Fundamentos de Programação

Filipe Francisco

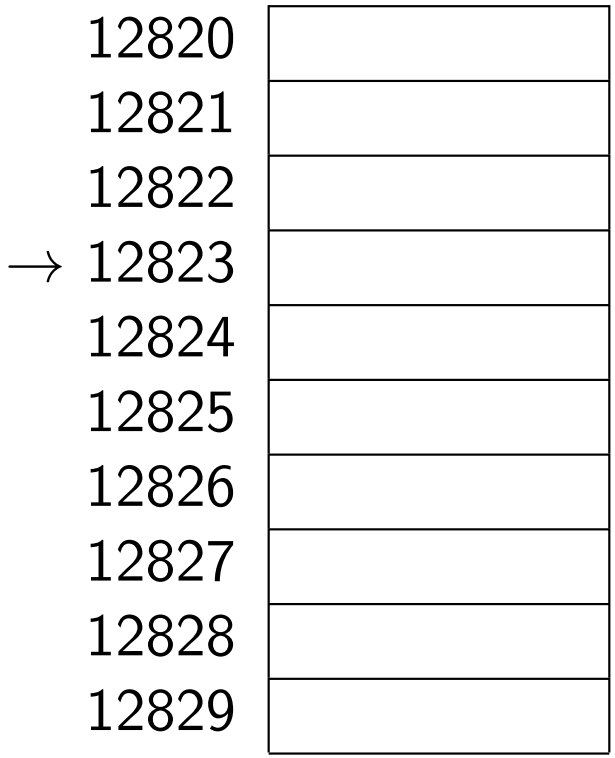
17 de maio de 2018

Problemas

- Recursão:
  - é um método utilizado para a solução de problemas
  - no caso de programação, representa uma função que é definida em função dela mesma
    - ou seja, dentro da função há uma chamada para a própria função
  - como montar uma função recursiva?
    - casos base: casos mais simples possível, para os quais a resposta é trivial
    - chamada(s) recursiva(s): resolve(m) o problema para casos gerais (diferentes do caso base)
  - se a função retorna algo, deve-se utilizar o retorno da função
    - pode-se armazenar numa variável ou retornar diretamente o valor
  - se a função não tem retorno, basta uma chamada normal para a função, sem atribui-la a uma variável

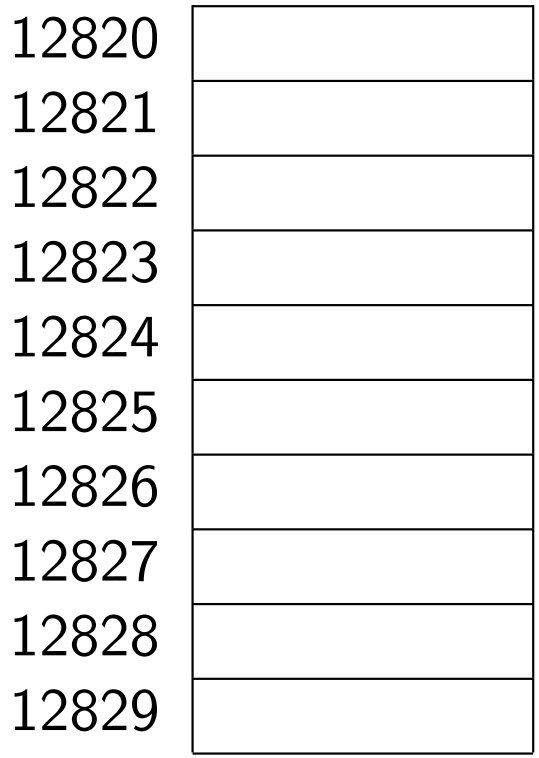
Ponteiros

- Ponteiro é um tipo especial de variável que aponta para alguma posição na memória
- A ideia de posição na memória trabalha com o conceito de endereço de memória
- A memória de um computador é composta de vários bytes, e cada um desses bytes tem seu próprio endereço
- A partir do endereço, acessar esta posição é algo extremamente rápido



Ponteiros

- Os endereços de memória são números sequênciais que podem ser utilizados para representar cada byte
- O endereço de uma variável normalmente é o endereço do primeiro byte desta variável na memória
- A partir do endereço, para ler uma variável basta sabermos quantos bytes ela "mede"
- Pergunta: como saber quantos bytes tem uma certa variável?

