Fundamentos de programação

aula 10: Programando com caixinhas

1 Introdução

Já faz algumas aulas que nós temos organizado os nossos programas em etapas.

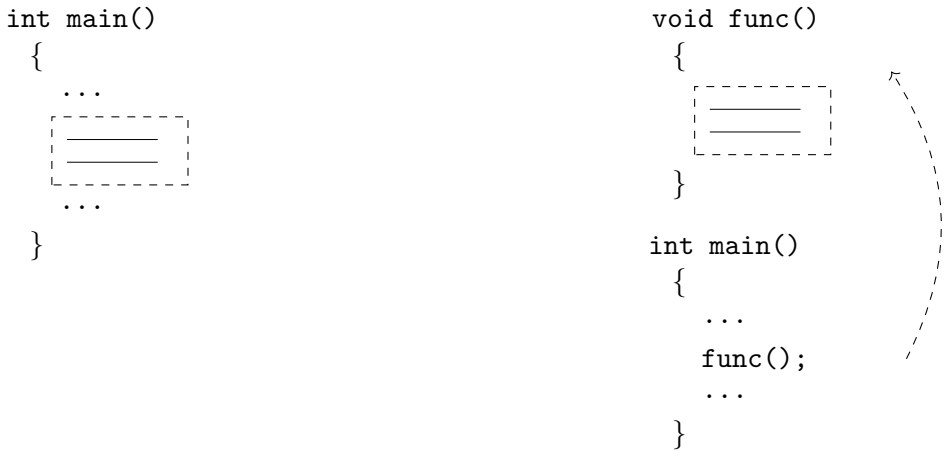
E faz mais tempo ainda que nós temos raciocinado em termos de caixinhas.

Essas ideias são tão naturais e tão úteis no contexto da programação que toda linguagem de programação oferece alguma maneira de trabalhar com elas.

Na linguagem C essa ideia aparece na forma de *funções*.

Por exemplo, imagine que existe uma parte do seu programa que é natural pensar como uma caixinha.

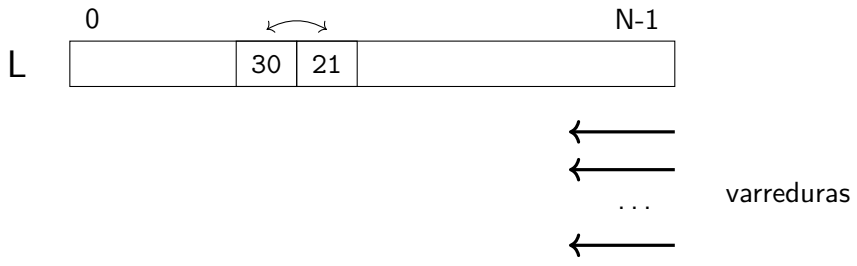
A ideia então é colocar essa parte literalmente dentro de uma caixinha (ou função) e colocar no seu lugar uma "*chamada*" a essa caixinha (ou função)



Vejamos um exemplo concreto.

A ideia da ordenação da bolha é realmente muito simples:

- fazer varreduras sucessivas na lista



onde uma varredura percorre a lista da direita para a esquerda,
trocando elementos invertidos de lugar

Daí que, faz sentido pensar na operação de varredura como uma caixinha.

```
int main()
{
    for (i=0; i<N-1; i++)
      ┌──────────────────────────────────────────────────┐
      │  for (j=N-1; j>0; j--)                           │
      │      if ( L[j] < L[j-1] )                        │   varredura
      │       {                                          │
      │         aux = L[j]; L[j] = L[j-1]; L[j-1] = aux; │
      │       }                                          │
}     └──────────────────────────────────────────────────┘
```

Na linguagem C, nós colocamos esse trecho de código dentro de uma caixinha da seguinte maneira

```
void varredura()
 {
    int j, aux;

    for (j=N-1; j>0; j--)
     {
       if ( L[j] < L[j-1] )
        {
          aux = L[j];   L[j] = L[j-1];   L[j-1] = aux;
     }  }
 }
```

E daí, nós reescrevemos o programa principal como

```
int main()
 {
    for (i=0; i<N-1; i++)
     {
       varredura();
 }   }
```

Observações

- Note que a declaração das variáveis `j` e `aux` agora foi feita dentro da caixinha

  ```
  void varredura()
   {
      int j, aux;
      ...
   }
  ```

  Isso faz sentido, porque essas variáveis só são utilizadas nesse lugar.

