Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Algoritmos e Estruturas de Dados I

Breve revisão de ponteiros e estruturas dinâmicas

Mirtha Lina Fernández Venero
Sala 529-2, Bloco A
mirtha.lina@ufabc.edu.br
http://professor.ufabc.edu.br/~mirtha.lina/aedi.html

22 de fevereiro de 2019

Agenda

Exercício aula hoje

Escreva um programa que leia duas sequências de letras terminadas no caractere `'\n'`, uma após a outra. Seu programa deve imprimir 0 se as duas sequências genéticas são complementares e 1 em outro caso, usando a menor quantidade de memória possível.

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Exercício aula hoje

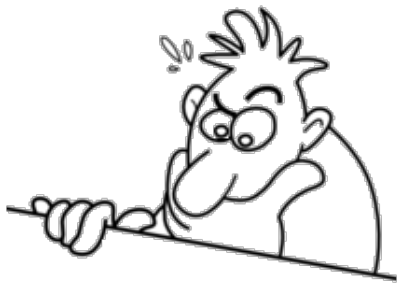
Escreva um programa que leia duas sequências de letras terminadas no caractere '\n', uma após a outra. Seu programa deve imprimir 0 se as duas sequências genéticas são complementares e 1 em outro caso, usando a menor quantidade de memória possível.

```
4   int lerSeq(char seq[]){
5     char letra; int sizeSeq = 0;
6     do {
7       scanf("%c", &letra);
8       if ( ! isBase(letra) )
9         return -1;
10      seq[sizeSeq] = letra;
11      sizeSeq ++;
12    } while(letra != '\n');
13    return sizeSeq;
14  }
```

Exercício aula hoje

Escreva um programa que leia duas sequências de letras terminadas no caractere '\n', uma após a outra. Seu programa deve imprimir 0 se as duas sequências genéticas são complementares e 1 em outro caso, usando a menor quantidade de memória possível.

```
4   int lerSeq(char seq[]){
5     char letra; int sizeSeq = 0;
6     do {···
12    } while(letra != '\n');
13    return sizeSeq;
14  }
```

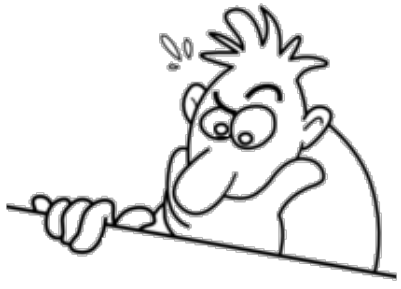


Como saber quantidade de memória que debemos reservar para armazenar os dados? Não dá pra saber! Precisamos reservar memória dinamicamente.

Exercício aula hoje

Escreva um programa que leia duas sequências de letras terminadas no caractere '\n', uma após a outra. Seu programa deve imprimir 0 se as duas sequências genéticas são complementares e 1 em outro caso, usando a menor quantidade de memória possível.

```
4   int lerSeq(char seq[]){
5     char letra; int sizeSeq = 0;
6     do {···
12    } while(letra != '\n');
13    return sizeSeq;
14  }
```



Como saber quantidade de memória que debemos reservar para armazenar os dados? Não dá pra saber! Precisamos reservar memória dinamicamente. **Como, onde e quando?**

Ponteiros

C permite o acesso e o gerenciamento dinâmico da memória do programa, usando o tipo de dado ponteiro.

- Uma variável ponteiro armazena um endereço de memória. Para **declarar** um ponteiro usa-se o operador `*` após o tipo dos dados cujos endereços ele vai armazenar (tipo base)

```
int * ip; char *cp = "Hello world";
```

- Para obter o **endereço** duma variável usa-se o operador `&`

```
int i = 100; pi = &i;
```

- Para obter o **dado** apontado pelo ponteiro (dereferenciar) também usa-se o operador `*`

```
printf("*ip = %d *cp = %c", *ip, *cp);
```

