

Tema: Introdução à programação

Atividade: Montagem de programas - Karel

- 01.) Editar e salvar um esboço de programa,
o nome do arquivo deverá ser Guia0301.cpp,
concordando maiúsculas e minúsculas, sem espaços em branco, acentos e cedilha.
Copiar o último arquivo com tarefas gerado no guia anterior, e
renomeá-lo para Tarefa0301.txt para ser usado em testes.

```
/*
  Guia0301 - v0.0. - __ / __ / ____
  Author: _____

  Para compilar em uma janela de comandos (terminal):

  No Linux   : g++ -o Guia0301  ./Guia0301.cpp
  No Windows: g++ -o Guia0301.exe Guia0301.cpp

  Para executar em uma janela de comandos (terminal):

  No Linux   : ./Guia0301.cpp
  No Windows: Guia0301

*/
// lista de dependencias
#include "karel.hpp"
#include "io.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 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( )
}

```

```

/**
    countCommands - Metodo para contar comandos de arquivo.
    @param fileName - nome do arquivo
*/
void countCommands( const char * fileName )
{
    // definir dados
    int x = 0;
    int length = 0;
    std::ifstream archive ( fileName );

    // repetir enquanto houver dados
    archive >> x;
    while ( ! archive.eof ( ) )
    {
        // contar mais um comando
        length = length + 1;
        // tentar ler a proxima linha
        archive >> x;
    } // end while
    // fechar o arquivo
    archive.close( );
    // informar a quantidade de comandos guardados
    sprintf ( msg_txt, "Commands = %d", length );
    has_Text = true;
    show_Text ( msg_txt );
} // end countCommands( )

}; // 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 ( "Guia0301.txt" );
  world->show ( );

  // preparar o ambiente para uso
  world->reset ( );               // limpar configuracoes
  world->read ( "Guia0301.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, NORTH, 0, "Karel" );

  // executar tarefa
  robot->countCommands ( "Tarefa0301.txt" );

  // 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. (OK)	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) – Guia0302.cpp.
- 05.) Realizar as mudanças de versão e
acrescentar ao programa as modificações indicadas abaixo:

```
/**
    countCommands - Funcao para contar comandos de arquivo.
    @return quantidade de comandos
    @param fileName - nome do arquivo
*/
int countCommands( const char * fileName )
{
    // definir dados
    int x = 0;
    int length = 0;
    std::ifstream archive ( fileName );

    // repetir enquanto houver dados
    archive >> x;
    while ( ! archive.eof() )
    {
        // contar mais um comando
        length = length + 1;
        // tentar ler a proxima linha
        archive >> x;
    } // end while
    // fechar o arquivo
    archive.close();
    // retornar resultado
    return ( length );
} // end countCommands( )
```

Na parte principal, alterar a chamada para testar a função.

```
// executar tarefa
sprintf ( msg_txt, "Commands = %d",
        robot->countCommands ( "Tarefa0301.txt" ) );
show_Text ( msg_txt );
```

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.

08.) Copiar a versão atual do programa para outra (nova) – Guia0303.cpp.

09.) Realizar as mudanças de versão e

acrescentar ao programa as modificações indicadas abaixo:

```
/**
    readCommands - Metodo para receber comandos de arquivo.
    @return grupo formado por todos os comandos
    @param filename - nome do arquivo
*/
int readCommands ( int commands [ ], const char * fileName )
{
    // definir dados
    int    x      = 0;
    int    action = 0;
    int    length = 0;
    std::ifstream archive ( fileName );

    // obter a quantidade de comandos
    length = countCommands ( fileName );

    // criar um armazenador para os comandos
    if ( length < MAX_COMMANDS )
    {
        // repetir para a quantidade de comandos
        for ( x=0; x<length; x=x+1 )
        {
            // tentar ler a proxima linha
            archive >> action;
            // guardar um comando
            // na posicao (x) do armazenador
            commands [ x ] = action;
        } // end for
        // fechar o arquivo
        // INDISPENSÁVEL para a gravacao
        archive.close( );
    } // end for
    // retornar quantidade de comandos lidos
    return ( length );
} // end readCommands( )
```

OBS.: A constante `COMMANDS` serve para indicar a quantidade máxima de comandos que poderão ser executados. Sua definição deverá ser externa e global.

Na parte principal, alterar a chamada para testar a função.