- Pensando dessa maneira, também faria sentido colocar a declaração da variável `i` dentro da função `main`

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

  porque, afinal de contas, a função `main` é uma caixinha como outra qualquer.

- Mas, a declaração da lista deve ficar do lado de fora

```
#define  N  10

#int  L[N]  = { ... };
```

  Porque, em princípio, ela pode ser usada em qualquer lugar do programa.

2 Programando com caixinhas

Pensar em termos de caixinhas acaba dando outras ideias pra gente.

Por exemplo, a gente pode pensar em uma caixinha que imprime o conteúdo da lista.

E daí, a gente chama essa caixinha antes e depois da ordenação, para ver se a coisa deu certo

```
int main()
 {
    int  i;

    imprime_lista();
     for (i=0; i<N-1; i++)
      {
        varredura();
      }
    imprime_lista();
 }
```

E agora, basta escrever o código dessa nova caixinha

```
void imprime_lista()
 {
    int  i;

    for (i=0; i<N; i++)   {  printf("%3d ", L[i]); }
    printf("\n");
 }
```

Quando a gente vê as coisas dessa maneira, também faz sentido colocar a ordenação dentro da sua própria caixinha

```
void ordenacao_bolha()
 {
    int  i;

    for (i=0; i<N-1; i++)
     {
       varredura();
 }   }

int main()
 {
    imprime_lista();
    ordenacao_bolha();
    imprime_lista();
 }
```

Observações

- Note que a nossa função `main` agora é um programa escrito em português.
- E note que agora nós temos um programa que tem uma caixinha que chama outra caixinha.

Outros exemplos:

a. **Mais uma caixinha**

Imagine que a gente queria testar o programa de ordenação com uma lista de tamanho 1000.

Daí, certamente seria muito chato ter que pensar em 1000 números aleatórios para colocar na lista no início do programa.

Mas, nós podemos ter uma caixinha que faz isso.

Quer dizer, a linguagem C tem a função

```
rand()
```

(mais uma caixinha) que nos dá um número pseudo-aleatório.

Daí, basta fazer uma caixinha que chama a função `rand`() sucessivamente, e coloca esses números na lista.

A coisa fica assim

```
void inicializacao_aleatoria()
 {
    int i;

    for (i=0; i<N; i++)   L[i] = rand();
 }

( . . . )

int main()
 {
    inicializacao_aleatoria();
 // imprime_lista();
    ordenacao_bolha();
 // imprime_lista();
 }
```

Observação

- Quando nós colocamos o símbolo `//` em alguma linha do programa, a linguagem C ignora tudo o que vem depois desse símbolo nessa linha.

  Aqui, nós usamos esse recurso para desabilitar a chamada à função que imprime a lista.

  Isso faz sentido, porque nós não queremos imprimir 1000 números na tela, não é?

◇

b. **Diálogo entre caixas**

O primeiro programa de ordenação que nós vimos na aula 06 era baseado na seguinte ideia

- encontrar o maior elemento da lista, e movê-lo para a última posição
- encontrar o segundo maior elemento da lista, e movê-lo para a penúltima posição
- e por aí vai ...

Esse algoritmo é conhecido como a *ordenação por seleção*.

Agora, nós vamos reorganizar o programa de ordenação por seleção em termos de caixas.

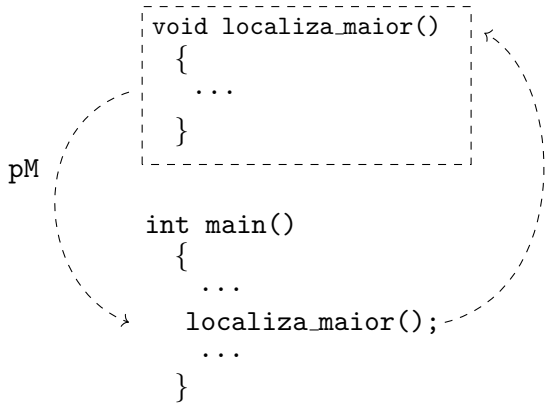
A ideia natural consiste em ter uma caixa que realiza a seguinte função

- localizar a posição do maior elemento da lista

o que corresponde ao seguinte trecho de código

```
pM = 0;
for (i=1; i<N; i++)
  if ( L[i] > L[pM] )  pM = i;
```

Mas, veja que agora a caixa precisa dizer para o programa principal que posição é essa



Na linguagem C isso é feito da seguinte maneira

```
int localiza_Maior()
 {
     . . .
   return (pM);
 }

int localiza_Maior()
 {
     . . .
   p = localiza_Maior();
     . . .
 }
```

Quer dizer, agora a função é definida como

```
int  localiza_Maior ()
```

par indicar que ela vai retornar um número inteiro.

