Notas de aula - 18/03

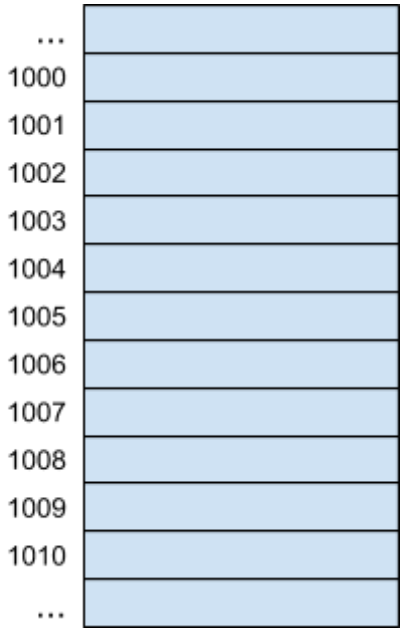
Endereços de memória

Quando declaramos uma variável na linguagem C, precisamos informar seu tipo, seu nome, e em algum momento, lhe atribuímos um valor. Por exemplo:

```
int n = 10;
char c = 'a';
```

Além de estar associada a um **tipo**, um **nome** e um **valor**, uma variável também está associada a um **endereço de memória**.

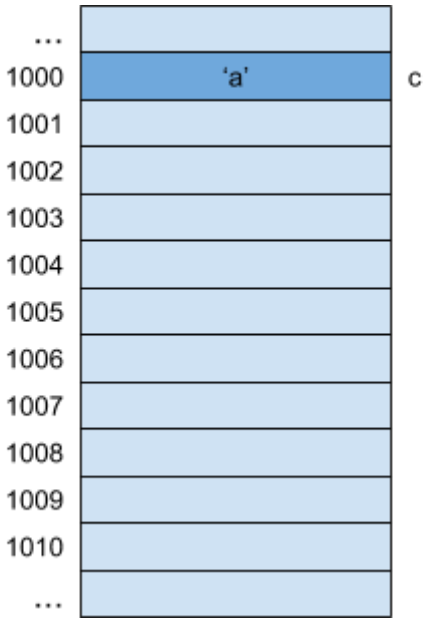
O desenho abaixo traz a representação da memória de um computador, onde cada retângulo representa 1 byte. À esquerda de cada retângulo está o seu endereço de memória.



Quando a variável:

```
char c = 'a';
```

é declarada, ela é guardada em algum lugar da memória do computador, por exemplo:

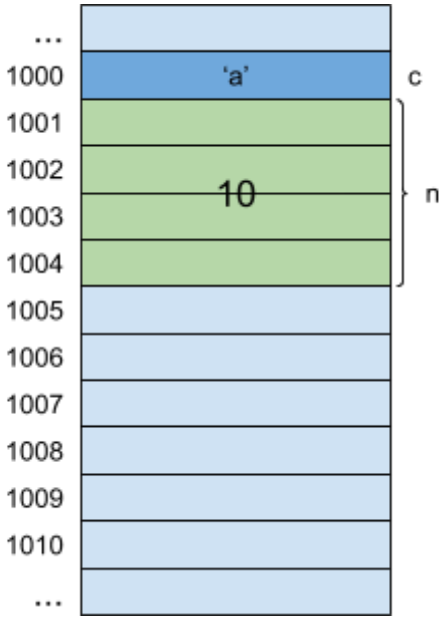


Uma variável do tipo char ocupa apenas 1 byte, portanto, no exemplo acima, a variável “c” ocupa a posição 1000. Dizemos então, que o endereço de “c” é 1000.

Considere agora uma variável do tipo int:

```
int n = 10;
```

Uma variável do tipo int, na maioria dos computadores atuais, ocupa 4 bytes:



No exemplo acima, a variável “n” ocupa os endereços 1001, 1002, 1003 e 1004. Quando uma variável ocupa mais do que um endereço na memória, dizemos que o seu endereço é onde ela começa, ou seja, o primeiro endereço ocupado por ela. Neste exemplo, endereço de memória da variável “n” é 1001.

O mesmo vale para um vetor ou matriz. O endereço de um vetor, por exemplo, sempre será onde ele “começa”, ou seja o primeiro endereço ocupado por ele.

