MC-102 — Aula 18
Apontadores e Alocação Dinâmica de Memória

Instituto de Computação – Unicamp

Segundo Semestre de 2009

Roteiro

Variáveis

Ao declararmos uma variável x como a abaixo:

```
int x = 100;
```

Temos associados a ela os seguintes elementos:

- Um nome (x);
- Um endereço de memória ou referência (0xbfd267c4);
- Um valor (100).

O operador *address-of* (&)

Para acessarmos o endereço de uma variável, usamos o operador &:

Exemplo

```
int x = 100;
printf("Valor de x = %d\n", x);
printf("Endereço de x = 0x%x\n",
       (unsigned int) &x);
```

Veja o exemplo em `x.c`.

Apontador

- Nós já vimos que existem tipos de dados para armazenar endereços de variáveis.
- Uma variável declarada como um destes tipos é chamada de apontador.
- Ao atribuir o endereço de uma variável a um apontador, dizemos que o mesmo aponta para a variável.

Exemplo

```
int x;
int *ap_x; /* apontador para inteiros */

ap_x = &x; /* ap_x aponta para x */
```

Veja o exemplo em `ap_x.c`.

Declaração de apontadores em C

Para declarar uma variável do tipo apontador utilizamos o operador unário *.

Exemplo

```
int    *ap_int;
char   *ap_char;
float  *ap_float;
double *ap_double;
```

Veja os exemplos em `apontadores.c` e `ap_tabela.c`.

Declaração de apontadores em C

Cuidado ao declarar vários apontadores em uma única linha. O operador * deve preceder o nome da variável e não suceder o tipo que o apontador apontará.

```
int *ap1, *ap_2, *ap_3;
```

Exercício

A declaração abaixo declara quantos inteiros e quantos apontadores para inteiro?

```
int *ap1, ap_ou_int1, ap_ou_int2;
```


Fazendo acesso aos valores das variáveis referenciadas

- Um endereço de variável por si só não é muito útil. Para acessarmos o valor de uma variável apontada por um endereço, também usamos o operador *:
- Ao precedermos um apontador com este operador, obtemos o equivalente a variável armazenada no endereço em questão:
- `*ap_x` pode ser usado em qualquer contexto que a variável `x` seria.

Exemplo

```
int x;
int *ap_x = &x;
*ap_x = 3;
```

Veja o exemplo em `valores.c`.

Apontadores para registros

- Para acessar os elementos de um registro através de um apontador, devemos primeiro acessar o registro e depois acessar o campo desejado.
- Os parênteses são necessários pois o operador * tem prioridade menor que o operador `.`.

Exemplo

```
struct ponto { double x; double y; };
typedef struct ponto Ponto;

Ponto *ap_p;
(*ap_p).x = 4.0;
(*ap_p).y = 5.0;
```


Apontadores para registros

- Para simplificar o acesso aos campos de um registro através de apontadores, foi criado o operador `->`.
- Usando este operador acessamos os campos de um registro diretamente através do apontador.

Exemplo

```
struct ponto { double x; double y; };
typedef struct ponto Ponto;

Ponto *ap_p;
ap_p->x = 4.0;
ap_p->y = 5.0;
```

Veja o exemplo em `ap_struct.c`.

Passagem de parâmetros por valor e referência

- Como já vimos, ao passarmos argumentos para uma função, estes são copiados como variáveis locais da função. Isto é chamado passagem por valor.
- Existe uma forma de alterarmos a variável passada como argumento, fazendo uma passagem por referência.
- O artifício corresponde a passarmos como argumento o endereço da variável, e não o seu valor.
- Ou seja, o mecanismo usado na linguagem C para fazer chamadas por referência corresponde a passarmos apontadores para as variáveis que queremos alterar na função.

Apontadores e vetores

- Uma variável que representa um vetor é implementada por uma apontador constante para o primeiro elemento do vetor.
- A operação de indexação corresponde a deslocar este apontador ao longo dos elementos alocados ao vetor.
- Isto pode ser feito de duas formas:
  - Usando o operador de indexação (`v[4]`).
  - Usando aritmética de endereços (`*(ap+4)`).
- Este dupla identidade entre apontadores e vetores é a responsável pelo fato de vetores serem sempre passados por referência e pela inabilidade da linguagem em detectar acessos fora dos limites de um vetor.

Veja o exemplos em `ap_e_vetor.c` e `cadeias.c`.

Vetores de apontadores

- Não existe diferença entre vetores de apontadores e vetores de tipos simples.
- Neste caso, basta observar que o operador * tem precedência menor que o operador de indexação [].

Exemplo

```
int *vet_ap[5];
char *vet_cadeias[5];

printf("%d %s", *vet_ap[0], vet_cadeias[0]);
```

Veja o exemplo em `vet_cadeias.c`.

Alocação dinâmica de memória

- Além de reservarmos espaços de memória com tamanho fixo na forma de variáveis locais, podemos reservar espaços de memória de tamanho arbitrário e acessá-los através de apontadores.
- Desta forma podemos escrever programas mais flexíveis, pois nem todos os tamanhos devem ser definidos aos escrever o programa.
- A alocação e liberação destes espaços de memória é feito por duas funções da biblioteca `stdlib.h`:
  - `malloc()`: Aloca um espaço de memória.
  - `free()`: Libera um espaço de memória.

Função `malloc()`

- Aloca um bloco consecutivo de bytes na memória e retorna o endereço deste bloco.
- Para determinarmos o tamanho necessário, devemos usar a função `sizeof()`.
- O espaço alocado por esta função pode ser usado para armazenar qualquer tipo de dados, logo devemos converter o tipo retornado (`void*`) para o tipo que iremos usar.

Exemplo: um vetor de 100 inteiros

```
int *p;
p = (int*) malloc(100 * sizeof(int));
```

Veja os exemplos em `malloc.c` e `nao_eh_infinita.c`.

Função `free()`

- Libera o uso de um bloco de memória, permitindo que este espaço seja reaproveitado.
- Deve ser passado para a função `free()` exatamente o mesmo endereço retornado por uma chamada da função `malloc()`.
- A determinação do tamanho do bloco a ser liberado é feita automaticamente.

Exemplo: liberando um vetor de 100 inteiros

```
int *p;
p = (int*) malloc(100 * sizeof(int));
free(p);
```

Veja o exemplo em `free.c`.