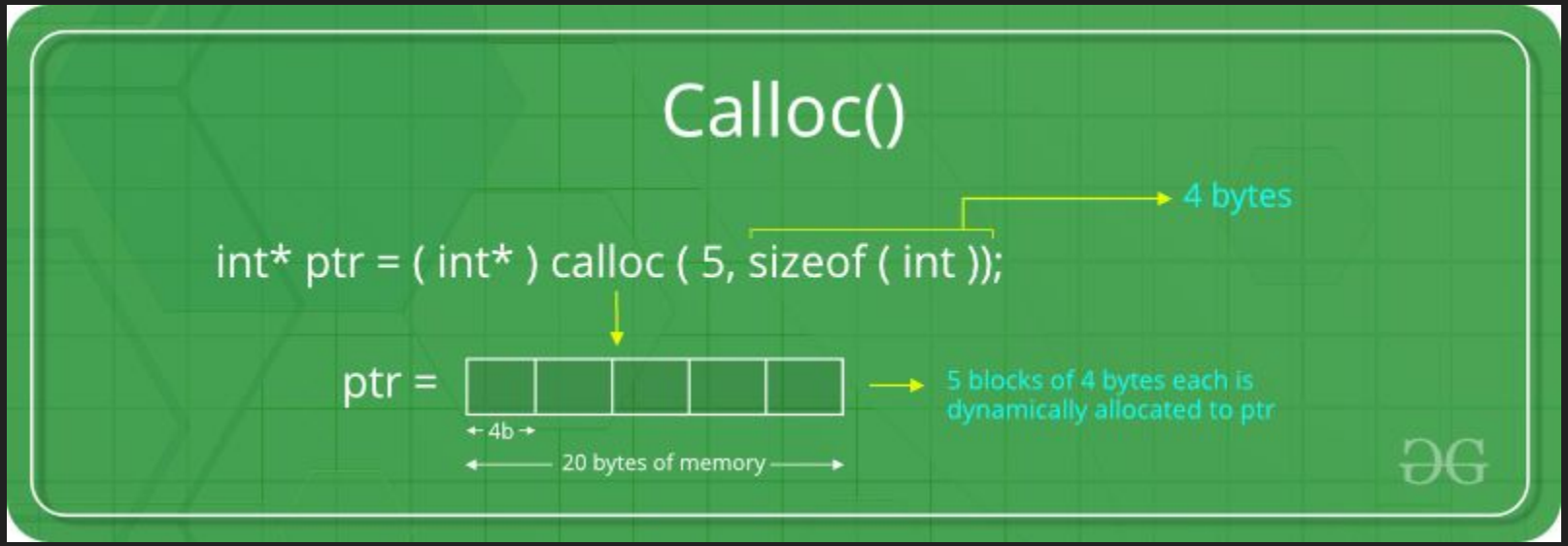
```c
int main() {
    int *ptr, n = 5, i;
    printf("Enter number of elements: %d\n", n);
    ptr = (int*)malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        printf("Memory successfully allocated using malloc.\n");
        for (i = 0; i < n; ++i)
            ptr[i] = i + 1;
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i)
            printf("%d, ", ptr[i]);
    }
    free (prt);
    return 0;
}
```

Alocação dinâmica [4]

- Se a alocação falhar (caso não tenha mais memória disponível) ela retorna um ponteiro *NULL*
  - É importante verificar isso sempre!
- Caso deseje-se inicializar todos os valores com 0, a função *calloc()* faz exatamente isso
  - A sintaxe é similar à do *malloc*(), exceto que o tamanho de cada elemento é passado separadamente
  - `ptr = (cast-type*)calloc(n, element-size);`

Alocação dinâmica [4]

