

A photograph of a modern architectural interior. In the foreground, a wide staircase with dark steps and black handrails leads upwards. To the left, a balcony with a metal railing is visible. The ceiling features a large, geometric glass skylight with a dark frame, through which a clear blue sky is visible. The overall lighting is bright and clean, highlighting the architectural details.

ArchiStation®

Guia de programação ASX 3.2

ArchiStation®

Guia de programação ASX 3.2

As informações contidas neste documento estão sujeitas a alterações sem aviso prévio. Nenhuma parte deste documento pode ser reproduzida ou transmitida de qualquer forma ou meio, eletrônico ou mecânico, sem a permissão por escrito da Hemero Software Ltda.

© 2023 Hemero Software Ltda. Todos os direitos reservados.

Hemero® e **ArchiStation®** são marcas registradas da Hemero Software Ltda.

Microsoft, Windows e logotipos associados são marcas registradas da Microsoft Corporation. Intel, Pentium e logotipos associados são marcas comerciais ou marcas registradas da Intel Corporation. OpenGL e logotipos associados são marcas registradas da Silicon Graphics, Inc. GeForce é marca registrada da NVIDIA Corporation. Radeon é marca registrada da ATI, Inc. POV-Ray e Persistence of Vision são marcas registradas da Persistence of Vision Raytracer Pty. Ltd.

As demais marcas citadas neste documento são de propriedade de seus respectivos fabricantes.

Revisão 015 – Junho/2023

Conteúdo

Notas da versão 3.2	7
Introdução	8
Linguagem de programação	8
A Interface de programação ASX	8
O primeiro script ASX	9
Explicação do funcionamento do script	10
Estrutura do C/C++	11
Sintaxe	11
Identificadores	11
Declaração de variáveis	11
Tipos básicos de variáveis	11
Exemplos de declaração de variáveis	11
Variáveis globais e locais	12
Avaliando expressões	12
Operador de atribuição	12
Operadores aritméticos	12
Incrementos e decrementos	13
Operadores de relação e lógicos	13
Instruções condicionais	14
if e else	14
Instruções de repetição	14
for	14
break e continue	15
while	16
while do	16
Declarando funções	16
Função sem retorno de valor	16
Função com passagem de parâmetros	17
Função com retorno de valor	17
Classes e objetos	18
Cadeias de caracteres	19
Código ASCII	19
Métodos associados	20
Funções relacionadas a strings	20
Definindo e manipulando pontos no espaço tridimensional	21
A classe point	21
Descrição dos métodos	21

Funções para Desenhar	23
Criando um linha	23
Outras funções.....	23
Alterando as propriedades de desenho	24
Camadas.....	24
Ajustando a espessura e a altura.....	25
Desenhando objetos com algoritmos e expressões matemáticas.....	25
 Obtendo dados do usuário	27
Fazendo com que o usuário indique um ponto.....	27
Obtendo números inteiros.....	28
Obtendo números reais	28
Obtendo cadeias de caracteres.....	29
Outras funções de entrada de dados	29
 Componentes ASX	30
Criando um Componente ASX.....	31
Editando as propriedades	33
Adicionando novas propriedades	34
Controlando os valores das propriedades.....	35
Organizando a disposição das propriedades.....	35
Descrição e nomes com acento no Inspetor de objetos	36
Outros tipos de propriedades	37
Exemplos de componentes.....	37
Cerca - função PointAt();.....	37
Escada.....	39
Espiral - senos e cossenos.....	40
 Programando a interface	42
Adicionando Guias ao Menu principal.....	42
Menu de ferramentas.....	42
Botões com ícones nos menus.....	43
Janelas de propriedades	44
 Tratamento de arquivos de dados	46
Operações de leitura do arquivo.....	46
Operações de escrita do arquivo	47
Manipulação do cursor do arquivo	47
 Obtendo dados de planilhas do MS Excel	50
 Listas de armazenamento	52
Definindo o item selecionado	53
Métodos genéricos associados às listas	53

Métodos especiais da classe stringList	53
Métodos especiais da classe pointList	54
Adicionando materiais, blocos, perfis e hachuras.....	55
Adicionando materiais.....	55
Adicionando blocos	55
Funções para inserir blocos	56
Adicionando perfis	56
Funções para criar perfis (Mudar para Shaped)	56
Adicionando hachuras	57
Funções para criar hachuras.....	57
Acessando os objetos do projeto	58
Métodos para modificar objetos	59
Acessando as propriedades dos objetos.....	60
Lista de objetos e suas propriedades principais	60
Exemplos de aplicações	62
Lista de materiais	62
Objetos BIM	64
Estilos de componentes - TStyle	64
Objetos BIM anexados.....	65
Componentes ASX	66
Acessando as tabelas	66
Objetos da tabela Layers (Camadas).....	66
Objetos da tabela Materials	66
Objetos da tabela Blocks	66
Selecionando objetos	68
Funções de seleção	68
Obtendo acesso aos objetos selecionados	68
Instalações	69
Lista de Funções	70
Funções para Desenhar	70
Alterando estados e tabelas	71
Espessura e altura	71
Camadas.....	71
Grupos.....	71
Pavimentos.....	71
Funções de controle	71
Entrada de dados.....	72
Funções Matemáticas.....	73

Funções Aritméticas.....	73
Gerador de números aleatórios.....	73
Funções Trigonométricas.....	73
Funções de geometria.....	74
Funções de conversão.....	74
Funções de formatação.....	74
Caixas de mensagem e diálogo.....	75
Message() - Mensagem de texto.....	75
MessageBox() - Caixa de mensagem de texto.....	75
GetFile() - Diálogo de seleção de nome de arquivo.....	76
Variáveis do sistema.....	76
Modificação da interface.....	77
Criando uma guia no menu principal.....	77
Criando um menu em uma guia.....	77
Adicionando um botão de comando a um menu.....	77
Obtendo dados de planilhas do MS Excel.....	78
Tipos de variáveis do AngelScript.....	79
Valores booleanos.....	79
Números inteiros.....	79
Números reais.....	79
Cadeia de caracteres.....	79

Notas da versão 3.2

Nesta versão do ASX foram implementados recursos para acessar dados de Estilos de componentes e seus tipos, além dos dados instanciados em objetos do desenho, com o objetivo de facilitar a criação de aplicativos para projetos de instalações.

CAPÍTULO 1

Introdução

A tecnologia **ASX** foi implementada no Archistation para permitir que os usuários com conhecimento de programação desenvolvam facilmente novos comandos e componentes utilizando a interface de programação integrada. Este guia pretende descrever os recursos e possibilidades que ajudarão você a criar soluções personalizadas de acordo com suas necessidades específicas.

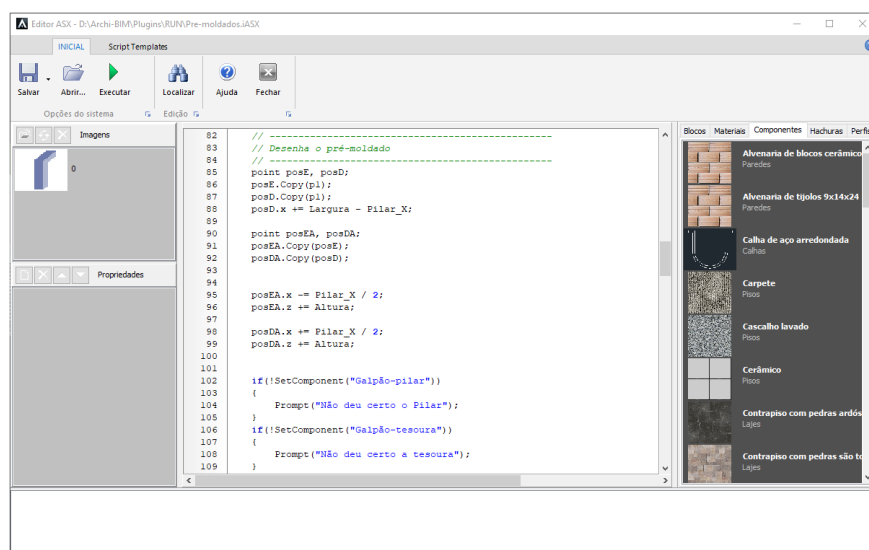
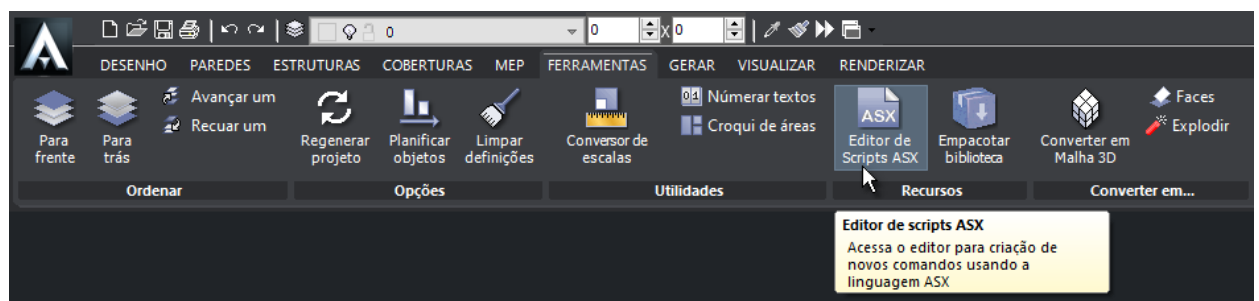
Linguagem de programação

Uma linguagem de programação é um método padronizado para expressar instruções ao computador, contendo regras sintáticas e semânticas para definir um programa, que permitirá especificar precisamente sobre quais dados o computador vai atuar.

As instruções **ASX** do **ArchiStation** são baseadas em linguagem **C/C++**, utilizando as bibliotecas do interpretador **AngelScript**. Para saber mais sobre a linguagem utilizada acesse <http://www.angelcode.com/angelscript>.

A Interface de programação ASX

ASX, sigla para *ArchiStation Extended*, é uma interface criada para fornecer ao usuário uma plataforma integrada de desenvolvimento, tornando prático o processo de programação. Para acessar selecione no menu principal o item **Ferramentas ► Editor ASX**.



O Editor ASX

De modo geral, com scripts **ASX** você poderá:

- Implementar novos comandos e aplicações para fazer aquilo que você precisa
- Adicionar novos itens a interface do ArchiStation
- Acessar arquivos do disco para ler e salvar informações em formatos personalizados
- Criar componentes inteligentes baseados em scripts

O primeiro script ASX

1. Acesse no menu principal **Ferramentas ► Editor de scripts ASX**.

Ao acessar pela primeira vez o editor de scripts, a função **void main()** já esta declarada. É por esta função que o ArchiStation vai iniciar a execução do script.

```
void main()
{
    //Adicione aqui o código do script
}
```

Na linguagem C/C++ os sinais '{' e '}' são utilizados para delimitar blocos de código, sendo que neste caso indicam o início e o fim da função **void main()**.

O sinal '/' é utilizado para inserir comentários no código, indica que todos os caracteres até o final da linha deverão ser ignorados no processo de execução do script.

2. Digite o código logo abaixo do comentário de maneira que o script fique assim:

Atenção! A linguagem C/C++ é *case sensitive* (diferencia maiúsculas de minúsculas) é importante digitar as maiúsculas e minúsculas conforme indicado.

Dica: Se você esta utilizando a versão digital deste documento em seu computador, poderá selecionar os códigos nos exemplos e copiar pressionando **<Ctrl+C>** e depois colar no editor ASX com **<Ctrl+V>**.


```
void main()
{
    //Adicione aqui o código do script

    point p1,p2;

    Prompt("Entre com o primeiro ponto:");
    GetPoint(p1);

    Prompt("Entre com o segundo ponto:");
    GetPoint(p2,p1);

    Line(p1,p2);
}
```

3. Pressione o botão  para salvar o projeto. A janela **Salvar como** é exibida e automaticamente direcionada para a pasta **Plugins** da instalação do ArchiStation. Entre com o nome **"teste"** pressione o botão **Salvar**.

4. Pressione o botão  para executar o script.

O script solicita ao usuário a indicação de dois pontos e cria uma linha entre eles. O script então é encerrado e o controle devolvido ao usuário.

Ao salvar o script na pasta **Plugins**, você estará implementando um novo comando que poderá ser acessado através do **Quadro comando**, digitando seu nome seguido da tecla **<ENTER>**. Neste caso, ao digitar **teste <ENTER>** o script será executado novamente.

DICA: Todos os arquivos de script com extensão **.asx** salvos na pasta **Plugins** da instalação do ArchiStation podem ser executados entrando com seu nome no **Quadro de comando**.

Explicação do funcionamento do script

Se você não tem experiência com linguagens de programação pode não entender muito bem a explicação a seguir. Mas fique tranquilo, as instruções utilizadas neste script serão descritas detalhadamente nos próximos capítulos.

A primeira linha digitada dentro da função **void main()**, declara dois objetos do tipo **point** (pontos do ArchiStation), denominados **p1** e **p2**, que serão usados pelo script.

```
point p1,p2;
```

A função **Prompt()** é chamada para exibir o texto entre aspas no **Quadro de comando**.

```
Prompt("Entre com o primeiro ponto:");
```

A função **GetPoint()** solicita ao usuário a entrada de um ponto que será armazenado em **p1**.

```
GetPoint (p1) ;
```

A função **Prompt()** é novamente chamada para exibir outro texto.

```
Prompt("Entre com o segundo ponto:");
```

Agora, uma variação da função **GetPoint()**, com dois parâmetros, solicita ao usuário a entrada do ponto **p2** de forma ancorada a **p1**. Ou seja, uma linha de apoio será desenhada entre **p1** e o cursor do mouse até que o usuário indique o segundo ponto que será armazenado em **p2**.

```
GetPoint (p2, p1) ;
```

Agora com os objetos **p1** e **p2** atualizadas com os valores introduzidos pelo usuário, a função **Line()** desenha uma linha entre os dois pontos.

```
Line (p1, p2) ;
```

CAPÍTULO 2

Estrutura do C/C++

Sintaxe

Todo o programa em C inicia sua execução pela função **void main()**, sendo obrigatória sua declaração no programa principal. As instruções de uma função devem estar contidas dentro de um bloco de dados, que podem conter várias instruções, são iniciados pelo sinal abrir chaves '{' e encerrado pelo fechar chaves '}'. Em uma função podem haver mais de um bloco de dados, e o mesmo número de chaves abertas deve ser o de chaves fechadas. Todas as instruções devem ser encerradas pelo sinal ponto e vírgula ';' e podem ser uma declaração, expressão ou a chamada de uma função.

Identificadores

Os identificadores são utilizados para nomear funções, variáveis e outros objetos que podem ser definidos pelo usuário. São formados por letras, números ou pelo sinal '_' (*underline*). O primeiro caractere deve sempre ser uma letra ou o sinal '_' (*underline*). As letras não devem possuir acentos ou cedilhas e há diferenciação entre maiúsculas e minúsculas.

Declaração de variáveis

Variáveis são identificadores em que podemos atribuir e modificar os valores. Antes de utilizar uma variável em C/C++, ela deve ser declarada de acordo com o tipo de dados que vai armazenar. A declaração de uma variável deve ser realizada com a seguinte sintaxe:

```
<tipo de variável> <identificador>;
```

Tipos básicos de variáveis

Os tipos mais comuns de variáveis são:

bool	Valores booleanos que podem ser false (falso ou 0) ou true (verdadeiro ou 1).
int	Números inteiros compreendidos entre -2.147.483.648 e 2.147.483.648
double	Números reais, decimais ou de ponto flutuante
distance	Distância no desenho. Funciona igual ao <i>double</i> , mas os valores são convertidos de acordo com a unidade do desenho para serem exibidos no Inspector de objetos para a edição do usuário.
string	Cadeia de caracteres

Exemplos de declaração de variáveis

Declara a variável 'a' como um inteiro:

```
int a;
```

Declara a variável 'Nome' como uma cadeia de caracteres:

```
string Nome;
```

Opcionalmente, no momento da declaração, você poderá atribuir um valor a variável:

```
int a=5;
double pi=3.1416;
string Nome="Teste";
```

E também poderá declarar várias variáveis do mesmo tipo em uma única instrução, separando-as por vírgula:

```
int a=5 , b=10 , c;

string Nome="Teste" , Descricao="Desenha uma linha";
```

Em C, as aspas são utilizadas para representar cadeias de caracteres. Por exemplo, 1234 é um valor numérico e é diferente de "1234" que é uma cadeia de caracteres. É por isso que os valores atribuídos as variáveis do tipo **string** são definidos entre Aspas.

