UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

Programação e Desenvolvimento de Software 2

Armazenamento de dados em memória

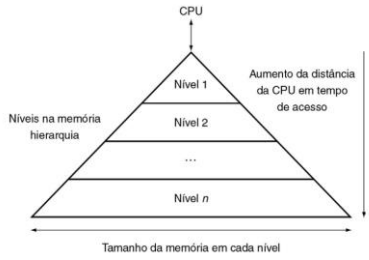
Prof. Douglas G. Macharet
douglas.macharet@dcc.ufmg.br

DCC
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

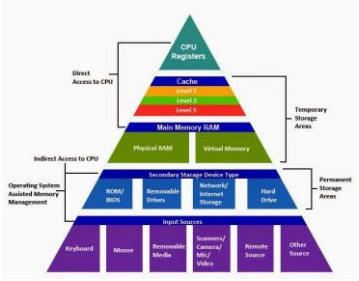
Hierarquia de memória

- Memória
  - Estrutura interna que armazena informações
- Memória principal
  - DRAM (Dynamic Random-Access Memory)
  - Armazenamento temporário
- Memória secundária
  - Tecnologias Magnéticas e Ópticas
  - Não voláteis

Hierarquia de memória



Hierarquia de memória



Hierarquia de Memória

- Corpo humano (inspiração?)
  - Lembranças recentes
    - Memórias menores, curta duração
  - Lembranças mais antigas
    - Memórias de maior capacidade, longa duração
- Princípio da localidade
  - Temporal
  - Espacial

Hierarquia de memória

Princípio da localidade – Temporal

- Dado acessados recentemente têm mais chance de serem usados novamente que dados usados há mais tempo
- Exemplo
  - Comandos de repetição
  - Funções
- Manter os dados e instruções usados recentemente no topo da Hierarquia

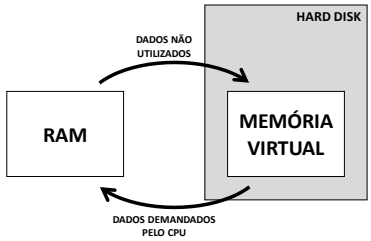
Hierarquia de memória
Princípio da localidade – Espacial

- Probabilidade de acesso maior para dados e instruções em endereços próximos àqueles acessados recentemente
- Exemplo
  - Acesso às posições de um vetor
- Variáveis são armazenadas próximas uma às outras, e vetores e matrizes armazenados em sequência de acordo com seus índices

Hierarquia de memória
Memória virtual

- Maior demanda da memória principal
  - Programas cada vezes maiores
  - Queda no custo não teve o mesmo ritmo
- Como resolver esse problema?
- Memória virtual
  - Utilizar a Memória Secundária (HD) como uma extensão da Memória Principal
  - Busca hierárquica pela informação

Hierarquia de memória
Memória virtual



Alocação de memória

- Segmentos da memória
  - Text/code
    - Guarda o código compilado do programa
  - Stack (pilha)
    - Espaço que variáveis dentro de funções são alocadas
  - Heap
    - Espaço mais estável de armazenamento
    - Programa aloca e desaloca porções de memória do heap durante a execução

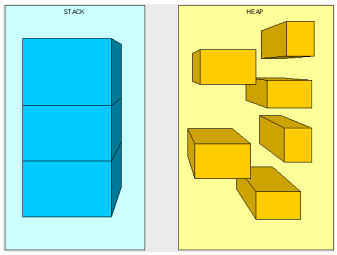
Alocação de memória
Stack (pilha)

- Porção contígua de memória
  - Incrementado toda vez que um certo método é chamado, esvaziado quando o ele é finalizado
- LIFO (last-in-first-out)
  - Último elemento a entrar é o primeiro a sair
- Não é necessário gerenciar manualmente
- Possui limites no seu tamanho

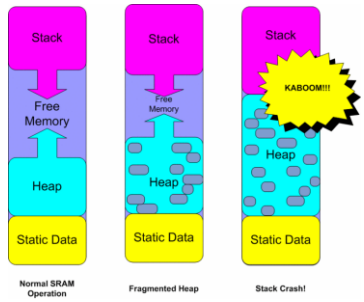
Alocação de memória
Heap

- *Pool* de memória de propósito geral
- Não impõe um padrão de alocação
  - Fragmentação ao longo do tempo
- Gerenciamento explícito
  - Alocação/desalocação
- "Não" possui um limite de tamanho

Alocação de memória



Alocação de memória



Alocação de memória

- Tipos de alocação
  - Estática
  - Dinâmica
- Estática
  - O espaço de memória alocado para as variáveis é reservado no início da execução
  - Pode ser acessado, mas não alterado depois