A instrução

```
return(pM);
```

está indicando que número é esse.

E no momento da chamada, nós escrevemos

```
p  =  localiza_Maior();
```

de modo que o valor retornado pela função é armazenado na variável `p`.

A coisa fica assim

```
int localiza_Maior()
 {
   int i, pM;

   pM = 0;
   for (i=1; i<N; i++)
     if ( L[i] > L[pM] )  pM = i;

   return (pM);
 }

int main()
 {
   int i, p, aux;

   for (i=1; i<N; i++)
    {
      p = localiza_Maior();

      aux = L[N-i];  L[N-i] = L[p];  L[p] = aux;
 }  }
```

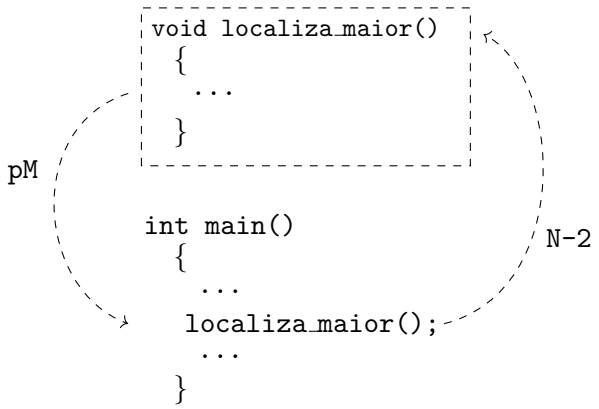
Mas, esse programa ainda não funciona.

Quer dizer, a função `localiza_Maior`() vai sempre retornar a posição do maior elemento da lista, e o programa principal vai sempre colocar esse elemento na posição $\texttt{N} - \texttt{i}$.

Mas não é isso o que a gente queria.

O problema é que, depois da primeira volta, a função `localiza_Maior`() não deve mais olhar para a última posição da lista.

Mas, então, é preciso dizer isso para ela



Na linguagem C isso é feito da seguinte maneira

```
int localiza_Maior (int m)
 {
     . . .
 }

int main()
 {
     . . .
   for (i=1; i<N; i++)
    {
      p = localiza_Maior( N-i-1 );
     . . .
 }
```

Quer dizer, agora a função é definida como

```
int  localiza_Maior (int m)
```

para indicar que

- ela vai receber um número inteiro
- ela vai retornar um número inteiro

No caso, o número `m` que ela recebe é a última posição da lista que ela deve examinar.

E no momento da chamada, nós escrevemos

```
p  =  localiza_Maior ( N-i-1 );
```

para fornecer essa informação para a função.

Abaixo nós temos o programa completo que implementa a ordenação por seleção

```
int localiza_Maior (int m)
 {
   int i, pM;

   pM = 0;
   for (i=1; i<=m; i++)
     if ( L[i] > L[pM] )  pM = i;

   return (pM);
 }

int main()
 {
   int i, p, aux;

   for (i=1; i<N; i++)
    {
      p = localiza_Maior();

      aux = L[N-i];  L[N-i] = L[p];  L[p] = aux;
 }  }
```

◇

c. **Revisitando a ordenação da bolha**

Agora que nós aprendemos a implementar o diálogo entre caixas, nós podemos melhorar um pouquinho o programa da ordenação da bolha.

Quer dizer, na versão que nós apresentamos acima, as varreduras sempre percorrem a lista inteira.

Mas nós já sabemos que isso não é necessário.

Quer dizer, após a primeira varredura, o menor elemento já se encontra na primeira posição.

Daí que, a segunda varredura não precisa ir até lá.

Mas para isso, é preciso dizer isso para ela.

A coisa fica assim

```
void varredura (int i)
 {
    int j, aux;

    for (j=N-1; j>i; j--)
     {
       if ( L[j] < L[j-1] )
        {
          aux = L[j];   L[j] = L[j-1];   L[j-1] = aux;
 }   }  }

int main()
 {
   int i;

    for (i=0; j<N; i++)  varredura(i);
 }
```

E aproveitando que nós estamos aqui, nós podemos colocar a operação de troca dos elementos dentro da sua própria caixinha também.

