

Tema: Introdução à programação

Atividade: Montagem de programas - Karel

- 01.) Editar e salvar um esboço de programa,  
o nome do arquivo deverá ser Guia0201.cpp,  
concordando maiúsculas e minúsculas, sem espaços em branco, acentos ou cedilha:

```
/*
  Guia_0201 - v0.0. - __ / __ / ____
  Author: _____

  Para compilar em uma janela de comandos (terminal):

  No Linux   : g++ -o Guia0201   ./Guia0201.cpp
  No Windows: g++ -o Guia0201.exe Guia0201.cpp

  Para executar em uma janela de comandos (terminal):

  No Linux   : ./Guia0201
  No Windows: Guia0201
*/
// lista de dependencias
#include "karel.hpp"

// ----- definicoes de metodos

/**
  decorateWorld - Metodo para preparar o cenario.
  @param fileName - nome do arquivo para guardar a descricao.
*/
void decorateWorld ( const char* fileName )
{
  // colocar paredes no mundo
  world->set ( 4, 4, VWALL );
  world->set ( 4, 4, HWALL );

  // colocar um marcador no mundo
  world->set ( 4, 4, BEEPER );

  // salvar a configuracao atual do mundo
  world->save( fileName );
} // decorateWorld ( )
```

```

/**
Classe para definir robo particular (MyRobot),
a partir do modelo generico (Robot)

Nota: Todas as definicoes irao valer para qualquer outro robo
criado a partir dessa nova descricao de modelo.
*/
class MyRobot : public Robot
{
public:

    /**
    turnRight - Procedimento para virar 'a direita.
    */
    void turnRight ( )
    {
        // definir dado local
        int step = 0;

        // testar se o robo esta' ativo
        if ( checkStatus ( ) )
        {
            // o agente que executar esse metodo
            // devera' virar tres vezes 'a esquerda
            for ( step = 1; step <= 3; step = step + 1 )
            {
                turnLeft( );
            } // end for
        } // end if
    } // end turnRight ( )

    /**
    moveN - Metodo para mover certa quantidade de passos.
    @param steps - passos a serem dados.
    */
    void moveN( int steps )
    {
        // definir dado local
        int step = 0;
        // testar se a quantidade de passos e' maior que zero
        for ( step = steps; step > 0; step = step - 1 )
        {
            // dar um passo
            move( );
        } // end if
    } // end moveN( )

    /**
    doPartialTask - Metodo para especificar parte de uma tarefa.
    */
    void doPartialTask( )
    {
        // especificar acoes dessa parte da tarefa
        moveN( 3 );
        turnLeft( );
    } // end doPartialTask( )
}

```

```
/**
    doTask - Relacao de acoes para o robo executar.
 */
void doTask( )
{
    // definir dado local
    int step = 4;

    // especificar acoes da tarefa
    while ( step > 0 )
    {
        // realizar uma parte da tarefa
        doPartialTask( );
        // tentar passar 'a proxima
        step = step - 1;
    } // end while

    // encerrar
    turnOff ( );
} // end doTask( )

}; // end class MyRobot
```

```

// ----- acao principal

/**
  Acao principal: executar a tarefa descrita acima.
*/

int main ( )
{
  // definir o contexto

  // criar o ambiente e decorar com objetos
  // OBS.: executar pelo menos uma vez,
  //   antes de qualquer outra coisa
  //   (depois de criado, podera' ser comentado)
  world->create ( "" );           // criar o mundo
  decorateWorld ( "Guia0201.txt" );
  world->show ( );

  // preparar o ambiente para uso
  world->reset ( );               // limpar configuracoes
  world->read ( "Guia0201.txt" ); // ler configuracao atual para o ambiente
  world->show ( );               // mostrar a configuracao atual

  set_Speed ( 3 );               // definir velocidade padrao

  // criar robo
  MyRobot *robot = new MyRobot( );

  // posicionar robo no ambiente (situacao inicial):
  // posicao(x=1,y=1), voltado para direita, com zero marcadores, nome escolhido )
  robot->create ( 1, 1, EAST, 0, "Karel" );

  // executar tarefa
  robot->doTask ( );

  // encerrar operacoes no ambiente
  world->close ( );

  // encerrar programa
  getchar ( );
  return ( 0 );

} // end main ( )

// ----- testes

/**
----- documentacao complementar

----- notas / observacoes / comentarios

----- previsao de testes