```
// executar tarefa
int quantidade = readCommands ( comandos, "Tarefa0301.txt" );
message [0] = '\0';           // limpar a mensagem
sprintf ( message, "Commands = %d", quantidade );
show_Text ( message );
```

- 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.
- 12.) Copiar a versão atual do programa para outra (nova) – Guia0304.cpp.
- 13.) Realizar as mudanças de versão e
acrescentar ao programa as modificações indicadas abaixo:

```
/**
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( )

```



```

/**
    metodo para executar comandos de arquivo.
    @param commands - grupo de comandos para executar
*/
void doCommands( int length, int commands [ ] )
{
    // definir dados
    int action = 0;
    int x      = 0;

    // repetir para a quantidade de comandos
    for ( x = 0; x < length; x = x + 1 )
    {
        // executar esse comando
        execute( commands [ x ] );
    } // end for
} // end doCommands( )

/**
    doTask - Metodo para executar comandos de arquivo.
    @param fileName - nome do arquivo
*/
void doTask( char * fileName )
{
    // definir dados locais
    int quantidade = 0;
    int comandos [ COMMANDS ];

    // ler quantidade e comandos
    quantidade = readCommands ( comandos, "Tarefa0301.txt" );
    message [0] = '\0';           // limpar a mensagem
    sprintf ( message, "Commands = %d", quantidade );
    show_Text ( message );

    // executar comandos
    doCommands ( quantidade, comandos );
} // end doTask( )

/**
    * doCommands - Metodo para executar comandos de arquivo.
    * @param length      - quantidade de comandos
    * @param commands - grupo de comandos para executar
    */
void doCommands( int length, int commands [ ] )
{
    // definir dados
    int action = 0;
    int x      = 0;

    // repetir para a quantidade de comandos
    for ( x = 0; x < length; x = x + 1 )
    {
        // executar esse comando
        execute( commands [ x ] );
    } // end for
} // end doCommands( )

```

Na parte principal, alterar a chamada para testar o método.

```
// executar tarefa
quantidade = robot->readCommands ( comandos, "Tarefa0301.txt" );
message [0] = '\0';           // limpar a mensagem
sprintf ( message, "Commands = %d", quantidade );
show_Text ( message );

robot->doCommands ( quantidade, comandos );
```

- 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) – Guia0305.cpp.
- 17.) Realizar as mudanças de versão e
acrescentar ao programa as modificações indicadas abaixo:

```
/**
doTask - Metodo para executar comandos de arquivo.
@param fileName - nome do arquivo
*/
void doTask( const char * fileName )
{
// definir dados locais
int quantidade = 0;
int comandos [ MAX_COMMANDS ];

// ler quantidade e comandos
quantidade = readCommands ( comandos, "Tarefa0301.txt" );
message [0] = '\0';           // limpar a mensagem
sprintf ( message, "Commands = %d", quantidade );
show_Text ( message );

// executar comandos
doCommands ( quantidade, comandos );
} // end doTask()
```

Na parte principal, alterar a chamada para testar o método.

```
// executar tarefa
robot->doTask ( "Tarefa0301.txt" );
```

- 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) – Guia0306.cpp.
- 21.) Realizar as mudanças de versão e
acrescentar ao programa as modificações indicadas abaixo:

```
/**
 * mapWorld - Metodo para o robot explorar o mundo.
 */
void mapWorld ( )
{
    // obter o tamanho do mundo
    if ( world != nullptr )
    {
        // informar o tamanho do mundo
        message [0] = '\0';           // limpar a mensagem
        sprintf ( message, "World is %dx%d",
                world->avenues( ), world->streets( ) );
        show_Text ( message );
    } // end if
} // end mapWorld( )
```

Na parte principal, alterar a chamada para testar o método.