Para isso, é preciso dizer para a caixinha que elementos serão trocados

```
void troca (int i, int j)
 {
    int  aux;

    aux = L[j];   L[j] = L[i];   L[i] = aux;
 }

int varredura (int i)
 {
   int i;

   for (j=N-1; j>i; j--)
    {
       if ( L[j] < L[j-1] )  troca (j-1,j);
 }
```

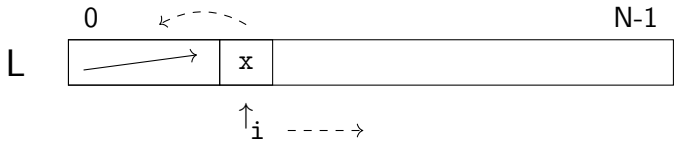
Note que, dessa maneira, a função varredura também já começa a parecer um programa escrito em português.

◇

d. **Ordenação por inserção** (dentro da caixa)

Agora só falta reorganizar o programa de ordenação por inserção.

Lembre que a ideia básica desse programa é inserir cada elemento na posição correta da porção inicial da lista que já se encontra ordenada



Essa operação pode ser implementada dentro de uma caixinha

```
void insercao (int i)
 {
    int  aux;

    aux = L[i];

    j = i;
    while ( j > 0  &&  L[j-1] > aux )
     {
        L[j] = L[j-1];   j--;
     }
    L[j] = aux;
 }
```

Quer dizer,

- nós guardamos o elemento a ser inserido na variável `aux`
- puxamos todo mundo que é maior do que ele para a direita
- e colocamos o elemento na posição que "ficou vazia"

◇

Agora, basta chamar essa função para todo elemento da segunda posição em diante

```
int main()
 {
    int  i;

    for (i=1; i<N; i++)
       insercao(i);
 }
```

◇

d. **Quem é o mais rápido?**

Legal.

Agora nós já reorganizamos todos os nossos programas de ordenação.

Mas, quem é o mais rápido?

Bom, basta executar os 3 programas e ver que leva menos tempo para realizar a tarefa.

E agora que os programas foram reorganizados, isso é bem fácil de fazer.

Quer dizer, nós podemos imaginar que temos as seguintes funções

```
void   ordenacao_selecao();

void   ordenacao_insercao();

void   ordenacao_bolha();
```

E a ideia é chamar cada uma delas a partir do programa principal.

Mas para ser justo, é preciso que as 3 funções trabalhem sobre a mesma lista.

Para fazer isso, nós vamos tirar uma cópia da lista original, e fazer as funções trabalharem sobre essa cópia.

A coisa fica assim

```
#define  N  1000

int  LO[N];            // lista original
int  L[N];             // lista que será ordenada

void inicializacao_aleatoria()
 {
    int i;

    for (i=0; i<N; i++)  LO[i] = rand();
 }

void copia_lista();
 {
    int i;

    for (i=0; i<N; i++)  L[i] = LO[i];
 }

int main()
 {
    inicializacao_aleatoria();
    copia_lista();   ordenacao_selecao();
    copia_lista();   ordenacao_insercao();
    copia_lista();   ordenacao_bolha();
 }
```

Certo.

Mas ainda falta medir o tempo.

Para isso, nós utilizamos a seguinte função da linguagem C

```
clock()
```

o que corresponde basicamente a olhar para o relógio.

Na prática, essa função retorna o número de ciclos do processador desde o início do programa.

Mas isso já é suficiente para comparar os tempos de execução das 3 funções.

A coisa fica assim

```
int main()
 {
    clock_t  t1,t2;

    inicializacao_aleatoria();

    copia_lista();
    t1 = clock();  ordena_selecao();  t2 = clock();
    printf(" ordenação por seleção: %7ld\n", t2 - t1);

    copia_lista();
    t1 = clock();  ordena_insercao();  t2 = clock();
    printf("ordenação por insercao: %7ld\n", t2 - t1);
```

```
    copia_lista();
    t1 = clock();  ordena_bolha();  t2 = clock();
    printf("    ordenação da bolha: %7ld\n", t2 - t1);
}
```

E o resultado que nós obtivemos foi o seguinte:

```
 ordenação por seleção:  643504
ordenação por insercao:  374627
    ordenação da bolha: 1337411
```

Porque?

◇