SCC0222 - Laboratório de Introdução à Ciência de Computação I

Funções

Prof.: Leonardo Tórtoro Pereira

leonardop@usp.br

Funções [1]

- Grupo de declarações com um nome, e que pode ser chamado de algum local do programa.
- Permite estruturar programas em segmentos de códigos que fazem tarefas individuais
- Sintaxe:
  - *tipo nome (parametro1, parametro2, ...) {declaracoes}*

Funções [1]

- *Tipo*
  - Tipo do valor retornado pela função
  - *void* se não retornar nada
- *Nome*
  - Identificador pelo qual a função é chamada
- *Parâmetros (quantos forem necessários)*
  - Cada um consiste num tipo seguido de um identificador, separados por vírgula

Funções [1]

- *Parâmetros (quantos forem necessários)*
  - Age como uma variável local à função
  - Passa argumentos às funções do local no qual ela foi chamada
- *Declarações*
  - É o corpo da função
  - Bloco de declarações que especificam o que a função faz

Funções [1]

- A função *main* é sempre chamada primeiro
  - O código de qualquer outra função só é executado se for chamado pela *main*
    - Direta ou indiretamente

Funções [1]

- Para chamar uma função, é preciso usar seu nome e valores (ou variáveis com tais valores) correspondentes a seus parâmetros
  - Caso ela retorne um valor, é preciso receber esse retorno (ou realizar alguma operação com ele)
    - Alguns níveis de compilação mais lenientes não obrigam que o valor retornado seja utilizado

Funções [1]

```
int add(int a, int b){
    return a+b;
}

int main(){
    int r;
    r = add(10, 20);
    printf("%d\n", r);
    return 0;
}
```

Funções [1]

- Ao chamar uma função, a execução da função que a chamou (e.g.: *main*) é parado e o controle é passado à função chamada.
  - O controle é retornado após a função terminar
- No caso da passagem de argumentos por parâmetros, eles são copiados para as variáveis locais da função na hora da chamada
  - Em nosso exemplo: *a* e *b*

Funções [1]

- A palavra-chave *return* indica o fim da função, retornando o controle de volta ao ponto na qual ela foi chamada
- Como no exemplo temos um retorno do tipo inteiro, o valor passado a esse retorno (30), é passado à variável *r* na função *main*
- Não é preciso “receber” o retorno em uma variável! Ele pode ser usado diretamente como parâmetro de uma função, por exemplo.

Funções [1]

```c
int add(int a, int b){
    return a+b;
}

int main()
{
    int x=10;
    printf("%d\n", add(x, 20));
    return 0;
}
```

Funções [1]

- Uma função pode não retornar nenhum valor. São as funções *void*. Você também pode usar essa palavra-chave para reforçar que uma função não recebe parâmetros (era obrigatório nos compiladores de C antigos)
- Ainda é preciso usar parênteses para chamar a função!

```
void printHello(void){
    printf("Hello World");
}
```

Funções [1]

- Retorno da *main* é considerado 0 caso nada seja retornado!
  - Só da main!
- Outros valores podem ter diferentes interpretações dependendo do ambiente em que você está executando

Funções [2]

- Argumentos podem ser passados por referência, para que possam ser acessados de dentro de um função.
  - Use o * após o tipo do parâmetro para passar uma variável por referência
- Nesse caso, não é passada mais uma cópia da variável, mas sim a própria, seu endereço.
- Qualquer alteração na função é feita nas variáveis que foram passadas!

Funções [2]

- É preciso passar um *endereço* para as funções que requerem argumentos por referência
  - Para um tipo primitivo, isso requer o operador &
  - Em um vetor, a variável é o endereço para a 1ª posição
- Lembre-se de alterar o *valor* do endereço na função usando o operador *
  - Se você usar só o nome da variável, vai alterar o endereço!

Funções [2]

```c
int divide(int a, int b, int *resto)
{
    *resto = a%b;
    return a/b;
}
int main()
{
    int r, resto;
    r = divide(30, 20, &resto);
    printf("resultado: %d resto: %d\n", r, resto);
    return 0;
}
```

Funções [2]

- Um adendo sobre eficiência
  - Fazer a cópia de um tipo primitivo ou usar seu endereço tem custos bem parecidos.
  - Porém, se você for passar argumentos bem maiores que um tipo primitivo (um vetor muito grande, ou estruturas de dados mais complexas que veremos depois - *structs*), isso pode ficar bem pesado

Funções [2]

- Para o caso de você passar argumentos grandes que **não serão alterados**, é possível passar seus endereços (passagem por referência), mas com a palavra-chave *const*, que “obriga” o programador a não alterar o valor de determinada variável (gera um erro de compilação caso exista alteração na variável precedida por *const*)
- É uma técnica mais avançada, pode deixar o código mais difícil de ler, e não **precisa** ser usada por vocês agora :)