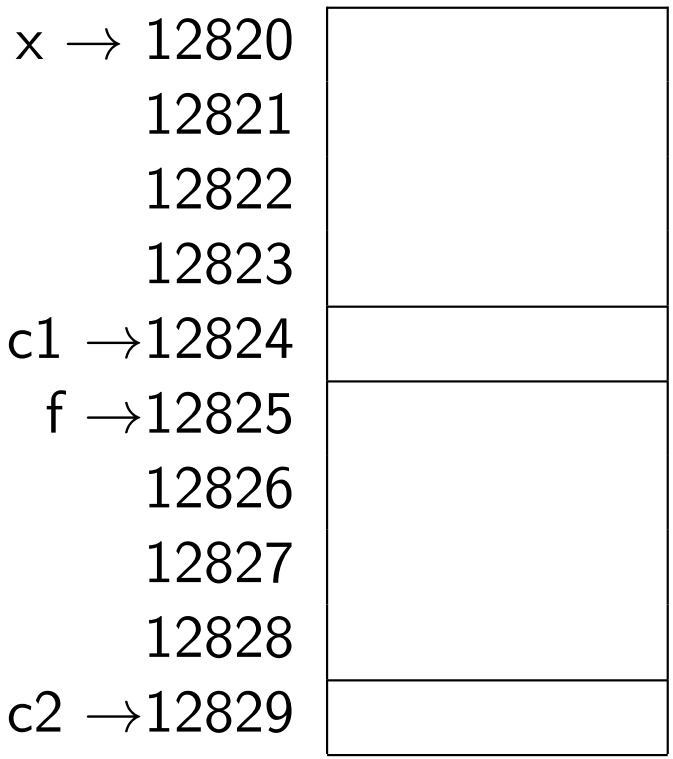


Ponteiros

- O tamanho em bytes de uma variável depende do tipo de variável
  - inteiro: 4 bytes
  - char: 1 byte
  - float: 4 bytes
  - double: 8 bytes
- Exemplo: ao alocar as seguintes variáveis

```
int x;
char c1;
float f;
char c2;
```

as variáveis basicamente armazenam o endereço de memória e o tipo de variável

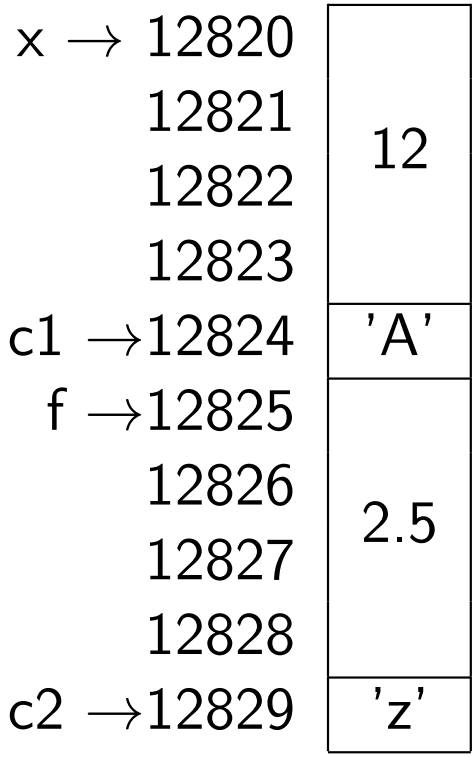


Ponteiros

- Para ler as variáveis, fazíamos:

```
scanf("%d",&x);
scanf("%c",&c1);
scanf("%f",&f);
scanf("%c",&c2);
```

- Basicamente, estamos indicando onde armazenar os valores lidos
- Suponha que lemos 12, 'A', 2.5, e 'z'
  - relembrando: 1 byte = 8 bits, portanto cada um destes valores é armazenado na sua forma binária