```
// executar tarefa
robot->mapWorld ( );
```

- 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) – Guia0307.cpp.

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

```
/**
 * mapWorld - Metodo para o robot explorar o mundo.
 */
void mapWorld ( )
{
    // definir dados locais
    int avenue   = 0,
        street   = 0;
    int beepers  = 0;

    // obter o tamanho do mundo
    if ( world != nullptr )
    {
        // informar o tamanho do mundo
        message [0] = '\0';
        sprintf ( message, "World is %dx%d", world->avenues( ), world->streets( ) );
        show_Text ( message );

        // percorrer o mundo procurando marcadores
        for ( street=1; street<=world->streets( ); street=street+1 )
        {
            for ( avenue=1; avenue<=world->avenues( ); avenue=avenue+1 )
            {
                // se proximo a um marcador
                if ( nextToABeeper( ) )
                {
                    // informar marcador nessa posicao
                    message [0] = '\0';
                    sprintf ( message, "Beeper at (%d,%d)", avenue, street );
                    show_Text ( message );
                    // encontrado mais um marcador
                    beepers = beepers + 1;
                } // end if
                // mover para a proxima posicao
                if ( avenue < world->avenues( ) )
                {
                    move( );
                } // end if
            } // end for
            turnLeft ( );
            turnLeft ( );
            moveN ( world->avenues( )-1 );
            if ( street < world->streets( ) )
            {
                turnRight ( );
                move ( );
                turnRight ( );
            } // end if
        } // end for
    } // end if
} // end mapWorld( )
```

- 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) – Guia0308.cpp.
- 30.) Realizar as mudanças de versão e acrescentar ao programa as modificações indicadas abaixo:

```
/**
    mapWorld    Metodo para o robot explorar o mundo
                e fazer um mapa.
    @param map - arranjo bidimensional ( matriz )
                onde guardar o mapa
*/
void mapWorld ( int map [ ][WIDTH] )
{
    // definir dados locais
    int avenue   = 0,
        street   = 0;
    int beepers  = 0;

    // obter o tamanho do mundo
    if ( world != nullptr )
    {
        // informar o tamanho do mundo
        message [0] = '\0';
        sprintf ( message, "World is %dx%d", world->avenues( ), world->streets( ) );
        show_Text ( message );

        // percorrer o mundo procurando beepers
        for ( street=1; street<=world->streets( ); street=street+1 )
        {
            for ( avenue=1; avenue<=world->avenues( ); avenue=avenue+1 )
            {
                // limpar posicao no mapa
                map [ street-1 ][ avenue-1 ] = 0;
                // se proximo a um marcador
                if ( nextToABeeper( ) )
                {
                    // informar marcador nesta posicao
                    message [0] = '\0';
                    sprintf ( message, "Beeper at (%d,%d)", avenue, street );
                    show_Text ( message );
                    // marcar posicao no mapa
                    map [ street-1 ][ avenue-1 ] = 1;
                    // encontrado mais um marcador
                    beepers = beepers + 1;
                } // end if
            }
        }
    }
}
```

```

        // mover para a proxima posicao
        if ( avenue < world->avenues( ) )
        {
            move( );
        } // end if
    } // end for
    turnLeft ( );
    turnLeft ( );
    moveN ( world->avenues( )-1 );
    if ( street < world->streets( ) )
    {
        turnRight ( );
        move ( );
        turnRight ( );
    } // end if
} // end for
} // end if

} // end mapWorld( )

```

Na parte principal, alterar a chamada para testar o método.

```

// definir armazenador para o mapa
int map [HEIGHT][WIDTH]; // altura x largura

...
// executar tarefa
robot->mapWorld ( map );

```

- 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) – Guia0309.cpp.

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

```
/**
    saveMap - Metodo para guardar um mapa em arquivo.
    @param filename - nome do arquivo onde guardar o mapa
    @param map      - arranjo bidimensional (matriz) com o mapa
*/
void saveMap ( const char * fileName, int map [ ][WIDTH] )
{
    // definir dados locais
    int  avenue  = 0,
        street   = 0;
    std::ofstream archive ( fileName );

    // testar se ha' informacao
    if ( world != nullptr )
    {
        // guardar o tamanho do mundo
        archive << world->avenues( ) << "\n";
        archive << world->streets( ) << "\n";

        // percorrer o mundo procurando beepers
        for ( street=1; street<=world->streets( ); street=street+1 )
        {
            for ( avenue=1; avenue<=world->avenues( ); avenue=avenue+1 )
            {
                // guardar informacao no arquivo
                if ( map [ street-1 ][ avenue-1 ] == 1 )
                {
                    archive << avenue << std::endl;
                    archive << street  << std::endl;
                    archive << map [street-1][avenue-1] << std::endl;
                } // end if
            } // end for
        } // end for
        // fechar arquivo
        archive.close ( );
    } // end if
} // end saveMap ( )
```

Na parte principal, alterar a chamada para testar o método.