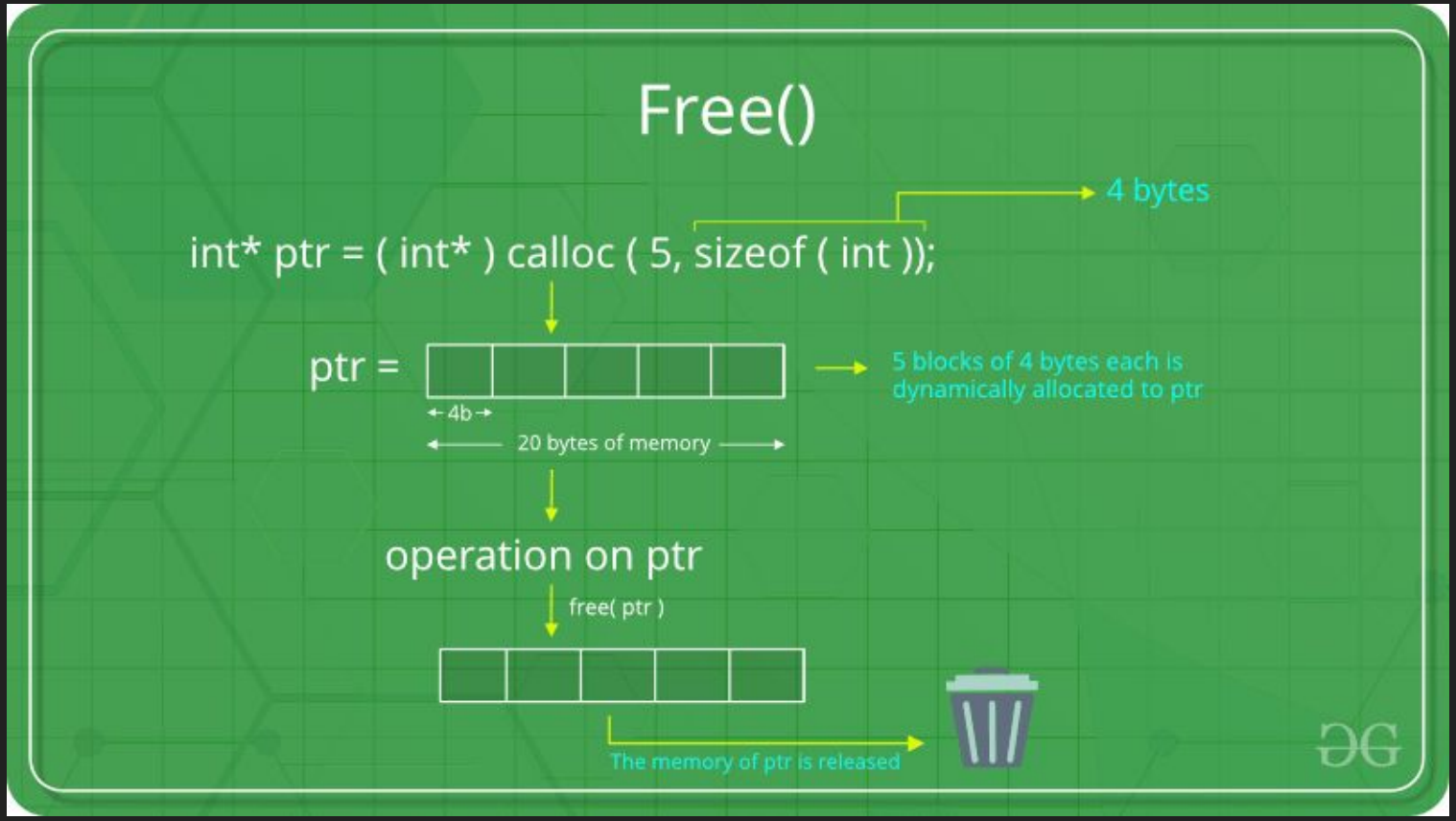
- Deve-se “desalocar” TODA memória alocada dinamicamente com o método *free()*.
  - O programa não libera ela automaticamente!
- Fazer isso para todos os “níveis” de ponteiros alocados
  - Ver próxima seção



Fonte:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Ponteiros [4]

```c
int main() {
    int *ptr, n = 5, i;
    printf("Enter number of elements: %d\n", n);
    ptr = (int*)calloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        printf("Memory successfully allocated using malloc.\n");
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i)
            printf("%d, ", ptr[i]);
    }
    free(ptr);
    return 0;
}
```



Fonte:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Alocação dinâmica [5]

- A função *free()* apenas libera o espaço de memória reservado pelas funções de alocação
  - O espaço fica disponível para que outras chamadas para tais funções possam usá-lo
  - Mas o ponteiro ainda aponta para a mesma região de memória
    - Ela só não é mais válida!

