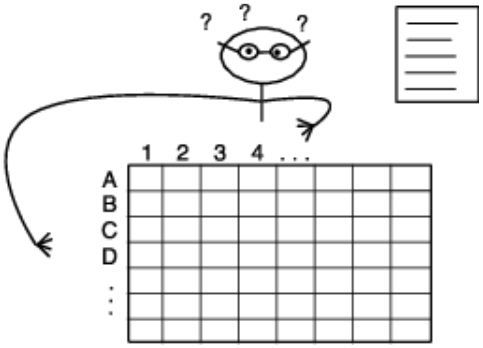
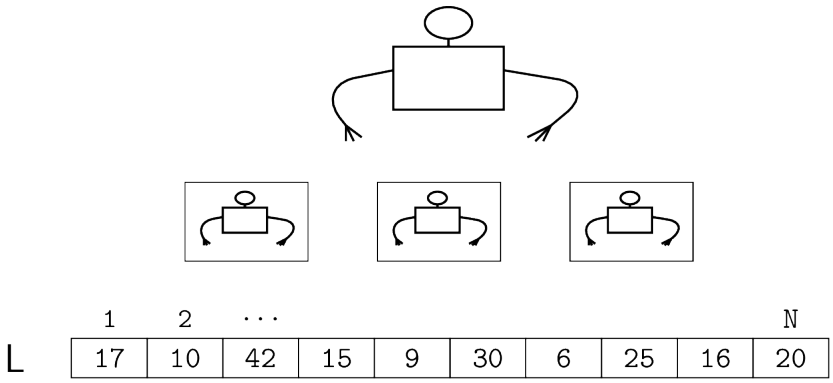
Programando com caixinhas
Professor Wladimir

Programando com caixinhas

Programar é como dar ordens para um bonequinho invisível, chamado processador, que consegue mexer nos dados guardados na memória do computador. A gente escreve uma lista de passos que ele deve seguir para fazer alguma coisa acontecer.



Para organizar melhor o processo, queremos que o bonequinho principal possa chamar outros bonequinhos, cada um responsável por uma tarefa específica. Dessa forma, conseguimos dividir o trabalho e resolver problemas mais complexos com mais facilidade.



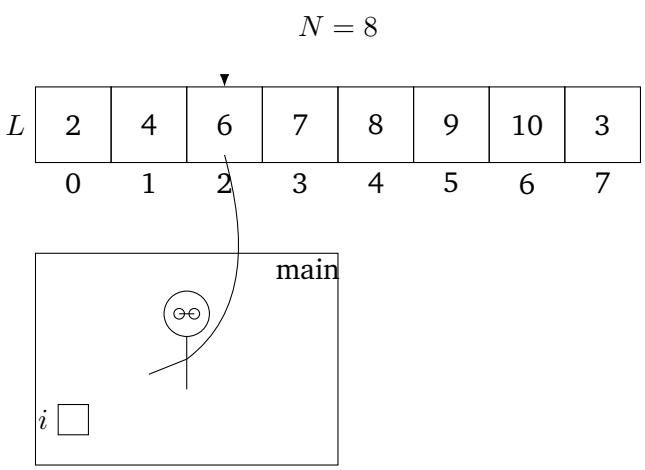
Então precisamos especificar o que cada bonequinho vai fazer e quais os dados eles podem mexer.

Considere o seguinte código:

```c
#include <stdio.h>
#define N 8
int L[N] = {2,4,6,7,8,9,10,3};

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

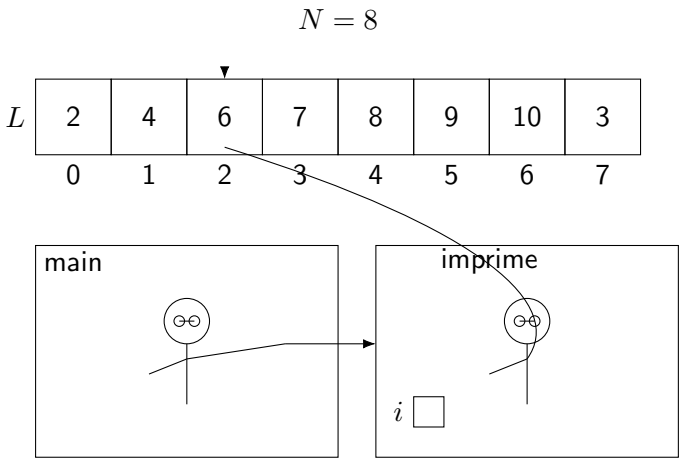
Podemos representar esse código da seguinte maneira:



Nesse exemplo, o bonequinho do main controla a variável i e consegue acessar a lista L e a variável N.

Agora, queremos fazer o seguinte:

O bonequinho main chama um outro bonequinho imprime que vai realizar a impressão da lista.



Isso pode ser feito da seguinte maneira:

```c
#include <stdio.h>
#define N 8
int L[N] = {2,4,6,7,8,9,10,3};

void imprime(){
    int i;
    for(i = 0; i < N; i++){
        printf("%d", L[i]);
    }
}

int main(){
    imprime();
}
```

Imprimindo duas listas

Imagine que agora precisamos imprimir duas listas diferentes, podemos fazer assim:

```c
#include <stdio.h>
#define N 8
int L1[N] = {2,4,6,7,8,9,10,3};
int L2[N] = {4,3,2,8,9,3,11,2};
void imprimeL1(){
    int i;
    for(i = 0; i < N; i++){
        printf("%d", L1[i]);
    }
}

void imprimeL2(){
    int i;
    for(i = 0; i < N; i++){
        printf("%d", L2[i]);
    }
}


int main(){
    imprimeL1();
    imprimeL2();
}
```

Note que ambas as funções executam os mesmos comandos, diferindo apenas quanto à lista à qual esses comandos se aplicam. Como poderíamos usar parâmetros para criar uma só função que sirva para ambas?

Neste caso, podemos fazer uma única função para imprimir as duas listas e indicando a lista que será impressa.

```c
#include <stdio.h>
#define N 8
int L1[N] = {2,4,6,7,8,9,10,3};
int L2[N] = {4,3,2,8,9,3,11,2};
void imprime(int L[]){
    int i;
    for(i = 0; i < N; i++){
        printf("%d", L[i]);
    }
}



int main(){
    imprime(L1);
    imprime(L2);
}
```

Imprimindo duas listas de tamanho diferente

Imagine que agora precisamos imprimir duas listas diferentes com tamanho diferente, podemos fazer assim:

```c
#include <stdio.h>
#define N 8
#define M 5
int L1[N] = {2,4,6,7,8,9,10,3};
int L2[M] = {4,3,2,8,9};
void imprimeL1(){
    int i;
    for(i = 0; i < N; i++){
        printf("%d", L1[i]);
    }
}

void imprime1L2(){
    int i;
    for(i = 0; i < M; i++){
        printf("%d", L2[i]);
    }
}


int main(){
    imprimeL1();
    imprimeL2();
}
```

Note que ambas as funções executam os mesmos comandos, diferindo apenas quanto à lista à qual esses comandos se aplicam e o tamanho da lista. Neste caso, podemos fazer uma única função para imprimir as duas listas e indicando a lista que será impressa e o seu tamanho.

```c
#include <stdio.h>
#define N 8
#define M 5
int L1[N] = {2,4,6,7,8,9,10,3};
int L2[M] = {4,3,2,8,9};

void imprime(int L[], int T){
    int i;
    for(i = 0; i < T; i++){
        printf("%d", L[i]);
    }
}

int main(){
    imprime(L1, N);
    imprime(L2, M);
}
```

Recebendo informações do bonequinho

Imagine que nós queremos que o bonequinho calcule a media de uma lista de tamanho N e devolva o valor da média. Podemos fazer assim:

```c
#include <stdio.h>
#define N 8
int L[N] = {2,4,6,7,8,9,10,3};

float media(int L[], int T){
    int i;
    float media = 0;
    for(i = 0; i < T; i++){
        media += L[i];
    }
    media = media/T;
    return media;
}

int main(){
    float m = media(L, N);

}
```

Note que agora o valor de retorno da função é float e usamos o comando return para devolver o valor calculado pela função.

Abaixo da média

Queremos calcular quantas valores da lista L estão abaixo da média, podemos fazer da seguinte maneira:

```c
#include <stdio.h>
#define N 8
int L[N] = {2,4,6,7,8,9,10,3};

float media(int L[], int T){
    int i;
    float media = 0;
    for(i = 0; i < T; i++){
        media += L[i];
    }
    media = media/T;
    return media;
}

int abaixo(int L[], int T, float k){
    int i, cont = 0;
    for(i = 0; i < T; i++){
        if( L[i] < k) cont++;
    }
    return cont;
}

int main(){
    float m = media(L, N);
    int c = abaixo(L, N, m);
}
```

Encontrando a posição do maior

Queremos fazer um bonequinho que encontra a posição do maior elemento do vetor, podemos fazer da seguinte maneira:

```
int pos_maior(){
    int i, M, posM;
    M = L[0];
    posM = 0;
    for(i = 1; i < N; i++){
        if(L[i] > M){
            M = L[i];
            posM = i;
        }
    }
    return posM;
}
```

Se quisermos trocar o maior elemento com o último podemos fazer da seguinte maneira:

```
int main(){
    int posM = pos_maior();
    int temp = L[posM];
    L[posM] = L[N-1];
    L[N-1] = temp;
}
```

Ordenando um vetor

Para ordenar um vetor precisamos modificar a função `pos_maior` para que ela não considere o vetor completo e podemos fazer da seguinte maneira:

```
int pos_maior(int T){
    int i, M, posM;
    M = L[0];
    posM = 0;
    for(i = 1; i < T; i++){
        if(L[i] > M){
            M = L[i];
            posM = i;
        }
    }
    return posM;
}
```

Agora, a função `pos_maior` encontrar a posição do maior entre as T primeiras posições. A função de ordenação pode ser feita da seguinte maneira:

```

void troca(int i, int j){
    int temp = L[i];
    L[i] = L[j];
    L[j] = temp;
}

int ordena(){
    int T;
    for(T  = N; T >= 1; T--){
        //Encontra a posição do maior entre os T primeiros
        int pos = pos_maior(T);
        //Troca a posicao do maior com a posição T-1
        troca(pos, T-1);
    }
}
```

Posição do menor em um intervalo do vetor

Imagine que queremos encontrar a posição do menor em um intervalo do vetor, podemos fazer da seguinte maneira:

```
int pos_menor_intervalo(int inicio, int fim){
    int i, M, posM;
    M = L[inicio];
    posM = inicio;
    for(i = inicio+1; i <= fim; i++){
        if(L[i] < M){
            M = L[i];
            posM = i;
        }
    }
    return posM;
}
```

Ordenando usando a posição do menor

```

void troca(int i, int j){
    int temp = L[i];
    L[i] = L[j];
    L[j] = temp;
}

int ordena(){
    int inicio;
    int fim = N-1;
    for(inicio  = 0; inicio < N; inicio++){
        //Encontra a posição do menor do intervalo começando do inicio até o fim
        int pos = pos_menor_intervalo(inicio, fim);
        //Troca a posicao do menor no intervalo com a posição inicio
        troca(inicio, pos);
    }
}
```

Varredura

Imagine que queremos fazer a operação de varredura que vai jogando os maiores para o fim do vetor, podemos fazer da seguinte maneira:

```

void troca(int i, int j){
    int temp = L[i];
    L[i] = L[j];
    L[j] = temp;
}

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

Ordenação por Varredura

```
int ordenacao_varredura(){
    int i;
    //Cada varredura garante que o elemento i será colocado na posição correta
    for(i = N-1; i >= 1; i--){
        varredura();
    }
}
```

Corrige

Imagine que queremos corrigir um vetor que tem apenas o último elemento fora de ordem podemos fazer da seguinte maneira:

```


int corrige(int T){
    int i;
    for(i = T-1; i >=1; i--){
        if(L[i-1] > L[i]) troca(i-1, i);
        else break;
    }
}
```

Ordenação por correção

```
int ordenacao_correcao(){
    int T;
    //Cada varredura garante que o elemento i será colocado na posição correta
    for(T = 2; T <= N; T++){
        corrige(T);
    }
}
```