UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Fundamentos de Programação

CP41F

Layout de memória. Ponteiros: conceito, operadores de referência e dereferência, operações, aritmética e indireção múltipla

Aula 12

Prof. Daniel Cavalcanti Jeronymo

Universidade Tecnológica Federal do Paraná (UTFPR)
Engenharia de Computação – 1º Período

Plano de Aula

- Layout de memória

- Ponteiros:
  - Conceitos
  - Operadores de referência e dereferência
  - Aritmética
  - Operações
  - Indireção múltipla

Layout de memória

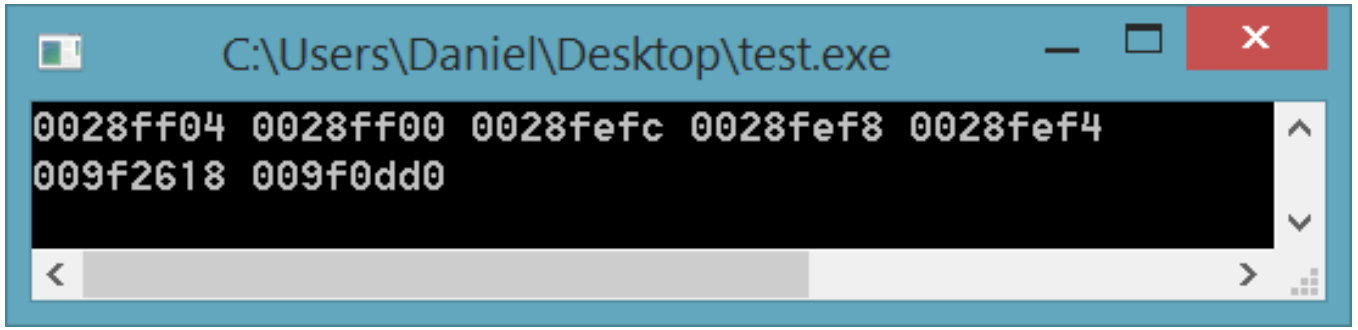
- Layout de memória
  - Define como a memória de um **executável** é organizada

- Terminologia:
  - Seções: regiões do executável com informações para montagem e relocação **(tempo de montagem/linking)**
  - Segmentos: regiões de memória com informações necessárias em tempo de execução, um segmento pode conter zero ou mais seções **(tempo de execução)**

Layout de memória

- Considere o seguinte programa:

```c
#include <stdio.h>
int main()
{
    int a, b, c, d, e;
    int *f = malloc(4096);
    int *g = malloc(1024);
    printf("%p %p %p %p %p\n", &a, &b, &c, &d, &e);
    printf("%p %p\n", f, g);
    return 0;
}
```



Layout de memória

- Obtendo o mapa de memória pelo OllyDbg 2.01:

Memory map

Address	Size	Owner	Section	Contains	Type	Access	Initial	Mapped as
00010000	00010000				Map	RW	RW	
00040000	0000F000				Map	R	R	
00085000	0000B000				Priv	RW Gua	RW Gua	
0028C000	00002000				Priv	RW Gua	RW Gua	
0028E000	00002000			Stack of main thread	Priv	RW	RW	
00290000	00004000				Map	R	R	
002A0000	00002000				Priv	RW	RW	
002B0000	0007E000				Map	R	R	C:\Windows\System32\locale.nls
00400000	00001000	test		PE header	Img	R	RWE Cop	
00401000	00006000	test	.text	Code	Img	R E	RWE Cop	
00407000	00001000	test	.data	Data	Img	RW	RWE Cop	
00408000	00001000	test	.rdata		Img	R	RWE Cop	
00409000	00001000	test	.eh_fram		Img	R	RWE Cop	
0040A000	00001000	test	.bss		Img	RW	RWE Cop	
0040B000	00001000	test	.idata	Imports	Img	RW	RWE Cop	
0040C000	00001000	test	.CRT		Img	RW	RWE Cop	
0040D000	00001000	test	.tls		Img	RW	RWE Cop	
005A0000	00006000				Priv	RW	RW	
00750000	00005000			Default heap	Priv	RW	RW	
009F0000	00005000			Heap	Priv	RW	RW	

Layout de memória

- Regiões importantes de memória:
  - Código (*text* ou *code*)
  - Dados (*data*)
  - Pilha (*stack*)
  - Memória dinâmica (*heap*)

Layout de memória

- Regiões importantes de memória – código
- Contem as instruções em código de máquina do programa
- Tamanho fixo
- Tipicamente, por segurança, de apenas leitura

Layout de memória

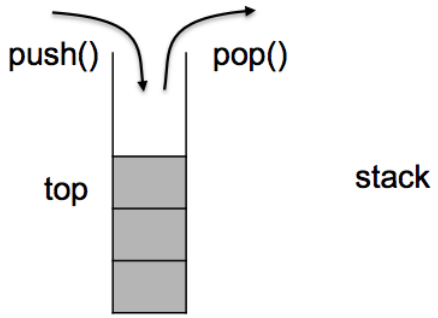
- Regiões importantes de memória – dados

- Contem as variáveis persistentes ao longo da execução de um programa
  - globais ou estáticas
  - constantes (alternativamente, podem ser colocadas no código)

- Tamanho fixo

Layout de memória

- Regiões importantes de memória – Pilha
  - Armazena as variáveis locais



  - Estrutura de dados análoga a uma pilha de objetos (e.g., pratos) – LIFO (*last in first out*)
  - Armazenamento temporário de variáveis – a permanência de uma variável na pilha é determinada pelo seu escopo
  - Como é a representação das variáveis a,b,c,d,e,f,g na pilha?