Variáveis globais e locais

As variáveis declaradas no início de código, antes da declaração da função **void main()**, são consideradas variáveis **globais**, podem ser acessadas em qualquer parte do programa, incluindo outras funções.

As variáveis declaradas dentro de uma função são consideradas variáveis **locais**, podem ser acessadas apenas dentro do blocos de dados onde foram declaradas ou de seus sub-blocos.

Exemplo:

```
int pi=3.14; // variável global

void main()
{
    double a=2; // variável local
}
```

Avaliando expressões

A avaliação de expressões é executada por meio de operadores.

Operador de atribuição

O operador de atribuição é o sinal de '=' (igual), utilizado para atribuir valores as variáveis.

Exemplos:

```
pi = 3.1416;

a = b;
```

Operadores aritméticos

Os principais operadores aritméticos do C são os mesmos utilizados na maioria das linguagens:

+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Módulo - Divisão na qual é retornado o resto.
(e)	Parênteses

Exemplos:

```
a = a + 5;
x = y + (3 * c);
```

O operador de adição também pode ser utilizado para unir duas ou mais cadeias de caracteres:

```
string Texto="Nome: ";
Texto = Texto + "ArchiStation";
```

No ambiente do *AngelScript*, ao tentarmos adicionar valores numérico de tipo **int** ou **double** a uma cadeia de caracteres, os valores numéricos são tratados como cadeia de caracteres:

```
double pi = 3.1416;
string A = "O valor de pi é igual a "+pi;
Prompt(A);
```

Executando este script obteremos como resposta no **Quadro de comando**:

O valor de pi é igual a 3.1416
COMANDO:

Incrementos e decrementos

Em C também podemos fazer o uso dos operadores de incremento e decremento para os valores de tipo numéricos:

++	Incrementa (adiciona 1)
--	Decrementa (subtrai 1)

Exemplos:

```
x++; // Equivalente a x=x+1;
y--; // Equivalente a y=y-1;
```

Operadores de relação e lógicos

Os operadores de relação retornam apenas os valores **1** para **verdadeiro** e **0** para **falso**, de acordo com a relação dos operandos entre si, enquanto que os operadores lógicos referem-se as maneiras como estas relações podem ser associadas. São normalmente utilizados em instruções condicionais.

Relacionais:

>	Maior que
>=	Maior ou igual
<	Menor que
<=	Menor ou igual
==	Igual
!=	Diferente

Lógicos:

&&	AND (e)
	OR (ou)
!	NOT (não)

Instruções condicionais

if e else

As instruções **if** e **else** permitem ao processador executar uma instrução ou outra de acordo com o resultado da avaliação de uma condição.

Sintaxe:

```
if ( <condição> ) < instrução se condição verdadeira >;
```

OU

```
if ( <condição> ) < instrução se condição verdadeira >;
else < instrução se condição falsa >;
```

Exemplos:

```
// Se o valor de 'A' é igual a B, adiciona 1 em A
if ( A==0 ) A=A+1;

// Se o valor de 'A' é negativo, torna positivo
if ( A<0 ) A=A*-1;

// Mostra no Quadro de comandos a mensagem se 'A' é maior que 20
if ( A>20 ) Prompt( "A é maior que 20." );
else Prompt( "A é menor ou igual a 20." );
```

Observação: Em C/C++ o sinal para verificar e igualdade são dois sinais de igual seguidos '=='. Apenas um sinal de igual '=' representa uma instrução de atribuição.

As instruções também podem estar agrupadas em blocos de dados:

```
// Mostra se 'A' e (operador &&) 'B' são maiores que 20
if ( A>20 && B>20 )
{
    Prompt("A e B são maiores que 20.");
}
else
{
    if ( A>20 ) Prompt( "A é maior que 20." );
    if ( B>20 ) Prompt( "B é maior que 20." );
}
```

Instruções de repetição

for

A instrução **for** permite definir uma variável de controle e implementar um loop de instruções que será executado iterativamente enquanto a condição for verdadeira, incrementado a variável de controle um valor definido a cada iteração.

Sintaxe:

```
for( <inicialização>; <condição>; <incremento> ) < instrução enquanto a
condição for verdadeira >;
```

A **inicialização** é um comando de atribuição usado para estabelecer a variável de controle do loop. A **condição** é uma expressão de relação que avalia a variável de controle e de acordo com o resultado executa as instruções do loop. Após cada execução do loop, o **incremento** determina como a variável de controle será alterada.

Exemplo: O trecho de código a seguir estabelece um loop em que a variável de controle 'a' é inicializada com o valor 1 e será incrementada a cada iteração enquanto 'a' for menor ou igual a 5.

```
for( int a=1; a<=5; a++ ) Prompt("Iteração: "+a+"/5.");
```

Com a execução deste código, o **Quadro de comando** mostrará :

```
Iteração: 1/5
Iteração: 2/5
Iteração: 3/5
Iteração: 4/5
Iteração: 5/5
COMANDO:
```

A instrução a ser repetida, também poderá ser um bloco de dados:

```
for( int a=1; a<=5; a++ )
{
    Prompt("Iteração: "+a+"/5.");
    if(a==5) Prompt("Fim.");
}
```

break e continue

As instruções **break** e **continue** permitem um controle maior sobre a execução dos blocos de loops em instruções de repetição.

Quando o comando **break** é encontrado em qualquer lugar dentro do bloco de loop, ele causa seu término. O processamento continuará pelo código seguinte ao loop.

Exemplo: A instrução **for** utilizada sem parâmetros na forma **for(;;)** causa um loop infinito. Empregaremos o comando **break** para interromper o loop quando 'a' for maior que 5. Obteremos a mesma resposta no **Quadro de comando** obtida com o exemplo anterior.

```
int a=0;
for(;;)
{
    a=a+1;
    if(a>5)
    {
        Prompt("Fim.");
        break;
    }
    Prompt("Iteração: "+a+"/5.");
}
```

O comando **continue** força a próxima iteração do loop. Esta operação salta o resto das instruções do bloco de loop, realiza o incremento e retorna ao início do bloco de loop.

Atenção! O uso de instruções de repetição podem fazer o programa cair em elo infinito, da modo a nunca permitir o fim da execução do script. Nestes e outros casos onde seja necessário interromper a execução do script, mantenha as duas teclas Ctrl pressionadas até paralisar a execução.

Exemplo: O seguinte trecho de código mostra todos os números entre 1 e 100 divisíveis por 7.

```
for( int a=1; a<=100; a++ )
{
    //Se o resto da divisão por 7 não for 0 continua
    if(a%7!=0)continue;
    Prompt("O número "+a+" é divisível por 7.");
}
```

while

A instrução **while** implementa um loop que será executado enquanto a condição for verdadeira.

Sintaxe:

```
while( <condição> ) < instrução enquanto a condição for verdadeira >;
```

while do

O conjunto de instruções **do while** primeiro executa o bloco de instruções, só então avaliará a condição que caso seja verdadeira repetirá o loop.

Sintaxe:

```
do < instrução > while( <condição> );
```

Declarando funções

Funções são trechos de código autônomos, separados do programa principal que servem para realizar determinadas tarefas. Várias funções já são pré-definidas pela interface do ArchiStation e serão apresentadas neste manual mais adiante. Você poderá declarar e criar novas funções utilizando a seguinte sintaxe:

```
<tipo de retorno> <identificador> ( <parâmetros> )
{
    <instruções da função>
}
```

Função sem retorno de valor

Quando a função não retorna valores ela deve ser declarada como **void**.

Exemplo: Este script completo mostra a função principal **void main()** chamando a função **void mensagem()**, declarada logo a seguir no código e mostra a mensagem "Olá!".

```
void main()
{
    //Chama a função mensagem()
    mensagem();
}

//Declara a função mensagem()
void mensagem()
{
    Prompt("Olá!");
}
```


Função com passagem de parâmetros

Para receber um parâmetro, a função deve declarar o tipo e nome da variável que deverá receber o valor, se for mais de um parâmetro, deverão ser separados por vírgula.

Exemplo:

```
void main()
{
    //Chama a função mensagem()
    mensagem("Olá");
}

//Declara a função mensagem() com o parâmetro 'A' do tipo string
void mensagem(string A)
{
    Prompt("A mensagem é: "+A);
}
```

Função com retorno de valor

Para criar uma função que retorna valores, inicialmente devemos informar o tipo de dados que será retornado em sua declaração. Usamos então o comando **return** para retornar o valor. Toda vez que retornamos um valor a função é terminada e o processamento retorna ao ponto onde foi chamada. O valor retornado pode ou não ser aproveitado, de acordo com as necessidades apresentadas pelo contexto.

Exemplo: Neste exemplo, chamamos a função **Soma()**, declarada para retornar valores do tipo **double**, e atribuímos o valor retornado a variável 'A'.

```
void main()
{
    //Declara as variáveis A, B e C
    double A,B=10,C=25;

    //Chama a função Soma(), o valor retornado será atribuído a 'A'
    A = Soma(B,C);
    Prompt("A soma de B com C é igual a "+A+".");
}

//Declara a função Soma() que retornará um valor do tipo double
double Soma(double v1,double v2)
{
    // Retorna a soma dos parâmetros de entrada v1 e v2
    return (v1+v2);
}
```

A execução deste script deve mostrar no **Quadro de comando**:

A soma de B com C é igual a 35.
COMANDO:

Classes e objetos

Classes são dados estruturados que permitem agrupar diversas variáveis em único objeto. As classes devem ser declaradas no início do script, antes da função **void main()**, e a sintaxe é:

```
class <identificador>
{
    <lista de membros>
};
```

Exemplo: Cria uma classe chamada "MinhaData " com três variáveis do tipo **int**.

```
class MinhaData
{
    int Dia;
    int Mes;
    int Ano;
}
```

Declarar um **objeto** da classe "MinhaData" é semelhante a declaração de uma variável.

Exemplo: Declaramos o objeto "Inicio" da classe "MinhaData".

```
MinhaData Inicio;
```

Para acessar as variáveis da classe, utilizamos o identificador do objeto seguido pelo sinal ponto '.' e o identificador da variável:

```
Inicio.Dia=10;
Inicio.Mes=6;
Inicio.Ano=2011;
```

Também podemos associar funções as classes, que chamamos **métodos**.

Exemplo: Adicionamos na classe "MinhaData" o método **AdicionaMes()** que tem por objetivo adicionar um Mês a data. Se o limite de 12 meses (Dezembro) for ultrapassado, então incrementará a variável "Ano" e retornará "Mes" ao valor inicial 1 (Janeiro).

```
class MinhaData
{
    //Variáveis
    int Dia;
    int Mes;
    int Ano;

    //Métodos
    void AdicionaMes ()
    {
        Mes++;

        if (Mes>12)
        {
            Ano++;
            Mes=1;
        }
    }
}
```

Os métodos das classes são acessados de maneira semelhante as variáveis, com o indentificador do objeto seguido de sinal ponto '.' e o nome da função.

Exemplo: Agora podemos usar o método para adicionar um mês a data do objeto "Inicio", criado anteriormente.

```
Inicio.AdicionaMes();
```

Cadeias de caracteres

A classe **string** é utilizada para armazenar e manipular cadeias de caracteres.

Exemplo:

```
//Declara a string A e atribui o valor "ABC"  
string A="ABC";
```

Você poderá realizar operações de adição entre cadeias de caracteres.:

```
//Declara a string B e atribui o valor "DEF"  
string B="DEF";  
  
//Soma as strings A e B obtendo em C o valor "ABCDE"  
string C=A+B;
```

Adição com variáveis numéricas também é possível, sendo que seus valores são convertidos em cadeias de caracteres antes da operação.

```
int D=456;  
  
string E="123"+D;
```

Repare que, neste caso, o valor atribuído a string **E** é "**123456**" e não a soma dos valores numéricos.

Código ASCII

Cada caractere da cadeia possui um código ASCII. Para acessar o código ASCII de um caractere utilize a notação `string[n]`, onde **n** representa a posição do caractere na cadeia, contagem que é iniciada em 0:

Exemplo:

```
//Declara a string str e atribui o valor "Olá!"  
string str="Olá!";  
  
//Declara a variável numérica 'a'  
//e atribui o valor em ASCII do primeiro caractere de str  
int a=str[0];
```

Neste exemplo, a variável numérica **a** passa a armazenar o valor 79, que é o código ASCII para 'O', o primeiro caractere da string **str**.

Métodos associados

```
int string.length();
```

Retorna o tamanho da string, ou seja, o número total de caracteres.

Exemplo:

```
string str="ArchiStation";  
int a=str.length();
```

Onde a passa **a** armazenar o número de caracteres da string **str**.

```
string string.substr (int posição, int comprimento);
```

Retorna parte de uma cadeia de caracteres a partir da posição e comprimento especificados.

```
int string.findFirst (const string str);
```

```
int string.findFirst (const string str, int início);
```

Retorna a posição da primeira ocorrência da string **str** dentro da cadeia de caracteres. Se nenhuma ocorrência for encontrada retorna -1. Opcionalmente, use o parâmetro início para especificar a posição de início da busca.

```
int string.findLast (const string str);
```

```
int string.findLast (const string str, int início);
```

Retorna a posição da última ocorrência da string **str** dentro da cadeia de caracteres. Se nenhuma ocorrência for encontrada retorna -1. Opcionalmente, use o parâmetro início para especificar a posição de início da busca.

Funções relacionadas a strings

```
string Chr(int ASCII);
```

Retorna uma string contendo o caractere correspondente ao código ASCII especificado.

```
string strUpperCase(string str);
```

Retorna a string com todos os caracteres convertidos em maiúsculas.

```
string strLowerCase (string str);
```

Retorna a string com todos os caracteres convertidos em minúsculas.

Exemplos:

```
// Coloca em strA a palavra "ArchiStation" ( Chr(110) = "n" )  
string strA="ArchiStatio"+Chr(110);  
  
// Coloca em strB os caracteres de strA em maiúsculas ("ARCHISTATION")  
string strB=strUpperCase(strA);
```

CAPÍTULO 3

Definindo e manipulando pontos no espaço tridimensional

A classe point

A classe **point** é utilizada pelo ArchiStation para criar e manipular pontos no espaço tridimensional. Definida com as propriedades **x**, **y**, e **z** para armazenar os valores de coordenadas no plano cartesiano, e a propriedade **alt**, utilizada de modo particular pelo ArchiStation para associar um valor relativo a altura de cada ponto. Seus métodos permitem copiar, mover, girar, espelhar e alterar a escala de suas coordenadas.

Definição:

```
class point
{
    // Variáveis
    double x;
    double y;
    double z;
    double alt;

    //Métodos
    void Copy(point@ p);
    void Move(double X, double Y, double Z);
    void Scale(point@ base, double EscX, double EscY, double EscZ);
    void Rotate(point@ base, double Ângulo);
    void Rotate3D(point@ I1, point@ I2, double Ângulo);
    void Mirror(point@ I1, point@ I2);
}
```

Descrição dos métodos

```
void point.Copy(point@ p);
```

Torna os valores iguais ao ponto p.

```
void point.Move(double X, double Y, double Z);
```

Move o ponto as distâncias especificadas para cada eixo.

```
void point.Scale(point@ base, double EscX, double EscY, double EscZ);
```

Escala o ponto em relação ao ponto base os fatores especificados para cada eixo.

```
void point.Rotate(point@ base, double Ângulo);
```

Gira o ponto em torno do eixo Z, a partir do ponto base o ângulo especificado.

```
void point.Rotate3D(point@ I1, point@ I2, double Ângulo);
```

Gira o ponto em torno do eixo definido pelos pontos **I1** e **I2** o ângulo especificado.

```
void point.Mirror(point@ l1, point@ l2);
```

Espelha o ponto em relação a reta que passa por **l1** e **l2**.