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

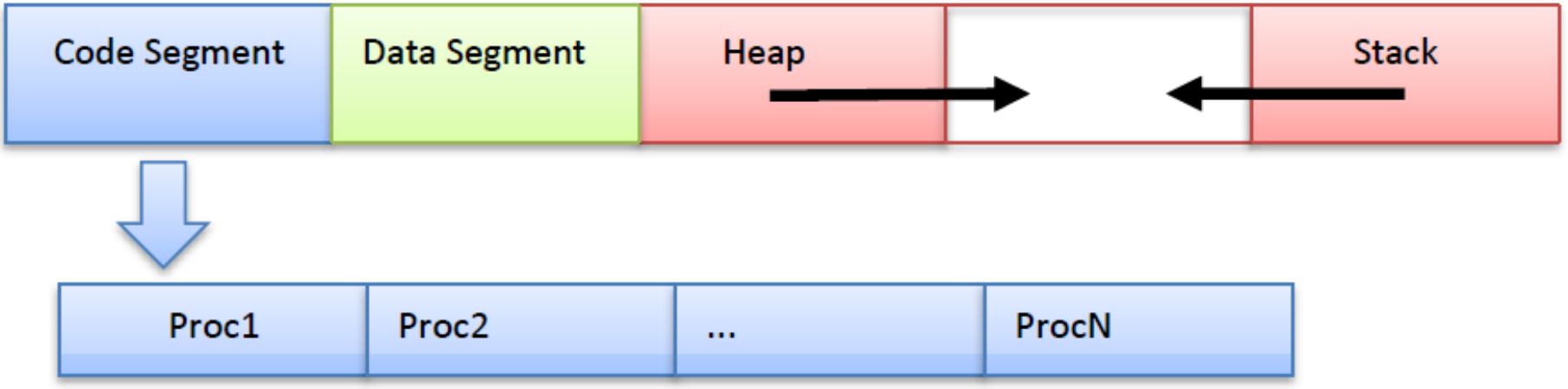
Ponteiros

```c
#include <stdio.h>

int main() {

   int  i = 20; char c='A';
   int  *ip;                        /* pointer  declaration */
   ip = &i;                         /* stores address of i in pointer */

   printf("Addresses of ip, c and i:%p - %p - %p\n", &ip,&c,&i);
   printf("Sizes of ip, c, i: %d - %d - %d\n",
            sizeof(ip), sizeof(i), sizeof(c));

   printf("Values stored in pi, c, i : %p - %x - %x\n", ip, c, i );

   /* access the value using the pointer */
   printf("Value of *ip variable: %X\n", *ip );

   return 0;
}
```

Variável	ip				c	i	
Valor	0x00007ffe	0x22bc368c	0x00000041			0x00000014	0x????????
Endereço	0x7ffe22bc3680	0x7ffe22bc3684	0x7ffe22bc3688			0x7ffe22bc368c	0x7ffe22bc3690

Ponteiros e Ambientes de Execução

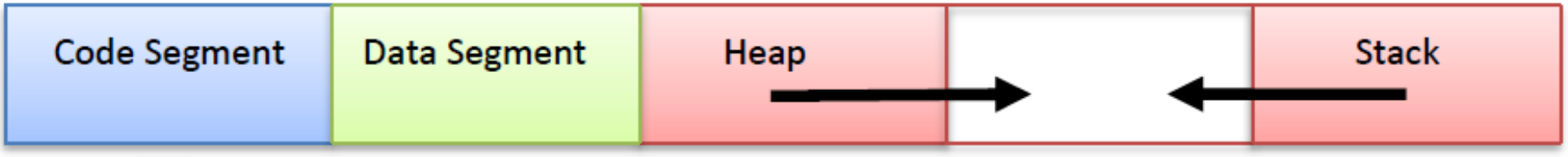
Como são organizados os recursos de um programa (código, variáveis, objetos de dados, funções) durante a execução?



- **Segmento de código**: Armazena o código das funções
- **Segmento de dados**: Armazena os dados globais e estáticos (tamanho e endereço não variam e são calculados em tempo de compilação; e.g. constantes, variáveis globais, etc)

Ponteiros e Ambientes de Execução

Como são organizados os recursos de um programa (código, variáveis, objetos de dados, funções) durante a execução?