```

----- historico

Versao	Data	Modificacao
0.1	__/__/__	esboco

----- testes

Versao	Teste	
0.1	01. (   )	identificacao de programa

\*/

\*/

- 02.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 03.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.
- 04.) Copiar a versão atual do programa para outra (nova) – Guia0202.cpp.
- 05.) Alterar as identificações e acrescentar na nova versão, na classe, antes da ação principal:

```
/**
doSquare - Metodo para especificar outro percurso.
*/
void doSquare( )
{
// definir dado local
int step = 4;

// especificar acoes da tarefa
while ( step > 0 )
{
// realizar uma parte da tarefa
moveN(3);
turnRight( );
// tentar passar 'a proxima
step = step - 1;
} // end while
turnOff ( );
} // end doSquare( )
```

- 05.) Editar mudanças no nome do programa e versão, tomando o cuidado de modificar todas as referências, inclusive as presentes em comentários. Incluir na parte principal uma chamada para testar o método novo.

```
...
// criar robo
MyRobot *robot = new MyRobot( );

// posicionar robo no ambiente (situacao inicial):
// posicao(x=1,y=1), voltado para direita, com zero marcadores, nome escolhido )
robot->create ( 1, 1, NORTH, 0, "Karel" );

// executar tarefa
robot->doSquare ( );
...
```

Incluir na documentação complementar as alterações feitas, acrescentar indicações de mudança de versão e prever novos testes.

```
// ----- testes

/*
Versao    Teste
0.1      01. ( )   - teste inicial
*/
```

- 06.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.

- 07.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.

```
// ----- testes
//
// Versao    Teste
// 0.1      01. ( OK )      teste inicial
// 0.2      01. ( OK )      teste da repeticao para virar 'a direita
//
```

- 08.) Copiar a versão atual do programa para outra (nova) – Guia0203.cpp.

- 09.) Realizar as mudanças de versão e acrescentar ao programa as modificações indicadas abaixo:

```
/**
    doSquare - Metodo para especificar outro percurso.
*/
void doSquare( )
{
    // definir dado local
    int step = 1;

    // especificar acoes da tarefa
    while ( step <= 4 )
    {
        // realizar uma parte da tarefa
        moveN(3);
        turnRight( );
        // tentar passar 'a proxima
        step = step + 1;
    } // end while
    turnOff ( );
} // end doSquare( )
```

- 10.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 11.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.

```
// ----- testes
//
// Versao    Teste
// 0.1       01. ( OK )       teste inicial
// 0.2       01. ( OK )       teste da repeticao para virar 'a direita
// 0.3       01. ( OK )       teste da repeticao para percorrer um quadrado
//
```

- 12.) Copiar a versão atual do programa para outra (nova) – Guia0204.cpp.

- 13.) Realizar as mudanças de versão e acrescentar ao programa as modificações indicadas abaixo:

```
/**
 * pickBeepers - Metodo para coletar marcadores.
 */
void pickBeepers ( )
{
    // repetir (com teste no inicio)
    // enquanto houver marcador proximo
    while ( nextToABeeper ( ) )
    {
        // coletar um marcador
        pickBeeper ( );
    } // end while
} // end pickBeepers ( )

/**
 * doSquare - Metodo para especificar outro percurso.
 */
void doSquare ( )
{
    // definir dado local
    int step = 1;
    // especificar acoes da tarefa
    while ( step <= 4 )
    {
        // realizar uma parte da tarefa
        moveN( 3 );
        pickBeepers ( );
        turnRight( );
        // tentar passar 'a proxima
        step = step + 1;
    } // end while
    turnOff ( );
} // end doSquare ( )
```

- 14.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 15.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.
- 16.) Copiar a versão atual do programa para outra (nova) – Guia0205.cpp.