Alocação dinâmica [5]

- Se o ponteiro que é passado para *free()* for NULL
  - A função não faz nada
- Se ele não aponta para um bloco de memória reservado pelas funções de alocação de memória
  - Causa comportamento indefinido

realloc()

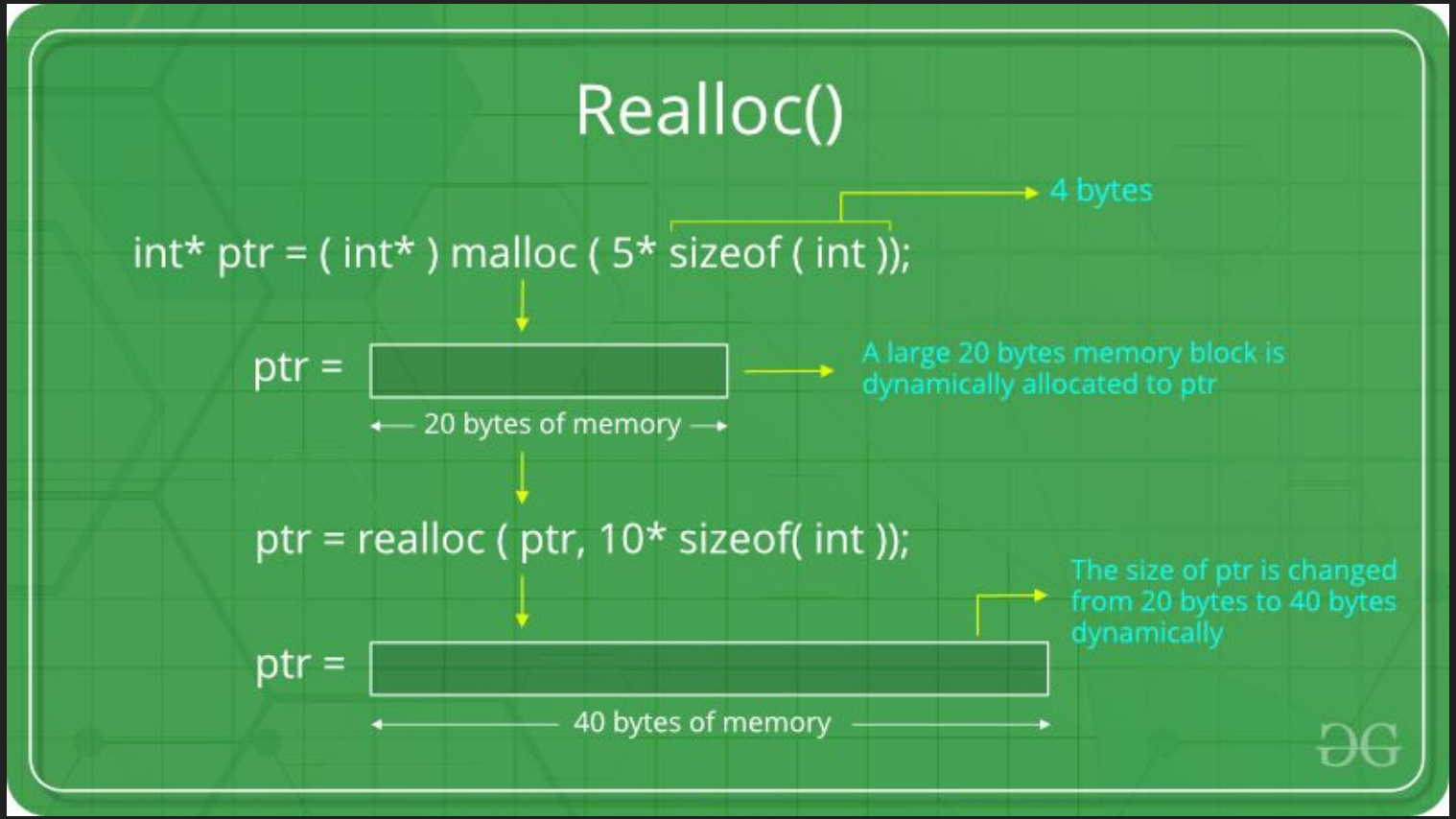
Alocação dinâmica [6]

- O método *realloc(ptr, size)* muda o tamanho do bloco de memória que o ponteiro *ptr* aponta para
- Caso não seja possível estender o bloco atual para o tamanho desejado, a função move o bloco para outro local
  - Por isso ela retorna um ponteiro!