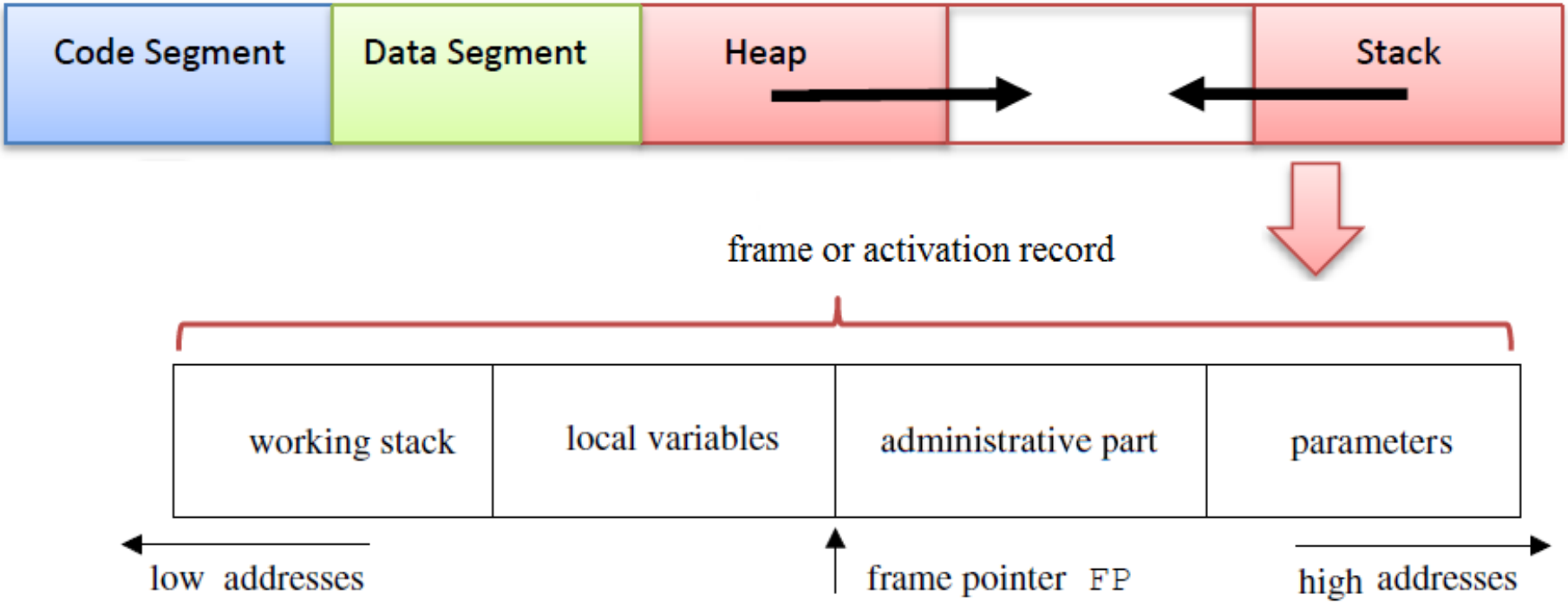


Stack e **Heap**: Armazenam dados alocados dinamicamente

- **Stack**: dados com tempo de vida conhecido em tempo de compilação porém limitado à função onde foram definidos, e.g. parâmetros de funções, variáveis locais, valor de retorno
- **Heap**: dados cujo tempo de vida não é conhecido em tempo de compilação e pode ir além da função onde foram criados

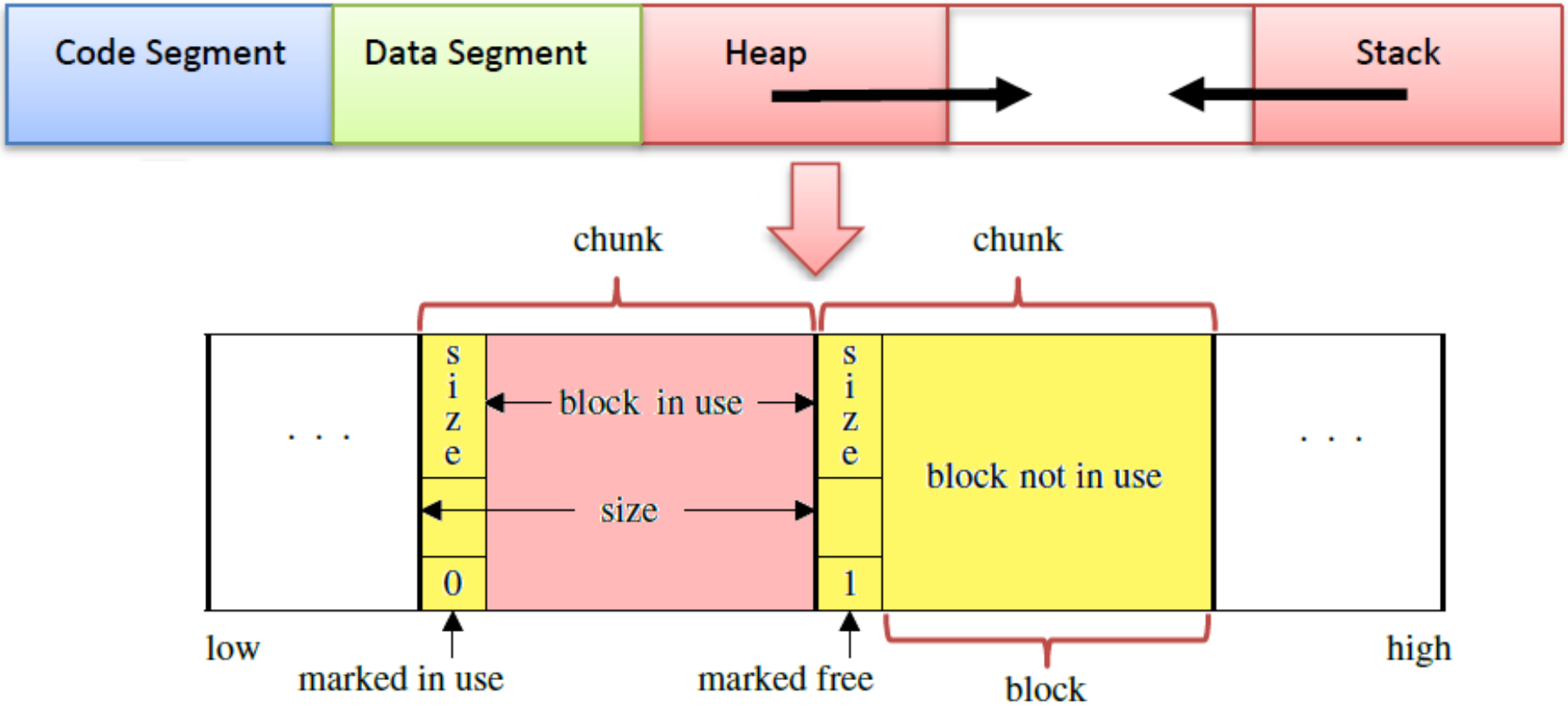
Organização do Stack (Pilha de Execução)