- 17.) Realizar as mudanças de versão e acrescentar ao programa as modificações indicadas abaixo:

```
/**
 * pickBeepers - Funcao para coletar marcadores.
 * @return quantidade de marcadores coletados
 */
int pickBeepers ( )
{
    // definir dado local
    int n = 0;
    // repetir (com teste no inicio)
    // enquanto houver marcador proximo
    while ( nextToABeeper ( ) )
    {
        // coletar um marcador
        pickBeeper ( );
        // contar mais um marcador coletado
        n = n + 1;
    } // end while
    // retornar a quantidade de marcadores coletados
    return ( n );
} // end pickBeepers( )

/**
 * doSquare - Metodo para especificar outro percurso.
 */
void doSquare( )
{
    // definir dado local
    int step = 1;
    int n = 0;
    char msg [80];
    // especificar acoes da tarefa
    while ( step <= 4 )
    {
        // realizar uma parte da tarefa
        moveN( 3 );
        n = pickBeepers ( );
        // testar se quantidade maior que zero
        if ( n > 0 )
        {
            // montar a mensagem para a saida
            sprintf ( msg, "Recolhidos = %d", n );
            // solicitar a exibicao da mensagem
            show_Text ( msg );
        } // end if
        turnRight( );
        // tentar passar 'a proxima
        step = step + 1;
    } // end while
    turnOff ( );
} // end doSquare( )
```

OBS.: A mensagem montada será exibida na próxima alteração do ambiente, se não houver erro.

- 18.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 19.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.
- 20.) Copiar a versão atual do programa para outra (nova) – Guia0206.cpp.
- 21.) Realizar as mudanças de versão e  
acrescentar ao programa as modificações indicadas abaixo.

No início, incluir a dependência de mais uma biblioteca (para tratar entradas e saídas):

```
#include "karel.hpp"  
#include "io.hpp" // para entradas e saídas
```

Antes da parte principal, incluir os métodos:

```
/**  
 * execute - Metodo para executar um comando.  
 * @param action - comando a ser executado  
 */  
void execute( int option )  
{  
    // executar a opcao de comando  
    switch ( option )  
    {  
        case 0: // terminar  
            // nao fazer nada  
            break;  
        case 1: // virar para a esquerda  
            if ( leftIsClear ( ) )  
            {  
                turnLeft( );  
            } // end if  
            break;  
        case 2: // virar para o sul  
            while ( ! facingSouth( ) )  
            {  
                turnLeft( );  
            } // end while  
            break;  
        case 3: // virar para a direita  
            if ( rightIsClear ( ) )  
            {  
                turnRight( );  
            } // end if  
            break;  
    }
```

```

case 4: // virar para o oeste
    while ( ! facingWest( ) )
    {
        turnLeft( );
    } // end while
    break;
case 5: // mover
    if ( frontIsClear ( ) )
    {
        move( );
    } // end if
    break;
case 6: // virar para o leste
    while ( ! facingEast( ) )
    {
        turnLeft( );
    } // end while
    break;
case 7: // pegar marcador
    if ( nextToABeeper( ) )
    {
        pickBeeper( );
    } // end if
    break;
case 8: // virar para o norte
    while ( ! facingNorth( ) )
    {
        turnLeft( );
    } // end while
    break;
case 9: // colocar marcador
    if ( beepersInBag( ) )
    {
        putBeeper( );
    } // end if
    break;
default:// nenhuma das alternativas anteriores
    // comando invalido
    show_Error ( "ERROR: Invalid command." );
} // end switch
} // end execute( )

```

```

/**
 * movel - Metodo para mover o robot interativamente.
 * Lista de comandos disponiveis:
 * 0 - turnOff
 * 1 - turnLeft           2 - to South
 * 3 - turnRight          4 - to West
 * 5 - move               6 - to East
 * 7 - pickBeeper         8 - to North
 * 9 - putBeeper
 */
void movel( )
{
    // definir dados
    int action;

    // repetir (com testes no fim)
    // enquanto opcao diferente de zero
    do
    {
        // ler opcao
        action = IO_readint ( "Command? " );

        // executar acao dependente da opcao
        execute ( action );
    }
    while ( action != 0 );
} // end movel( )

```

Na parte principal, incluir a chamada ao método interativo:

```

// executar tarefa
robot->movel ( );

```

- 22.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 23.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 24.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.
- 25.) Copiar a versão atual do programa para outra (nova) – Guia0207.cpp.