```
// executar tarefas
robot->mapWorld ( map );
robot->saveMap ( "Mapa0309.txt", map );
```

- 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) – Guia0310.cpp.

```
/**
 showMap - Metodo para ler um mapa em arquivo.
 @param avenues - largura do mapa
 @param streets - altura do mapa
 @param map - arranjo bidimensional (matriz) com o mapa
 */
void showMap ( int avenues, int streets, char map [ ][HEIGHT] )
{
    // definir dados
    int x = 0,
        y = 0;

    // percorrer o mundo procurando marcadores
    clrscr ( );
    std::cout << " Mapa de marcadores\n\n";

    std::cout << " ";
    for ( x=0; x<streets; x=x+1 )
    {
        std::cout << (x+1);
    } // end for
    std::cout << std::endl;
    for ( y=0; y<avenues; y=y+1 )
    {
        std::cout << (y+1);
        for ( x=0; x<avenues; x=x+1 )
        {
            std::cout << map [ y ][ x ];
        } // end for
        std::cout << std::endl;
    } // end for
    IO_pause ( " Apertar ENTER para continuar." );
} // end showMap ( )

/**
 readMap - Metodo para ler um mapa em arquivo.
 @param fileName - nome do arquivo com o mapa
 */
void readMap ( const char * fileName )
{
    // definir dados
    int avenue = 0,
        street = 0;
    int avenues = 0,
        streets = 0;
    int x = 0,
        y = 0,
        z = 0;
    std::ifstream archive ( fileName );

    // reservar area para guardar o maior mapa possivel
    char map [ WIDTH ][ HEIGHT ];
```



```

// obter o tamanho do mundo
archive >> avenues;
archive >> streets;

// testar configuracao do mapa
if ( ( 0 < avenues && avenues < world->width ) &&
    ( 0 < streets && streets < world->height ) )
{
    // percorrer o mundo procurando marcadores
    for ( y=0; y<streets; y=y+1 )
    {
        for ( x=0; x<avenues; x=x+1 )
        {
            map [ y ][ x ] = '.';
        } // end for
        map [ y ][ x ] = '\0';
    } // end for

    // repetir enquanto houver dados
    archive >> avenue; // tentar ler a primeira linha
    while ( ! archive.eof() ) // testar se nao encontrado o fim
    {
        // contar mais um comando
        archive >> street;
        archive >> z ;
        // testar se informacoes validas
        if ( (1 <= avenue && avenue <= world->width ) &&
            (1 <= street && street <= world->height ) &&
            (z == 1) )
        {
            map [ street-1 ][ avenue-1 ] = 'X';
        } // end if
        archive >> avenue; // tentar ler a proxima linha
    } // end while
    // fechar o arquivo
    // RECOMENDAVEL para a leitura
    archive.close ();

    // mostrar o mapa
    showMap ( avenues, streets, map );
} // end if
// end readMap ( )

```

Na parte principal, alterar a chamada para testar o método.