Exemplo de como declarar um objeto **point** e atribuir valores as suas coordenadas:

```
//Declara o identificador p1 como objeto point
point p1;

//Atribui valores as coordenadas x, y, z, e alt.
p1.x=10;
p2.y=20;
p3.z=0;
p4.alt=270;
```

Você também poderá declarar um objeto **point** e atribuir valores na mesma instrução adicionando as coordenadas desejadas entre parênteses, separadas por vírgula. Os valores para as coordenadas **Z** e **alt** podem ser omitidos, neste caso considerados iguais a 0.

```
//Declara o ponto p1 e atribui valores para x e y
point p1(10,20);

//Declara o ponto p2 e atribui valores para x, y e z
point p2(10,20,30);

//Declara o ponto p3 e atribui valores para x e y, z e alt
point p3(10,20,30,270);
```

Exemplos de utilização dos métodos da classe **point** para manipulação das coordenadas de posição no espaço:

```
//Copia os valores das coordenadas (x,y,z e alt) do ponto p2 para o ponto
p1
p1.Copy(p2);

//Move o ponto p1 a distância indicada em cada um dos eixos X, Y e Z
p1.Move(10,20,30);
```

Quando necessário, os pontos podem ser definidos sem que seja necessário declarar um identificador, apenas indicando **point** e os valores das coordenadas especificadas entre parênteses separadas por vírgula. Exemplo:

```
//Gira o ponto p1 45 graus em torno do eixo definido na posição 50,50
p1.Rotate( point(50,50), 45);
```

CAPÍTULO 4

Funções para Desenhar

Neste capítulo serão apresentados os comandos que podem ser utilizados para adicionar novos objetos ao desenho na área de edição do ArchiStation.

Criando um linha

Para desenhar uma linha entre dois pontos, utilizamos a função **Line()**.

```
void Line(point@ p1, point@ p2);
```

A linha será desenhada na camada e espessura correntes, conforme especificado na barra de formatação. A altura da linha será definida pelo valor **alt** dos pontos indicados.

Exemplo: Cria uma linha entre os pontos p1 e p2.

```
Line (p1, p2);
```

Os pontos, objetos da classe **point**, podem ser definidos dentro dos parâmetros da função, sem que seja necessário declarar identificadores:

Exemplo: Cria uma linha entre coordenadas (0, 0, 0) e as coordenadas (100, 100, 0).

```
Line ( point(0, 0, 0) , point(100, 100) );
```

Na definição dos pontos também podemos utilizar variáveis, expressões matemáticas, coordenadas de pontos existentes e até aproveitar o valor retornado por chamadas a outras funções:

```
Line(point(p1.x, p1.y, z) , point(p2.x+100, p2.y+50 , CalculaZ(p2.z) ) );
```

Outras funções

```
void Circle(point@ Centro, double Raio);
```

Desenha um círculo com centro e raio indicados.

```
void Arc(point@ Centro, double Raio, double ÂnguloInicial, double ÂnguloFinal);
```

Desenha um arco definido por centro, raio, ângulos inicial e final.

```
void Ellipse(point@ Centro, double RaioX, double RaioY, double ÂnguloInicial, double ÂnguloFinal, double ÂnguloRotacao);
```

Cria uma elipse definida por Centro, Raios no eixo X e Y, ângulos inicial, final e de rotação.

```
void Face(point@ p1, point@ p2, point@ p3, point@ p4);
```

Desenha uma face definida pelos quatro pontos indicados.

```
void Face(point@ p1, point@ p2, point@ p3, point@ p4, bool edge1, bool edge2, bool edge3, bool edge4);
```

Desenha uma face definida pelos quatro pontos indicados, sendo que as variáveis **edge** definem quais das arestas serão visíveis ou invisíveis.

```
bool Face(point@ p1, point@ p2, point@ p3, point@ p4,
bool edge1, bool edge2, bool edge3, bool edge4,
double u1, double v1, double u2, double v2, double u3, double v3, double u4, double v4);
```

Desenha uma face definida pelos quatro pontos indicados, sendo que as variáveis **edge** definem quais das arestas serão visíveis ou invisíveis, e associa as coordenadas de mapeamento **uv** de textura u1, v1, u2, v2, u3, v3, u4 e v4 para cada um dos pontos. As coordenadas de mapeamento padrão são (0,0), (0,1), (1,1) e (1,0).

```
void Text(point@ Posicao, string Texto, double Tamanho, double Rotacao);
```

Desenha um texto na posição, tamanho e rotação indicados.

Alterando as propriedades de desenho

Camadas

Para adicionar um nova camada ao desenho utilize a função **AddLayer()**.

```
void AddLayer(string Nome, int Cor, string LType);
```

Cria uma camada com nome, cor e estilo de traço especificados.

LType é o estilo de traço.

```
void AddLayer(string Nome, int Cor, string LType, string Material);
```

Cria uma camada com nome, cor, estilo de traço e material especificados (ASX 2.0).

Se o material especificado não for encontrado no desenho principal, será procurado no script ASX e adicionado ao desenho se encontrado.

Para tornar corrente uma camada use a função **SetLayer()**.

```
bool SetLayer(string Nome);
```

Se a camada não existir ou não for possível a tornar corrente, a função retorna **false**.

Exemplo: Torna corrente a camada "Cerca", se ela não existir, adiciona a nova camada.

```
if(!SetLayer("Cerca"))
{
    // Cria a camada "Cerca" na cor Verde
    AddLayer("Cerca", RGB(0,255,0) , "");
    SetLayer("Cerca");
}
```


Ajustando a espessura e a altura

Para especificar a espessura e altura correntes utilizamos as funções **SetWidth()** e **SetHeight()**:

```
bool SetWidth(double Espessura);  
Torna corrente a espessura indicada.
```

```
bool SetHeight(double Altura);  
Torna corrente a altura indicada.
```

Exemplo:

```
// Ajusta a espessura corrente para 10 unidades  
SetWidth(15);  
  
// Ajusta a altura corrente para 60 unidades  
SetHeight(60);
```

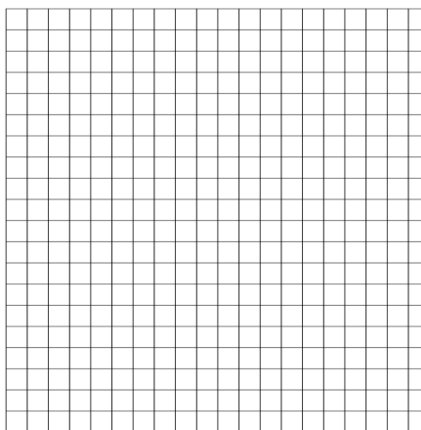
Desenhando objetos com algoritmos e expressões matemáticas

Você poderá utilizar scripts que combinam algoritmos, expressões matemáticas e funções de desenho para criar objetos mais complexos.

1º Exemplo - Grade

Podemos utilizar o comando de repetição **for()** e a função **Line()** para criar uma grade.

```
void main()  
{  
    //Cria as linhas no sentido do eixo Y  
    for(int x=0; x<=1000; x=x+50)  
        Line( point( x, 0), point(x,1000) );  
  
    //Cria as linhas no sentido do eixo X  
    for(int y=0; y<=1000; y=y+50)  
        Line( point( 0, y), point(1000,y) );  
}
```



Grade de linhas 1000 x 1000 unidades com espaçamento 50 unidades criada pelo exemplo acima.

2º Exemplo - Relógio

Você poderá utilizar a função **Circle()** para desenhar o contorno do relógio.

```
Circle( point(0,0), 100 );
```

Mas para desenhar os marcadores das horas? Funções matemáticas podem ser empregadas para realizar estes cálculos. Uma volta completa em um círculo tem 360 graus, ou $2 \times \pi$ em radianos. No relógio, os números estão posicionados a cada 30 graus, ou $\pi / 6$ radianos.

Vamos criar uma comando **for()** para fazer a variável **angulo** percorrer os valores de 0 a 360 graus, saltando 30 graus por vez.

```
for(int angulo=0; angulo<360; angulo=angulo+30)
```

Para cada valor de **angulo** utilizamos a função **radians()** para converter de graus para radianos e utilizamos as funções de trigonometria **sin()** e **cos()** para calcular o valor de "seno" e "cosseno" do angulo, utilizados para se calcular a posição de **x** e **y** de cada hora.

```
double x= cos( radians(angulo) );
double y= sin( radians(angulo) );
```

Por fim, multiplicamos os valores de obtidos de **x** e **y** pelo raio do círculo que desejamos e criamos as linhas de marcação das horas com a função **Line()**.

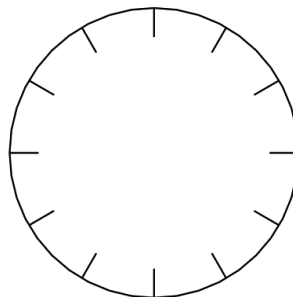
```
Line( point(x*80, y*80), point(x*100, y*100));
```

Código completo do exemplo:

```
void main()
{
    Circle( point(0,0), 100);

    for(int angulo=0; angulo<360; angulo=angulo+30)
    {
        double x=cos( radians(angulo) );
        double y=sin( radians(angulo) );

        Line( point(x*80, y*80), point(x*100, y*100));
    }
}
```



Relógio desenhado com operações matemáticas seno e cosseno.

Observação: Divisões por zero interrompem a execução do script. Antes de realizar uma divisão, procure sempre verificar se o denominador é diferente de 0.

CAPÍTULO 5

Obtendo dados do usuário

Neste capítulo serão descritos os comandos que permitem interagir com o usuário, obtendo dados como pontos, distâncias, valores numéricos ou cadeias de caracteres (**strings**).

Fazendo com que o usuário indique um ponto

Uma das principais funções de entrada de dados é a **GetPoint()**, que interrompe a execução do script e aguarda até que o usuário especifique ponto na tela ou digite suas coordenadas no **Quadro de comando**. O ponto é então armazenado em um objeto do tipo **point** indicado.

```
bool GetPoint(point@ in);
```

Exemplo: O script pede ao usuário que indique um ponto e cria um círculo com raio de 25 unidades no local indicado.

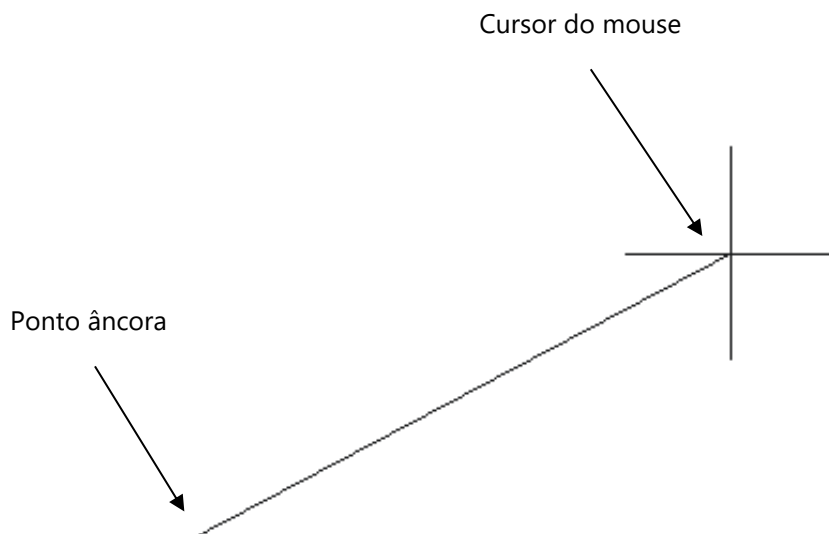
```
void main()
{
    // Declara p1 como objeto point
    point p1;

    // Escreve a mensagem no Quadro de comando
    Prompt("Especifique um ponto:");

    // Solicita que o usuário entre com um ponto e armazena em p1
    GetPoint(p1);

    // Cria um círculo com centro em p1 e raio 25
    Circle(p1, 25);
}
```

Outra variação da função **GetPoint()** permite que o usuário especifique um ponto mantendo o cursor do mouse ancorado a um ponto especificado.



```
bool GetPoint(point@ in, point@ ancora);
```

Exemplo:

```
void main()
{
    // Declara p1 e p2 como objetos point
    point p1,p2;

    Prompt("Especifique o primeiro ponto:");

    // Solicita o ponto p1
    GetPoint(p1);

    Prompt("Especifique o segundo ponto:");

    // Solicita o ponto p2, mantendo o cursor ancorado a p1
    GetPoint(p2,p1);

    // Cria um círculo com centro em p1
    // e raio definido pela distância a entre p1 e p2
    Circle(p1, Distance(p1,p2) );
}
```

No exemplo foi utilizada a função **Distance()** para calcular a distância entre dois pontos indicados e determinar o raio para criar o círculo.

Obtendo números inteiros

A função **GetInt()** interrompe a execução do script até que o usuário entre com um número inteiro através do **Quadro de comando**. Se a entrada digitada pelo usuário for incorreta, a mensagem ***Entrada descartada*** será exibida e o sistema torna a aguardar até que um valor satisfatório seja digitado.

```
void GetInt(int &in);
    Solicita a entrada de um número inteiro.
```

Exemplo:

```
// Declara uma variável numérica inteira do tipo int
int i;

// Escreve a mensagem e solicita ao usuário a entrada de um valor inteiro
Prompt("Entre com um número inteiro:");
GetInt(i);
```

Obtendo números reais

A função **GetReal()** é semelhante a **GetInt()**, mas obtém valores reais.

```
void GetReal(double &in);
    Solicita a entrada de um número real.
```

Exemplo:

```
//Declara uma variável numérica de precisão double
double N;

// Escreve a mensagem e solicita ao usuário a entrada de um valor real
Prompt("Entre com um número real:");
GetReal(N);
```

Obtendo cadeias de caracteres

A função **GetString()** pode ser utilizada para obter uma cadeia de caracteres do usuário.

```
bool GetString(string &in);  
    Solicita a entrada de uma cadeia de caracteres.
```

Exemplo:

```
//Declara uma variável string  
string S;  
  
// Escreve a mensagem e solicita ao usuário a entrada de uma string  
Prompt("Entre com uma cadeia de caracteres:");  
GetString(S);  
  
// Mostra o que o usuário digitou  
Prompt("Voê digitou: "+S);
```

Outras funções de entrada de dados

```
bool GetCorner(point@ in, point@ p1);  
    Solicita a entrada de um ponto usando como referência a área retangular criada a partir do ponto p1.
```

```
void GetDist(double &in);  
    Solicita a entrada de uma distância. Se o usuário indicar um ponto no desenho, será solicitada a entrada de mais um ponto e retornada a distância entre os dois.
```

```
void GetDist(double &in, point@ p1);  
    Solicita a entrada de uma distância. Se o usuário indicar um ponto no desenho, será retornada a distância deste ponto ao ponto p1.
```