Alocação dinâmica [6]

- O conteúdo no bloco de memória é preservado até o menor tamanho entre o antigo e o novo, mesmo se ele for movido
  - Se o novo bloco for maior, os blocos novos tem valor indeterminado
- Se o ponteiro *ptr* for NULL, a função age como o *malloc*

```c
// Exemplo em http://www.cplusplus.com/reference/cstdlib/realloc/
int main () {
  int input,n, count = 0, *numbers = NULL, *more_numbers = NULL;
  do {
     printf ("Enter an integer value (0 to end): ");
     scanf ("%d", &input);
     count++;
     more_numbers = (int*) realloc (numbers, count * sizeof(int));
     if (more_numbers!=NULL) {
       numbers=more_numbers;
       numbers[count-1]=input;
     }
     else {
       free (numbers);
       puts ("Error (re)allocating memory");
       exit (1);
     }
  } while (input!=0);
  printf ("Numbers entered: ");
  for (n=0;n<count;n++) printf ("%d ",numbers[n]);
  free (numbers);
  return 0;
}
```



Fonte:
https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/

Alocação dinâmica [4]

- Quando usamos *realloc* em um ponteiro passado para uma função corremos o risco de perder a referência original dele fora do escopo da função!
- Não devemos dar *realloc* em ponteiros em uma função se não retornarmos este (possível) novo ponteiro
- Uma alternativa é passar um ponteiro para o ponteiro que será realocado!

Exemplo Realloc em Funções

```
//Retornando o ponteiro realocado
char* concatStrings(char* string1, char* string2) {
    //String1 + String2 + \n + espaço
    int newSize = strlen(string1) + strlen(string2) + 2;
    int index1 = strlen(string1);
    string1 = (char*) realloc(string1, sizeof(char)*newSize);
    string1[index1++] = ' ';
    int index2 = 0;
    while(string2[index2] != '\0') {
        string1[index1++] = string2[index2++];
    }
    string1[index1] = '\0';
    return string1;
}
```

Exemplo Realloc em Funções

```
int main(){
    char* stringPtr = NULL;
    char string1[] = "fique";
    char string2[] = "em";
    char string3[] = "casa";
    stringPtr = (char*) malloc(sizeof(char)*strlen(string1));
    strcpy(stringPtr, string1);
    printf("\n%s\n", stringPtr);
    stringPtr = concatStrings(&stringPtr, string2);
    printf("\n%s\n", stringPtr);
    stringPtr = concatStrings(&stringPtr, string3);
    printf("\n%s\n", stringPtr);
    return 0;
}
```

Alternativa também correta:

```
//Usando ponteiro para o ponteiro a ser realocado
void concatStrings2(char** string1, char* string2){
    //String1 + String2 + \n + espaço
    int newSize = strlen(*string1) + strlen(string2) + 2;
    int index1 = strlen(*string1);
    *string1 = (char*) realloc(*string1, sizeof(char)*newSize);
    (*string1)[index1++] = ' ';
    int index2 = 0;
    while(string2[index2] != '\0')    {
        (*string1)[index1++] = string2[index2++];
    }
    (*string1)[index1] = '\0';
}
```

Alternativa errada:

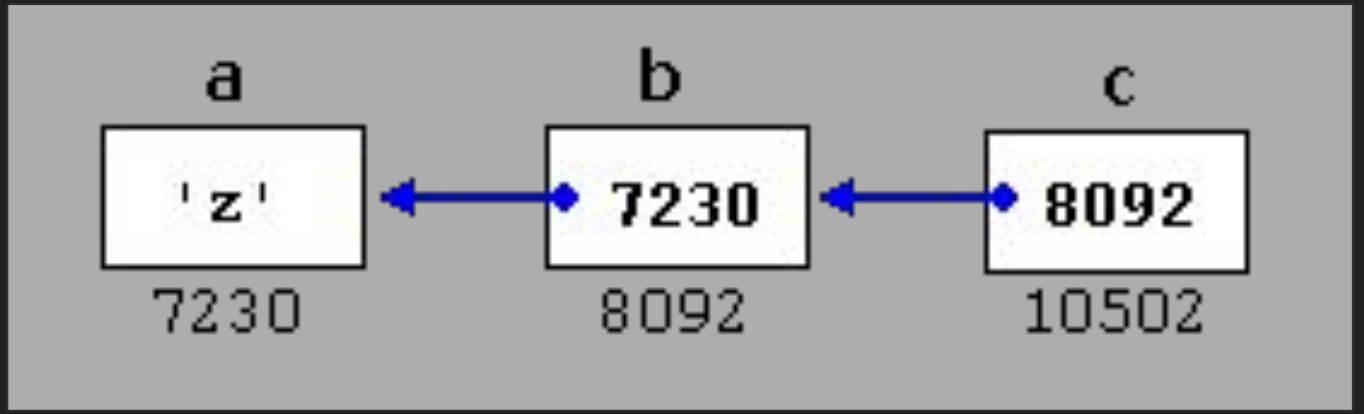
```
//Não retornando o ponteiro reallocado - ERRADO!!!
void concatStringsWrong(char* string1, char* string2){
    //String1 + String2 + \n + espaço
    int newSize = strlen(string1) + strlen(string2) + 2;
    int index1 = strlen(string1);
    string1 = (char*) realloc(string1, sizeof(char)*newSize*100);
    string1[index1++] = ' ';
    int index2 = 0;
    while(string2[index2] != '\0')    {
        string1[index1++] = string2[index2++];
    }
    string1[index1] = '\0';
}
```

Ponteiros de ponteiros

Ponteiros de ponteiros

- Ponteiros podem apontar para outros ponteiros
  - E assim sucessivamente...
  - Cada novo nível requer um `*` a mais na declaração

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```



Fonte: http://www.cplusplus.com/doc/tutorial/pointers/

Ponteiros de ponteiros

- No exemplo anterior temos os seguintes tipos e valores
  - *c* é um *char*** com valor 8092
  - **c* é um *char** com valor 7230
  - ***c* é um *char* com valor ‘z’
- No geral, eles são equivalentes a vetores multidimensionais
- Considere a declaração seguinte:
  - `int nums[2][3] = { {16, 18, 20}, {25, 26, 27} };`

Ponteiros de ponteiros

Anotação de ponteiro	Anotação de vetor	Valor
**nums	nums[0][0]	16
*(*nums+1)	nums[0][1]	18
*(*nums+2)	nums[0][2]	20
((nums+1)	nums[1][0]	25
((nums+1)+1)	nums[1][1]	26
((nums+1)+2)	nums[1][2]	27

Ponteiros de ponteiros

```
int main () {
    int nums[2][3]  =  { {16, 18, 20}, {25, 26, 27} };
    int **numsp;
    numsp = (int**) malloc(sizeof(int*)*2);
    for(int i = 0; i < 2; ++i)
        numsp[i] = (int*)malloc(sizeof(int)*3);
    printf("\nVetor:\n");
    for(int i = 0; i < 2; ++i) {
        for(int j = 0; j < 3; ++j) {
            printf("%d - ", nums[i][j]);
            *(*(numsp+i)+j) = nums[i][j];
        }
        printf("\n");
    }
```

Ponteiros de ponteiros

```
    printf("\nPonteiro:\n");
    for(int i = 0; i < 2; ++i) {
        for(int j = 0; j < 3; ++j) {
            printf("%d - ", *(*(numsp+i)+j));
        }
        printf("\n");
    }

    for(int i = 0; i < 2; ++i)
        free(numsp[i]);
    free(numsp);
}
```

Ponteiros de ponteiros

- ➔ Podemos fazer quantas “dimensões” quisermos de ponteiros
- ➔ Inclusive para ler dinamicamente strings
- ➔ Vamos ao exemplo!

Structs

Structs [1]

- Um grupo de elementos de dados agrupados com um mesmo nome é conhecido como *estrutura de dados* ou *struct*
- Esses elementos de dados são chamados de **membros**
- Podem ter diferentes tipos e tamanhos
- Eles são muito úteis para representarmos informações complexas de uma maneira mais organizada!