- 26.) Realizar as mudanças de versão e acrescentar ao programa as modificações indicadas abaixo:

```
/**
    recordActions - Metodo para mover o robot interativamente
    e guardar a descricao da tarefa em arquivo.
    @param fileName - nome do arquivo
*/
void recordActions ( const char *fileName )
{
    // definir dados
    int action;
    // definir arquivo onde gravar comandos
    std::ofstream archive ( fileName );

    // repetir enquanto o comando
    // for diferente de zero
    do
    {
        // ler opcao
        action = IO_readint ( "Command? " );
        // testar se opcao valida
        if ( 0 <= action && action <= 9 )
        {
            // executar comando
            execute ( action );
            // guardar o comando em arquivo
            archive << action << "\n";
        } // end if
    }
    while ( action != 0 );
    // fechar o arquivo
    // INDISPENSÁVEL para a gravacao
    archive.close ( );
} // end recordActions ( )
```

Na parte principal, incluir uma chamada ao método para testá-lo e guardar a definição da tarefa em arquivo do tipo texto.

```
// executar e gravar acoes
robot->recordActions ( "Tarefa0207.txt" );
```

- 27.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 28.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.
- 29.) Copiar a versão atual do programa para outra (nova) – Guia0208.cpp.

- 30.) Realizar as mudanças de versão e acrescentar ao programa as modificações indicadas abaixo.

```
/**
    playActions - metodo para receber comandos de arquivo.
    @param fileName - nome do arquivo
*/
void playActions ( const char *fileName )
{
    // definir dados
    int action;
    // definir arquivos de onde ler dados
    std::ifstream archive ( fileName );

    // repetir enquanto houver dados
    archive >> action;          // tentar ler a primeira linha
    while ( ! archive.eof ( ) ) // testar se nao encontrado o fim
    {
        // mostrar mais um comando
        IO_print( IO_toString ( action ) );
        delay ( stepDelay );
        // executar mais um comando
        execute ( action ) ;
        // tentar ler a proxima linha
        archive >> action ;    // tentar ler a próxima linha
    } // end for
    // fechar o arquivo
    // RECOMENDAVEL para a leitura
    archive.close ( );
} // end playActions ( )
```

Na parte principal, incluir uma chamada ao método para testá-lo e guardar a definição da tarefa em arquivo do tipo texto.

```
// executar tarefas
robot-> recordActions ( "Tarefa0208.txt" );
robot-> playActions   ( "Tarefa0208.txt" );
```

- 31.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 32.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.
- 33.) Copiar a versão atual do programa para outra (nova) – Guia0209.cpp.

- 34.) Realizar as mudanças de versão e acrescentar ao programa as modificações indicadas abaixo:

```
/**
 * dictionary - Metodo para traduzir um comando.
 * @param action - comando a ser traduzido
 */
chars dictionary( int action )
{
    // definir dado
    static char word [80];
    strcpy ( word, "" ); // palavra vazia
    // identificar comando
    switch ( action )
    {
        case 1: // virar para a esquerda
            strcpy ( word, "turnLeft( ); " );
            break;
        case 2: // virar para o sul
            strcpy ( word, "faceSouth( ); " );
            break;
        case 3: // virar para a direita
            strcpy ( word, "turnRight( ); " );
            break;
        case 4: // virar para o oeste
            strcpy ( word, "faceWest( ); " );
            break;
        case 5: // mover
            strcpy ( word, "move( ); " );
            break;
        case 6: // virar para o leste
            strcpy ( word, "faceEast( ); " );
            break;
        case 7: // pegar marcador
            strcpy ( word, "pickBeeper( );" );
            break;
        case 8: // virar para o norte
            strcpy ( word, "faceNorth( ); " );
            break;
        case 9: // colocar marcador
            strcpy ( word, "putBeeper( ); " );
            break;
    } // end switch
    // retornar palavra equivalente
    return ( &(word[0]) );
} // end dictionary( )
```

OBS.: A especificação **static** estabelecerá um local para armazenamento que permanecerá disponível para acesso após o encerramento da função. O comportamento padrão é reciclar todos os recursos utilizados.