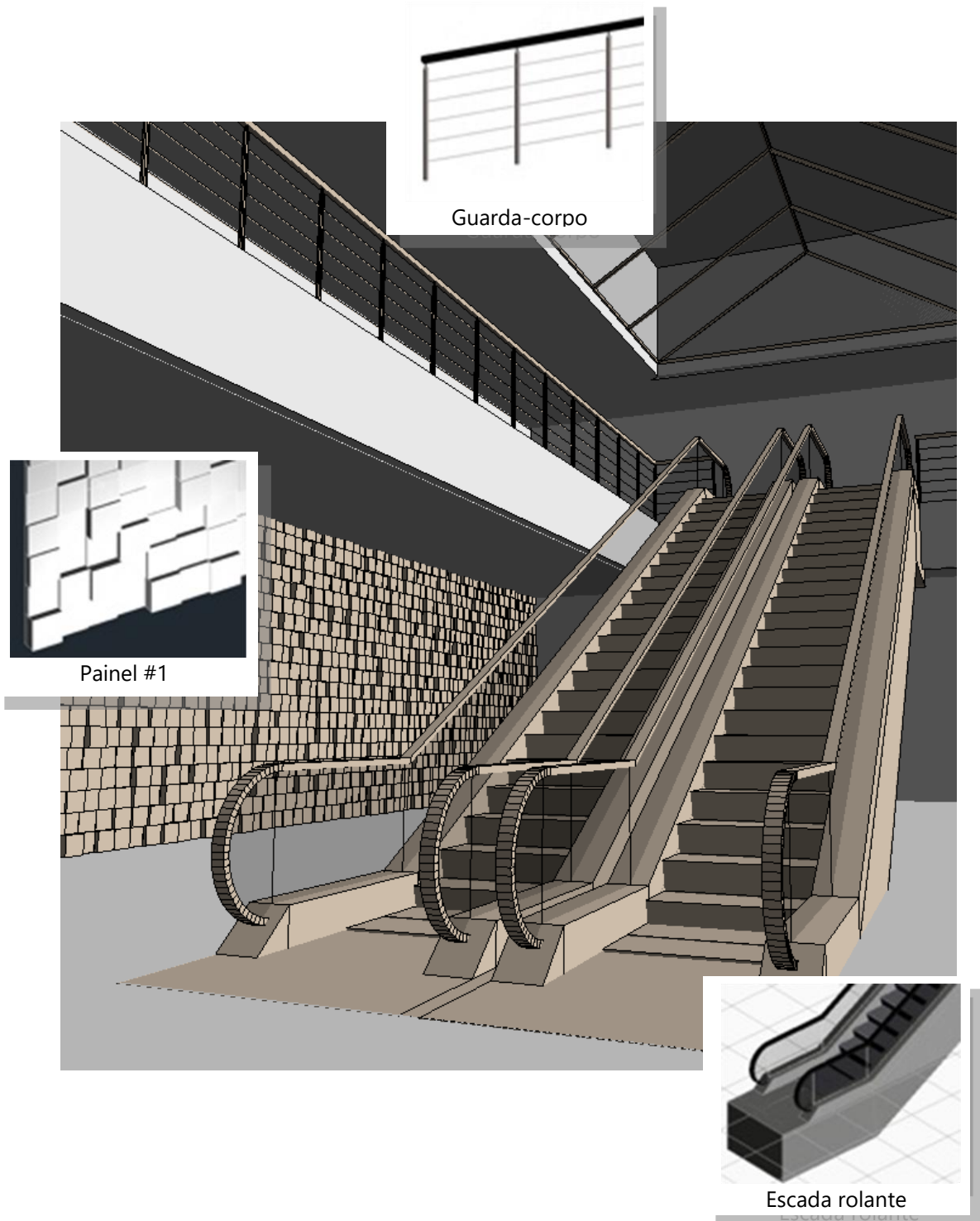
```
void GetAngle(double &in);  
    Solicita a entrada de um valor angular. Se o usuário indicar um ponto no desenho, será solicitada a entrada de mais um ponto e retornado o ângulo do segmento formado com os dois pontos.
```

```
void GetAngle(double &in, point@ p1);  
    Solicita a entrada de um valor angular. Se o usuário indicar um ponto no desenho, será retornado o ângulo do segmento formado com o ponto indicado e o ponto p1.
```

CAPÍTULO 6

Componentes ASX

A criação de **Componentes ASX** é uma das mais interessantes e produtivas formas de se utilizar a tecnologia ASX. São componentes inteligentes, desenhados a partir de scripts, que podem mudar de forma de acordo com os ajustes em suas propriedades. A imagem capa deste tutorial teve o emprego amplo destes componentes:

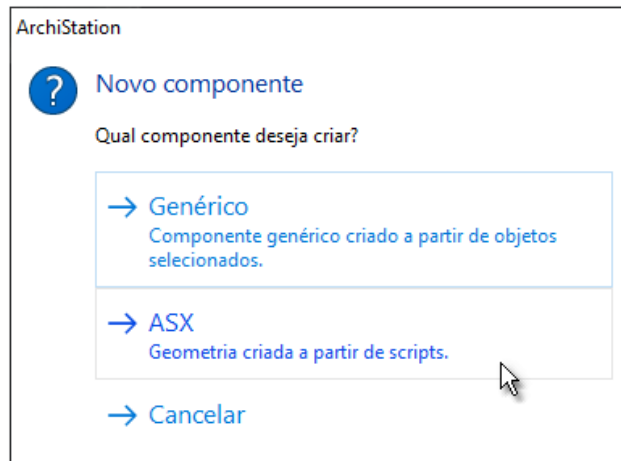


Agora que você já sabe um pouco sobre a linguagem de programação, algumas funções para desenhar e como obter dados do usuário, está pronto para criar seu primeiro **Componente ASX**.

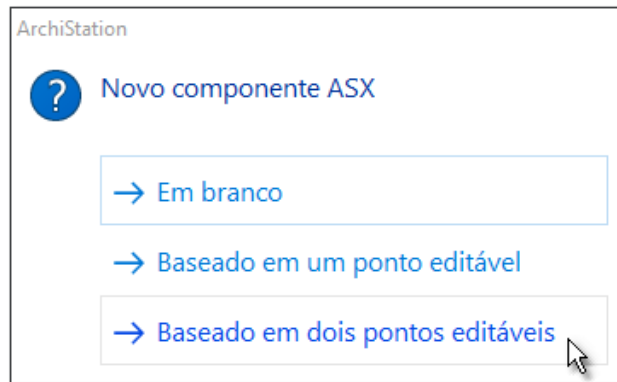
Criando um Componente ASX

1. No **Quadro Lateral <F11>** selecione a guia **Componentes**.

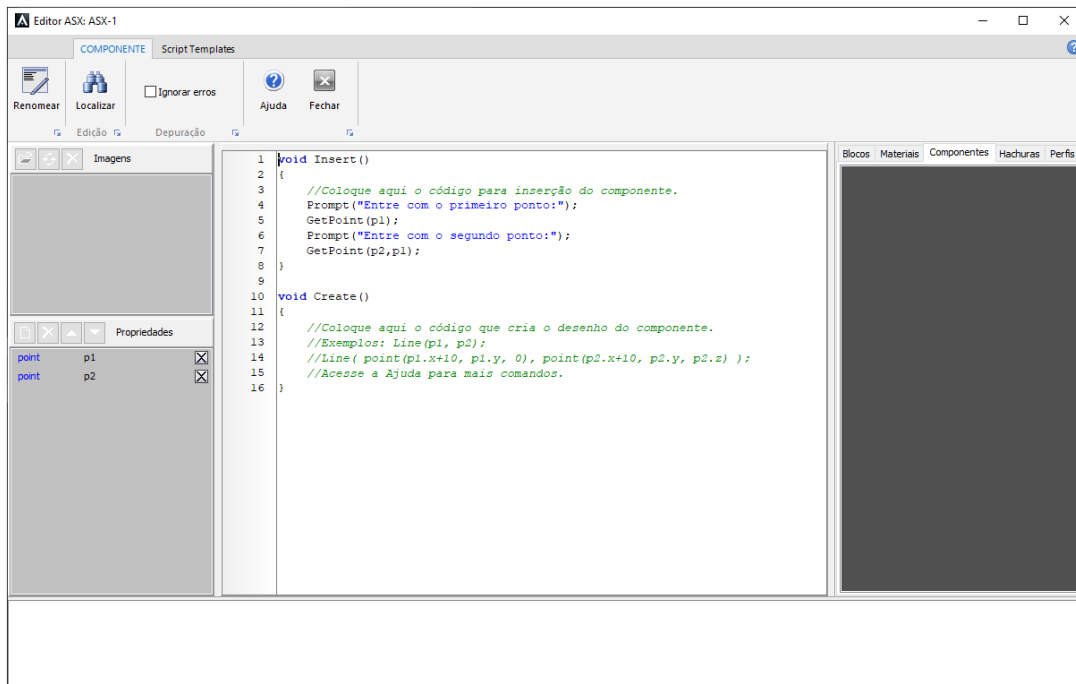
2. Clique sobre o ícone **Criar um novo componente** .



3. Na caixa de diálogo selecione a opção **ASX - Baseado em Script**.



4. A caixa de diálogo seguinte conterá opções de modelos de script que podem ser utilizados como base para seu componente. Para este exemplo optamos por selecionar a opção **Baseado em dois pontos editáveis**.



Editor ASX

O modelo escolhido cria automaticamente nas propriedades do componente os pontos **p1** e **p2** que podem ser acessados pelo script, e declara duas funções essenciais para o funcionamento do componente. A função **Insert()**, chamada para administrar a coleta de informações do usuário ao inserir o componente no desenho, e a função **Create()**, chamada pelo sistema quando for necessário criar ou recriar a sua geometria.

5. Altere o nome do componente modificando na caixa **Nome do componente** o texto de "**ASX-1**" para "**Linha ASX**".
6. Edite a função **Create()** para adicionar as instruções mostradas no quadro a seguir:

```

void Create()
{
    Line(p1, p2);
}
  
```

7. Pressione o botão **Fechar** para encerrar a edição do script.

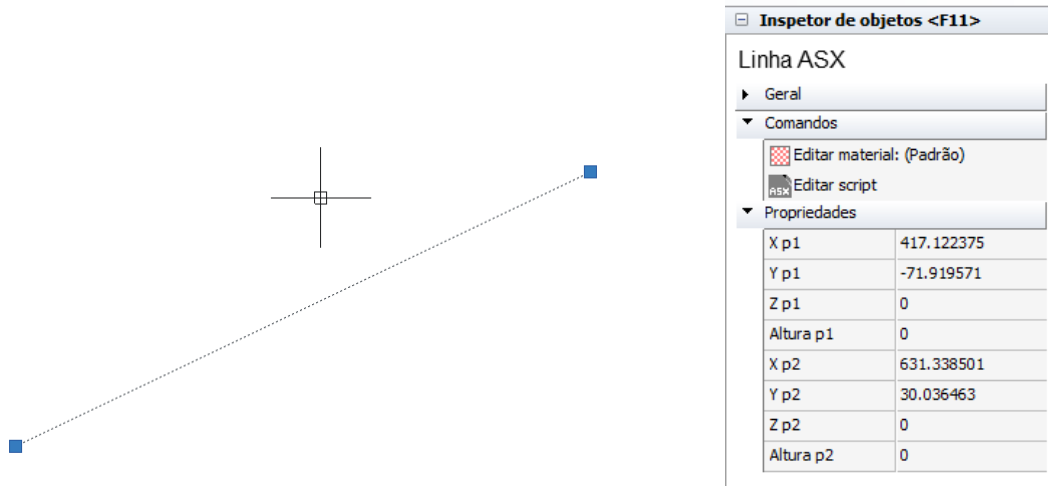
Pronto! Você criou seu primeiro componente ASX. Ele desenha apenas uma linha entre os dois pontos indicados pelo usuário. Mas já é um bom começo. Para inserir-lo selecione o componente **Linha ASX** no gerenciador de componentes do **Quadro Lateral** e arraste para a tela de edição.

Ao inserir o componente no desenho a função **Insert()** é chamada. Repare que as frases que aparecem no **Quadro de comando** enquanto indica os pontos de inserção do componente são as das instruções **Prompt()** declaradas no código da função **Insert()**. Você poderá modificar as instruções desta função para coletar os dados de acordo com as propriedades que seu componente necessitar.

Logo depois que indicar os dois pontos a função termina, e então a função **Create()** é chamada para criar a geometria do componente, que simplesmente usa a instrução **Line()** para desenhar a linha entre os pontos **p1** e **p2** declarados nas propriedades do componente como objetos da classe **point**.

Editando as propriedades

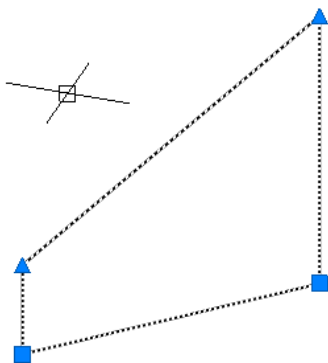
As propriedades dos componentes declaradas como **point** são administradas pelo ArchiStation como **pontos editáveis** pelo usuário, ou seja, ao selecionar o componente os pontos serão realçados e, ao clicar sobre um deles, poderá indicar visualmente uma nova posição no desenho para mover o ponto ou teclar **<ESC>** para cancelar o movimento.



Componente "Linha ASX" inserido no desenho e selecionado, pontos editáveis realçados em azul.

Ao selecionar individualmente o componente e pressionar tecla **<F11>** para exibir o **Inspecor de objetos**, as propriedades do componente serão relacionadas associadas aos seus respectivos valores.


Qualquer alteração nos valores fará com que a função **Create()** seja chamada novamente para recriar a geometria do componente. Nestes casos, o ArchiStation armazena e gerencia os dados para possibilitar as operações **Desfazer...** e **Refazer...** caso o usuário deseje desfazer a alteração nos valores das propriedades.

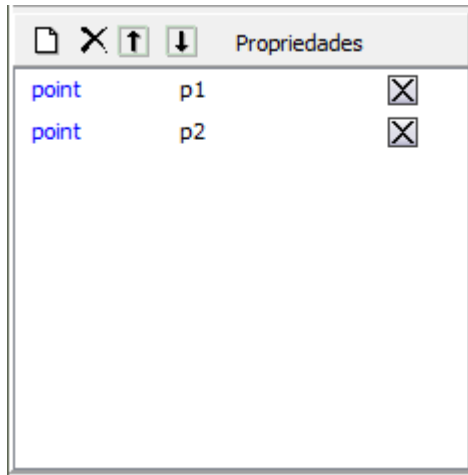


Observe que particularmente no ArchiStation, cada ponto possui as coordenadas **x, y, z** e ainda a coordenada de altura **alt**. Alterando a propriedade **alt** do ponto para um valor diferente de 0, a linha passa a ser desenhada com altura atribuída, e os **pontos editáveis de altura** (triângulos em azul), também serão realçados. A edição destes pontos altera apenas a propriedade de altura **alt** do ponto para o valor que ajuste ao nível indicado, sem afetar **x, y**, ou **z**, ou seja, o ponto permanece fixo na posição.

Adicionando novas propriedades

Para adicionar ou remover propriedades de um componente ASX, ou mesmo editar os nomes desta propriedades, não poderá haver nenhuma instância dele inserida no desenho. Então selecione e pressione a tecla **<Delete>** para apagar qualquer componente do tipo **"Linha ASX"** que criamos anteriormente que esteja inserido no desenho.


Agora, no gerenciador de componentes do Quadro lateral, selecione o componente **"Linha ASX"** e pressione o botão **Propriedades do componente**  para acessar novamente o **Editor ASX**.



No quadro a esquerda do Editor você verá a guia **Propriedades**, e logo abaixo a relação das propriedades declaradas no **Editor ASX** para o componente.

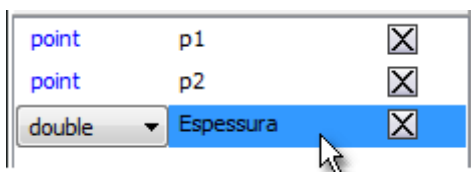
A primeira coluna mostra o **tipo** da propriedade, a segunda coluna mostra seu identificador, ou seja, o nome que associa a propriedade ao código do script.

Se o quadro de seleção na última coluna estiver marcado, indica que a propriedade é visível e pode ser editada pelo usuário através do **Inspetor de objetos**.

