Programação Estruturada

- Funções (cont.) -

Prof. Ulysses Santos Sousa

ulyssessousa@ifma.edu.br

Aula 07

Roteiro

- Classes de armazenamento
- O Pré-Processador C
- Principais diretivas

Classes de armazenamento

- **A classe de armazenamento de uma variável determina:**
  - Em qual momento será criada;
  - Em qual momento será destruída;
  - Em que parte da memória será armazenada;
  - Qual será o seu valor inicial.
- **São quatro as classes de armazenamento em C:**
  - auto
  - extern
  - static
  - register

Classes de armazenamento

- **Classe *auto***
  - São variáveis "visíveis" e "acessíveis" somente às funções em que foram declaradas.

São criadas em tempo de execução, especificamente quando o programa encontra a instrução de sua declaração, e são destruídas ao término da execução do bloco a qual pertencem.

Classes de armazenamento

- **Classe *auto***
  - Podem ser acessadas somente pelas instruções do mesmo bloco.
  - Quando uma variável automática é criada, o programa não a inicializa com nenhum valor específico.

> Variáveis automáticas conterão um valor inicial aleatório, chamado ***lixo***.

Classes de armazenamento

- **Classe *auto***
  - A palavra chave **auto** pode ser usada para especificar uma variável automática, mas não é necessária, visto que a classe **auto** é padrão.

```
int main()
{
     int n;
     ...
}
```

equivale a

```
int main()
{
     auto int n;
     ...
}
```

Classes de armazenamento

- **Classe *extern***
  - São variáveis declaradas fora de qualquer função.
  - A instrução de declaração de uma variável externa é idêntica à de uma variável automática.
  - O acesso a elas é permitido por todas as funções do programa, e elas existirão enquanto estiver sendo executado.

Essas variáveis são criadas em tempo de compilação, quando o programa estiver sendo compilado, e são inicializadas com zero por falta de inicialização explícita.

Classe de armazenamento

- **Classe *extern***

```c
#include <stdio.h>
#include <stdlib.h>

int i;
int j=234;

void func(){
    i = 25; j = 48;
}

int main()
{
    printf("%d\t%d\n", i, j);
    func();
    printf("%d\t%d\n", i, j);
    system("pause");
    return 0;
}
```

i inicializa com 0 (zero)
j inicializa com 234

Saída:
0 234

Saída:
25 48

Classes de armazenamento

- **Classe *extern***
  - Variáveis automáticas têm precedência sobre variáveis externas.
  - A palavra chave ***extern*** não é usada para criar variáveis da classe ***extern*** e sim para informar ao compilador que a variável em questão foi criada em outro programa-fonte, compilado separadamente, que será linkeditada com este para formar o programa final.

```
int main()
{
      extern int x;
      ...
}
```

Não cria a variável

Classes de armazenamento - extern

arquivoPrincipal.c

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    extern int z;
    printf("z = %d\n", z);
    return 0;
}
```

outroArquivo.c

```c
int z = 43;

void imprimirLinha()
{
    printf("---------------\n");
}
```

Classes de armazenamento

- **Classe *static***
  - São conhecidas somente na função que as declaram, porém os seus valores são mantidos mesmo quando a função termina.

Essas variáveis são criadas em tempo de compilação, quando o programa estiver sendo compilado, e são inicializadas com zero por falta de inicialização explícita.

Classes de armazenamento

- Classe *static*

```c
#include <stdio.h>
#include <stdlib.h>

int teste()
{
    static int quant;
    quant++;
    return quant;
}

int main()
{
    int y = 10;
    printf("A funcao teste() foi chamada %d vezes.\n", teste());
    printf("A funcao teste() foi chamada %d vezes.\n", teste());
    printf("A funcao teste() foi chamada %d vezes.\n", teste());
    return 0;
}
```

Classes de armazenamento

- **Classe *static extern***
  - A classe ***static*** pode ser associada a declarações externas, criando um mecanismo de privacidade.

Uma variável estática externa tem as mesmas propriedades de uma variável externa, exceto pelo fato de que variáveis externas podem ser usadas em qualquer parte do programa, enquanto variáveis estáticas externas somente podem ser acessadas pelas funções do mesmo programa-fonte e definidas abaixo de suas declarações.

Classes de armazenamento – *static extern*

arquivoPrincipal.c

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    extern int z;
    printf("z = %d\n", z);
    return 0;
}
```

outroArquivo.c

```c
static int z = 43;

void imprimirLinha()
{
    printf("--------------\n");
}

void imprimeZ()
{
    printf("%d\n", z);
}
```