Acrescentar também ao programa o método indicado abaixo:

```
/**
  translateActions - Metodo para receber comandos de arquivo e traduzi-los.
  @param fileName - nome do arquivo
*/
void translateActions ( const char *fileName )
{
  // definir dados
  int action;
  // definir arquivo de onde ler dados
  std::ifstream archive ( fileName );

  // repetir enquanto houver dados
  archive >> action;      // tentar ler a primeira linha
  while ( ! archive.eof( ) )  // testar se nao encontrado o fim
  {
    // tentar traduzir um comando
    IO_print ( dictionary ( action ) );
    getchar ( );
    // guardar mais um comando
    execute ( action );
    // tentar ler a proxima linha
    archive >> action ;    // tentar ler a proxima linha
  } // end for
  // fechar o arquivo
  // RECOMENDAVEL para a leitura
  archive.close ( );
} // end translateActions ( )
```

Na parte principal, incluir uma chamada ao método para testá-lo e guardar a definição da tarefa em arquivo do tipo texto.

```
// executar tarefas
robot->recordActions ( "Tarefa0209.txt" );
robot->translateActions ( "Tarefa0209.txt" );
```

- 35.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 36.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.
- 37.) Copiar a versão atual do programa para outra (nova) – Guia0210.cpp.



38.) Realizar as mudanças de versão e acrescentar ao programa as modificações indicadas abaixo:

```
/**
 * recordActions - Metodo para mover o robot interativamente
 * e guardar a descricao da tarefa em arquivo.
 * @param fileName - nome do arquivo
 */
void recordActions ( const char * fileName )
{
    // definir dados
    int action;
    // definir arquivo onde gravar comandos
    std::ofstream archive ( fileName );

    // ler acao
    action = IO_readint ( "Command? " );
    // repetir enquanto acao maior ou igual a zero
    while ( action >= 0 )
    {
        // testar se opcao valida
        if ( 0 <= action && action <= 9 )
        {
            // executar comando
            execute ( action );
            // guardar o comando em arquivo
            archive << action << "\n";
        } // end if
        // ler acao
        action = IO_readint ( "Command? " );
    } // end while
    // fechar o arquivo
    // INDISPENSÁVEL para a gravacao
    archive.close ( );
} // end recordActions ( )
```

```

/**
    appendActions - Metodo para acrescentar comandos ao arquivo e traduzi-los.
    @param filename - nome do arquivo
*/
void appendActions ( const char *fileName )
{
    // definir dados
    int action;
    // definir arquivo para receber acrescimos ao final
    std::fstream archive ( fileName, std::ios::app );

    // repetir enquanto acao diferente de zero
    do
    {
        // ler acao
        action = IO_readint ( "Command? " );
        // testar se opcao valida
        if ( 0 <= action && action <= 9 )
        {
            // executar comando
            execute ( action );
            // guardar o comando em arquivo
            archive << action << std::endl;
        } // end if
    }
    while ( action != 0 );
    // fechar o arquivo
    // INDISPENSÁVEL para a gravacao
    archive.close ( );
} // end appendActions ( )

```

Na parte principal, acrescentar chamadas para testar os métodos.

```

// executar tarefa
robot->recordActions ( "Tarefa0210.txt" );

// dar uma pausa na entrada de comandos
show_Text ( "Pause on recording" );

// mostrar configuracao atual do mundo
world->show ( );

// retomar a entrada de comandos
robot->appendActions ( "Tarefa0210.txt" );

// reproduzir todos os comandos
robot->playActions ( "Tarefa0210.txt" );

```

- 39.) Compilar o programa novamente.  
Se houver erros, resolvê-los e compilar novamente, até que todos tenham sido resolvidos.  
Se não houver erros, seguir para o próximo passo.
- 40.) Executar o programa.  
Observar as saídas.  
Registrar os resultados com os valores usados para testes.

Exercícios:

DICAS GERAIS: Consultar o Anexo CPP para mais informações e outros exemplos.

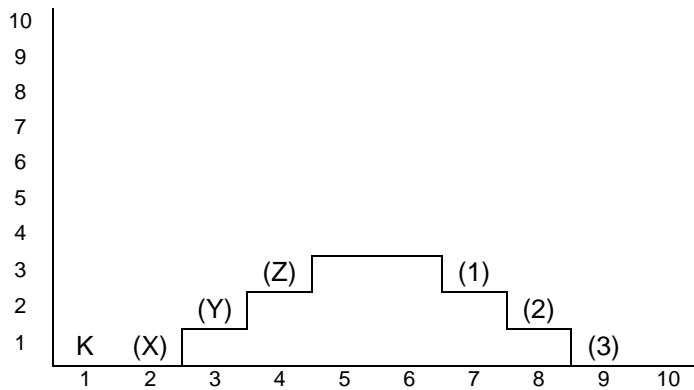
Prever, realizar e registrar todos os testes efetuados.