Structs [1]

➔ Sintaxe de uma *struct*

```
struct nome_tipo {
tipo_membro1 nome_membro1;
tipo_membro2 nome_membro3;
tipo_membro3 nome_membro3;
.
.
} nomes_variaveis;
```

Structs [1]

- ➔ *nome_tipo* é o nome do tipo da *struct*
- ➔ *nomes_variaveis* pode ser um conjunto de identificadores válidos para as variáveis que pertencem ao tipo dessa *struct*
- ➔ Entre chaves estão a lista de membros de dados, cada um com seu tipo e identificador válido como nome.

Structs [1]

➔ Exemplo:

```
struct product {
   int weight;
   double price;
} apple, banana, melon;
```

Structs [1]

- Assim declaramos uma *struct* de nome *product*
- A *struct* possui dois membros
  - *peso* e *preço*
  - Cada um com seu tipo
- Essa declaração cria um novo tipo: *product*
- Foram criadas 3 variáveis desse tipo:
  - *apple, banana* e *melon*
- Porém, podemos declarar variáveis de outro jeito:

Structs [1]

```
struct product {
  int weight;
  double price;
} ;

struct product apple;
struct product banana, melon;
```

Structs [1]

- Uma vez que *product* é declarado, ele pode ser usado como um tipo comum, desde que seja colocada a palavra-chave *struct* antes do nome da *struct*
- Nota: em C++ é possível declarar variáveis sem o uso da palavra-chave *struct*. Cuidado para não confundir!
  - Existe um jeito de fazer isso em C, vamos ver mais adiante nesta aula

Structs [1]

- ➔ Uma vez criada uma variável do tipo da *struct*, é possível acessar diretamente seus membros.
- ➔ Para isso, é só colocar um ponto “.” entre o nome da variável e seu membro:

```
apple.weight    banana.weight    melon.weight
apple.price     banana.price     melon.price
```

Structs [1]

```
struct manga_t
{
    char title[30];
    int year;
} mine, yours;

void printManga (struct manga_t manga)
{
    printf("\nManga Title:\n%s", manga.title);
    printf("\nManga Year of Release:\n%d\n", manga.year);
}
```

Structs [1]

```c
int main ()
{
    strcpy(mine.title, "One Piece");
    mine.year = 1997;

    scanf("%s", yours.title);
    scanf("%d", &yours.year);

    printManga(mine);
    printManga(yours);
    return 0;
}
```

Structs [1]

- ➔ É possível observar que as variáveis yours.year é um inteiro e yours.name é uma *string*, ambos totalmente válidos
- ➔ Além disso, as variáveis *mine* e *yours* são, por si só, também variáveis (do tipo manga_t)
- ➔ Por isso, é possível passá-las como argumentos de função, verificar seu tamanho entre outras coisas

Structs [5]

➔ É possível inicializar uma struct com o uso de chaves, passando os valores na ordem exata da declaração dos argumentos

```
struct manga_t other = {"Yu Yu Hakusho", 1990};
```

Structs [1]

➔ Como *structs* são tipos de dados, elas também podem ser usadas como tipos de vetores.

```
struct manga_t {
    char title[30];
    int year;
} manga[2];
```

Structs [1]

- ➔ É possível observar que as variáveis yours.year é um inteiro e yours.name é uma *string*, ambos totalmente válidos
- ➔ Além disso, as variáveis *mine* e *yours* são, por si só, também variáveis (do tipo manga_t)
- ➔ Por isso, é possível passá-las como argumentos de função, verificar seu tamanho entre outras coisas

Structs [1]

- Também é possível fazer ponteiros de *structs*!
- As operações seguem o mesmo princípio de qualquer outra variável
- Mas existe uma diferença:
  - Para acessar os valores de uma variável de ponteiro de *struct* usa-se o operador “->” no lugar de “.”
  - Ou *(*pstruct).value*
- Os dois trechos de código a seguir são equivalentes:

Structs [1]

```c
struct manga_t amanga;
struct manga_t *pmanga;
pmanga = &amanga;

strcpy(pmanga->title, "Bleach");
pmanga->year = 2001;

printManga(pmanga);
```

```c
struct manga_t amanga;
struct manga_t *pmanga;
pmanga = &amanga;

strcpy((*pmanga).title, "Bleach");
(*pmanga).year = 2001;

printManga(pmanga);
```

Structs [1]

- O operador de seta “->” **NÃO** é a mesma coisa que
  - **pmanga.title*
    - Este é a mesma coisa que **(pmanga.title)*
    - Ambos acessariam o valor apontado pelo ponteiro hipotético *title* da estrutura *pmanga*
      - Mas *title* não é do tipo ponteiro!