Code::Blocks | Search results | Cccc | Build log | **Build messages** | CppCheck

File	Line	Message
		=== Build: Debug in ClassesVariaveis (compiler: GNU GCC Compiler) ===
obj\Debug\main.o		In function `main':
F:\OneDrive\Ba...	7	undefined reference to `z'
		error: ld returned 1 exit status
		=== Build failed: 2 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===

Classes de armazenamento

- **Funções *static extern***
  - Toda função C é da classe ***extern***, ou seja, é visível a todas as outras.
  - No entanto, uma função também pode ser estática.

Uma função ***static extern*** é acessível somente às funções do mesmo programa-fonte definidas abaixo dela, ao contrário das outras que podem ser acessadas por outros arquivos compilados em módulos separados.

Classes de armazenamento - Funções *static extern*

arquivoPrincipal.c

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    imprimeZ(); //CORRETO
    imprimirLinha(); //ERRO
    return 0;
}
```

outroArquivo.c

```c
static int z = 43;

static void imprimirLinha()
{
    printf("--------------\n");
}

void imprimeZ()
{
    printf("%d\n", z);
    imprimirLinha();
}
```

A variável *z* e a função ***imprimeLinha()*** são acessíveis apenas pelas funções deste arquivo-fonte onde foram definidas.

Classes de armazenamento

- **Classe *register***
  - Indica que, se possível, a variável associada deve ser guardada fisicamente numa memória de acesso muito mais rápido chamada ***registrador***.
  - São semelhantes às automáticas, mas se aplicam apenas às variáveis do tipo ***int*** e ***char***.
  - São utilizadas para aumentar a velocidade de processamento. Logo, o programador deve escolher as variáveis que são mais frequentemente acessadas e declará-las como da classe ***register***.
  - Geralmente, utiliza-se como ***register*** as variáveis de laço e os argumentos de funções.

Classes de armazenamento

- Classe *register*

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
    int i, j;
    register int m, n;
    double t;
    t = clock();
    for(i = 0; i < 50000; i++)
        for(j=0; j < 50000; j++);
    t = clock() - t;
    printf("Tempo dos lacos nao register: %lf\n", t);

    t=clock();
    for(m = 0; m < 50000; m++)
        for(n=0; n < 50000; n++);
    t = clock() - t;
    printf("Tempo dos lacos register: %lf\n", t);

    system("pause");
    return 0;
}
```

O Pré-Processador C

- É um programa que examina o programa-fonte em C e executa certas modificações nele antes da compilação.
- As modificações são realizadas com base em instruções chamadas de **diretivas**.
- O pré-processador faz parte do compilador e é executado automaticamente antes da compilação.

As diretivas iniciam com o caractere #

O Pré-Processador C

- **Principais ações:**
  - Inclusão de novos arquivos;
  - Definição de constantes e macros;
  - Compilação condicional do código do programa;
  - Execução condicional das diretivas do pré-processador.

As diretivas podem ser colocadas em qualquer lugar do programa, mas geralmente são escritas no início, antes de qualquer função.

Elas se aplicam somente do ponto onde são escritas ao final do programa-fonte.

O Pré-Processador C

- Diretivas mais comuns:
  - #define
  - #undef
  - #include
  - #if
  - #ifdef
  - #ifndef
  - #else
  - #elif
  - #endif
  - #error

Diretiva *#define*

- **Utilizadas para criação de constantes simbólicas e macros.**
- **Forma geral:**

#define identificador texto_de_substituição

  - todas as ocorrências subsequentes de ***identificador*** serão substituídas por ***texto_de_substituição*** antes do programa ser compilado.

Diretiva *#define*

- **Exemplo:**

```
#define PI 3.14159
```

  - Substitui todas as ocorrências subsequentes da constante simbólica ***PI*** pela constante numérica ***3.14159***.
  - Constantes simbólicas permitem que o programador crie um nome para a constante e use o nome ao longo de todo o programa.