Os dados necessários para executar uma função são armazenados en fragmentos de memória chamados **registros de ativação** (**frames or activation records**)



Organização do Heap

Fragmentos de memória (*chunks*) que contém uma área administrativa usada pelo gerenciador do heap e outra usada pelo programador (*block*)



Gerenciamento do Heap

1. **Manual**: O programador solicitar explicitamente a alocação e liberação dos dados (e.g. C, C++, Pascal)
   - \+ Simples.
   - \+ **O programador pode usar a memória como quer!**
   - \- **O programador pode usar a memória como quer!**

2. **Automático**: O compilador gera código para solicitar a alocação de dados no heap e o gerenciador de memória é responsável pela liberação deles quando não são usados (lixo-garbage, e.g. Java, C#, PHP, JavaScript)
   - \- O compilador precisa identificar os apontadores.
   - \- A coleta de lixo não é simples e consome tempo.
   - \- **O programador não pode usar a memória como quer!**
     $\Longrightarrow$ (talvez) pior desempenho
   - \+ **O programador não pode usar a memória como quer!**
     $\Longrightarrow$ (com certeza) maior segurança

Ponteiros e Alocação dinâmica de memória

- É possível alocar e liberar memória em tempo de execução usando as funções `malloc`, `calloc`, `realloc` e `free`
- A constante `NULL` (que representa um pointero nulo `==0`) é retornada caso não houver memória disponível pra ser alocada

```c
#include <stdlib.h>

int main(void)
{
    int *p1 = malloc(4*sizeof(int));  // allocates an array of 4 int
    int *p2 = malloc(sizeof(int[4])); // same

    if(p1) {
        for(int n=0; n<4; ++n) // populate the array
            p1[n] = n*n;
    }

    free(p1);
    free(p2);
}
```

Ponteiros e Vetores

- A variável de um vetor armazena o endereço do primeiro elemento ⇒ é possível acessar aos elementos dum vetor usando ponteiros
- "Aritmética de ponteiros" com os operadores `++`, `--`, `+`, `-`

```
4   int arrSize = 5, i;
5   double arr[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
6   double *p = arr, *q = p;
7
8   printf( "\nArray values using pointers\n");
9   for ( i = 0; i < arrSize; i++ ) {
10    printf("\nAddress of arr[%d] = %p\n", i, q );
11    printf("arr[%d] = %f =  %f\n",  i, *(arr + i), arr[i]);
12    printf("*(p + %d) = %f =  %f\n",  i, *(p + i), *q );
13    q++;
14   }
```

Erros comuns com ponteiros

- Não inicializar ponteiros ou criar alias. Dereferenciar ponteiros nulos ou com referencias não válidas (*dangling references*).
- Não liberar memória que não vai ser mais usada. Quando não puder ser mais acessada vira **lixo** (*garbage or memory leak*)

```
int *p1; p1 = malloc(sizeof(int)); *p1 = 4;
int *p2 = p1;                       //alias of p1
int *p3; p3 = malloc(sizeof(int));

free(p1);                          // p2 is now dangling

printf("%d\n", *p2);               // print ????

p3 = malloc(sizeof(int));          // former p3 is now garbage
```

- Abusar da aritmética de ponteiros e acessar memória fora dos limites dum vetor ou estrutura
- Liberar memória que não foi alocada ou já foi liberada

Gerenciamento do Heap

1. **Manual**: O programador solicitar explicitamente a alocação e liberação dos dados (`malloc`, `calloc`, `realloc` e `free`)
   + Simples.
   - **O programador pode usar a memória como quer! Isto pode conduzir a erros graves e difíceis de detectar!!!**


```
int *a = new int[46]; ...
*(a+64) = ...; a += 4; free(a); ...;
a[2] = ...; a = &x; ...; free(a);

// C: Undefined behavior
// Java:  No pointer arithmetic nor free +
//        Automatic Garbage collection
```

http://jayesh.profitfromprices.com/Fun_Best_Computer_Cartoons.htm

Ponteiros e Funções

Relembrando: A transferência de parâmetros em C é por valor, i.e. somente o valor é do argumento na chamada é transferido para o parâmetro formal da função e esse valor não pode ser mudado.

- Os ponteiros permitem modificar o conteúdo dum parâmetro. Como argumento deve ser transferido o endereço duma variável
- Os ponteiros podem ser retornados. Isso, permite criar um vetor ou estrutura dentro duma função e retornar seu endereço

```
int* swapAlloc(int *xa, int *ya) {
    int temp = *xa;
    *xa = *ya;
    *ya = temp;
    return malloc(*xa*sizeof(int));
}

int main() {
    int x, y;
    scanf("%d", &x);
    scanf("%d", &y);
    int *arr = swapAlloc(&x, &y);
    printf("x = %d, y = %d", x, y);
    return 0;
}
```

Exercício aula hoje

Escreva um programa que leia duas sequências de letras terminadas no caractere `'\n'`, uma após a outra. Seu programa deve imprimir 0 se as duas sequências genéticas são complementares e 1 em outro caso, usando a menor quantidade de memória possível.