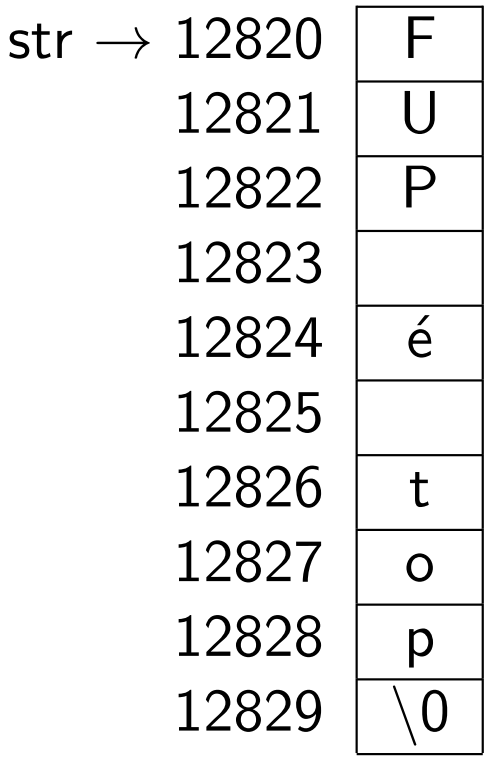


Ponteiros

- Vimos casos de como são armazenadas variáveis simples
- Pergunta: como armazenar um vetor?
- É simples: utilizamos posições consecutivas de memória
- Exemplo: lendo a string "FUP é top"

```
scanf("%100[^\n]",str);
```

  - note que não utilizamos '&'!



Ponteiros

- Portanto, ponteiros são estruturas um pouco diferentes de variáveis propriamente ditas
- Ponteiros apontam para uma certa posição de memória (ou não)
  - se ele aponta para uma posição de memória, seu valor é o endereço para o qual ele aponta
  - se ele não aponta para nada, seu valor é NULL (já vimos casos deste tipo)
- A partir desta posição de memória, podemos ler os dados a partir desta posição como desejarmos
- Por mais que pareça ter pouco uso, ponteiros são muito úteis para alguns problemas
  - inclusive, alguns já podem ter sido usados por vocês!
  - teremos alguns slides de exemplos já já

Ponteiros

- Criando um ponteiro:
  - precisamos de duas informações: tipo de variável, e identificador do ponteiro
    - tipo: indica como leremos a informação
    - identificador: "nome" da variável
  - além disso, adicionamos um asterisco (*), o qual indicará que tratamos de um ponteiro
  - exemplo: **int** *ptr;
    - obs: também podemos criar um vetor de ponteiros

Ponteiros

- Atribuindo e acessando endereços de ponteiros:
  - ponteiros fazer referência a endereços de memória, portanto atribuir algo a um ponteiro significa fazer com que ele aponte para algum lugar
    - ou seja, mudar o endereço armazenado nele
  - exemplo: se quisermos fazer dois ponteiros apontarem para os endereços de duas variáveis x e y na memória, fazemos "p1 = &x;" e "p2 = &y;"
    - se estamos fazendo isso no momento da criação do ponteiro, fazemos "**int** p1 = &x;" e "**int** p2 = &y;"
    - '&' indica que queremos o endereço de memória de uma variável
  - também podemos acessar o endereço de memória de um ponteiro para fazer alguma operação
  - exemplo: se quisermos somar dois inteiros com endereços armazenados nos ponteiros $p1$ e $p2$, podemos fazer "soma = *p1 + *p2;"

Ponteiros

- Utilização de ponteiros em funções:
  - uma função pode receber um ponteiro como argumento
  - deste modo, estamos recebendo o endereço onde alguma informação está guardada
    - neste caso, podemos acessar diretamente a memória para recuperar o valor, e também podemos escrever diretamente sobre o valor anterior
  - exemplo: "**float** complexo2(**float** *r, **float** *t)"

  - uma função também pode retornar um ponteiro
  - neste caso, em vez de apresentar apenas o tipo de retorno, adicionamos um asterisco
    - exemplo: em vez de apenas "**int** fun()", temos "**int** * fun()"