Pressione o primeiro botão **Criar uma nova propriedade**  para adicionar uma propriedade ao componente. A nova propriedade deverá aparecer no final da lista.

Selecione a propriedade recém criada. Uma caixa deve ser mostrada sobre o **tipo** de propriedade. Clique sobre a caixa e selecione o tipo **double**.

Clique agora sobre o identificador da propriedade, na coluna central. Com o texto em edição, altere para **Espessura**.



Pronto! Declaramos uma nova propriedade chamada **Espessura**. Vamos agora fazer uso dela no código.

Para inicializar a propriedade quando o componente for inserido no desenho, adicione a seguinte linha no final da função **Insert()**:

```
Espessura=15;
```

Agora, na função **Create()**, adicione a instrução **SetWidth()** para ajustar a espessura de desenho:

```
SetWidth( Espessura );
```

O código todo deve ficar assim:

```
void Insert()
{
    Prompt("Entre com o primeiro ponto:");
    GetPoint(p1);
    Prompt("Entre com o segundo ponto:");
    GetPoint(p2,p1);

    //Inicializa a propriedade espessura
    Espessura=15;
}

void Create()
{
    //Ajusta a espessura
    SetWidth( Espessura);

    //Desenha a linha
    Line(p1, p2);
}
```

Agora, ao inserir o componente no desenho, a **Linha ASX** será desenhada com 15 unidades de espessura, valor com que a propriedade **Espessura** foi inicializada. Ao selecionar o componente e acessar o **Inspetor de objetos <F11>** a propriedade **Espessura** que acabamos de criar estará disponível para a edição do usuário. Qualquer ajuste no valor desta propriedade será correspondido prontamente no desenho do componente.

Controlando os valores das propriedades

No exemplo, a propriedade **Espessura** foi inicializada com o valor de 15 unidades e pode assumir posteriormente qualquer valor que o usuário venha a especificar. Para impor limites aos valores que uma propriedade pode atingir você poderá adicionar restrições no código do script.

Por exemplo, supondo que desejamos criar um componente que permita o valor máximo para **Espessura** de 50 unidades, então, colocamos uma linha para verificar e ajustar seu valor antes que seja utilizada pelo script, ou seja, antes da instrução **SetWidth()**:



```
// Se a Espessura for maior que 50, ajusta para 50
if( Espessura>50 ) Espessura=50;
```

Assim, se o usuário tentar ajustar a **Espessura** para um valor superior a 50, ela assumirá o valor 50 antes que o componente seja totalmente recriado.

Organizando a disposição das propriedades



Alguns componentes podem possuir muitas propriedades de diferentes tipos, e cabe ao desenvolvedor organizar as informações para torná-las claras ao usuário dos componentes. A primeira coisa que pode ser feita é a ordenação da lista.

A ordem que as propriedades são mostradas no **Editor ASX** será a mesma visualizada posteriormente no **Inspetor de objetos** quando o componente for selecionado. Você poderá


alterar esta ordem selecionando a propriedade e clicando sobre o botão  para ascender uma posição na ordem ou  para descer uma posição na ordem.

Outra forma de manter os dados organizados é criando grupos de propriedades separadas por categorias. Para isso, você deve adicionar propriedades do tipo **separator**. Este tipo de propriedade não pode ser acessada pelo código, mas surge como uma guia visível no **Inspetor de objetos** quando o componente é selecionado agrupando todas as propriedades incluídas na seqüência da lista, até um novo **separator** ou o fim da lista.

Linha ASX

▶ Geral	
▼ Comandos	
	Editar material: (Padrão)
	Editar script
▼ Propriedades	
X p1	721
Y p1	392
Z p1	0
Altura p1	0
X p2	1217
Y p2	602
Z p2	0
Altura p2	0
▼ Geometria	
Espessura	0

separator "Geometria"

Por fim, se uma propriedade não é mais necessária, selecione e clique em  para excluir a propriedade do componente.

Descrição e nomes com acento no Inspetor de objetos

Como em C/C++ não são permitidos acentos nos nomes de identificadores, se você desejar que nomes com acentos, espaços ou outros caracteres especiais sejam mostrados na indicação da propriedade no **Inspetor de objetos**, você deverá utilizar a descrição da propriedade.

No **Editor ASX**, clique sobre o identificador da propriedade com o botão direito do mouse e adicione a descrição desejada para a propriedade. Esse nome, quando preenchido, substituirá o identificador da propriedade nos controles visíveis ao usuário.

Descrição da variável 'Espessura' nos controles:

Caixa de diálogo para edição da descrição.

Outros tipos de propriedades

A tabela a seguir mostra os 11 tipos possíveis de propriedades:

Tipo	Descrição
bool	Valor booleano, pode ser true ou false . No Inspetor aparecem como Sim ou Não .
int	Valor numérico inteiro.
double	Valor numérico de precisão double.
string	Cadeia de caracteres.
point	Ponto com coordenadas x, y, z e alt. Se for visível, passa a ser selecionável.
separator	Utilizado apenas para separar os dados quando relacionados no Inspetor de objetos .
information	É uma cadeia de caracteres acessível da mesma forma que o tipo string, mas o usuário não consegue alterar seu valor através do Inspetor de objetos . Serve apenas para informar o usuário sobre alguma circunstância do componente.
intList	Lista de números inteiros. O funcionamento das listas será explicado adiante.
doubleList	Lista de valores de precisão double.
stringList	Lista de valores strings.
pointList	Lista de pontos.

Exemplos de componentes

Agora que você já aprendeu sobre o funcionamento dos **Componentes ASX**, e já sabe criar e controlar suas propriedades, está pronto para criar os mais variados tipos de componentes que poderão agilizar suas tarefas e aprimorar seus projetos.

Incluímos aqui códigos de componentes e algumas explicações que poderão servir de base para o desenvolvimento de seus próprios componentes.

Cerca - função PointAt();

Tabela de propriedades:

Tipo	Identificador
point	p1
point	p2

```

void Insert()
{
    Prompt("Entre com o primeiro ponto:");
    GetPoint(p1);
    Prompt("Entre com o segundo ponto:");
    GetPoint(p2,p1);

    //Ajusta a altura da cerca
    p1.alt=150;
    p2.alt=150;
}

void Create()
{
    //Variáveis
    double Tabua=10, Espaco=3;

    //Ajusta a espessura da cerca
    SetWidth(3);

```

```

//Calcula a distância entre p1 e p2
double Distancia=Distance(p1,p2);

//Declara os pontos que serão usados no cálculo
point ini,fim;

//Percorre a distância entre p1 e p2 colocando as tábuas
for(double Posicao=0; Posicao<Distancia-Tabua; Posicao+=(Tabua+Espaco)
)
{
    // Calcula os pontos de início e fim da tábua
    PointAt(ini, p1, p2, Posicao);
    PointAt(fim, p1, p2, Posicao+Tabua);

    //Desenha a tábua
    Line( ini, fim);
}
}

```

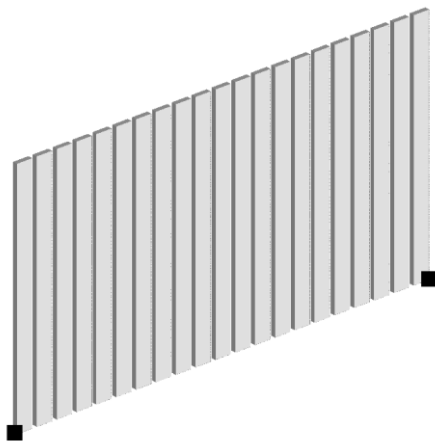
A função **PointAt()** é uma das instruções mais importantes deste código, e uma das mais utilizadas no desenvolvimento de **Componentes ASX**. Sua definição é:

```
void PointAt(point@ p, point@ p1, point@ p2, double Distancia);
```

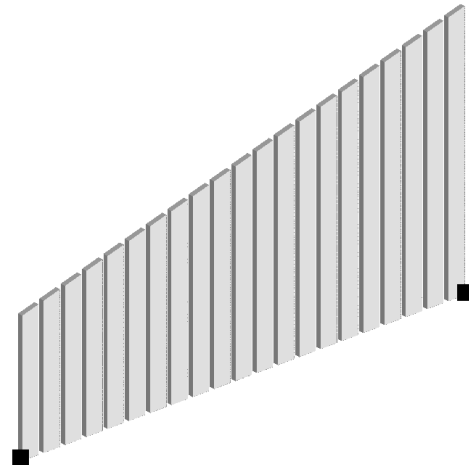
Ela atualiza o ponto **p** para a posição a contada a partir de **p1**, percorrida a distância indicada em direção a **p2**. Considerando no calculo, inclusive, as diferenças na cota **Z** e as variações da altura **alt** dos pontos.

Neste caso, utilizamos a função **PointAt()** para calcularmos o ponto de início de cada tábua, usando como distância de referência a **Posição**, e também o ponto onde cada tábua termina, usando como distância a **Posição + Largura da Tábua**.

A cada tábua desenhada, incrementamos a **Posição** com a **Largura da Tábua** e o **Espaço** entre elas.



Cerca com 150 unidades de altura em p1 e p2



Cerca com alturas diferentes em cada ponto

Como a função **PointAt()** considera até diferenças de altura entre os pontos em seu calculo, você poderá ajustar alturas diferentes para cada ponto do componente inserido através das propriedades no **Inspetor de objetos** e observar os resultados.

Dica: Você pode aprimorar este componente removendo a declaração das variáveis **Tabua** e **Espaco**, e adicionando-as como propriedades de tipo **double**, editáveis pelo usuário. Não esqueça de inicializar seus valores na função **Insert()**.

Escada

Tabela de propriedades:

Tipo	Identificador
point	p1
point	p2
double	Degrau_Altura
double	Degrau_Largura

```

void Insert()
{
    Prompt("Entre com o primeiro ponto:");
    GetPoint(p1);
    Prompt("Entre com o segundo ponto:");
    GetPoint(p2,p1);

    //Inicializa as propriedades
    Degrau_Altura=16;
    Degrau_Largura=31;
}

void Create()
{
    //Ajusta p2 para o mesmo nível de p1
    p2.z=p1.z;

    //Calcula a distância entre p1 e p2
    double Distancia=Distance(p1,p2);

    //Largura da escada
    SetWidth(120);

    point ini,fim;

    //Altura inicial
    double Z=Degrau_Altura;

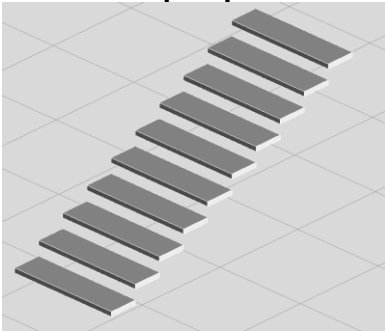
    //Percorre a distância entre p1 e p2 desenhando os degraus
    for(double Posicao=0; Posicao<Distancia;
        Posicao=Posicao+Degrau_Largura)
    {
        //Calcula os pontos de início e fim do degrau
        PointAt(ini, p1, p2, Posicao);
        PointAt(fim, p1, p2, Posicao+Degrau_Largura);

        //Desenha o degrau
        Line( point(ini.x, ini.y, p1.z+Z-5, 5),
            point(fim.x, fim.y, p1.z+Z-5, 5));

        //Incremento da altura
        Z=Z+Degrau_Altura;
    }
}

```

Neste exemplo, devemos criar além das propriedades **point p1** e **p2**, as propriedades do tipo **double Degrau_Largura** e **Degrau_Altura**. O número de degraus será obtido em função da distância entre **p1** e **p2**.



Para desenhar os degraus da escada é seguido o mesmo princípio utilizado para desenhar as tábuas da cerca no exemplo anterior, utilizando a função **PointAt()** e uma posição de referência para cada degrau.

A diferença é basicamente o desenho do degrau utilizando coordenadas definidas por **point** para especificar a coordenada **Z** com o mesmo valor, e a altura fixa de 5 unidades para o elemento do degrau.

Espiral - senos e cossenos

Tabela de propriedades:

Tipo	Identificador
point	p1
point	p2

```

void Insert()
{
    Prompt("Entre com o primeiro ponto:");
    GetPoint(p1);
    Prompt("Entre com o segundo ponto:");
    GetPoint(p2,p1);
}

void Create()
{
    //Calcula a distância entre p1 e p2, sem considerar as coordenadas Z
    double Distancia=Distance( point(p1.x,p1.y), point(p2.x,p2.y) );

    //Inicializa o Raio em 0
    double Raio=0;

    //Calcula o acréscimo que deverá ser somado ao raio
    //para 4 voltas (cada volta tem 360 graus)
    double Acrescimo=Distancia/(360*4);

    //Obtém o angulo de saída (em radianos)
    double Angulo=Angle(p1,p2);

    point ini,fim;

    //Percorre as quatro voltas de ângulo em ângulo
    for(double a=0; a<=360*4; a++)
    {
        //Calcula a posição com as funções seno e cosseno
        fim.x=cos( radians(a)+Angulo ) *Raio+p1.x;
        fim.y=sin( radians(a)+Angulo ) *Raio+p1.y;

        //Se já possui os valores ini e fim desenha a linha
        if(a>0) Line(ini,fim);

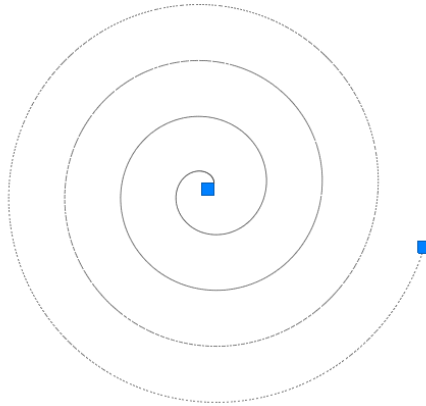
        //Copia as coordenadas do ponto fim no ponto ini
        ini.Copy(fim);

        //Soma o acréscimo para aumentar o raio a cada ângulo
        Raio=Raio+Acrescimo;
    }
}

```



```
}
```



Este componente desenha uma espiral, com centro em **p1**, com 4 voltas terminando exatamente sobre **p2**. Primeiramente é calculada a distância entre **p1** e **p2** para obter o raio final da espiral. O valor inicial do Raio é 0 e será incrementado a cada ângulo pelo valor de **Acréscimo**. O ângulo inicial, entre os pontos **p1** e **p2**, é obtido com a função **Angle()**, que retorna o valor em radianos. Este ângulo vai nos permitir chegar ao ponto final da espiral exatamente sobre o ponto **p2**.

A função **for()** inicia o loop que fará o valor de **a** passar de 0 até 360×4 , ou 1440, o total angular para 4 voltas completas. As funções de trigonometria de **seno** e **coosseno** são então aplicadas para se obter a posição de cada ponto da espiral.

Na realidade, se removêssemos do código a soma do acréscimo e especificássemos um valor para o **Raio** constante, obteríamos um círculo perfeito. Lembrando que nesse caso, seria necessário apenas uma volta, ou 360 graus.

CAPÍTULO 7

Programando a interface

Depois de criar código para novos comandos, com certeza você desejará criar atalhos para acessá-los com mais rapidez, como botões em barras de ferramentas ou um item no menu principal do ArchiStation. Este capítulo descreve as funções que permitem a incorporação de recursos customizáveis a interface do programa.