Alocação estática
Exemplo 1

```c
#include <stdio.h>

double multiplyByTwo(double input) {
  double twice = input * 2.0;
  return twice;
}

int main() {
  int age = 30;
  double salary = 12345.67;
  double myList[3] = {1.2, 2.3, 3.4};

  printf("Double is %.3f\n", multiplyByTwo(salary));

  return 0;
}
```

https://goo.gl/sy7Mn8

Alocação dinâmica

- Não se sabe o tamanho na implementação
  - Armazenamento de grandes quantidades de dados cujo tamanho máximo é desconhecido
  - Tamanho pode variar após o início da execução
- C/C++
  - Utilização de ponteiros
  - Manuseio da memória de maneira explícita

Ponteiros

- Ponteiros
  - Variáveis alocadas dinamicamente (Heap)
  - Armazenam um endereço de memória
- Referência
  - &x
  - Endereço de memória da variável x
- Deferência
  - *x
  - Conteúdo do endereço apontado por x

Ponteiros

```
int main() {
  int i = 10;
  int *ponteiro = &i;
  int **ppp = &ponteiro;

  return 0;
}
```

https://goo.gl/iR6zEUS

Variável	Endereço	Valor
i	0x80	10
ponteiro	0x84	0x80
ppp	0x88	0x84

Ponteiros
Operadores

- C
  - Alocação: malloc
  - Liberação: free
- C++
  - Alocação: new
  - Liberação: delete

Ponteiros
Exemplo 2

```
#include <iostream>
using namespace std;

int main() {

  int *p1, *p2;
  p1 = new int;
  *p1 = 42;
  p2 = p1;
  cout << "*p1 == " << *p1 << endl;
  cout << "*p2 == " << *p2 << endl;

  *p2 = 53;
  cout << "*p1 == " << *p1 << endl;
  cout << "*p2 == " << *p2 << endl;

  p1 = new int;
  *p1 = 88;
  cout << "*p1 == " << *p1 << endl;
  cout << "*p2 == " << *p2 << endl;
  return 0;
}
```

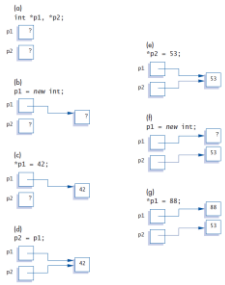
https://goo.gl/VBuCn9

```
*p1 == 42
*p2 == 42

*p1 == 53
*p2 == 53

*p1 == 88
*p2 == 53
```


Ponteiros
Exemplo 2



Ponteiros
Exemplo 3

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```

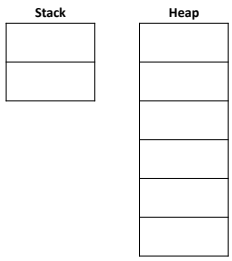
- Qual o valor de 'a' ao final?
- Qual o problema com esse código?

Ponteiros
Exemplo 3

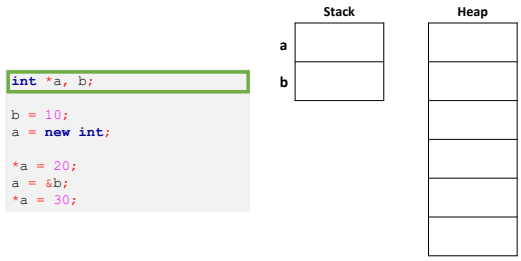
```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```



Ponteiros
Exemplo 3




```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```

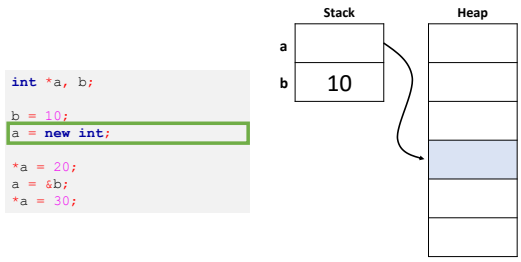

Ponteiros
Exemplo 3


```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```


Ponteiros
Exemplo 3




```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```

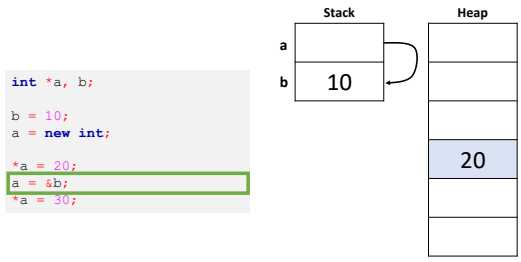

Ponteiros
Exemplo 3


```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```


Ponteiros
Exemplo 3




```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```


Ponteiros
Exemplo 3


```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```

Ponteiros
Exemplo 3

- Como melhorar esse código?