```

// executar tarefas
robot->mapWorld ( map );
robot->saveMap ( "Mapa0310.txt", map );
robot->readMap ( "Mapa0310.txt" );

```

- 38.) 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.
- 39.) 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.

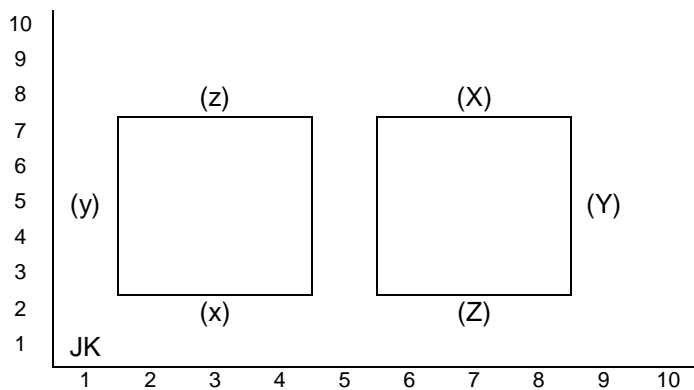
Prever, realizar e registrar todos os testes efetuados.

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 Guia0311 para:

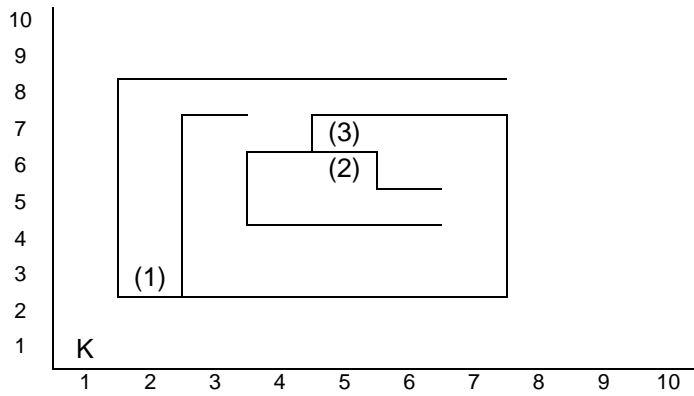
- definir um robô na posição (1,1), voltado para leste, sem marcadores;
- dispor blocos em uma configuração semelhante a dada abaixo:



- definir dois quadrados, lado a lado, separados por uma passagem entre eles;
- definir marcadores em volta do primeiro quadrado, um de cada lado, nas posições (x-y-z);
- tarefa:
o robô deverá buscar os marcadores em (x-y-z), e movê-los até as novas posições indicadas (X-Y-Z), juntas ao segundo quadrado;
- restrição:
o robô deverá passar pelo "corredor" entre os quadrados, pelo duas vezes, na ida e na volta, como em um '8' deitado, antes de voltar a posição inicial;
a especificação da tarefa deverá ser feita por um arquivo (Tarefa0311.txt);
se necessário, consultar como foi feito no Guia_02.

02.) Definir um conjunto de ações em um programa Guia0312 para:

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



- definir um labirinto com os marcadores indicados segundo o modelo acima;
- tarefa:
 - o robô deverá buscar os marcadores, na ordem crescente indicada pelas quantidades, e trazê-los à posição inicial;
 - a especificação da tarefa deverá ser feita por um arquivo (Tarefa0052.txt);
 - se necessário, consultar o Guia_02;
- métodos deverão ser criados para ajudar o robô a mover-se no labirinto:

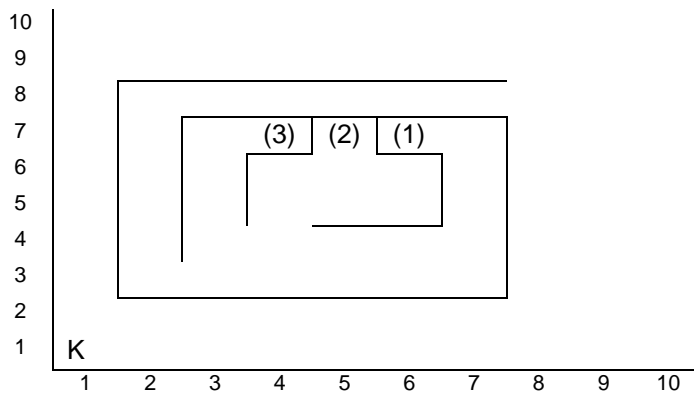
turnAround() - inverter a direção do movimento