Diretiva *#define*

- **Macro**
  - É uma operação definida em uma diretiva ***#define*** do pré-processador.
  - Macro sem Argumentos:
    - é processada como uma constante simbólica.
  - Macro com argumentos:
    - Os argumentos são substituídos no texto de substituição, então a macro é expandida, ou seja, o texto de substituição substitui o identificador e a lista de argumentos no programa.

Diretiva *#define*

- Macro (exemplo):

```
#define PI 3.14159
#define AREA_CIRCULO(x) (PI*x*x)
```

No código, a instrução

```
x =  AREA_CIRCULO(4)
```

Seria substituída por:

```
x = (3.14159 * (4) * (4));
```

Diretiva #*define*

- **Macro:**
  - A macro AREA_CIRCULO, poderia ser definida como uma função:

```
double areaCirculo(double x)
{
    return 3.14159 * x * x;
}
```

Diretiva *#define*

- **Macro:**
  - Para evitar erros, deve-se envolver cada argumentos com parênteses.

```
#define PROD(x,y) (x*y)
...
z = PROD(2 + 3, 4);

A instrução processada é:
z = PROD(2 + 3 * 4);  //resultado não esperado

Solução:
#define PROD(x,y) ((x)*(y))
```

Diretiva #*define*

- **Macro**
  - Vantagens:
    - Como macros são simples substituições dentro de programas, a execução é mais rápida que a utilização de funções.
    - Não há a necessidade de especificar o tipo dos argumentos.
  - Desvantagem:
    - O código do programa será aumentado, porque o código da macro será duplicado cada vez que esta for chamada.

Diretiva *#undef*

- **Anula a definição de uma constante simbólica ou macro.**

```
#undef AREA_CIRCULO
```

  - Para remover uma macro, somente o nome da macro deve constar na diretiva ***#undef***, não devemos incluir a lista de argumentos.

Diretiva *#include*

- **Utilizada para incluir outro arquivo no programa-fonte.**

> **O Compilador substitui a diretiva *#include* do programa pelo conteúdo do arquivo indicado, antes de compilar o programa.**

  - Exemplo:
    - #include <stdio.h>

Diretiva *#include*

- **O nome do arquivo incluído pode ser delimitado pelos símbolos <> ou “”.**
- **<>**
  - Utiliza-se quando o arquivo a ser incluído está na pasta **include**.
- **“”**
  - Utiliza-se quando o arquivo a ser incluído está na mesma pasta do arquivo que está sendo compilado.
  - Esse método é utilizado normalmente para incluir arquivos de cabeçalho definidos pelo usuário.

Compilação condicional

- **O pré-processador oferece diretivas que permitem a compilação condicional de um programa.**
- **Permitem a escrita de códigos com maior portabilidade de uma máquina para outra ou de um ambiente para outro.**
- **Diretivas:**
  - #if, #ifdef, #ifndef, #elif, #else e #endif

Compilação condicional

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #define BRASIL 1
4   #define FRANCA 2
5   #define ESPANHA 3
6   #define EUA 4
7   #define PAIS BRASIL
8
9   int main()
10  {
11      #if PAIS == BRASIL
12          printf("Seja bem-vindo!\n");
13      #elif PAIS == FRANCA
14          printf("Bienvenue!\n");
15      #elif PAIS = ESPANHA
16          printf("Sea bienvenido!\n");
17      #else
18          printf("Welcome!\n");
19      #endif // PAIS
20      return 0;
21  }
```

Neste exemplo, a linha **12** seria considerada durante a compilação e as linhas 14, 16 e 18 seriam descartadas, pois o valor da constante **PAIS** foi definida com o valor da constante **BRASIL**.

Compilação condicional

```
1  #include <stdio.h>
2  #define MAX 100
3  int main()
4  {
5      #ifndef MAX
6          printf("MAX nao foi definida.\n");
7      #else
8          printf("MAX foi definida.\n");
9      #endif // MAX
10
11     return 0;
12 }
```

Neste exemplo, a linha **6** não seria compilada e a linha **8** seria, pois a constante **MAX** foi definida na linha **2**.

Referências

- MIZRAHI, V. V. Treinamento em Linguagem C. 2ª Edição. São Paulo: Person Prentice Hall, 2008.

- SCHILDT, H. C, Completo e Total. 3ª Edição revista e atualizada; Tradução e revisão técnica: Roberto Carlos Mayer. São Paulo: Makron Books, 1996.