Organização do código

À medida em que um programa cresce em tamanho e funcionalidades, seu código-fonte deve ser organizado corretamente para facilitar sua compreensão, manutenção e evolução. É importante quebrar o código-fonte em arquivos separados, dividindo-o de acordo com os módulos e/ou funcionalidades do sistema.

São duas as razões para dividir um programa grande e complexo em módulos:

1. **separation of concerns** Não sei traduzir a expressão; seria algo como "focar em uma coisa de cada vez". Cada funcionalidade do programa é encapsulada em um módulo, e um conjunto (relativamente) pequeno de funções e de variáveis é disponibilizado aos demais módulos. Isso garante que a programadora pode se concentrar em um pedaço pequeno do problema, o que geralmente simplifica as tarefas de projeto e implementação;

2. **evitar efeitos colaterais** Se, ao longo de todo o programa, todas as estruturas de dados são visíveis, e podem ser alteradas, então é muito provável que ao longo da vida do programa, eventuais remendos ou extensões, provoquem efeitos colaterais. Uma alteração introduzida sem cuidado pode causar modificações incompletas nas estruturas de dados e então provocar erros que são difíceis de reproduzir e detectar.

Por enquanto, estamos lidando com programas pequenos e simples e as ideias de separação e efeitos colaterais podem parecer bobagem de quem não tem o que fazer. Não, estas ideias são importantíssimas. Não se esqueça delas.

Modularidade

Um **módulo** provê uma *funcionalidade* a outros componentes do programa. Tipicamente, a funcionalidade é construída sobre uma estrutura de dados e consiste de um conjunto de operações que alteram o estado desta estrutura.

O módulo **exporta** algumas poucas funções que permitem alterar a estrutura de dados de uma forma bem-definida. Estas funções exportadas são descritas pela *application programming interface* (API) do módulo (páginas de manual).

O módulo **esconde** outras funções que permitem manipulações das estruturas de dados que, a bem de segurança e integridade do serviço, devem ficar indisponíveis ao programador que usa o módulo. O módulo também pode esconder a definição da estrutura de dados, enquanto expõe somente um conjunto pequeno de funções para a sua manipulação.

Em sua forma mais simples, um *módulo* consiste de uma estrutura de dados e das funções que operam sobre esta estrutura, tais como inicialização da estrutura, inserção, remoção, e modificação de elementos.

A forma típica de organizar o código de um módulo é agrupar todas as definições de tipos de dados, constantes, e protótipos das funções relevantes num arquivo de cabeçalho, que será importado (#include) em todos os arquivos com código do módulo.

O código das funções pode ser distribuído em um ou mais arquivos com sufixo ".c" e que importam (#include) o arquivo de cabeçalho.

Ao dividir o código-fonte em arquivos separados, alguns cuidados devem ser tomados:

- Agrupe as definições de constantes, tipos de dados e estruturas de dados num arquivo de cabeçalho

".h", com o mesmo prefixo (nome) do arquivo ".c" onde estão definidas as *funções exportadas* que atuam sobre a(s) estrutura(s) de dados.

Neste ".h" devem estar também os protótipos de todas as funções que são *exportadas* pelo módulo.

- Somente efetue inclusões ("#include") de arquivos de cabeçalho (.h).

Considere o arquivo "abobrinha.h" que contém os protótipos das funções **exportadas** (tornadas públicas) pelo módulo, além das definições das constantes relevantes ao programador que fará uso do módulo para a manipulação de abóboras. O arquivo "abobrinha.c" contém as declarações das estruturas de dados e das funções **exportadas** que operam sobre estas estruturas, enquanto que o arquivo "sementes.c" contém as declarações das funções auxiliares na manipulação de abóboras e que devem ficar **invisíveis** ao usuário do módulo, possivelmente porque seu uso pode comprometer a integridade das estruturas de dados.

Qualquer programa que manipule abóboras deve incluir "abobrinha.h" para ter acesso às funções públicas, e deve ser compilado juntamente com "abobrinha.c" e "sementes.c".

Pecados imperdoáveis

- **fazer inclusão de arquivos de código (#include "arquivo.c")**
- **colocar código real (for, if, while, ...) em arquivos de cabeçalho (.h)**

Exemplo

O exemplo a seguir implementa um conjunto de funções que permitem definir e operar sobre números complexos.

O arquivo que usa o módulo de números complexos (neste exemplo, "main.c") deve incluir todos os arquivos de cabeçalho necessários para sua compilação e também deve definir a função "main":

```
// -- main.c ------------------------------------------------
#include "complex.h"

int main()
{
  complex_t a, b, c ;             // tipo definido em complex.h

  complex_define (&a, 10, 17) ;   // operação definida em complex.h
  complex_define (&b, -2, 4) ;
  complex_sum (&c, a, b);         // operação definida em complex.h
  ...
}
// ------------------------------------------------------------
```

Como o arquivo "main.c" provavelmente não define funções (ou estruturas, tipos, etc) que serão usadas em outros programas, não é necessário criar um arquivo "main.h".