Obs.: para verificar o tamanho de um tipo de dados, a linguagem C possui a função *sizeof*. Ela recebe por parâmetro um tipo de dado e retorna quantos bytes esse tipo ocupa na memória (o retorno é um número inteiro). Exemplos de uso:

```c
#include <stdio.h>

typedef struct aluno Aluno;
struct aluno {
	char nome[20];
	int RA;
	float coeficiente;
}

main() {
	printf("%d\n", sizeof(char));
	printf("%d\n", sizeof(int));
	printf("%d\n", sizeof(float));
	printf("%d\n", sizeof(Aluno)); // funciona também com novos tipos
}
```

Verificando o endereço de uma variável na linguagem C

Na linguagem C estamos acostumados a acessar os valores das variáveis, por exemplo:

```c
printf("O valor da variável c é: %c", c);
printf("O valor da variável i é: %d", n);
```

E se quiséssemos saber os endereços das variáveis “c” e “n” em vez de seus valores? Para isso existe o operador de endereço, representado pelo símbolo “&”. Quando colocado do lado esquerdo de uma variável, em vez de termos o valor da variável, temos o seu endereço na memória:

```c
printf("O endereço da variável c é: %c", &c);
printf("O endereço da variável n é: %d", &i);
```

Já estamos acostumados com o operador & no scanf:

```c
int x;
printf("Informe o valor de x: ");
scanf("%d", &x);
```

Os argumentos básicos da função scanf são: o tipo do valor que será lido e o endereço onde esse valor será guardado.

Tipo ponteiro

Na linguagem C, existem variáveis que podem guardar endereços em vez de valores. Essas variáveis são chamadas **ponteiros** ou **apontadores**.

Esse tipo de variável possui uma declaração especial. O nome da variável deve estar precedido por um asterisco (*) para indicar que ela vai guardar um endereço de memória e não um valor:

tipo *nome_variavel;
ou:
tipo* nome_variavel;
ou ainda:
tipo * nome_variavel;

As 3 formas são aceitas pela linguagem C. Leia mais em: https://www.ime.usp.br/~pf/algoritmos/aulas/footnotes/pointer-decl.html

Abaixo, um exemplo de declaração de um ponteiro para um inteiro:

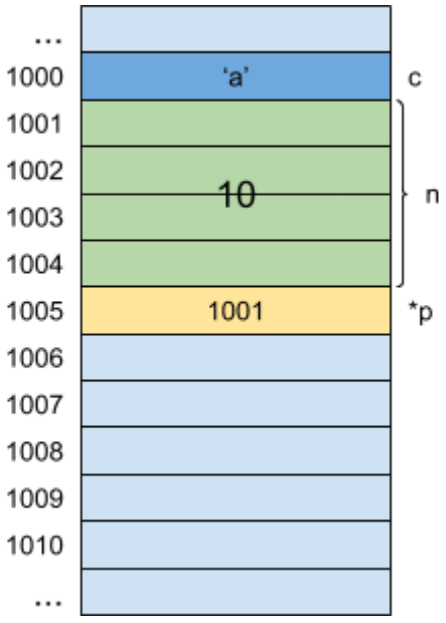
```
int *p;
```

O operador & - “o endereço de”

Agora vamos passar o endereço de “n” para o ponteiro “p”. Visto que “p” não vai guardar o valor, mas sim o endereço de “n”, a variável “n” deve estar precedida por um “&”:

```
p = &p;
```

Como “p” guarda o endereço da variável “n”, podemos dizer que **“p” aponta para “n”**.



Teste o código abaixo. Você deverá ver que os dois últimos printf irão imprimir o mesmo valor, que é o endereço de “n”. Usamos “%d” nos printf com endereços, pois um endereço sempre será um número inteiro.

```c
#include <stdio.h>

main() {
	int n = 10;
	int *p;
	p = &n; // p aponta para n;
	printf("Valor de n: %d\n", n);
	printf("Endereco de n: %d\n", &n);
	printf("Endereço apontado por p: %d\n", p);
}
```

```
Valor de n: 10
Endereco de n: 1220074716
Valor de p: 1220074716
```

O operador * - “no endereço… “

Quando um ponteiro aponta para um endereço de memória, a operação para acessar o conteúdo desse endereço é chamada de **deferência.** O operador utilizado é o asterisco (*):

```c
#include <stdio.h>

main() {
	int i = 10;
	char c = 'a';
	int *pi = &i;
	char *pc = &c;
	printf("O valor de i é: %d\n", i);
	printf("O valor de c é: %c\n", c);
	printf("O endereço de i é: %d\n", &i);
	printf("O endereço de c é: %d\n", &c);
	printf("pi aponta para o endereço de i, que é: %d\n", pi);
	printf("pc aponta para o endereço de c, que é %d\n", pc);
	printf("No endereço apontado por pi está o conteúdo: %d\n", *pi);
	printf("No endereço apontado por pc está o conteúdo: %c\n", *pc);
}
```

```
O valor de i é: 10
O valor de c é: a
O endereço de i é: 2064975428
O endereço de c é: 2064975427
pi aponta para o endereço de i, que é: 2064975428
pc aponta para o endereço de c, que é 2064975427
No endereço apontado por pi está o conteúdo: 10
No endereço apontado por pc está o conteúdo: a
```