Structs [1]

- É possível fazer estruturas aninhadas!
  - Ou seja, uma *struct* que tem outra *struct* como variável
- Com isso é possível criar diversos tipos complexos de dados com uma organização de código muito melhor!

Structs [1]

```c
struct manga_t{
    char title[40];
    int year;
} manga;

struct anime_t{
    struct manga_t manga;
    char studio[30];
    int year;
} anime;

void printManga (struct manga_t manga);
void printAnime (struct anime_t anime);
```

Structs [1]

```
int main () {

    strcpy(manga.title,
       "Kono Subarashii Sekai ni Shukufuku o!");
    manga.year = 2012;

    strcpy(anime.studio, "Studio Deen");
    anime.year = 2016;

    anime.manga = manga;
    printAnime(anime);
    return 0;
}
```

Structs [1]

```
void printManga (struct manga_t manga){

    printf("\nManga Title:\n%s\n", manga.title);
    printf("\nManga Year of Release:\n%d\n", manga.year);
}

void printAnime (struct anime_t anime){

    printManga(anime.manga);

    printf("\nAnime Studio:\n%s\n", anime.studio);
    printf("\nAnime Year of Release:\n%d\n", anime.year);
}
```

Structs [2]

- Também é possível realizar alocação dinâmica de variáveis de *structs*
- O processo é exatamente igual com qualquer outra variável
  - Colocando o tipo como struct
- Veja o trecho de código a seguir:

Structs [2]

```
struct manga_t *pmanga;

pmanga = (struct manga_t*) malloc(sizeof(struct manga_t)*1);

strcpy(pmanga->title, "Kubera");
pmanga->year = 2010;

printManga(pmanga);
```

Structs [3, 4]

- Ok... é bem chato ficar digitando "*struct*" toda hora, né?
- Em C é possível dar um "apelido" (*alias*) para um tipo
  - Isso faz com que um tipo seja identificado com um nome diferente
- É só usar a palavra-chave *typedef* seguida pelo tipo e seu novo nome
  - Pode-se usar em tipos primitivos ou *structs*, por exemplo

Structs [3, 4]

```
typedef char C;

typedef unsigned int WORD;

typedef char * pChar;

typedef char field [50];

typedef struct {
    char title[30];
    int year;
} manga_t;
```

Structs [3, 4]

```
C mychar, anotherchar, *ptc1;
WORD myword;
pChar ptc2;
field name;
manga_t manga;
```

FILE

FILE [6, 7]

- Existe uma *struct* muito importante chamada FILE, que é essencial para a manipulação de arquivos
- Os conteúdos internos dela não são muito relevantes para a maioria dos programadores, mas vamos apresentar a seguir:

Structs [7]

```
typedef struct
{
 short level;
 short token;
 short bsize;
 char fd;
 unsigned flags;
 unsigned char hold;
 unsigned char *buffer;
 unsigned char *curp;
 unsigned istemp;
}FILE ;
```

Member	Use / Function
level	Fill / Empty level of Buffer
token	Validity Checking
bsize	Buffer size
fd	File descriptor using which file can be identified
flags	File status flag
hold	ungetc character if no buffer space available
buffer	Data transfer buffer
curp	Current active pointer
istemp	Temp. File indicator

Fonte: [7]

FILE [6]

- Os conteúdos de FILE são feitos para serem acessados somente por funções padrões de C, como as da *stdio.h*
- A alocação de memória dele é feita automaticamente
  - Através de funções como *fopen()*
- Assim como a liberação de memória após a chamada de *fclose()*
- Além disso, *stdin*, *stdout* e *stderr* são do tipo *FILE*

FILE [6]

```
int main() {
   FILE *pFile;
   char buffer [100];
   pFile = fopen ("aizen.txt" , "r");
   if (pFile == NULL) perror ("Error opening file");
   else {
     while ( ! feof (pFile) ) {
       if ( fgets (buffer , 100 , pFile) == NULL ) break;
       fputs (buffer , stdout);
     }
     fclose (pFile);
   }
   return 0;
}
```