Por sua vez, o arquivo de cabeçalho "complex.h" deve declarar somente informações públicas para serem **exportadas**, tais como os tipos de dados e protótipos de funções que devem ser fornecidos para quem utilizar as funcionalidades providas por "complex.c".

```
// -- complex.h ---------------------------------------------
#define __COMPLEX__

typedef struct {
  float r,i;
} complex_t ;

void complex_define (complex_t *v, float r, float i) ;
void complex_sum    (complex_t *v, float r, float i) ;
```

```
// ------------------------------------------------------------
```

O arquivo correspondente "complex.c" contém as informações privativas do módulo: estruturas de dados internas, variáveis globais e o código das funções. Esse arquivo deve incluir todos os cabeçalhos necessários à implementação das funções.

```
// -- complex.c -----------------------------------------------

#include <math.h>       // biblioteca de matemática
#include "complex.h"    // declarações sobre números complexos

// hidden internal function, used only in this file, declared STATIC
static void convert_polar_rect (float r, float a, float *x, float *y)
{
  // function body
}

// function "exported" through complex.h
void complex_define (complex_t *v, float r, float i)
{
  // function body
}

// function "exported" through complex.h
void complex_sum (complex_t *v, float r, float i)
{
  // function body
}
// ------------------------------------------------------------
```

Em resumo:

- "complex.c": contém a implementação das funções de manipulação de números complexos;
- "complex.h": contém a interface (protótipos) das funções **exportadas** definidas em "complex.c";
- "main.c": programa que usa as funções declaradas em "complex.h" e implementadas em "complex.c"; e

Para compilar, devem ser explicitados os dois arquivos fonte (main.c e complex.c), o nome do executável (my_prog), e que a biblioteca de matemática deve ser usada (-lm).

```
gcc -Wall main.c complex.c -o my_prog -lm
```

O arquivo "complex.c" também pode ser compilado separadamente, gerando um arquivo objeto "complex.o" que poderá ser ligado ao arquivo "main.o" posteriormente. Essa organização torna mais simples a construção de programas grandes e a distribuição de código binário para incorporação em outros projetos (reuso de código). Além disso, essa estruturação agiliza a compilação de grandes projetos, através do sistema Make, que estudaremos em uma próxima aula.

Regras de Escopo de Visibilidade

A palavra 'visibilidade' no título se refere aos locais, no código fonte, em que variáveis ou funções podem ser referenciadas. 'Escopo' refere-se ao conjunto de locais em que uma certa variável ou função é visível.

Escopo de Variáveis

Uma variável é **global** se ela está definida fora do corpo de uma função. Por global entenda-se que a variável pode ser lida e atualizada em qualquer das funções que são definidas depois da sua definição -- depois no arquivo fonte.

Uma variável é **local** se ela está definida no corpo de uma função, e somente pode ser lida e atualizada dentro daquela função. Os argumentos de uma função são variáveis locais à função.

```
void f(int a)
{
   int p;       // p e a são locais à f(), v é inacessível a f()
   ...
}

int v;          // variável global

void g(int b)
{
   int q;       // v, b e que são visíveis dentro do corpo de g()
   ...
}
```

O **escopo** de uma variável é o bloco em que ela é definida. Por bloco entenda-se o que está entre um par de chaves: { ... }.

Se a variável é definida *fora de um bloco* então ela pode ser vista em todos os blocos que seguem sua definição.

Blocos podem ser aninhados, e uma definição num bloco interno se sobrepõe a uma definição num bloco externo.

```
{
   int a, b, c;

   a = b = c = 2;
   printf("%d %d %d\n", a, b, c); // imprime 2 2 2

   {
      int a;

      a = 5;
      b = 1;
      printf("%d %d %d\n", a, b, c); // imprime 5 1 2
   }

   printf("%d %d %d\n", a, b, c); // imprime 2 1 2
}
```

Blocos são ditos 'paralelos' se não são aninhados. Funções são blocos paralelos. 'Paralelo' aqui é com relação ao escopo de visibilidade e não com a forma de execução das funções.

```
{
   int a;              // a é visível aos blocos 1 e 2

   // bloco_1
   {
      int b, c;        // b e c são locais ao bloco_1

      a = b = c = 2;
      printf("%d %d %d\n", a, b, c); // imprime 2 2 2
   }

   // bloco_2
   {
      int b, c;        // b e c são locais ao bloco_2

      a = a + 5;
      b = 1;
      c = 3;
```

```
      printf("%d %d %d\n", a, b, c); // imprime 7 1 3
   }
}
```

Uma variável global declarada como **static** só é visível no arquivo em que é declarada.

Classes de armazenagem

São quatro as classes de armazenagem de variáveis:

- **auto** Variáveis automáticas são aquelas definidas dentro de um bloco. Quando a execução 'entra' no bloco, espaço é alocado para as variáveis automáticas; quando a execução 'sai' do bloco, o espaço é desalocado e seu conteúdo se perde. Variáveis automáticas *devem* ser inicializadas antes que sejam lidas. Variáveis locais e argumentos de funções são da classe **auto**. Não é necessário enfeitar a declaração de variáveis locais com **auto**.
- **extern** Variáveis declaradas com **extern** são definidas em outro arquivo, e o **extern** informa ao compilador que a variável está definida (e seu espaço alocado) "em outro lugar".
- **register** O programador declara variáveis com a classe **register** na esperança de que o compilador dê a estas variáveis atenção especial, forçando sua alocação aos registradores do processador, porque assim pode-se aumentar o desempenho do programa. Compiladores modernos ignoram esta declaração porque eles são, geralmente, melhores do que os humanos para julgar questões de eficiência de código.
- **static** Variáveis locais da classe **static** preservam seus valores entre execuções consecutivas do bloco em que estão declaradas.

Vejamos um exemplo de variáveis estáticas.

```
static int x;               // x só é visível neste arquivo;

int f(void)
{
   static int cnt = 0;      // cnt mantém seu valor entre execuções

   return( cnt++ );
}

int main(void)
{
   for (x = 0; x < 5; x+=1)
     printf( "%d ", f() );    // imprime 0 1 2 3 4

   return(0);
}
```

Um outro aspecto importante da organização do código é o uso de declarações "extern" para variáveis globais usadas em vários arquivos de código-fonte. Esta página contém uma excelente explicação sobre o uso correto da declaração "extern".