Como saber quantidade de memória que debemos reservar para armazenar os dados? Não dá pra saber! Precisamos reservar memória dinamicamente **usando ponteiros.**

```
5  #define SEQTYPE int
6
7  SEQTYPE lerBase();
8  SEQTYPE *lerSeq (int *seqSize);
9  int cmpSeqs(SEQTYPE *seq1, int seqSize1);
```

Exercício aula hoje

Escreva um programa que leia duas sequências de letras terminadas no caractere `'\n'`, uma após a outra. Seu programa deve imprimir 0 se as duas sequências genéticas são complementares e 1 em outro caso, usando a menor quantidade de memória possível.

Como saber quantidade de memória que debemos reservar para armazenar os dados? Não dá pra saber! Precisamos reservar memória dinamicamente **usando ponteiros.**

Estratégia 1: Começar com um tamanho de vetor minimo (pex. 10) e, se não for suficiente, dobrar o tamanho.

Exercício aula hoje

Escreva um programa que leia duas sequências de letras terminadas no caractere `'\n'`, uma após a outra. Seu programa deve imprimir 0 se as duas sequências genéticas são complementares e 1 em outro caso, usando a menor quantidade de memória possível.

Como saber quantidade de memória que debemos reservar para armazenar os dados? Não dá pra saber! Precisamos reservar memória dinamicamente **usando ponteiros.**

Estratégia 1: Começar com um tamanho de vetor minimo (pex. 10) e, se não for suficiente, dobrar o tamanho. **Desvantagens???**

Exercício aula hoje

Escreva um programa que leia duas sequências de letras terminadas no caractere `'\n'`, uma após a outra. Seu programa deve imprimir 0 se as duas sequências genéticas são complementares e 1 em outro caso, usando a menor quantidade de memória possível.

Como saber quantidade de memória que debemos reservar para armazenar os dados? Não dá pra saber! Precisamos reservar memória dinamicamente **usando ponteiros.**

Estratégia 1: Começar com um tamanho de vetor minimo (pex. 10) e, se não for suficiente, dobrar o tamanho. **Desvantagens???**

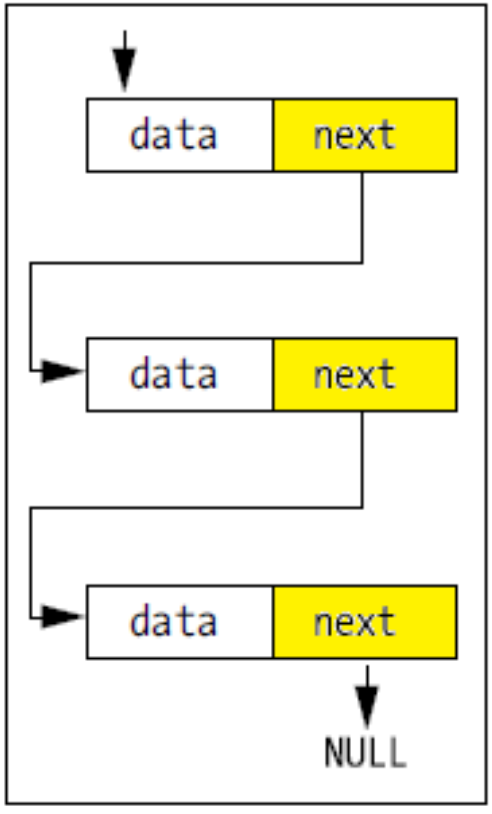
Estratégia 2: usar uma estrutura enlaçada

Agenda

Estruturas enlaçadas (ligadas, encadeadas)

Estrutura de dados que permite armazenar uma sequência ou conjunto de elementos de dados do mesmo tipo de forma não consecutiva na memória.

- Cada elemento de dado (*nó*) contem uma referência a outro elemento da sequência
- É necessário armazenar uma referência ao primeiro elemento e distinguir o último
- Podem ser estáticas (usando vetores) ou dinâmicas (ponteiros); lineares ou não
- Variantes dependem das operações de busca, inserção e remoção: listas, pilhas, filas, árvores, grafos



Ponteiros e Estruturas

- É possível definir ponteiros a qualquer outro tipo, em particular estruturas

Live Demo

```c
#include <stdio.h>
#include <string.h>

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books *book );
```

Ponteiros e Estruturas

- É possível definir ponteiros a qualquer outro tipo, em particular estruturas
- Para acessar o conteúdo de um campo da estrutura através dum ponteiro pode ser usado o operador `->`. Por exemplo `book->title` é equivalente a escrever `(*book).title` (não `*book.title`)

```
void printBook( struct Books *book ) {

   printf( "Book title : %s\n", book->title);
   printf( "Book author : %s\n", book->author);
   printf( "Book subject : %s\n", book->subject);
   printf( "Book book_id : %d\n", book->book_id);
}
```

- para evitar escrever o `struct` toda vez que for usado o tipo deve-se usar um `typedef`, e.g.
`typedef struct Books *PBooks; PBooks book1;`

Estruturas lineares enlaçadas

Permitem armazenar uma sequência ou conjunto de elementos de dados do mesmo tipo de forma não consecutiva na memória.