Funções [2]

```
int divide(const int *a, const int *b, int *resto)
{
    *resto = *a%*b;
    return *a/(*b);
}
int main()
{
    int r, resto;
    int a = 30, b = 20;
    r = divide(&a, &b, &resto);
    printf("resultado: %d resto: %d\n", r, resto);
    return 0;
}
```

Funções [1, 2, 4, 5, 6]

- Funções devem ser declaradas antes de serem chamadas
- Caso não seja declarada, o compilador assume uma função do tipo "*int function name()*"
  - Se no final a função não for assim, o compilador mostrará um erro
- *O C considera por padrão um retorno de *int* se nada for declarado*
- Porém, é possível declarar um protótipo de função antes, e seu corpo depois do uso

Funções [2]

```
int divide(int a, int b, int* resto);
int main()
{
    int r, resto;
    r = divide(30, 20, &resto);
    printf("resultado: %d resto: %d\n", r, resto);
    return 0;
}
int divide(int a, int b, int *resto)
{
    *resto = a%b;
    return a/b;
}
```

Funções [7]

- Funções são globais por padrão
- Porém, é possível restringir o escopo da função para apenas o arquivo *.c* na qual ela se encontra usando a palavra-chave *static*
- Isso pode ser útil mais para frente, quando aprendermos a fazer programas maiores, que usam diversos arquivos *.c*
- *static int add(int a, int b);*

Funções [8]

- Vetores são sempre passados como endereço para a função
- Mas você não sabe o tamanho! É sempre necessário passar o tamanho do vetor!
- sizeof(arr) não vai adiantar!

Funções [8]

```
void fun(int arr[])  // SAME AS void fun(int *arr)
{
   unsigned int n = sizeof(arr)/sizeof(arr[0]);
   printf("\nArray size inside fun() is %d", n);
}
int main()
{
   int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
   unsigned int n = sizeof(arr)/sizeof(arr[0]);
   printf("Array size inside main() is %d", n);
   fun(arr);
   return 0;
}
```

Passing array to function in C

Pointer to arr

Length of arr

```c
void func( int a[] , int size )
{

}

int main( )
{
      int n=5;
      int arr[5] = { 1, 2, 3, 4, 5 };
      func( arr , n);
      return 0;
}
```

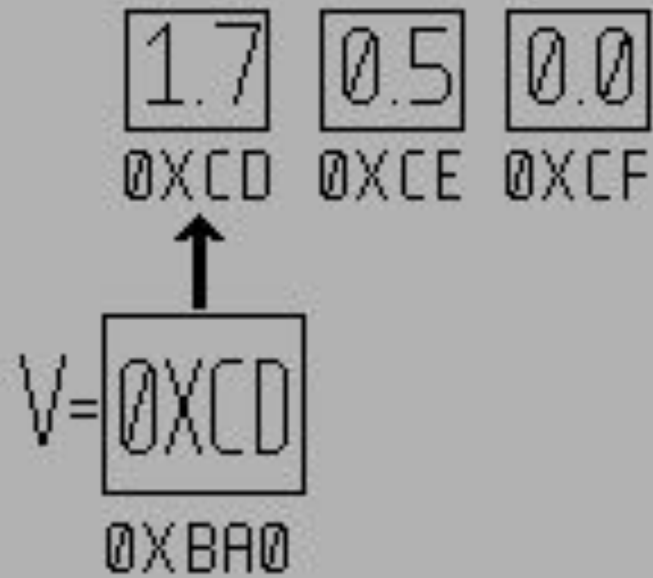
Pointer a takes the base address of array arr

The length of arr is passed. It is compulsory to pass size as is just a pointer

Fonte: [8]

Funções [8]

- Se você quiser alterar o endereço de um vetor **dentro** de uma função, você precisa passar o endereço do ponteiro do vetor (&)
- Se você alterar o valor dele dentro da função, irá perder a referência deste ponteiro na variável que o armazena **fora** da função (*main*)



Fonte: Leonardo :)

Referências

1. http://www.cplusplus.com/doc/tutorial/functions/
2. https://www.geeksforgeeks.org/functions-in-c/
3. https://en.wikibooks.org/wiki/C_Programming/Procedures_and_functions
4. https://www.geeksforgeeks.org/what-is-the-purpose-of-a-function-prototype/
5. https://www.geeksforgeeks.org/implicit-return-type-int-c-language/
6. https://www.geeksforgeeks.org/g-fact-95/
7. https://www.geeksforgeeks.org/what-are-static-functions-in-c/
8. https://www.geeksforgeeks.org/how-arrays-are-passed-to-functions-in-cc/
9. https://www.geeksforgeeks.org/return-local-array-c-function/
10. https://www.geeksforgeeks.org/pass-2d-array-parameter-c/