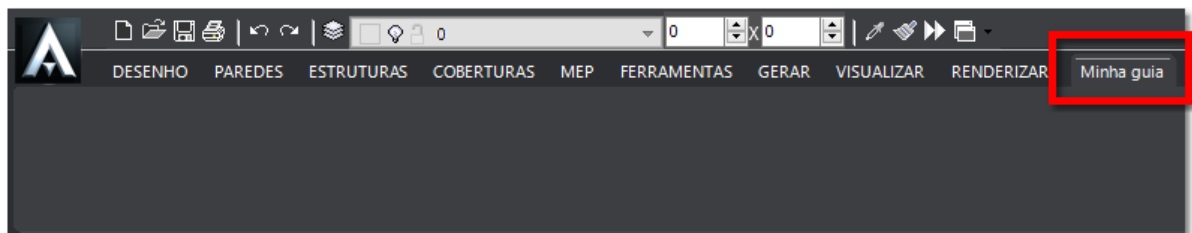
Adicionando Guias ao Menu principal

Você poderá adicionar uma nova **Guia** ao Menu principal com a função **CreateToolBarPage()** que retornará um objeto da classe **ToolBarPage@**.

```
ToolBarPage@ CreateToolBarPage(string Título);
```

Exemplo:

```
ToolBarPage@ Page = CreateToolBarPage("Minha guia");
```



Guia "Minha guia" criada com o comando acima

Menu de ferramentas

Para criar um novo menu de ferramentas chamamos a função **CreateToolBar()** que retornará um objeto da classe **ToolBar@**.

```
ToolBar@ ToolBarPage@.CreateToolBar(string Título, string Ajuda);
```

Exemplo: O código declara o objeto **Bar** como **ToolBar@** e cria um novo menu de ferramentas com o título "Meus comandos". O texto **Ajuda** deve aparecer para o usuário quando o cursor do mouse for deixado sobre o menu.

Exemplo:

```
ToolBar@ Bar = Page.AddToolBar("Meus comandos", "Ajuda do menu Meus comandos");
```

Botões com ícones nos menus

Para adicionar botões a um menu de ferramentas, utilizamos a função **ToolBar.AddButton()**:

```
ToolBar.AddButton(string Título, int Imagem, string Comando, string Ajuda);
```

Adiciona um botão a barra de ferramentas.

Título	Nome associado ao botão
Imagem	Um valor inteiro (int) que deve indicar o número da imagem na lista de imagens do container ASX . A imagem poderá ser de qualquer tamanho, mas para manter o padrão da interface recomendamos que utilize preferencialmente imagens no tamanho 32 x 32 pixels .

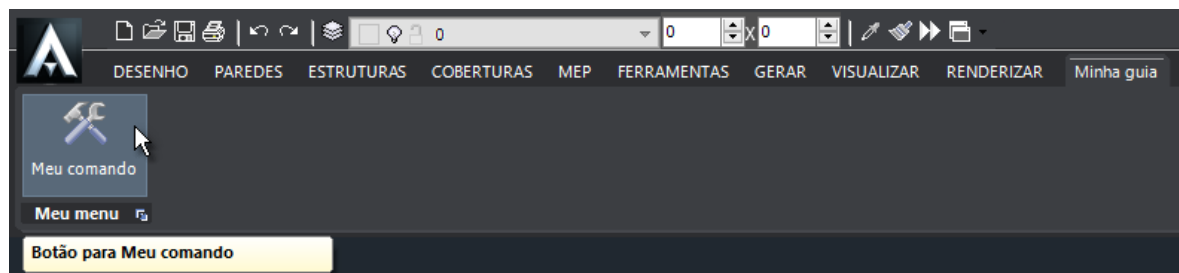
Comando	Comando que deve ser chamado ao se clicar o botão.
----------------	--

O comando poderá ser:

- Um comando do ArchiStation, exemplo: "**linha**"
- Chamada a uma função definida no próprio script. O nome da função deve ser precedido por um duplo sinal de dois pontos ("::"). Exemplo: "**::funcao()**"
- Chamada a uma função de outro script localizado na pasta "/Plugins" da instalação do ArchiStation. Neste caso deve ser indicado o nome do arquivo seguido do sinal "::" e o nome da função a ser chamada. Exemplo: "**arquivo::funcao()**"

Observação: As funções chamadas devem ser definidas como "**void**".

Ajuda	Texto que aparece para o usuário quando mantém o cursor do mouse sobre o botão por alguns instantes.
--------------	--



Guia "Minha guia" com a adição do menu "Meu menu" e o botão "Meu comando"

Exemplo: Segue o código completo para criação de uma nova guia com menu e um botão que chama uma função dentro do script.

```
void main()
{
    // Cria a guia "Minha guia" se não existir
    ToolbarPage@ Page = CreateToolbarPage("Minha guia");

    // Adiciona o menu "Meu menu" se não existir
    Toolbar@ Bar = Page.AddToolbar("Meu menu", "Ajuda para meu menu");

    // Adiciona um botão com a imagem obtida na lista de imagens que ao
    // clicado chamará a função Ola() definida no próprio script
    Bar.AddButton("Meu comando", 0, ":::Ola", "Botão para Meu comando");
}

// Definição da função Ola que será chama quando clicar sobre o botão
void Ola()
{
    // Escreve Olá! na barra de comando
    Prompt("Olá!");
}
}
```

Janelas de propriedades

Você poderá criar uma nova janela de diálogo com o usuário usando a função **CreateDialog()** que retornará um objeto da classe **Dialog@**.

```
Dialog@ Janela = CreateDialog(string Título, string Pannel);
```

Cria uma janela de diálogo para exibir e obter informações do usuário.
O comando já adiciona o primeiro painel retrátil.

Controles que podem ser adicionados a janela de diálogo:

```
void Dialog.AddPanel(string Titulo)
```

Adiciona um painel retrátil a janela.

```
void Dialog.AddString(string Titulo, const string &in)
```

Adiciona uma caixa de edição de string.

```
void Dialog.AddInt(string Titulo, const int &in)
```

Adiciona uma caixa de edição de valor inteiro (**int**).

```
void Dialog.AddDouble(string Titulo, const double &in)
```

Adiciona uma caixa de edição de de valor real (**double**).

```
void Dialog.AddDistance(string Titulo, const double &in)
```

Adiciona uma caixa de edição de distância.

```
void Dialog.AddLabel(string Titulo)
```

Adiciona um texto de informação ao usuário.

Depois de criar a janela de diálogo, utilize a função **Show()** para mostrar ao usuário e que ele possa interagir com ela.

```
void Dialog.Show()
```

Apresenta a caixa de dialogo ao usuário.

CAPÍTULO 8

Tratamento de arquivos de dados

Para ler ou escrever dados de um arquivo, os seguintes procedimentos devem ser realizados:

- Abrir o arquivo para leitura ou escrita;
- Ler ou escrever os dados;
- Fechar o arquivo.

Os objetos da classe **file** são responsáveis pelo tratamento de arquivos de dados. Os métodos associados a classe **file** são:

```
int file.open(string filename, string mode);  
Abre um arquivo para leitura, escrita ou edição.
```

Parâmetros:

filename: Nome do arquivo

mode: "r" - Abre o arquivo em modo leitura;

"w" - Abre o arquivo para escrita. Sobrescreve o arquivo existente se houver;

"a" - Abre o arquivo para adicionar dados.

```
int file.close();  
Fecha o arquivo.
```

```
int file.getSize();  
Retorna o tamanho do arquivo em bytes.
```

```
bool file.isEndOfFile();  
Retorna true se o final do arquivo foi atingido.
```

Operações de leitura do arquivo

```
int file.readString(uint length, string &str);  
Lê um número específico de bytes para a string str.
```

```
int file.readLine(string &str);  
Lê todo o conteúdo até a próxima linha de caracteres.
```

```
int64 file.readInt(uint bytes);  
Lê um número inteiro com sinal com o tamanho especificado em bytes.
```

```
uint64 file.readUInt(uint bytes);  
Lê um número inteiro sem sinal com o tamanho especificado em bytes.
```

```
float file.readFloat();  
Lê um número de precisão float.
```

```
double file.readDouble();  
Lê um número de precisão double.
```

Operações de escrita do arquivo

```
int file.writeString(string in);
```

Escreve uma cadeia de caracteres **string** no arquivo.

```
int file.writeInt(int64 v, uint bytes);
```

Escreve um número inteiro com sinal com o tamanho especificado em *bytes*.

```
int file.writeUInt(uint64 v, uint bytes);
```

Escreve um número inteiro sem sinal com o tamanho especificado em *bytes*..

```
int file.writeFloat(float v);
```

Escreve um número de precisão **float** no arquivo.

```
int file.writeDouble(double v);
```

Escreve um número de precisão **double** no arquivo.

Manipulação do cursor do arquivo

```
int file.getPos();
```

Retorna a posição do manipulador de leitura e escrita no arquivo.

```
int file.setPos(int pos);
```

Ajusta a posição do manipulador de leitura e escrita do arquivo.

```
int file.movePos(int delta);
```

Faz um deslocamento de *delta* bytes na posição do manipulador de leitura e escrita.

Observação: As funções de tratamento de arquivos não estão disponíveis na versão de demonstração.

Exemplo: Script básico que faz a leitura do arquivo file.txt.

```
void main()
{
    // Declara o objeto file
    file f;

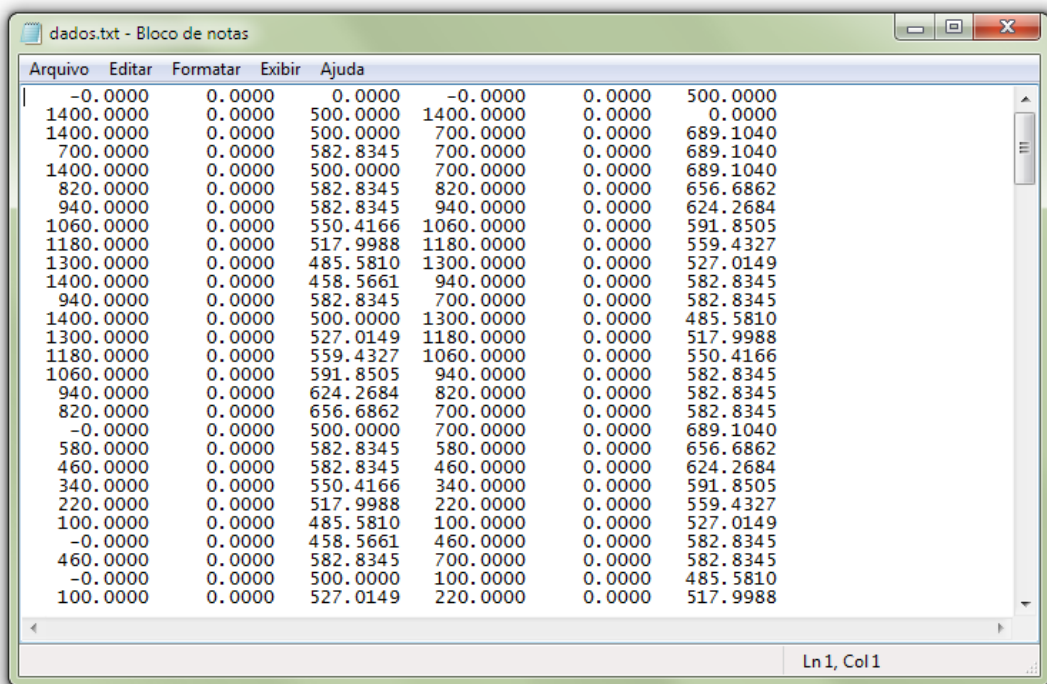
    // Declara a string str
    string str;

    // Abre o arquivo em modo de leitura "r"
    if( f.open("file.txt", "r") >= 0 )
    {
        // Lê todo arquivo para a string str
        f.readString(f.getSize(), str);

        // Fecha o arquivo
        f.close();
    }
}
```

Para um exemplo prático de acesso a arquivos de dados desenvolvemos um script que lê o arquivo "dados.txt", disponível na pasta Plugins da instalação do ArchiStation, que contém coordenadas de pontos que serão utilizadas para a criação de uma estrutura metálica tridimensional composta por linhas. As informações do arquivo dados.txt estão organizadas no seguinte formato:

- Cada linha do texto contém informações para criação de uma linha no desenho;
- As informações estão organizadas em dois conjuntos de três números que representam as coordenadas dos pontos da linha, sendo x_1, y_1, z_1 e x_2, y_2 e z_2 ;
- x_1 = começa na posição 0 e ocupa 10 dígitos (string inicia pelo caractere na posição 0);
- y_1 = começa na posição 11 e ocupa 10 dígitos;
- z_1 = começa na posição 22 e ocupa 10 dígitos;
- x_2 = começa na posição 33 e ocupa 10 dígitos;
- y_2 = começa na posição 44 e ocupa 10 dígitos;
- z_2 = começa na posição 55 e ocupa 10 dígitos.



Arquivo dados.txt

Arquivo **IO.asx** disponível na pasta Plugins do ArchiStation:

```
void main()
{
    //Solicita ao usuário para indicar um arquivo com extensão *.txt
    string Arquivo=GetFile("", "", "txt");

    //Declara o objeto file
    file f;

    // Abre o arquivo em modo de leitura
    if( f.open(Arquivo, "r") >= 0 )
    {
        //Declara as variáveis que serão utilizadas
        string str;
        double x1,y1,z1,x2,y2,z2;

        //Inicia um loop infinito
        for(;;)
        {
```



```

//Le uma linha do arquivo
f.readLine(str);

//Se o final do arquivo foi atingido encerra o loop
if( f.isEndOfFile() )break;

//Obtem os caracteres de cada coordenada com o método substr();
//e converte em números (double) com a função atof();
x1=atof( str.substr(0,10) );
y1=atof( str.substr(11,10) );
z1=atof( str.substr(22,10) );

x2=atof( str.substr(33,10) );
y2=atof( str.substr(44,10) );
z2=atof( str.substr(55,10) );


//Cria a linha com os dados dos pontos obtidos do arquivo
Line( point(x1,y1,z1), point(x2,y2,z2) );
}

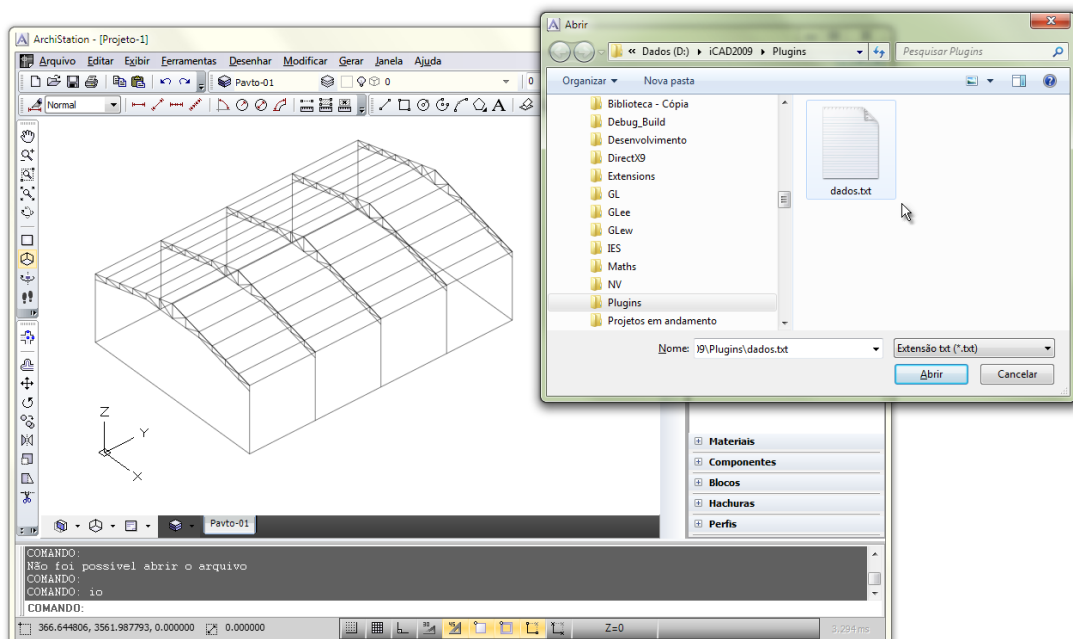
//Fecha o arquivo
f.close();
}
else
{
    //Mensagem no caso de ocorrer erro na tentativa de abrir o arquivo
    Prompt("Não foi possível abrir o arquivo");
}
}

```

Você pode executar este script digitando **IO** e teclando <ENTER> no **Quadro de comando** do ArchiStation. A função **GetFile()**, no início do script, faz com que uma caixa de diálogo de seleção de arquivos apareça, indique o arquivo **dados.txt** e clique em **Abrir**.