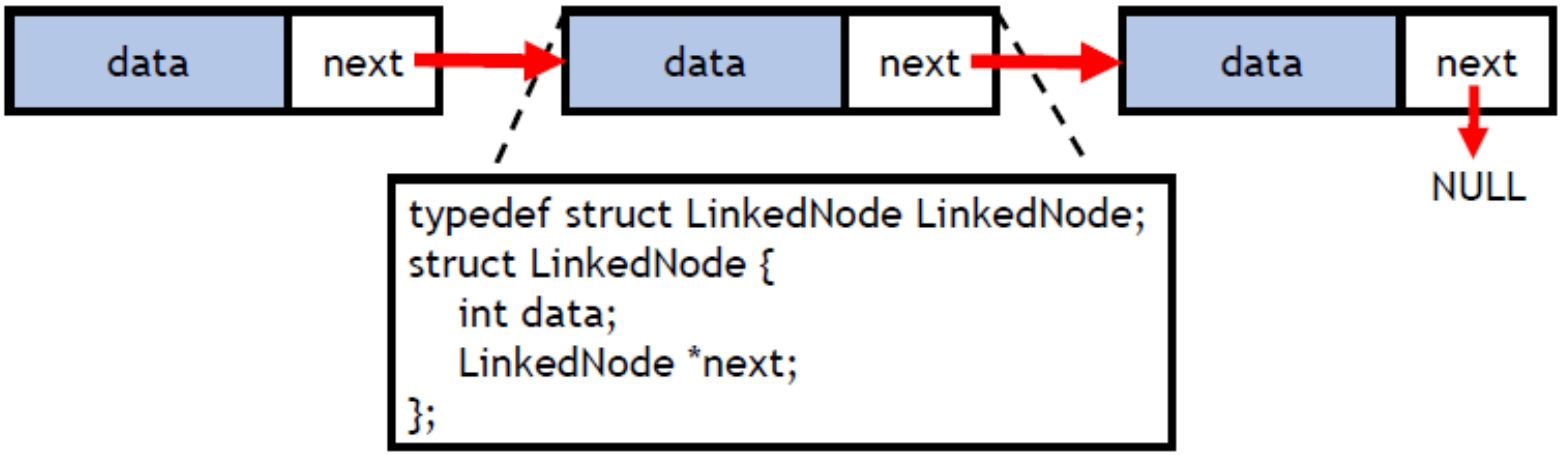


Figura das aulas Prof. Paulo Pisani

Estruturas lineares enlaçadas

Permitem armazenar uma sequência ou conjunto de elementos de dados do mesmo tipo de forma não consecutiva na memória.