Analise o código abaixo:

```c
#include <stdio.h>

main() {
	int x = 10;
	int *px;
	px = &x;
	printf("Valor de x: %d\n", x);
	printf("px aponta para x, que esta no end.: %d\n", px);

	*px = 20;
	printf("px ainda aponta para o end.: %d\n", px);
	printf("O objeto apontado por px foi alterado, logo, x passou a valer: %d", x);
}
```

Quando queremos mexer no endereço apontado por px, usamos apenas **px**. Quando queremos mexer no objeto apontado por px, usamos ***px** (com o operador de deferência). Lembrando que alterar o objeto apontado por px irá refletir na variável x, que usa esse objeto:

```
Valor de x: 10
px aponta para x, que esta no end.: -1125344260
px ainda aponta para o end.: -1125344260
O objeto apontado por px foi alterado, logo, x passou a valer: 20
```

Cuidados ao fazer atribuições

```c
int i = 10;
char c = 'a';
int *p;
int *pi;
char *pc;
pi = &i;
pc = &c;

pi = pc; // Warning: os ponteiros não são do mesmo tipo (pi é int e pc é char)
pi = i; // Warning: ponteiro recebendo um valor, não um endereço
*p = 10; // Erro: p não está inicializado, ou seja, ainda não está
```

```
associado a nenhum endereço de memória
```

Aritmética com ponteiros

Com ponteiros é possível fazer apenas operações de soma, subtração e comparação. Exemplo de soma:

```c
#include <stdio.h>

main() {
	int i = 10;
	int *pi = &i;
	printf("O endereço apontado por pi é: %d\n", pi);
	printf("O valor do objeto apontado por pi é: %d\n", *pi);
	pi = pi + 1;
	printf("O endereço apontado por pi é: %d\n", pi);
}
```

```
O endereço apontado por pi é: 1928620876
O valor do objeto apontado por pi é: 10
O endereço apontado por pi é: 1928620880
```

Ao fazer a adição pi = pi + 1, o valor 1 é somado ao endereço e não o valor 10. O endereço resultante está 4 endereços a frente, uma vez que tipo int ocupa 4 bytes. Se imprimirmos o valor contido no endereço resultante, teremos um valor de pouca utilidade, visto que o endereço pode conter algum lixo de memória.

Podemos perceber que a operação de soma no exemplo acima não tem muito sentido prático. Uma aplicação um pouco mais útil sobre operações de soma e subtração pode ser mostrada com vetores:

```c
#include <stdio.h>

main() {
	int v[5] = {10, 15, 90, 45, 26};
	int *pv = &v;
	printf("pv aponta para o endereço: %d\n", pv); // endereço de v[0]
	printf("O valor guardado no endereço acima é: %d\n", *pv);
	pv = pv + 2;
	printf("pv aponta para o endereço: %d\n", pv); // endereço de v[2]
	printf("O valor guardado no endereço acima é: %d\n", *pv);
	pv = pv - 1;
	printf("pv aponta para o endereço: %d\n", pv); // endereço de v[1]
	printf("O valor guardado no endereço acima é: %d\n", *pv);

}
```

```
pv aponta para o endereço: 701222592
O valor guardado no endereço acima é: 10
pv aponta para o endereço: 701222600
O valor guardado no endereço acima é: 90
pv aponta para o endereço: 701222596
O valor guardado no endereço acima é: 15
```

É possível ver que ao adicionar uma unidade ao endereço guardado por pv, temos a posição seguinte do vetor. Ao diminuir uma unidade, tempos a posição anterior do vetor.