Os dados em formato de cadeias de caracteres são convertidos para valores numéricos pela função **atof()**, que converte strings em números de precisão **double**.

Logo após o término da execução do script, selecione o modo de **Edição em Vista 3D**  e a estrutura metálica deve ser visualizada na tela.



Estrutura metálica obtida a partir da leitura do arquivo dados.txt

CAPÍTULO 9

Obtendo dados de planilhas do MS Excel

Utilizando a tecnologia *OLE Automation* o ArchiStation pode se conectar a planilhas do MS Excel e obter dados que podem ser utilizados para desenhar ou complementar informações de componentes do desenho.

Os procedimentos devem ser realizados nesta ordem:

- Abrir o arquivo do Excel;
- Selecionar a planilha que vai ser utilizada;
- Obter o valor das células desejadas;
- Fechar o arquivo do Excel;

Para cada um dos procedimentos listados acima existe uma função, descritas a seguir:

```
bool ExcelOpenFile(string Arquivo);
```

Abre um arquivo do Excel para manipulação. O MS Excel deve estar instalado no seu computador. Se o nome do arquivo não especificar a pasta, o ArchiStation procurará o arquivo na pasta **Plugins**. Retorna **true** se obteve êxito, caso contrário retorna **false**.

```
bool ExcelSetSheet(string Sheet);
```

Seleciona qual planilha do arquivo será manipulada. Retorna **true** se obteve êxito, caso contrário retorna **false**.

```
string ExcelGetItem(int Col, int Line);
```

Retorna uma string com o valor do item na coluna e linha indicados.

```
bool ExcelCloseFile();
```

Fecha o arquivo. Retorna **true** se obteve êxito, caso contrário retorna **false**.

Exemplo - Arquivo **Excel.aspx** disponível na pasta Plugins do ArchiStation:

```

void main()
{
    //Solicita ao usuário para indicar um arquivo com extensão *.xls
    string Arquivo=GetFile("Localizar o arquivo
'dados.xls'",PluginsDir(),"xls");

    //Abre o arquivo do Excel
    if(ExcelOpenFile(Arquivo))
    {
        //Seleciona a planilha que será manipulada
        if(ExcelSetSheet("Plan1"))
        {
            //Declara a string que vai receber o valor dos itens
            double x1,y1,z1,x2,y2,z2;

            //Pega os valores da linha 1 até a linha 177
            for(int Linha=1; Linha<=177; Linha++)
            {
                //Obtém os caracteres de cada célula com a função
                //string ExcelGetItem(int Coluna, int Linha);
                //e converte em números (double) com a função atof();
                x1=atof( ExcelGetItem(1,Linha) );
                y1=atof( ExcelGetItem(2,Linha) );
                z1=atof( ExcelGetItem(3,Linha) );

                x2=atof( ExcelGetItem(4,Linha) );
                y2=atof( ExcelGetItem(5,Linha) );
                z2=atof( ExcelGetItem(6,Linha) );

                //Cria a linha com os dados dos pontos obtidos
                Line( point(x1,y1,z1), point(x2,y2,z2) );
            }
        }
        else
        {
            MessageBox(
                "A planilha 'Plan1' não existe no arquivo.", "", "ArchiStation",5);
        }

        //Fecha o arquivo
        ExcelCloseFile();
    }
    else
    {
        MessageBox("Não foi possível abrir o arquivo.", "", "ArchiStation",5);
    }
}

```

Neste exemplo, ao executar o script a função **GetFile()** exibe a caixa de diálogo de seleção de arquivo. Selecione o arquivos **dados.xls** e clique em **Abrir**. Este arquivo contém os mesmos valores do arquivo **dados.txt** do exemplo do capítulo anterior, mas com os valores das coordenadas organizados da coluna 1 a coluna 6, e da linha 1 até a linha 177.

A conexão com o Excel é iniciada e o arquivo é aberto com a função **ExcelOpenFile()**. A planilha "Plan1" é selecionada com a função **ExcelSetSheet()**. As informações das células são obtidas com a função **ExcelGetItem()** e convertidas para números de precisão **double** com a função **atof()**. Os dados são então utilizados para criar as linhas da estrutura até a linha 177. Por fim a função **ExcelCloseFile()** é chamada para fechar o arquivo e encerrar a conexão com o Excel.

A função **MessageBox()** exibirá uma caixa de diálogo no caso de ocorrer algum problema na leitura do arquivo. Esta função está descrita detalhadamente no **Apêndice A**, no final deste documento.

CAPÍTULO 10

Listas de armazenamento

O ArchiStation permite criar quatro tipos de listas de itens para armazenamento de dados, são elas:

- **intList** - Lista de valores inteiros `int`;
- **doubleList** - Lista de valores reais de precisão `double`;
- **stringList** - Lista de Cadeias de caracteres do tipo `string`;
- **pointList** - Lista de objetos da classe `point`.

Para utilizar uma lista de armazenamento, o primeiro passo é a declaração do tipo de lista e seu identificador:

```
//Declara uma lista para valores inteiros
intList Lista;
```

Para adicionar itens na lista utilize o método **Add(tipo) Valor** ou o método **Insert(int Índice, (tipo) Valor)**. O índice do primeiro item armazenado na lista será 0, e índice do último item da lista será o número total de itens da lista menos 1.

```
//Adiciona itens na lista
Lista.Add(10);
Lista.Add(30);

//Inserir o item de valor 20 na posição 1, no meio da lista
Lista.Insert(1,20);
```

Para saber quantos itens estão armazenados na lista utilize o método **Count()**, e para recuperar seus valores utilize o método **GetItem(int Índice)**.

```
//Percorre todos os valores da lista
for(int a=0; a<Lista.Count(); a++)
{
    //Mostra os valores da lista no Quadro de comando
    Prompt("O item na posição "+a+" é igual a "+Lista.GetItem(a)+".");
}
```

Com a execução deste script obterá como resultado a seguinte mensagem no **Quadro de comando**:

```
O item na posição 0 é igual a 10.
O item na posição 1 é igual a 20.
O item na posição 2 é igual a 30.
```

Observação: Os valores armazenados nas listas de componentes ASX inseridos no desenho declaradas através da interface do **Editor ASX** como propriedades do componente, serão salvos automaticamente no arquivo do projeto.

Os valores da lista podem ser alterados com a função **SetItem()**:

```
Lista.SetItem(1,15);
```

Definindo o item selecionado

Você poderá definir qual item da lista está selecionado através dos métodos **GetItemIndex()** e **SetItemIndex()**.

As listas podem aparecer como propriedades editáveis no **Inspetor de Objetos** quando o componente é selecionado. O item selecionado será mostrado em uma caixa de seleção. Caso o usuário clicar sobre a caixa, todos os itens da lista serão mostrados. Se o usuário escolher um item diferente, o valor do **ItemIndex** será alterado.

O **ItemIndex** ajustado para o valor **-1** indica que nenhum item da lista está selecionado, e a caixa de seleção aparecerá em branco.

Métodos genéricos associados às listas

```
void (tipo)List.Add((tipo)Item);
```

Adiciona um item na lista.

```
void (tipo)List.Insert(int index, (tipo)Item);
```

Insere um item na lista na posição especificada.

```
(tipo) (tipo)List.GetItem(int index);
```

Retorna o valor de um item da lista.

```
void (tipo)List.SetItem(int index, (tipo)Valor );
```

Define o valor de um item da lista.

```
int (tipo)List.Count();
```

Retorna quantos itens a lista possui armazenados.

```
void (tipo)List.Move(int ÍndiceAtual, int NovoÍndice);
```

Troca a posição de um item na lista.

```
void (tipo)List.Delete(int index);
```

Apaga um item da lista.

```
void (tipo)List.Clear();
```

Apaga todos os itens da lista.

```
int (tipo)List.GetItemIndex();
```

Retorna o índice selecionado da lista. Se nenhum item estiver selecionado retorna -1.

```
void (tipo)List.SetItemIndex(int index);
```

Define o índice selecionado da lista.

Métodos especiais da classe **stringList**

```
void stringList.Sort ();
```

Ordena alfabeticamente os itens da lista.

```
int stringList.IndexOf(string Valor);
```

Retorna o índice do primeiro item da lista que possuir o valor especificado. Se o valor não for encontrado na lista, retorna -1.

Exemplo:

```
void main()
{
    //Declara uma lista de strings
    stringList Lista;

    //Adiciona alguns pontos na lista
    Lista.Add("Projetos");
    Lista.Add("Tridimensionais");
    Lista.Add("ArchiStation");

    //Coloca os itens da lista em ordem alfabética
    Lista.Sort();

    //Mostra a posição do item "ArchiStation"
    Prompt("A posição de 'ArchiStation' na lista é "+
        Lista.IndexOf("ArchiStation")+".");
}
```

Este script cria uma lista de strings, adiciona alguns valores, chama a função **Sort()** para ordenar os itens e usa a função **IndexOf()** para retornar a nova posição do item "ArchiStation". Ao executar este código, você deve obter no Quadro de comando a seguinte mensagem:

A posição de 'ArchiStation' na lista é 0.

Métodos especiais da classe pointList

point& pointList.Item(int index);

Retorna o objeto armazenado na posição indicada. Diferente da função **GetItem()**, que retorna uma cópia do objeto armazenado, a função **Item()** retorna o próprio objeto armazenado. Sendo que, qualquer alteração nos valores do objeto retornado por esta função afetará diretamente os valores armazenados na lista, não sendo necessário chamar **SetItem()** para esta função.

Exemplo:

```
void main()
{
    //Declara a lista de pontos
    pointList ListadePontos;

    //Adiciona alguns pontos na lista
    ListadePontos.Add( point(10,10,0) );
    ListadePontos.Add( point(20,20,0) );
    ListadePontos.Add( point(30,30,0) );

    //Percorre todos os itens da lista
    for(int a=0; a<ListadePontos.Count(); a++)
    {

        //Declara um ponteiro para um objeto da classe point
        //e aponta para o objeto da lista
        point @p=ListadePontos.Item(a);

        //Move o ponto
        p.Move(10,10,0);
    }
}
```

Este script cria uma lista de pontos, adiciona alguns valores e movimenta todos os pontos da lista 10 unidades no eixo X e 10 unidades no eixo Y.

CAPÍTULO 11

Adicionando materiais, blocos, perfis e hachuras

É possível adicionar coleções de materiais, blocos, perfis e hachuras a um script ASX. Estes elementos podem ser utilizados no código para adicionar novos estilos e elementos ao desenho.

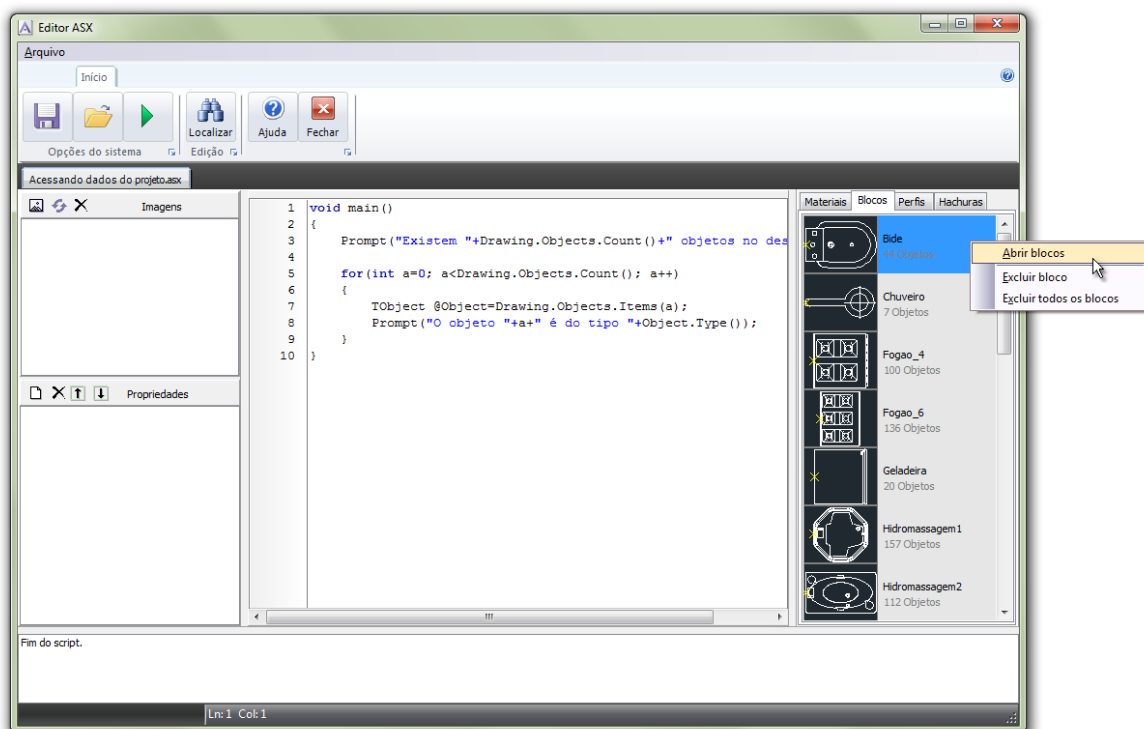
Adicionando materiais

► Para adicionar uma coleção de blocos ao script ASX

1. No quadro a direita do código do script clique sobre a guia **Blocos**.
2. Clique como botão direito do mouse sobre o quadro.
3. No menu, selecione a opção **"Abrir blocos"**.
4. Selecione a pasta e o arquivo de blocos que deseja adicionar e pressione **OK**.

Os materiais podem ser utilizados na criação de novas camadas.

Adicionando blocos



Editor ASX – Inserindo uma coleção de blocos

► Para adicionar uma coleção de blocos ao script ASX

1. No quadro a direita do código do script clique sobre a guia **Blocos**.
2. Clique com o botão direito do mouse sobre o quadro.
3. No menu, selecione a opção **"Abrir blocos"**.
4. Selecione a pasta e o arquivo de blocos que deseja adicionar e pressione **OK**.

Funções para inserir blocos

```
bool Insert(string Nome, point@ Inserção, double Rotação);
```

Inserir um bloco no desenho com a escala e rotação indicados.

```
bool Insert(string Nome, point@ Inserção, double Escala, double Rotação);
```

Inserir um bloco no desenho com a escala e rotação indicados.

```
bool Insert(string Nome, point@ Inserção, double EscX, , double EscY, double EscZ, double Rotação);
```

Inserir um bloco no desenho com escalas diferentes para cada um dos eixos axiais.

```
bool Insert(string Nome, point@ Inserção, double EscX, , double EscY, double EscZ, double Rotação, point@ VetorDeExtrusão);
```

Inserir um bloco no desenho com escalas diferentes para cada um dos eixos axiais.

Ao inserir um bloco no desenho com um dos comandos relacionados acima, caso a definição de bloco indicada pelo nome não existir no desenho, ela será procurada na relação de blocos do script ASX e inserida no desenho. Se a definição de bloco indicada não for encontrada, o bloco não será inserido e a função retornará **false**.

Adicionando perfis

► Para adicionar uma coleção de perfis ao script ASX

1. No quadro a direita do código do script clique sobre a guia **Perfis**.
2. Clique com o botão direito do mouse sobre o quadro.
3. No menu, selecione a opção **"Abrir perfis"**.
4. Selecione a pasta e o arquivo de perfis que deseja adicionar e pressione **OK**.