Fazer um programa para atender a cada uma das situações abaixo envolvendo definições e ações básicas.

Os programas deverão ser desenvolvidos em C++ com as bibliotecas indicadas.

01.) Definir um conjunto de ações em um programa Guia0211 para:

- configurar o mundo semelhante ao diagrama abaixo:



- definir uma "escada" com seis marcadores, em cada degrau do lado oposto, conforme a figura acima;
- tarefa:  
o robô deverá começar o trajeto ao pé da "escada", buscar os marcadores, e deixá-los do outro lado nas posições correspondentes (X,Y,Z); e voltar à posição inicial;
- métodos deverão ser criados e usados para deslocar um robô na "escada":

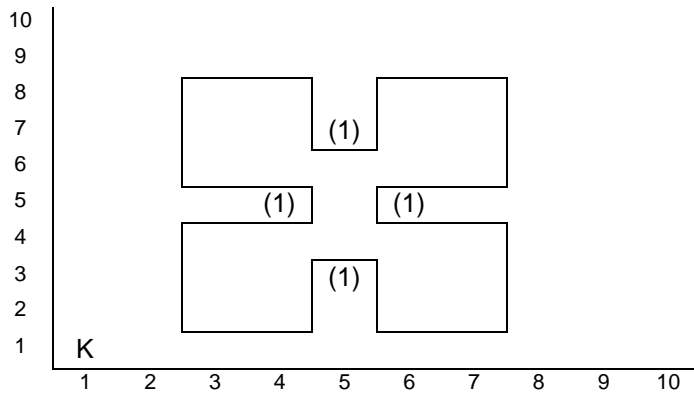
stepUpRight( ) - um degrau para cima e à direita  
stepDownRight( ) - um degrau para baixo e à direita  
stepUpLeft( ) - um degrau para cima e à esquerda  
stepDownLeft( ) - um degrau para baixo e à esquerda.

Exemplo para se descrever um método (semelhante ao modelo):

```
/*  
  Descricao:  
*/  
void stepUpRight( )  
{  
  // acoes para subir um degrau  
} // fim stepUpRight( )
```

02.) Definir um conjunto de ações em um programa Guia0212 para:

- configurar o mundo semelhante ao descrito abaixo:



- definir uma estrutura com quatro marcadores, um em cada "nicho";
- tarefa:  
o robô deverá começar o trajeto abaixo da estrutura, buscar os marcadores, no sentido horário, e trazê-los à posição inicial;
- métodos deverão ser criados e usar as seguintes condições nativas em testes ou repetições:

rightIsClear( )     - se caminho livre à direita  
leftIsClear( )       - se caminho livre à esquerda  
nextToABeeper( )   - se próximo a um marcador.

Exemplos de como usar uma condição nativa:

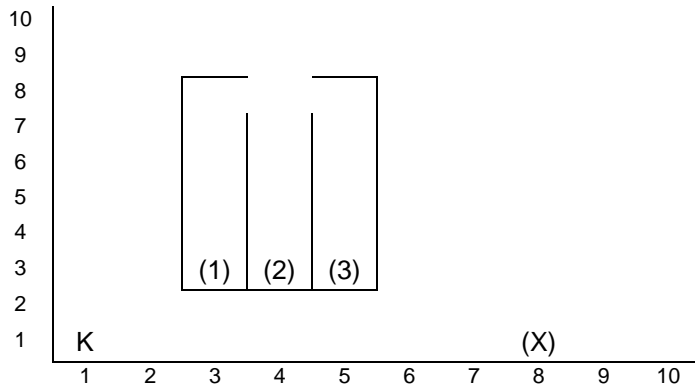
```
// testar se está próximo a um marcador,  
// antes de tentar pegá-lo  
if ( nextToABeeper( ) )  
{  
    pickBeeper( );  
} // fim se
```

```
// testar se poderá virar e mover-se  
// para a esquerda  
if ( leftIsClear( ) )  
{  
    turnLeft( );  
    move( );  
} // fim se
```

DICA: Verificar a possibilidade de pegar mais de um marcador por vez.