Comparação de ponteiros

Comparações só terão algum sentido quando forem feitas sobre ponteiros que apontam para o mesmo objeto:

```c
#include <stdio.h>

main() {
	int x, y, *p1, *p2;
	x = 5;
	y = 6;
	p1 = &x;
	p2 = &y;
	printf("p1 e p2 são iguais? %d\n", p1 == p2);
	p1 = &x;
	p2 = &x;
	printf("p1 e p2 são iguais? %d\n", p1 == p2);
}
```

```
p1 e p2 são iguais? 0
p1 e p2 são iguais? 1
```

Ponteiros como argumentos de funções

Imagine que você possui uma variável “a” com valor igual a 10 e uma variável “b” com valor igual a 15. Você quer trocar o conteúdo das duas variáveis, ou seja, “a” tem que passar a valer 15 e “b” tem que passar a valer 10. Uma ideia de resolver o problema seria:

```c
#include <stdio.h>

main() {
	int a = 10;
	int b = 15;
	int t;
	t = a;
	a = b;
	b = t;
	printf("A: %d - B: %d", a, b);
```

```
}
```

E se você precisasse fazer uma função para fazer essa troca? A função abaixo conseguiria trocar os valores de “a” e “b”?

```
#include <stdio.h>

void troca(int a, int b) {
	int t;
	t = a;
	a = b;
	b = t;
}

main() {
	int a = 10;
	int b = 15;
	troca(a, b);
	printf("A: %d - B: %d", a, b);
}
```

A função troca trabalha com cópias de “a” e “b”, portanto, mesmo após chamar esta função, veremos no printf que “a” e “b” permanecem inalteradas.

Considere agora que a função troca, em vez de receber cópias de “a” e “b” recebe ponteiros que apontam para “a” e “b”.

```
void troca(int *a, int *b) {
	int t;
	t = *a; // t recebe o conteúdo do end. apontado por a
	*a = *b; // o conteúdo do end. apontado por a recebe o conteúdo do
end. apontado por b
	*b = t; // o conteúdo do end. apontado por b recebe o valor de t
}
```

Na chamada da função em vez de passar cópias de “a” e “b”, devemos passar os endereços de “a” e “b” para que a função possa fazer as alterações diretamente sobre os conteúdos guardados nesses endereços. Dizemos que “a” e “b” são passados por **referência** para a função troca.

```
main() {
	int a = 10;
	int b = 15;
	troca(&a, &b);
	printf("A: %d - B: %d", a, b);
}
```

E se eu quiser criar uma função que recebe um vetor? Quando passamos um vetor por parâmetro para uma função na linguagem C, na verdade, não estamos passando o vetor inteiro como parâmetro, estamos passando um ponteiro para a primeira posição do vetor. Portanto, vetores **sempre** são passados por referência e não por valor. É por isso que a função do exemplo abaixo consegue alterar o conteúdo do vetor “v”:

```
#include <stdio.h>

int altera_vetor_1(int vetor[], int n) {
      vetor[0] = 2;
}

main() {
      int i, n = 5;
      int v[] = {1, 2, 3, 4, 5};

      altera_vetor_1(v, n);
      for(i = 0; i < n; i++) {
      printf("%d\t", v[i]);
      }
}
```

A função altera_vetor_1 é quivalente a função altera_vetor_2 mostrada abaixo. A chamada para a função altera_vetor_2 pode ser exatamente igual a chamada da função altera_vetor_1. Não é necessário usar o operador & em sua chamada.

```
int altera_vetor_2(int *vetor, int n) {
      vetor[0] = 2;
}
```

Ponteiros como retorno de funções

Para retornar um ponteiro deve-se declarar a função como tendo tipo de retorno um ponteiro. Considere uma função que retorna um ponteiro para a primeira ocorrência de um caractere “c” em uma string “s”:

```
#include <stdio.h>

char * match(char c, char *s) {
   /*
   *s - pega o conteúdo do ponteiro
   c != *s - verifica se c é diferente do conteúdo do ponteiro.
   *s - verifica se ainda não está no fim da string, ao chegar no fim da
string o ponteiro terá \0 como conteúdo.
   */
      while(c != *s && *s)
          s++; // se ainda não encontrou, o ponteiro é movido para o
```

```
próx. endereço
      return s; // retorna ponteiro
}

main() {
      char s[] = "boa noite";
      char c = 'a';
      char *p;
      p = match(c, s);
      if(*p) // encontrou (ponteiro não tem \0 como conteúdo)
          printf("%s", p);
      else
          printf("Nao encontrou");
}
```

Referências:

Livro C completo e Total - Herbert Schildt - 3ª edição.
Notas de aula Profª. Carmem Hara e Prof. Wagner Zola:
http://www.inf.ufpr.br/nicolui/Docs/Livros/C/ArmandoDelgado/notas-28_Ponteiros.html