 (virar 180°, para a direção contrária)

turnAroundCornerLeft() - fazer curva fechada à esquerda ("U")
 (acompanhar uma parede interna,
 com aquelas mais internas)

DICAS: Inserir novos comandos no método execute().

03.) Definir um conjunto de ações em um programa Guia0313 para:

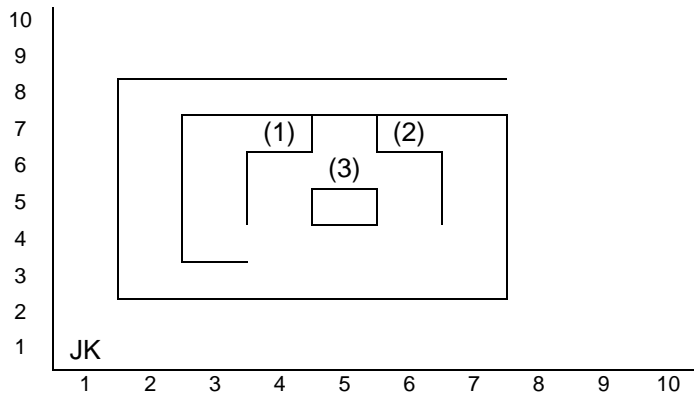
- definir um robô na posição (1,1), voltado para leste, sem marcadores;
- dispor blocos em uma configuração semelhante a dada abaixo:



- definir um labirinto com os marcadores indicados, segundo o modelo acima;
- tarefa:
o robô deverá buscar os marcadores, na ordem indicada, e trazê-los à posição inicial;
guardar em tabelas separadas as coordenadas (x, y) e a quantidade de marcadores recolhidos.

DICAS: Seguir a parede pelo lado direito, combinando testes nativos `rightIsClear()` e `frontIsClear()`.

- 04.) Definir um conjunto de ações em um programa Guia0314 para:
- definir um robô na posição (1,1), voltado para leste, sem marcadores;
 - dispor blocos em uma configuração semelhante a dada abaixo:

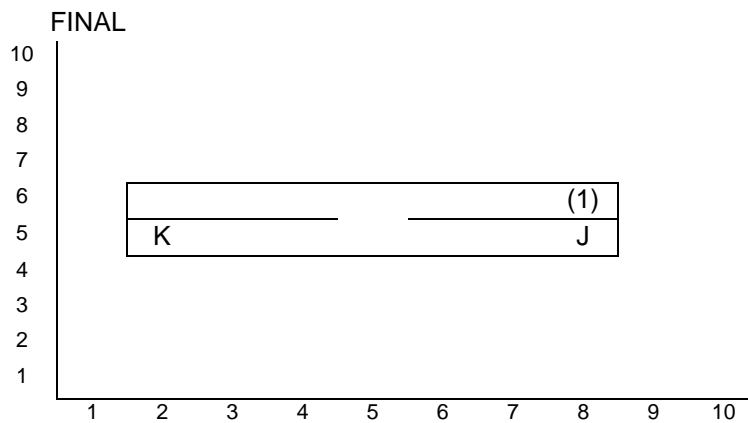
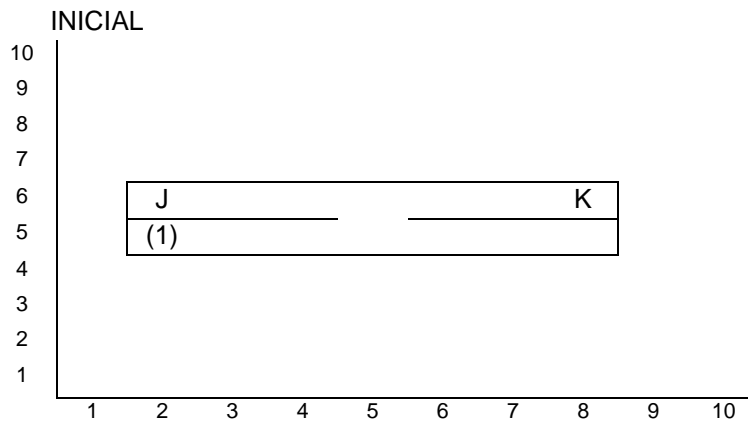


- o robô deverá buscar os marcadores indicados, preferencialmente da direita para a esquerda;
- retornar à posição inicial, voltar-se para leste e desligar-se;
- poderá ser marcado em um mapa o deslocamento efetuado, se as posições percorridas forem marcadas ou se forem guardadas em arquivo e mostradas ao final como o roteiro percorrido.

DICA: Ao mover o robô, colocar uma marca ('x') na posição correspondente no mapa, ou gravar as coordenadas (x,y) em arquivo.

05.) Definir um conjunto de ações em um programa Guia0315 para:

- dispor blocos em uma configuração semelhante a abaixo:



- definir dois robôs, conforme indicado na configuração inicial, por exemplo,