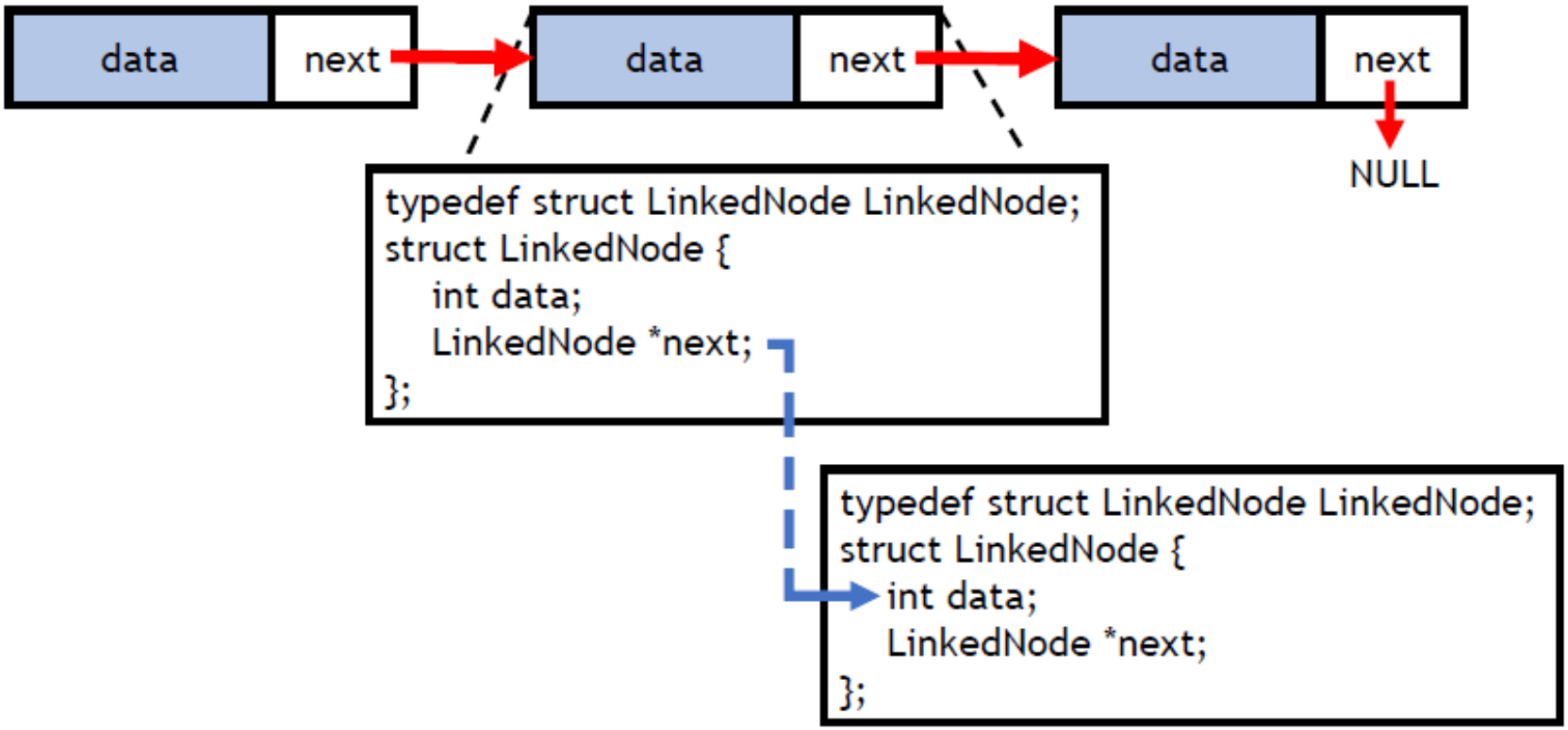
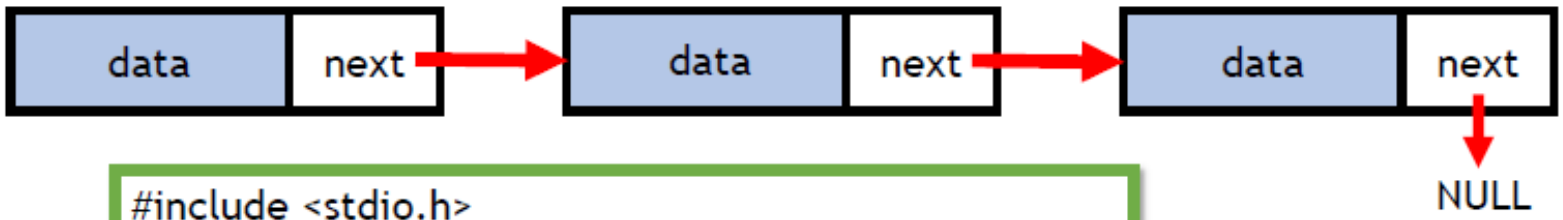


Figura das aulas Prof. Paulo Pisani

Estruturas lineares enlaçadas

Permitem armazenar uma sequência ou conjunto de elementos de dados do mesmo tipo de forma não consecutiva na memória.



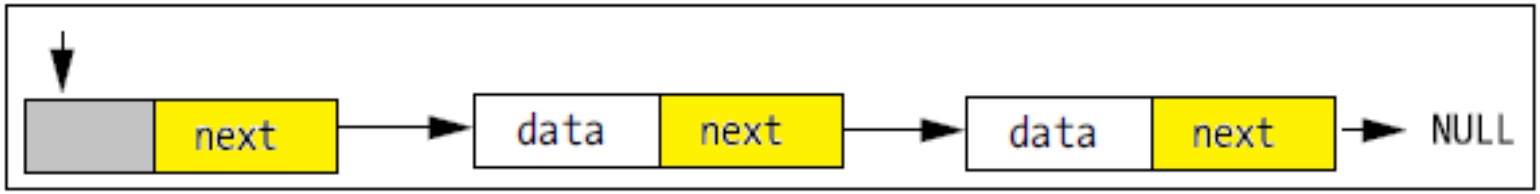

```
#include <stdio.h>
#include <stdlib.h>

typedef struct LinkedNode LinkedNode;
struct LinkedNode {
    int data;
    LinkedNode *next;
};

    ...

    LinkedNode *first = NULL;

    ...
```

O ponteiro para o primeiro item deve ser salvo.

