Ponteiros

Professor: Guilherme Corredato Guerino

Disciplina: Algoritmos e Técnicas de Programação

Ciência da Computação
Universidade Estadual do Paraná

Introdução

- A linguagem C possibilita o **armazenamento** e a **manipulação** de valores de **endereços de memória**.

- Em um **endereço de memória**, existe um valor de **determinado tipo**.

Exemplo

- **Considere o seguinte trecho de código:**

```
include<stdio.h>
include<stdlib.h>
int main(){
   int v;
   return 0;
}
```

Exemplo

- **Possível explicação:**
  - Uma variável de nome **v** é declarada. Tal variável pode armazenar **valores inteiros**.
  - Após a declaração, um **espaço na memória** é **reservado**.
  - O espaço é **suficiente** para **armazenar valores inteiros**.

Introdução

- Assim como existem variáveis para **armazenar valores inteiros**, também existem variáveis para **armazenar valores de endereços de memória**.

- Uma **variável** que armazena **valores de endereços** é conhecida como **ponteiro**.

Observações

- Para cada **tipo primitivo de dado**, existe uma **variável ponteiro correspondente**.

- A linguagem C **não reserva** uma **palavra especial** para a **declaração de ponteiros**.

- Utiliza-se a **mesma palavra do tipo**, com os nomes das variáveis **precedidos** pelo **caractere** *.

Exemplo

```
include<stdio.h>
include<stdlib.h>
int main(){
  int *p; // declaração ponteiro
  return 0;
}
```

Exemplo

- No slide anterior, tem-se a declaração de uma variável de nome p, que pode armazenar um **endereço de memória**.

- No **endereço armazenado**, um **valor inteiro** pode estar **armazenado**.

Ponteiro

- Assim como foi declarado um ponteiro do tipo **int**, pode-se declarar ponteiros de outros tipos:
  - **float *m**;
  - **char *s**;

- Para **atribuir** e **acessar** endereços de memória, a linguagem C oferece os seguintes **operadores unários**: **&** e *.

Ponteiro

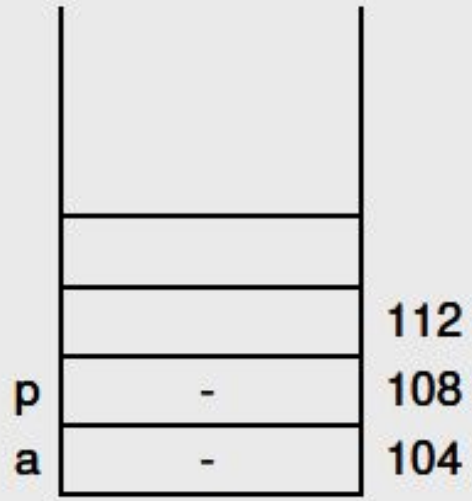
- O operador unário **&** (“**endereço de**”), acessa o **endereço da posição de memória**, reservada para a variável.

- O operador ***** (“**conteúdo de**”), acessa o **conteúdo do endereço de memória** armazenado pela variável ponteiro.

Exemplo

```
/*variável inteiro */
 int a;

/*variável ponteiro p/ inteiro */
 int *p;
```



Exemplo

- As variáveis **a** e **p** foram declaradas.
  - A variável **a** é do **tipo inteiro**.
  - A variável **p** é do **tipo ponteiro de inteiro**.

- Os números **104**, **108** e **112** representam **endereços de memória**.

Observações

- Após as declarações das variáveis **a** e **p**, não se sabe quais são os **valores armazenados nas mesmas**.
- Esses valores são considerados "**lixo**", uma vez que **não são importantes** para a **computação em questão**.
- Tais valores foram considerados em **processamentos** e/ou **computações anteriores**.

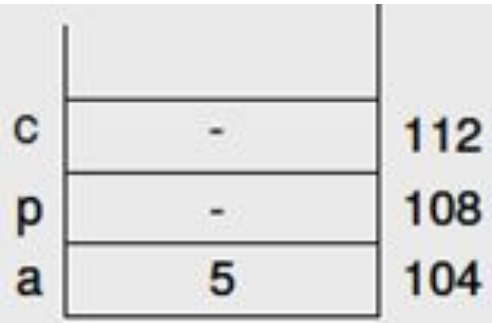
Exemplo

- Considerando as variáveis **a** e **p** já declaradas, tem-se que **atribuições** podem ser realizadas.