Ponteiros

- Exemplo 1: leitura de variáveis básicas
  - quando lemos uma variável, queremos armazenar o valor na memória
    - mais especificamente, queremos armazenar no endereço de memória onde está a variável lida
  - para isso, colocamos um '&' antes da variável armazenando
    - '&': operador utilizado para acessar o endereço de memória de uma variável
  - uma possível pergunta que pode surgir sobre isso é: por que não utilizamos o '&' para ler uma string?
    - sabemos que lemos uma string até encontrarmos um '\0'
    - então, de fato nós devemos saber apenas onde a string começa!
    - por isso, armazenamos só o endereço do primeiro caractere
  - generalizando a explicação acima, para acessarmos qualquer posição de um vetor V, basta sabermos onde está $V[0]$

Ponteiros

- Exemplo 2: função strchr(str,c)
  - retorna NULL, se não encontrar o caractere *c* na string *str*
  - retorna um ponteiro para o caractere *c*, se encontrar
    - quando o caractere não existia na string, o programa imprimia um "(NULL)"
    - quando imprimíamos a saída desta função, na verdade estávamos imprimindo o ponteiro retornado
    - e no caso de ele não ser nulo, imprimíamos tudo a partir deste ponteiro retornado!
    - desta forma, pulamos, caractere a caractere (byte a byte), imprindo os caracteres até encontrarmos um '\0'
  - obs: o mesmo ocorre com a função strstr(str1,str2)

Ponteiros

- Exemplo 3: alterar o valor de uma variável externa dentro da função
  - quando criamos uma função "**int** Fatorial(**int** n)" e chamamos Fatorial(n) dentro do main, vimos que o valor de *n* na main não é alterado, mesmo que mudemos o valor de *n* dentro da função
    - esta é a ideia da passagem de argumento por valor
    - ou seja, o valor é copiado para uma variável auxiliar da função
  - se desejarmos, podemos utilizar ponteiros para alterar o valor das variáveis fora da função
  - como?
    - se tivermos os ponteiros indicando onde estas variáveis estão armazenadas, podemos escrever algo diretamente nesta posição
    - deste modo, estamos alterando o valor da variável fora da função!
    - esta é a ideia da passagem de argumento por referência
    - ou seja, você referencia onde a variável é armazenada
  - normalmente se utiliza passagem por valor para evitar problemas
  - a passagem por referência pode causar "efeitos colaterais" e só deve ser usada se estes efeitos forem desejáveis

Ponteiros

- Exemplo 4: passar vetor ou matriz como argumento de função
  - considere que você quer montar uma função para ordenar um vetor
    - pode ser uma função que faz o Bubble Sort
  - a ideia é que esta função receba o vetor V de inteiros e a quantidade (inteira) n de elementos no vetor
    - BubbleSort(V, n);
  - pergunta: como passar o vetor como argumento da função?
  - resposta: ponteiros!
  - quando passamos V como argumento, não passamos n variáveis inteiras
    - na verdade, passamos somente a posição de $V[0]$
    - o n como outro argumento nos auxilia a trabalhar somente dentro do vetor, a não exceder a quantidade de posições de V
  - portanto, o protótipo desta função BubbleSort deve ser

    void BubbleSort(**int** *V, **int** n);

Ponteiros

- Exemplo 5: fazer uma função que retorne mais de um valor
  - como exemplo, vamos fazer uma função que eleve um número complexo ao quadrado
  - relembrando: um número complexo é do tipo $a + bi$, onde $i = \sqrt{-1}$
    - a é a parte real e b é a parte imaginária
  - considere, então, que queremos elevar o número complexo n ao quadrado e ainda desejamos aproveitar memória, escrevendo sobre os valores anteriores de a e b
  - podemos criar duas funções para calcular as duas partes em separado
  - porém, podemos criar uma única função para fazer tudo isso ao mesmo tempo, escrevendo o resultado diretamente na memória
  - para isso, utilizaremos a passagem de parâmetros por referência!