FILE [6]

- *fopen(filename, mode)* é responsável por abrir um arquivo "*filename*" e retornar um ponteiro para *FILE*
  - Existem diversos modos de abertura de arquivos, vamos ver mais exemplos aula que vem
- *feof(stream)* é uma função que retorna 0 enquanto o fim do arquivo não é encontrado
  - Este fim é representado pela macro EOF

FILE [6]

- *fgets(str, num, stream)*
  - é uma função utilizada para ler um vetor de *chars* "*str*" do arquivo "*stream*", até um número máximo determinado de caracteres "*num*"
- *fputs(str, stream)*
  - é uma função que escreve uma string *str* para um arquivo *stream*
- *fclose(stream)* fecha o arquivo *stream* e libera memória

Documentação

Documentação

- Um código bem claro e documentado é ESSENCIAL para qualquer programador.
- Você com quase toda certeza trabalhará junto com outros programadores e terão de compartilhar códigos entre si.
- Você não vai querer receber um código indecifrável!
- Nem o seu colega!
- Nem os professores e monitores corrigindo os trabalhos!

Documentação

- ➔ Existem vários guias e boas práticas sobre como elaborar um bom código.
- ➔ Algumas dessas diretrizes mudam de acordo com a convenção de cada linguagem.
- ➔ Mas a maioria é igual para todos.
- ➔ Tudo pode se resumir a: seja claro e tenha bom senso.

Documentação

- Variáveis, funções e arquivos devem ter nomes intuitivos!
  - Evite usar “x”, “aux”, “var1”, “var2”
  - Exceto nos casos de contadores de laços, em que é convenção usar “i”, “j”, “k”, etc... ou em caso de variáveis temporárias em funções que realmente vão receber algo não muito claro.
    - Mesmo assim, deixe o mais intuitivo possível

Documentação

- Faça comentários em qualquer trecho mais complexo do código.
- Tente sempre fazer um comentário antes de funções explicando, resumidamente, o que ela faz, o que são seus parâmetros e seu retorno.
- Para trechos dentro das funções, faça comentários *inline* ou antes do trecho caso ele seja complexo e você sinta que deveria explicá-lo

Documentação

- Mantenha seu código identado corretamente (use Tab)
- Tenha consistência na indentação!
  - Existem vários padrões, vários certos, mas tente usar sempre o mesmo.
- Evite linhas muito grandes de código. Quebre em linhas menores sempre que possível!

Makefile

Makefile

- ➔ Arquivo que define regras para compilação de projetos de software.
- ➔ O programa “make” interpreta o conteúdo do arquivo e executa as regras contidas nele.
- ➔ Tem como fazer no Windows com o NMake, mas é um pouco mais complicado.
- ➔ Vamos ver no Linux (no WSL pra ser mais preciso)

Makefile

- Pode evitar a compilação de arquivos desnecessários
  - Descobre qual arquivo foi alterado e compila apenas a biblioteca necessária (vamos ver o que é biblioteca mais pra frente)
- Automatiza tarefas como limpeza de vários arquivos temporários

Makefile

```
#Comentarios

all: main

main: main.o

	gcc -o main main.o

main.o: main.c

	gcc -o main.o -c main.c -Werror -Wall

clean:

	rm -rf *.o

cleanmain: clean

	rm -rf main
```

Arquivos .h

Arquivos .h

- São arquivos “header”
  - Ajudam na organização do código
  - De maneira geral, é onde ficam declarações de typedef e cabeçalhos de função
  - São acompanhados de um “.c” que contém a implementação de tais cabeçalhos e definições das structs

Arquivos .h

- Na prática, o compilador copia e cola o código dos arquivos .h aonde eles são incluídos
  - `include “arquivo.h”`
- Precisa tomar cuidado com ordem das inclusões, caso tenham dependência cíclica

Arquivos .h

- Ele pode ser definido múltiplas vezes, portanto, usamos um “if” para verificar se já foi construído
- `#ifndef ARQUIVO_H`
  - `#define ARQUIVO_H`
- `#endif`

Referências

- ZIVIANI, N. Projeto de Algoritmos, Thomson, 2a. Edição, 2004.