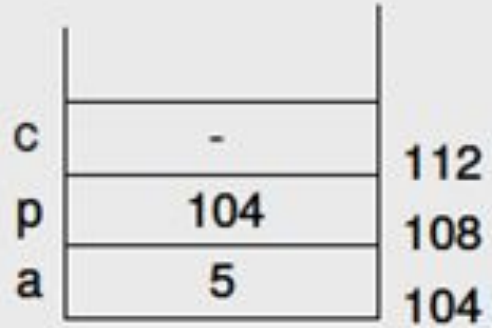
- Tais atribuições são apresentadas a seguir...

Exemplo

```
/ * a recebe o valor 5 */
a = 5;
```

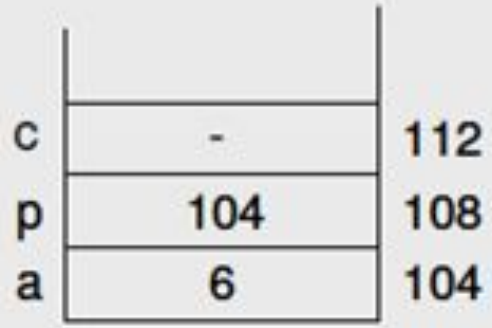



```
/* p recebe o endereço de a
   ou seja, p aponta para a */
p = &a;
```

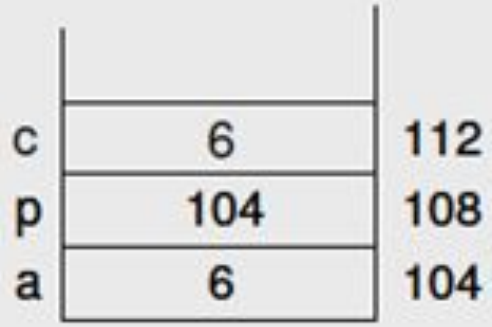


Exemplo

```
/* posição de memória apontada por p
   recebe 6 */
*p = 6;
```




```
/* c recebe o valor armazenado
   na posição de memória apontada por p */
c = *p;
```