```
int *a, b;

b = 10;
a = new int;

*a = 20;
a = &b;
*a = 30;
```

```
int *a = nullptr;
int b = 10;

a = new int;
*a = 20;
delete a;
a = &b;
*a = 30;
```


Alocação dinâmica de vetores

- Criar vetores em tempo de execução
  - Só ocupar a memória quando necessário
- Apontador guarda o endereço (aponta) da primeira posição do vetor

```
int main(){
  int *p = new int[10];

  p[0] = 99;

  delete[] p;

  return 0;
}
```

https://goo.gl/pGRggS

Ponteiros nulos

- **`nullptr`** (**`NULL`**)
  - Constante simbólica
  - Útil para representar ponteiros não inicializados ou condições de erro
- Nenhum ponteiro válido possui esse valor!
- Esse valor não pode ser acessado
  - Falha de segmentação

Ponteiros para **`void`**

- Não existe um objeto do tipo **`void`**
- Valor utilizado com um coringa
  - Pode apontar para qualquer tipo de objeto

```
int i = 10;
int *int_ptr;
void *void_ptr;
double *double_ptr;

int_ptr = &i;
void_ptr = int_ptr;        // OK
double_ptr = int_ptr;      // !OK
double_ptr = void_ptr;     // OK
```

Variável	Endereço	Valor
i	0x80	10
int_ptr	0x84	0x80
void_ptr	0x88	0x80
double_ptr	0x8c	0x80

Ponteiros para estruturas

- Declaração e inicialização

```
struct data {int dia; int mes; int ano;};
struct data d1;
struct data *ptr = &d1;
int i = 0;
```

- Acesso aos campos

```
(*ptr).dia = 8;
(*ptr).mes = 3;
(*ptr).ano = 2012;

ptr->dia = 8;
ptr->mes = 3;
ptr->ano = 2012;
```

Variável	Endereço	Valor
d1.dia	0x80	8
d1.mes	0x84	3
d1.ano	0x88	2012
ptr	0x8c	0x80
i	0x90	0

Alocação dinâmica de memória
Exemplo 4

```
#include <iostream>

using namespace std;

int main() {

  int *ptr_a = nullptr;
  ptr_a = new int;

  if (ptr_a == nullptr) {
      cout << "Memoria insuficiente!" << endl;
      exit(1);
  }

  cout << "Endereco de ptr_a: " << ptr_a << endl;
  *ptr_a = 90;
  cout << "Conteudo de ptr_a: " << *ptr_a << endl;
  delete ptr_a;

  return 0;
}
```

https://goo.gl/HPb8P9

Passagem de parâmetros

- Parâmetros passados de duas formas
- Valor
  - Parâmetro formal (recebido na função) é uma cópia do parâmetro real (passado na chamada)
- Referência (ponteiro)
  - Parâmetro formal (recebido) é uma referência para o parâmetro real (passado)
  - Modificações refletem no parâmetro real

Passagem de parâmetros
Exemplo 5 – Valor

```
#include <iostream>
using namespace std;

void addOneValue(int x) {
  x = x + 1;
}

int main() {
  int a = 0;
  cout << "Antes: " << a << endl;

  addOneValue(a);
  cout << "Depois: " << a << endl;

  return 0;
}
```

https://goo.gl/37XRTy

Passagem de parâmetros
Exemplo 6 – Referência

```
#include <iostream>
using namespace std;

void addOneReference(int &x) {
  x = x + 1;
}

void addOnePointer(int* x) {
  *x = (*x) + 1;
}

int main() {
  int a = 0;
  cout << "Antes: " << a << endl;
  //addOneReference(a);
  //addOnePointer(&a);
  cout << "Depois: " << a << endl;

  return 0;
}
```

Essa referência não pode ser null!

Esse ponteiro pode ser null!

https://goo.gl/5uFe9k

Passagem de parâmetros

- Use referências (&) sempre que possível, e ponteiros (*) quando for necessário.

```
void function(int &i) {
  int j = 10;
  i = &j;
}
```

- Erro de compilação!

```
void function(int *i) {
  int j = 10;
  i = &j;
}
```

- Compila, mas ótima fonte de bugs!

Passagem de parâmetros

Considerações finais
Erros comuns

- Tentar acessar o conteúdo de uma posição sem ter alocado memória previamente
  - Ou após já ter sido desalocada
- Copiar o valor do apontador (endereço) ao invés do valor da variável apontada
- Esquecer de desalocar memória
  - Desalocada ao fim do programa (função) da declaração, pode ser um problema em loops