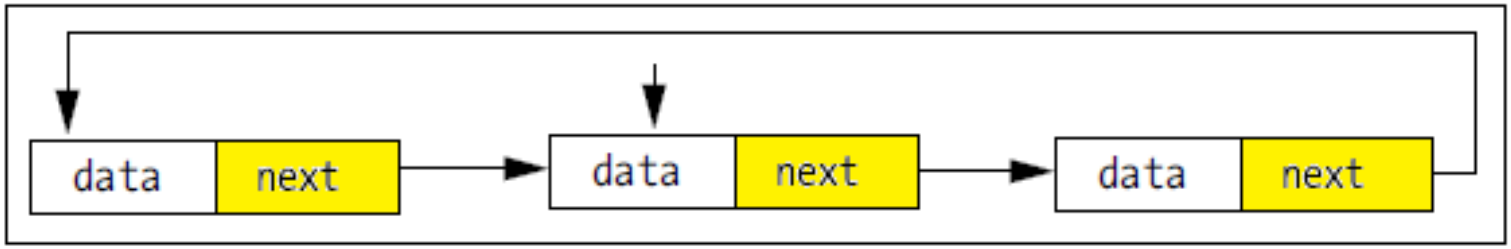
Figura das aulas Prof. Paulo Pisani

Variantes de listas enlaçadas

- **Lista com cabeça**: O primeiro nó é usado para armazenar dados que não pertencem ao conjunto (e.g. tamanho da lista). **Vantagem**: não precisa checar se a lista está vazia.

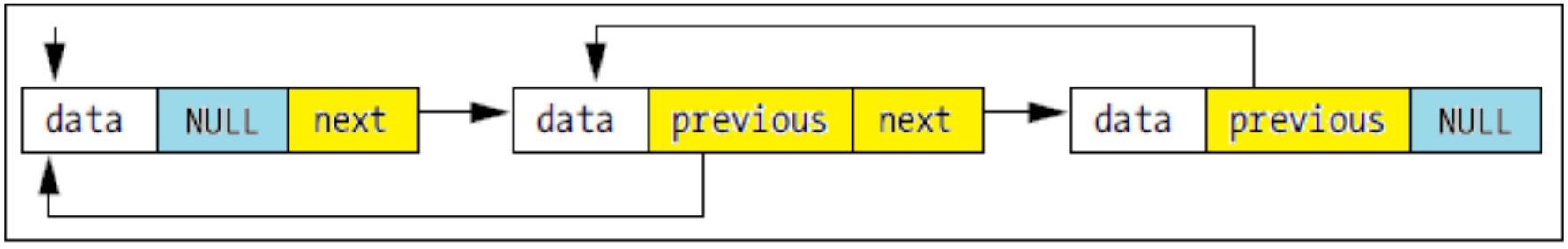


- **Lista circular**: O último nó aponta ao primeiro
Vantagem: Usada para gerenciar recursos de forma tal que nenhum usuário use mais duma vez um recurso antes que todos os outros tenham usado (round robin algorithm).



Lista duplamente enlaçada (doubly or two-way linked)

Cada nó tem referências ao nó anterior e seguinte da lista



Vantagens:

- A lista pode ser percorrida em ambas direções
- Um nó pode ser removido em tempo constante fornecendo seu endereço.

Desvantagem:

- Precisa de mais espaço

Universidade Federal do ABC

CMCC
Centro de Matemática, Computação e Cognição

Agenda

Função `main` e seus argumentos

- Não pode ser usada em nenhum lugar do programa
- Pode retornar `void` or `int`; não precisa ter `return` no corpo.
- Pode ter ou não parâmetros. Quando o programa é chamado na linha de comandos os argumentos são cadeias

```c
#include <stdio.h>

int main( int argc, char *argv[] )  {

   printf("Program name %s\n", argv[0]);

   if( argc == 2 ) {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 ) {
      printf("Too many arguments supplied.\n");
   }
   else {
      printf("One argument expected.\n");
   }
}
```

Agenda

Bibliografia e Links úteis

- **C How to Program**, Paul J. Deitel & Harvey Deitel, 8th ed. 2015
- **Beginning C**, Ivor Horton, 5th ed. 2013
- **C Programming Language**, Brian W. Kernighan & Dennis Ritchie. 1988
- **Essential C**, Nick Parlante. 2003
  http://cslibrary.stanford.edu/101/EssentialC.pdf
- **Material extra sobre Ponteiros em C**, Prof. Paulo Pisani, http://professor.ufabc.edu.br/~paulo.pisani/2019Q1/AEDI/index.html

Universidade Federal do ABC

CMCC Centro de Matemática, Computação e Cognição

Agenda

Exercícios para casa

1. Resolva o exercício em sala usando a estratégia 1, i.e. comece reservando memória para uma cadeia genética de 10. Toda vez que ler uma letra se o vetor estiver cheio, crie um novo vetor com o dobro do tamanho do vetor atual, copie os dados no vetor criado e coloque letra lida no. Não esquecer checar se há memoria disponível e não deixar lixo!

2. Escreva funções C para realizar as seguintes operações sobre vetores dinâmicos de tipo `double`.

   - Concatenar dois vetores
   - Dividir um vetor em duas metades. Se o número de elementos for ímpar, a segunda metade terá tamanho ímpar.
   - Inserir um elemento na posição i do vetor
   - Remover o elemento na posição i do vetor