03.) Definir um conjunto de ações em um programa Guia0213 para:

- definir um robô na posição (1,1), voltado para leste, sem marcadores
- dispor blocos em uma configuração semelhante a mostrada abaixo
- buscar os três marcadores nas posições indicadas



- descarregar todos os marcadores obtidos na posição (9,1) mediante um novo procedimento (a ser criado)

putBeepers( )

que poderá usar em testes as condições nativas

beepersInBag( ) - está portando marcadores ? (rever o método execute( ) )

areYouHere(x, y) - está na posição (x,y) ?

Exemplos de como usar uma condição nativa:

```
// testar se esta' na posicao desejada ...
```

```
if ( areYouHere(1,1) )
```

```
{
```

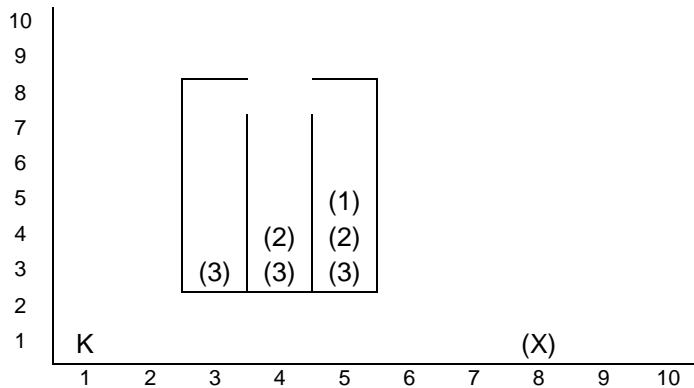
```
    // ... antes de fazer algo
```

```
} // fim se
```

- retornar à posição inicial, voltar-se para leste e desligar-se.

04.) Definir um conjunto de ações em um programa Guia0214 para:

- definir um robô na posição (1,1), voltado para leste, sem marcadores;
- dispor blocos em uma configuração semelhante a dada abaixo:



- buscar os marcadores nas posições indicadas, na ordem crescente das quantidades;
- descarregar os marcadores na posição indicada (X);
- retornar à posição inicial, voltar-se para leste e desligar-se;
- todas as posições visitadas pelo robô que tiverem marcadores deverão ser guardadas em arquivo, cujo nome deverá ser Tarefa0214b.txt.

DICA:

As posições poderão ser guardadas quando o robô "pegar o marcador" (no procedimento pickBeepers( )).

Para se obter as coordenadas do robô, definir nesse procedimento armazenadores inteiros para receber as coordenadas:

`int x, y;`

`x = xAvenue( ); // obter posicao atual (avenue)`

`y = yStreet( ); // obter posicao atual ( street )`

Para guardar as coordenadas, primeiro, deverá definir e abrir um arquivo para receber acréscimos (Guia0210).

Ao recolher um marcador, gravar as coordenadas, acrescentando no arquivo, um valor por linha de cada vez:

`arquivo << x << std::endl;`

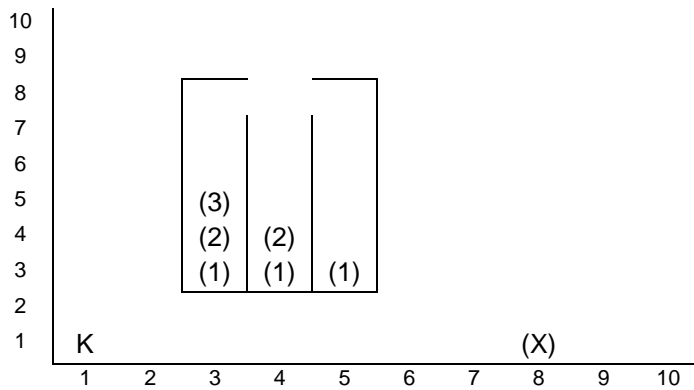
`arquivo << y << std::endl;`

Para garantir a gravação, deverá fechar o arquivo:

`arquivo.close ( );`

05.) Definir um conjunto de ações em um programa Guia0215 para:

- reutilizar parte da configuração do problema anterior;
- o robô deverá partir da posição inicial (coluna=1, linha=1), voltado para leste e com nenhum marcador;
- buscar os marcadores nas quantidades indicadas e na ordem crescente das quantidades, e descarregar os marcadores na posição indicada (X), seguindo os comandos guardados em arquivo;
- retornar à posição inicial, voltar-se para o leste e, antes de desligar-se, reposicionar os marcadores na configuração abaixo:



- todas os códigos das ações necessárias para a execução deverão ser primeiro guardados em arquivo (por treinamento ou por edição direta), cujo nome deverá ser Tarefa0215.txt, e depois aplicados mediante leitura.