```
robot1->create ( 1, 1, EAST, 0, "Jarel" );
robot2->create ( 1, 1, EAST, 0, "Karel" );
```

- definir um marcador na posição indicada, inicialmente;

- definir paredes entre os robôs, exceto na metade do caminho;

- tarefa:

o robô R1 deverá buscar o marcador (1),

mover-se até a passagem; ir à parte de acima, aguardar a aproximação de R2,

e entregar o marcador; depois, o robô R2 levará o marcador até posição final indicada

e ambos retornarão às suas respectivas posições iniciais;

- dois métodos adicionais deverão ser criados:

halfPathRight() - andar metade do caminho para a direita

halfPathLeft() - andar metade do caminho para a esquerda

- outros métodos envolvendo sensores deverão ser usados para a percepção de um robô em relação ao outro, antes da transferência do marcador:

a.) testar se próximo a outro robô

```
if ( nextToARobot( ) ) // robo (1)
{
    // comandos dependentes da condição
    putBeeper( );      // exemplo
}
else
{
    // comandos dependentes do contrário
} // end if
```

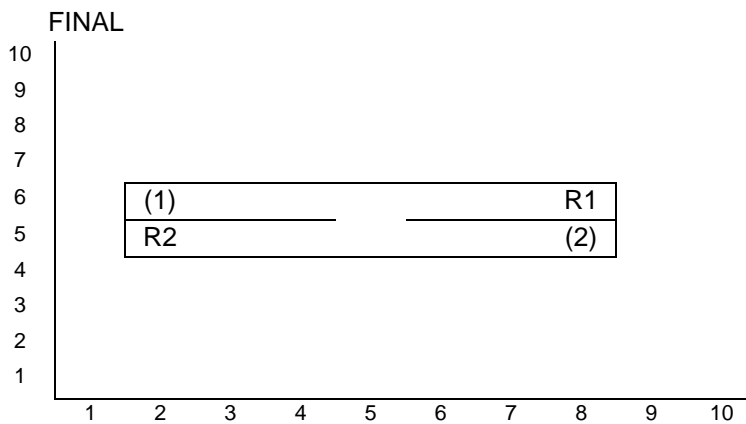
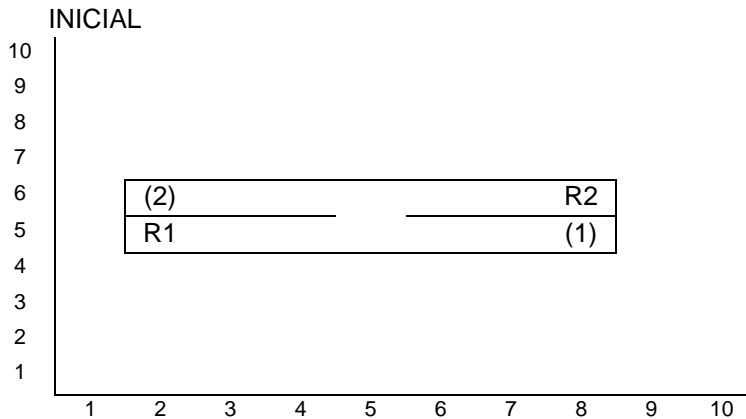
b.) testar se próximo a um marcador

```
if ( nextToABeeper( ) ) // robo (2)
{
    // comandos dependentes da condição
    pickBeeper( );      // exemplo
}
else
{
    // comandos dependentes do contrário
} // end if
```

- o robô R1 deverá testar se está próximo ao outro (ambos na mesma posição); se estiver, deverá deixar o marcador para o outro robô (R2) pegar; então, o segundo deverá testar se há um marcador disponível e recolher esse marcador, antes de completar a tarefa, e voltar à posição inicial.
Se o primeiro robô chegar à posição combinada para a entrega, e o outro não estiver lá, deverá retornar à posição inicial com o marcador.
DICA: Dividir a tarefa em subtarefas.

Tarefa extra

- E1.) Definir um conjunto de ações em um programa Guia03E1 para:
dividir as tarefas do último exercício e distribuí-las em arquivos diferentes,
chamando-os para execução na ordem esperada.
- E2.) Definir um conjunto de ações em um programa Guia02E2 para:
definir um conjunto de ações para resolver o seguinte problema:
dispor blocos em uma configuração semelhante a abaixo:



- onde cada robô deverá buscar os marcadores de seu respectivo "andar",
irem até o ponto de encontro (metade superior),
trocarem os marcadores, voltar aos seus "andares",
guardar os marcadores recebidos e retornarem às posições iniciais.

DICA: Um robô só poderá receber marcadores de outro,
se não estiver carregando algum.

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.

Conclusão

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