Ponteiros

- Exemplo 5: fazer uma função que retorne mais de um valor (continuação)
  - deste modo, teremos uma função como a seguinte:

```
void complexo2(float *r, float *t) {
        float real = (*r * *r) - (*t * *t);
        *t = 2 * *r * *t;
        *r = real;
}
```

  - note que a função sequer precisa de retorno!
    - afinal, ela já escreve o resultado na memória!
  - a partir disto, basta lermos os valores de *a* e *b* e chamarmos "complexo2(&a,&b);"
  - deste modo, calculamos os valores desejados e substituímos os valores anteriores de *a* e *b*

Ponteiros

- Exemplo 6: referência para listas, pilhas e filas
  - vocês verão em Estruturas de Dados!
- Exemplo 7: alocação dinâmica
  - uma das aplicações mais importantes!
  - é o próximo tópico da aula! :)

Alocação dinâmica

- Em vários problemas envolvendo vetores, trabalhamos com uma quantidade de valores que, a princípio, era desconhecida
- Para conseguir armazenar esta quantidade desconhecida, resolvemos de duas maneiras:
  - 1: criar bem mais posições do que provavelmente precisaremos
    - exemplos: questões de strings (criando str[100])
  - 2: ler a quantidade *n* de elementos e criar um vetor com *n* elementos
    - exemplos: questões de vetor (criando V[n])
- Porém, para alocar um vetor com uma quantidade variável de posições, esta maneira de se alocar não é muito legal
  - não é adequado ler *n* e criar "**int** V[n];"
  - não é porque o compilador aceita, que isso é adequado!
- Para casos como este, trabalhamos com alocação dinâmica!

Alocação dinâmica

- O que é alocação dinâmica?
  - é um tipo especial de alocação de variáveis
- Quais as vantagens de utilizar alocação dinâmica?
  - permite uma maior alocação de memória
    - existe um tamanho máximo para se alocar um vetor
    - alocar demais pode resultar em crashs
  - ideal para se utilizar quando o tamanho a ser alocado só é conhecido durante a execução (ou seja, lendo o valor de n)
    - evita correr o risco de criar um vetor exageradamente grande (ou muito pequeno) para comportar os dados
    - também evita o risco de tentar criar um vetor maior do que se pode
  - a memória é mantida de uma função para outra
    - variáveis criadas em uma função eram perdidas uma vez que a função terminava
    - já variáveis alocadas dinamicamente são mantidas até serem liberadas!

Alocação dinâmica

- Quais as vantagens de utilizar alocação dinâmica?
  - a memória pode ser reaproveitada!
    - se utilizamos um vetor e não precisamos mais dele, podemos liberar os espaços de memória ocupados por ele!
  - permite alocar mais e mais memória, sendo que o tamanho é desconhecido
    - desconhecido, neste caso, significa que não é dado pelo usuário
    - ou seja, você aloca mais e mais, até um ponto em que não há mais necessidade de alocar
    - vocês verão aplicações disto em Estruturas de Dados!

Alocação dinâmica

- Como trabalhar com alocação dinâmica?

- Existem algumas funções da biblioteca "stdlib.h" para isso:
  - **void** *malloc (size_t size)
    - aloca memória
  - **void** *calloc (**void** *ptr, size_t size)
    - aloca memória e inicializa com zero
  - **void** *realloc (size_t nitems, size_t size)
    - faz uma tentativa de mudar o tamanho da memória alocada pelas duas funções acima
  - **void** free (**void** *ptr)
    - libera (desaloca) memória alocada por uma das três funções acima
- E o que é "size_t size"?
  - representa a quantidade de bytes que queremos alocar!
  - para isso, utilizaremos a função "sizeof"!
  - exemplo: "sizeof(**int**)" retorna o tamanho, em bytes, de um inteiro (4)

Alocação dinâmica

- Como trabalhar com alocação dinâmica? (continuação)

- Deste modo, considere que queremos armazenar *n* valores em um vetor, com *n* variável (ou seja, recebemos o valor de *n*)

- Antes, fazíamos o seguinte:

```
int n;
scanf("%d",&n);
int V[n];
```

- Utilizando alocação dinâmica, devemos fazer do seguinte modo:

```
int n, *V;
scanf("%d",&n);
V = malloc(n*sizeof(int));
```

  - obs: se desejarmos, após o uso do vetor, podemos liberar a memória alocada utilizando "free(V);"