Layout de memória

- Regiões importantes de memória – Memória dinâmica (heap)
  - Armazena memória alocada dinamicamente
  - Em C por funções: malloc, free, realloc
  - Em C++ pelos operadores: new e delete
  - Aula específica mais pra frente!

Layout de memória

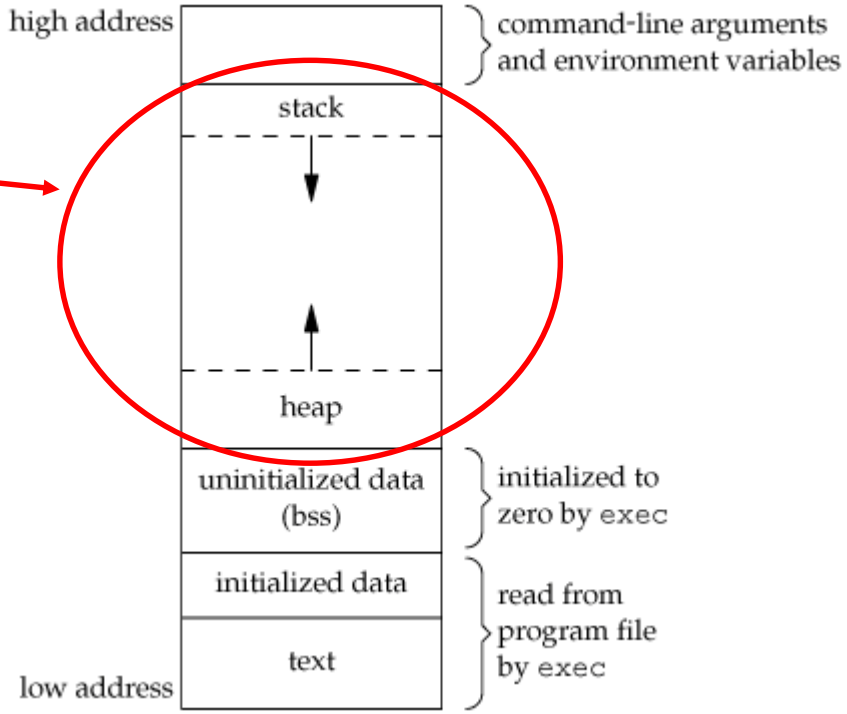
- Sumário das seções importantes:

- .text – instruções de máquina (código)
- .data – variáveis inicializadas globais ou estáticas
- .rdata – variáveis inicializadas constantes ou dados inicializados de apenas leitura (e.g., *string literals*)
- .bss – variáveis globais ou estáticas inicializadas em zero ou não inicializadas – região de memória nula

- Essas seções não são especificadas, dependem da implementação do compilador

Layout de memória

- Layout antigo de memória
  - Cuidado com a internet!!!
- Pilha e heap em direções opostas
- Conceito válido há décadas atrás (~1970)



- Tornado obsoleto pelos modelos de memória virtual
- Na prática o gerenciamento de memória **moderno** é muito mais complexo, onde a pilha e a heap são alocadas e como elas se comportam depende da arquitetura e do sistema operacional

Ponteiros - Conceito

- Variáveis
  - Quando uma variável é declarada o compilador prepara um espaço para esta variável em um **endereço único** na memória
  - O compilador associa o endereço com o **nome** da variável
  - Quando um programa usa o **nome** de uma variável, o endereço apropriado é acessado

Ponteiros - Conceito

- Ponteiros
  - Tipos de dados que permitem endereçar a memória referente à algum tipo
  - Definição: variável que armazena um endereço de memória
  - Sintaxe:

```
tipo *nome;
```

Ponteiros - Operadores

- Declaração:

```
int *p;
```

- Operador de referência:

&p – retorna o **endereço** da variável p

- Operador de dereferência:

*p – retorna o **valor apontado** por p

Ponteiros - Operadores

- Declaração - cuidados:
  - O operador de ponteiro (*) é associado ao nome da variável, não ao tipo

```
int *p, q;
```

- Equivale a:

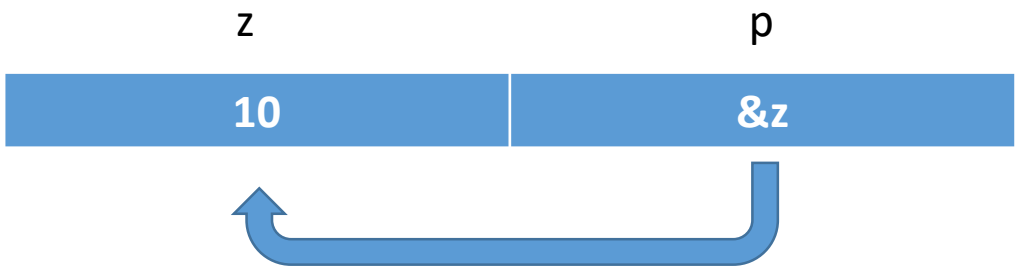
```
int *p;
int q;
```

Ponteiros - Operadores

- Exemplo:

```
int z = 10;

int *p = &z;
```



- p contém o **endereço** de z

- *p acessa o conteúdo endereçado

```
*p =20;

printf("Valor de z: %d\nValor de *p:%d", z, *p);
```

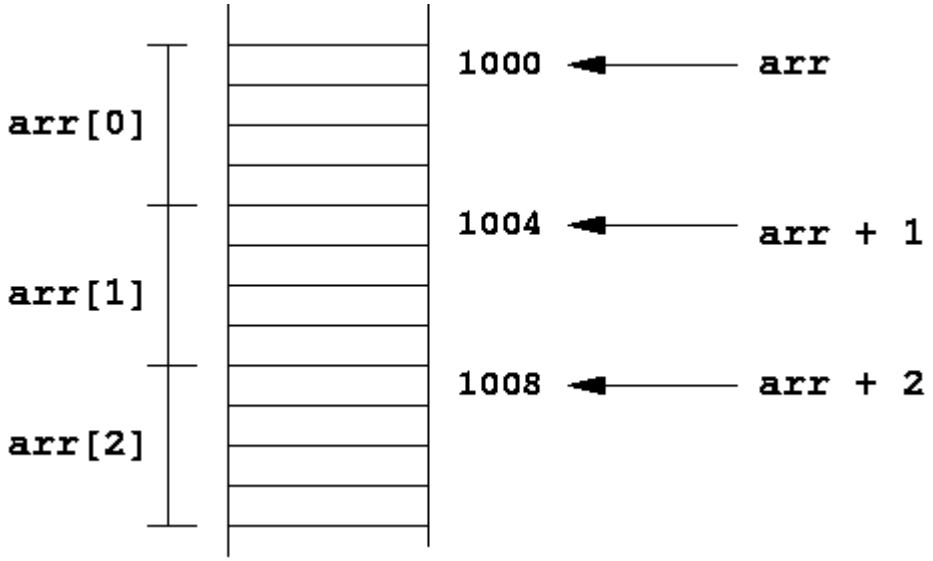
Ponteiros - Aritmética

- A lógica aritmética de ponteiros segue a mesma regra de acesso de índices

- Deve-se ao fato de que acesso a índices é definido usando aritmética de ponteiros:

```
int *arr = (int*)1000;

arr[i] == *(arr + i)
```



Ponteiros - Aritmética

- Ponteiros podem ser somados ou subtraídos de valores constantes

- O deslocamento na memória depende do tipo do ponteiro
  - endereço + valor => endereço + valor*sizeof(tipo)
  - mesma lógica de índices em vetores!

Ponteiros - Aritmética

```
int buf[10];

int *p1 = &buf[0];

int *p2 = &buf[3];
```

- Ponteiros podem ser subtraídos de outros ponteiros

```
p2 – p1;   /* ok – numero de elementos entre os dois */
```

- Ponteiros não podem ser somados com outros ponteiros

```
p1 + p2;   /* erro! – não tem significado */
```

Ponteiros - Operações

```
int buf[10];

int *p = &buf[0];
```

ou

```
int *p = buf;
```

- &buf[0] e buf são ambos do tipo (int*) e apontam pro mesmo endereço
- p[3] == buf[3]

Ponteiros - Operações

- Atenção!
  - Ponteiros e vetores compartilham aritmética e operadores
  - Porém ponteiros e vetores não são a mesma coisa!

- Vetores tem espaço fixo e alocado automaticamente

- Ponteiros devem ser atribuídos para apontar à algum endereço na memória e podem ser reatribuídos arbitrariamente.

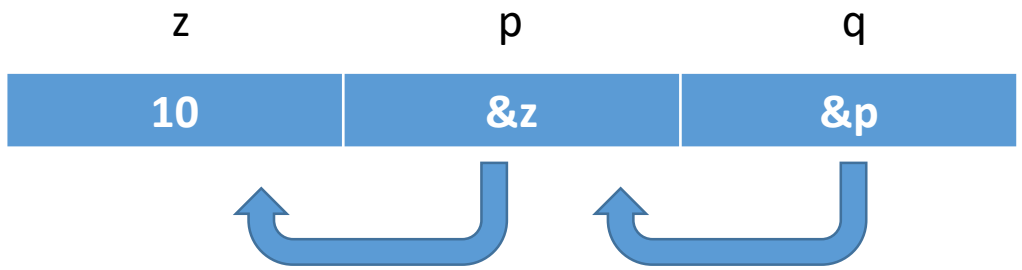
Ponteiros – Indireção múltipla

- Ponteiros que apontam para ponteiros

```
int z = 10;

int *p = &z;

int **q = &p;
```



- p contém o **endereço** de z
- q contém o **endereço** de p