## Tarefa extra

E1.) Definir um conjunto de ações em um programa Guia02E1 para:  
contar e informar o número de comandos (linhas)  
em um arquivo contendo a descrição de uma tarefa.

DICA: Definir um contador  
e contar mais uma linha lida,  
ao tentar e conseguir ler uma linha (de cada vez).

E2.) Definir um conjunto de ações em um programa Guia02E2 para:  
ler o número de comandos (linhas)  
em um arquivo que descreva uma tarefa e, em seguida,  
tentar ler e executar cada um desses comandos.

DICA: Copiar um arquivo contendo a descrição de uma tarefa,  
editar esse arquivo para conter, na primeira linha,  
a quantidade de comandos nele existentes.

Exemplo:

Arquivo original (TAREFA000.TXT)

5  
5  
5

Arquivo novo I (TAREFA001.TXT)

**3**  
5  
5  
5



## Atividade suplementar

Associar os conceitos de representações de dados e a metodologia sugerida para o desenvolvimento de programa (passo a passo), para modificar o modelo proposto (e exemplos associados) e introduzir, pouco a pouco, as modificações necessárias, cuidando de realizar a documentação das definições, procedimentos e operações executadas.

## Para pensar a respeito

Qual a estratégia de solução?

Como definir uma classe com um método principal que execute essa estratégia?

Serão necessárias definições prévias (extras) para se obter o resultado?

Como dividir os passos a serem feitos e organizá-los em que ordem?

Que informações deverão ser colocadas na documentação?

Como lidar com os erros de compilação?

Como lidar com os erros de execução?

## Fontes de informação

apostila de C++ (anexos)

exemplos (0-9) na pasta de arquivos relacionada

bibliografia recomendada

lista de discussão da disciplina

websites

## Processo

1 relacionar claramente seus objetivos e registrar isso na documentação necessária para o desenvolvimento;

2 organizar as informações de cada proposição de problema:

2.1 escolher os armazenadores de acordo com o tipo apropriado;

2.2 realizar as entradas de dados ou definições iniciais;

2.3 realizar as operações;

2.4 realizar as saídas dos resultados;

2.5 projetar testes para cada operação, considerar casos especiais

3 especificar a classe:

- 3.1 definir a identificação do programa na documentação;
- 3.2 definir a identificação do programador na documentação;
- 3.3 definir armazenadores necessários (se houver)
- 3.4 definir a entrada de dados para cada valor
- 3.5 testar se os dados foram armazenados corretamente
- 3.6 definir a saída de cada resultado ou (execução de cada ação)
- 3.7 testar a saída de cada resultado com valores (situações) conhecidas
- 3.8 definir cada operação
- 3.9 testar isoladamente cada operação, conferindo os resultados

4 especificar as ações da parte principal:

- 4.1 definir o cabeçalho para identificação;
- 4.2 definir as constantes, armazenadores e dados auxiliares (se houver);
- 4.3 definir a estrutura básica de programa que possa permitir a execução de vários dos testes programados;

5. realizar os testes isolados de cada operação e depois os testes de integração;

5.1 registrar todos os testes realizados.

## Dicas

- Digitar os exemplos fornecidos e testá-los.
- Identificar exemplos que possam servir de modelos para os exercícios, e usá-los como sugestões para o desenvolvimento.
- Fazer rascunhos, diagramas e esquemas para orientar o desenvolvimento da solução, previamente, antes de começar a digitar o novo programa.
- Consultar os modelos de programas e documentação disponíveis.
- Anotar os testes realizados e seus resultados no final do texto do programa, como comentários.
- Anotar erros, dúvidas e observações no final do programa, também como comentários. Usar /\* ... \*/ para isso.

## Conclusão

Analisar cada resultado obtido e avaliar-se ao fim do processo.