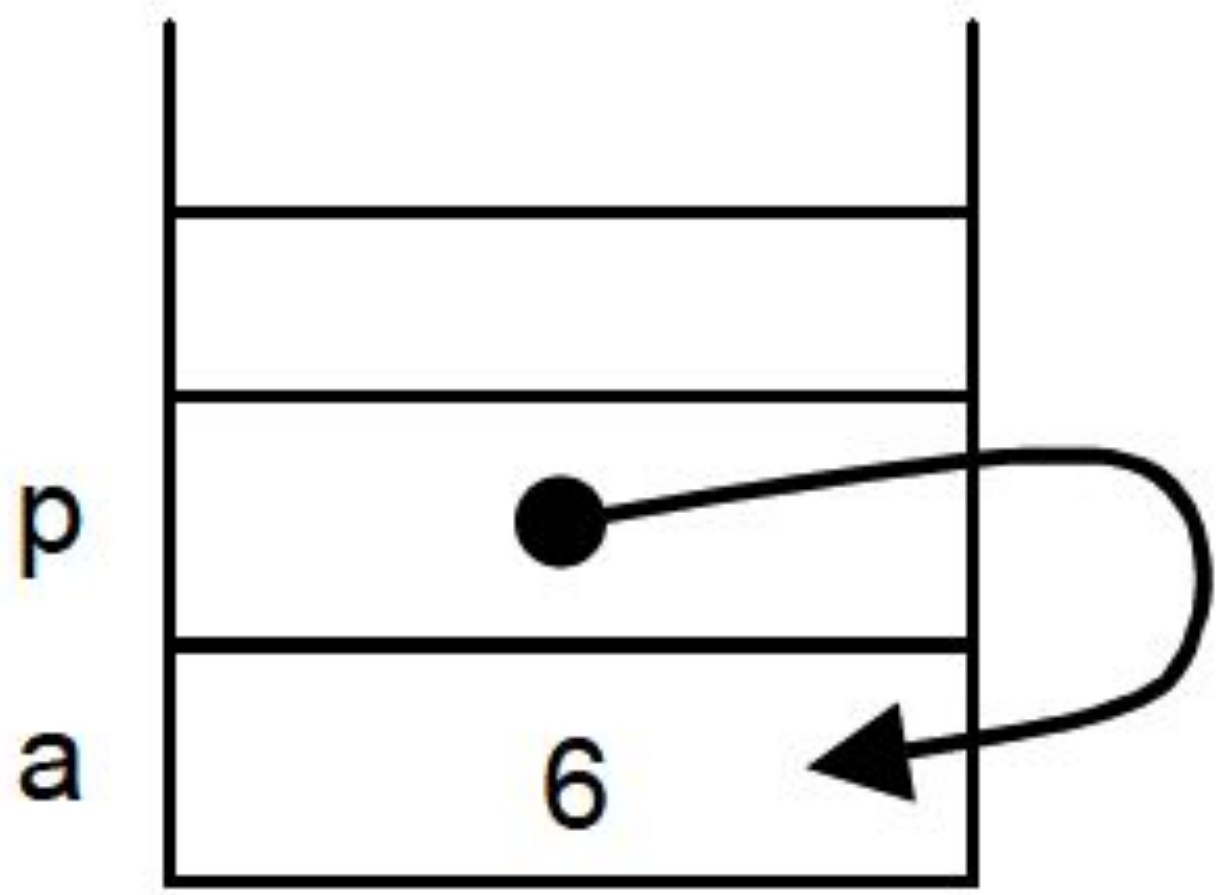
Exemplo

- Após as atribuições mostradas, a variável **a** recebe, **indiretamente**, o valor 6.

- Isso acontece pois acessar **a** é equivalente a acessar ***p**, uma vez que **p** armazena o endereço de **a**.

- Dada essa situação, pode-se dizer que **p aponta para a**.

Observação



Observação: o fato da variável **p** "**apontar**" para a variável **a**, significa que **p** armazena o **endereço** de **a**.

Possíveis Problemas

- O uso de ponteiros representa uma das maiores **potencialidades** da linguagem C.

- Por outro lado, programas podem se tornar um **problema**, por conta da **manipulação indevida dos ponteiros**.

Possíveis Problemas

- Isso significa que os programas **podem não funcionar**, ou pior, **podem funcionar e gerar efeitos não previstos**.

- Para trabalhar com ponteiros de **maneira adequada**, é preciso considerar **determinados cuidados**.

Exemplo

```
int main ( void )
{
   int a;
   int *p;
   p = &a;
   *p = 2;
   printf(" %d ", a);
   return;
}
```

Exemplo

```
int main ( void )
{
   int a, b, *p;
   a = 2;
   *p = 3;
   b = a + (*p);
   printf(" %d ", b);
   return 0;
}
```

Observações

- No exemplo 1, as instruções são executadas sem **nenhum problema** a princípio.

- No exemplo 2, tem-se um **erro típico, presente na manipulação de ponteiros**.

Observações

- Esse erro consiste na **não inicialização** do ponteiro **p**, ou seja, o ponteiro p **não recebeu um endereço de memória antes de ser manipulado**.

- Com isso, a variável p considerou o "**endereço lixo**" **armazenado**.

Observações

- A atribuição ***p = 3** armazena o valor 3 em um **endereço de memória desconhecido**.

- Esse endereço pode representar tanto um **espaço não utilizado** quanto um **espaço utilizado**, que armazena **informações importantes** sobre **outros programas** e/ou **aplicações**.

Orientações

- Para evitar problemas com a manipulação de ponteiros, é importante **inicializar o ponteiro**, **antes de qualquer operação com o mesmo**.

- Sem a inicialização do(s) ponteiro(s), o programa pode apresentar **diferentes comportamentos**.

Orientações

- **Entre os possíveis comportamentos estão:**
  - **Funcionar bem**, se o endereço de memória **não estiver** sendo utilizado; ou
  - **Apresentar problemas imprevisíveis**, por conta da manipulação de **espaços de memória** associados com outros **programas** e/ou **aplicações**.

Referências

- Introdução a Estruturas de Dados – 1ª edição. Waldemar Celes, Renato Cerqueira e José Lucas Rangel. Campus, 2004.