Funções para criar perfis (Mudar para Shaped)

```
bool Perfilado(string Perfil, pointList@ ListaDePontos);
```

Cria um perfilado com o perfil indicado através da lista de pontos.

```
bool Perfilado(string Perfil, pointList@ ListaDePontos, double Escala);
```

Cria um perfilado com o perfil indicado através da lista de pontos na escala indicada.

```
bool Perfilado_Orto(string Perfil, pointList@ ListaDePontos);
```

Cria um perfilado ortogonal com o perfil indicado através da lista de pontos.

```
bool Perfilado_Orto (string Perfil, pointList@ ListaDePontos, double Escala);
```

Cria um perfilado ortogonal com o perfil indicado através da lista de pontos na escala indicada.

Observação: Perfilados ortogonais sempre mantêm o perfil de extrusão 90 graus em relação ao chão, giram apenas pelo eixo Z, variando a direção nos eixos X e Y. Veja no **Guia do usuário** mais sobre os perfis ortogonais.

Adicionando hachuras

► Para adicionar uma coleção de hachuras ao script ASX

1. No quadro a direita do código do script clique sobre a guia **Hachuras**.
2. Clique com o botão direito do mouse sobre o quadro.
3. No menu, selecione a opção "**Abrir hachuras**".
4. Selecione a pasta e o arquivo de hachuras que deseja adicionar e pressione **OK**.

Funções para criar hachuras

```
bool Hatch(string Hachura, pointList@ ListaDePontos);
```

Cria uma região hachurada na região definida pela lista de pontos.

```
bool Hatch(string Hachura, pointList@ ListaDePontos, double Escala, double Rotação);
```

Cria uma região hachurada com a escala e rotação indicadas na região definida pela lista de pontos.

CAPÍTULO 12

Acessando os objetos do projeto

Todos os dados do projeto são organizados em elementos da classe **TDrawing**.

Definição da classe TDrawing:

```
class TDrawing
{
    TList Objects; // Lista de objetos do projeto (Pavimento em edição)
    TList Layers; // Lista de camadas
    TList Materials; // Lista de materiais
    TList Blocks; // Lista de definições de bloco
    TList Hatches; // Lista de definições de padrões de hachuras
    TList Shapedes; // Lista de definições de perfis
}
```

O projeto em edição fica armazenado no objeto **Drawing** e seus objetos são armazenados na lista **Drawing.Objects** em elementos da classe **TObject**.

Definição da classe TObject:

```
class TObject
{
    //Retorna uma string com o nome da classe do objeto
    string ClassName();

    //Métodos para modificar objetos
    void Copy(point@ p);
    void Move(double X, double Y, double Z);
    void Scale(point@ base, double EscX, double EscY, double EscZ);
    void Rotate(point@ base, double Ângulo);
    void Rotate3D(point@ I1, point@ I2, double Ângulo);
    void Mirror(point@ I1, point@ I2);

    //Métodos para obter valores das propriedades
    bool GetBoolean(string Propriedade);
    bool SetBoolean(string Propriedade, bool Valor);
    int GetInt(string Propriedade);
    bool SetInt(string Propriedade, int Valor);
    double GetDouble(string Propriedade);

    //Métodos para alteras valores das propriedades
    bool SetDouble(string Propriedade, double Valor);
    string GetString(string Propriedade);
    bool SetString(string Propriedade, string Valor);
    bool GetPoint(string Propriedade, point@ p);
    bool SetPoint(string Propriedade, point@ p);
}
```

Segue um exemplo mostrando como pode ser feito o acesso a lista de objetos do projeto **Drawing.Objects**:

```
void main()
{
    //Exibe quantos objetos existem no projeto
    Prompt("Existem "+Drawing.Objects.Count()+" objetos no desenho.");

    //Pega objeto por objeto
    for(int a=0; a<Drawing.Objects.Count(); a++)
    {
        //Pega o objeto da lista
        TObject @Object=Drawing.Objects.Items(a);

        //Exibe o tipo de objeto
        Prompt("O objeto "+a+" é do tipo "+Object.ClassName());
    }
}
```

O método **ClassName()** utilizado no código retorna uma string com o tipo de objeto. Os tipos mais comuns de objetos são: **LINE, CIRCLE, ARC, ELLIPSE, TEXT, INSERT, HATCH, SHAPED** e **ASX**.

Métodos para modificar objetos

```
void TObject.Move(double X, double Y, double Z);
```

Move o objeto o valor indicado para cada eixo.

```
void TObject.Scale(point@ base, double EscX, double EscY, double EscZ);
```

Escala o objeto em relação ao ponto base com os fatores especificados para cada eixo.

```
void TObject.Rotate(point@ base, double Ângulo);
```

Gira o objeto em torno do eixo Z, a partir do ponto base o ângulo especificado.

```
void TObject.Rotate3D(point@ I1, point@ I2, double Ângulo);
```

Gira o objeto em torno do eixo definido pelos pontos **I1** e **I2** o ângulo especificado.

```
void TObject.Mirror(point@ I1, point@ I2);
```

Espelha o objeto em relação a reta que passa por **I1** e **I2**.

Exemplo de código que move todos os objetos do projeto:

```
void main()
{
    //Pega objeto por objeto
    for(int a=0; a<Drawing.Objects.Count(); a++)
    {
        //Pega o objeto da lista
        TObject @Object=Drawing.Objects.Items(a);

        //Move o objeto
        Object.Move(100,0,0);
    }
}
```

Acessando as propriedades dos objetos

O acesso as propriedades individuais de cada objeto do projeto do é feito através dos métodos da classe **TObject**, compostos por um conjunto de instruções **Get** – para obter os valores e instruções **Set** – para modificar os valores:

```
bool TObject::GetBoolean(string Propriedade);
```

Retorna o valor de uma propriedade booleana do objeto.

```
bool TObject::SetBoolean(string Propriedade, bool Valor);
```

Modifica o valor de uma propriedade booleana do objeto.

```
int TObject::GetInt(string Propriedade);
```

Retorna o valor de uma propriedade int do objeto.

```
bool TObject::SetInt(string Propriedade, int Valor);
```

Modifica o valor de uma propriedade int do objeto.

```
double TObject::GetDouble(string Propriedade);
```

Modifica o valor de uma propriedade double do objeto.

```
bool TObject::SetDouble(string Propriedade, double Valor);
```

Modifica o valor de uma propriedade double do objeto.

```
string TObject::GetString(string Propriedade);
```

Retorna o valor de uma propriedade string do objeto.

```
bool TObject::SetString(string Propriedade, string Valor);
```

Modifica o valor de uma propriedade string do objeto.

```
bool TObject::GetPoint(string Propriedade, point@ p);
```

Copia para o ponto **p** o valor de uma propriedade point do objeto.

```
bool TObject::SetPoint(string Propriedade, point@ p);
```

Modifica o valor de uma propriedade point do objeto o valor para do ponto **p**.

Lista de objetos e suas propriedades principais

Propriedades genéricas		
double	Xvector	Coordenada X do vetor de extrusão
double	Yvector	Coordenada Y do vetor de extrusão
double	Zvector	Coordenada Z do vetor de extrusão

Objeto	LINE (Linha)	
Tipo	Propriedade	Descrição
point	Start	Ponto inicial
point	End	Ponto final

Objeto CIRCLE (Círculo)		
Tipo	Propriedade	Descrição
point	Center	Centro
double	Radius	Raio

Objeto ARC (Arco)		
Tipo	Propriedade	Descrição
point	Center	Centro
double	Radius	Raio
double	StartAngle	Ângulo inicial em graus
double	EndAngle	Ângulo final em graus

Objeto ELLIPSE (Elipse)		
Tipo	Propriedade	Descrição
point	Center	Centro
point	StartPoint	Ponto de início
double	Ratio	Razão entre o maior raio e o menor raio
double	StartAngle	Ângulo inicial em graus
double	EndAngle	Ângulo final em graus
double	Rotation	Ângulo de rotação em graus

Objeto TEXT (Texto)		
Tipo	Propriedade	Descrição
point	Insertion	Posição do texto
string	Text	Texto
string	Style	Estilo de texto
float	Size	Tamanho do texto
float	Rotation	Ângulo de rotação em graus
char	JustificationH	Justificação horizontal do texto
char	JustificationV	Justificação vertical do texto

Objeto 3DFACE (Face)		
Tipo	Propriedade	Descrição
point	P1, P2, P3, P4	Os quatro pontos que determinam face
bool	Edge1	Primeira aresta visível
bool	Edge2	Segunda aresta visível
bool	Edge3	Terceira aresta visível
bool	Edge4	Quarta aresta visível
double	U1, V1	Coordenada UV de mapeamento para textura do ponto 1
double	U2, V2	Coordenada UV de mapeamento para textura do ponto 2
double	U3, V3	Coordenada UV de mapeamento para textura do ponto 3
double	U4, V4	Coordenada UV de mapeamento para textura do ponto 4

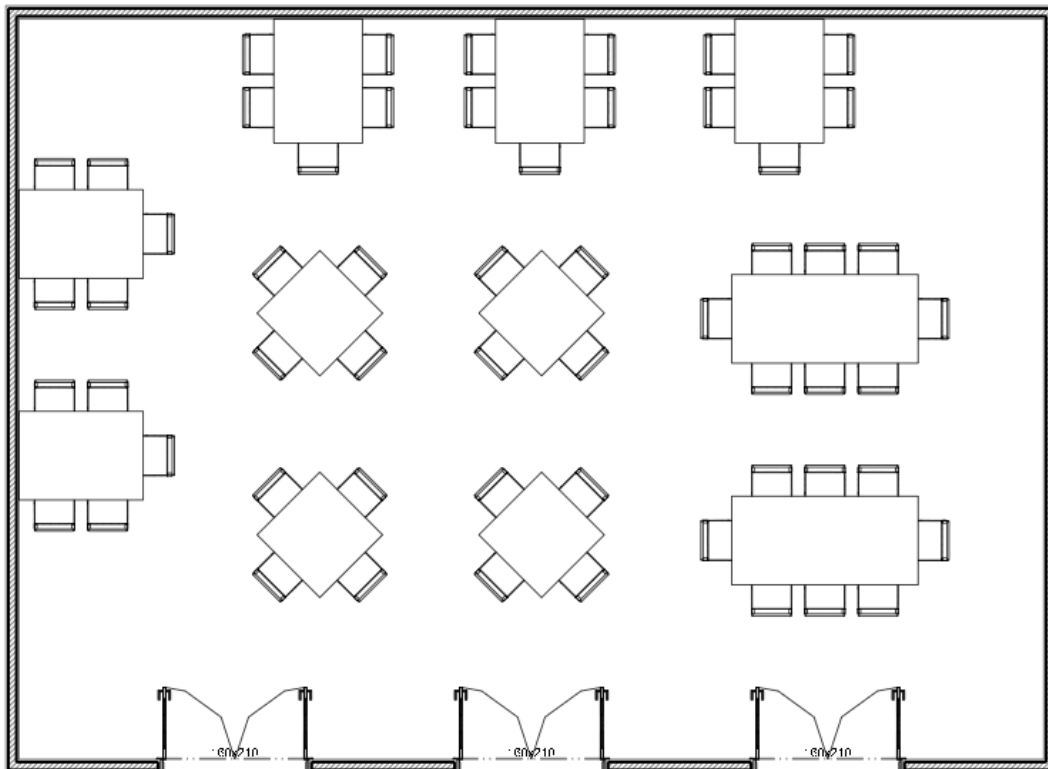
Objeto	INSERT (Inserção - Bloco inserido no desenho)	
Tipo	Propriedade	Descrição
point	Insertion	Ponto de inserção
string	Name	Nome do bloco
float	Rotation	Ângulo de rotação em graus
double	Xscale	Fator de escala no eixo X
double	Yscale	Fator de escala no eixo Y
double	Zscale	Fator de escala no eixo Z

Objeto	HATCH (Preenchimento com hachuras e gradientes)	
Tipo	Propriedade	Descrição
string	Name	Nome do padrão da hachura
float	Scale	Fator de escala para desenhar o padrão
float	Rotation	Ângulo de rotação do padrão em graus

Objeto	SHAPED (Perfilado)	
Tipo	Propriedade	Descrição
string	Name	Nome do perfil
double	Xscale	Fator de escala no eixo X
double	Yscale	Fator de escala no eixo Y
double	Zscale	Fator de escala no eixo Z

Exemplos de aplicações

Lista de materiais



Este exemplo como criar uma tabela mostrando as quantidades de blocos do tipo especificado que estão inseridos no desenho:

```

void main()
{
    //Define as variáveis para a contagem
    int Mesa4=0, Mesa5=0 ,Mesa8=0 ,Cadeiras=0;

    //Faz a contagem dos blocos
    for(int a=0; a<Drawing.Objects.Count(); a++)
    {
        TObject @Object=Drawing.Objects.Items(a);
        if(Object.ClassName()=="INSERT")
        {
            string Bloco=Object.GetString("NAME");
            //Mesa de 4 lugares
            if(Bloco=="Mesa_d4")
            {
                Cadeiras+=4;
                Mesa4+=1;
            }
            //Mesa de 5 lugares
            if(Bloco=="Mesa_d5_canto")
            {
                Cadeiras+=5;
                Mesa5+=1;
            }
            //Mesa de 8 lugares
            if(Bloco=="Mesa_d8")
            {
                Cadeiras+=8;
                Mesa8+=1;
            }
        }
    }

    //Pede o ponto de inserção da tabela
    point p1;
    Prompt("Indique o ponto inserção da tabela:");
    GetPoint(p1);

    //Cria a tabela
    SetLayer("0");

    Text( point(p1.x+10, p1.y, p1.z) , "Objeto", 10, 0);
    Text( point(p1.x+240, p1.y, p1.z) , "Quantidade", 10, 0);
    p1.Move(0,-5,0);

    Line( p1, point(p1.x+320, p1.y, p1.z) );
    p1.Move(0,-15,0);

    Text( point(p1.x+10, p1.y, p1.z) , "Mesa para 4", 10, 0);
    Text( point(p1.x+240, p1.y, p1.z) , ""+Mesa4, 10, 0);
    p1.Move(0,-5,0);

    Line( p1, point(p1.x+320, p1.y, p1.z) );
    p1.Move(0,-15,0);

    Text( point(p1.x+10, p1.y, p1.z) , "Mesa para 5", 10, 0);
    Text( point(p1.x+240, p1.y, p1.z) , ""+Mesa5, 10, 0);
    p1.Move(0,-5,0);

    Line( p1, point(p1.x+320, p1.y, p1.z) );

```

```

p1.Move(0,-15,0);

Text( point(p1.x+10, p1.y, p1.z) , "Mesa para 8", 10, 0);
Text( point(p1.x+240, p1.y, p1.z) , ""+Mesa8, 10, 0);
p1.Move(0,-5,0);

Line( p1, point(p1.x+320, p1.y, p1.z) );
p1.Move(0,-15,0);

Text( point(p1.x+10, p1.y, p1.z) , "Cadeiras", 10, 0);
Text( point(p1.x+240, p1.y, p1.z) , ""+Cadeiras, 10, 0);
p1.Move(0,-5,0);

Line( p1, point(p1.x+320, p1.y, p1.z) );
}

```

Objeto	Quantidade
Mesa para 4	4
Mesa para 5	5
Mesa para 8	2
Cadeiras	57

Objetos BIM

Estilos de componentes - TStyle

Os objetos BIM podem possuir **Estilos de componentes** associados, que contém seus dados parametrizados. O métodos de **TObject GetBoolean, SetBoolean, GetInt, SetInt, GetDouble, SetDouble, GetString, SetString** também podem ser utilizados para acessar as propriedades do Estilo do componente ou seus valores instanciados armazenados no próprio objeto.

Você também poderá acessar diretamente os dados de um **Estilo de componentes**. Utilize a função **GetStyle()** para obter um objeto **TStyle** com os dados do **Estilo de componente** do objeto. Caso o objeto não possua estilo associado, será retornado o valor **null**.

```
TStyle@ TObject::GetStyle();
```

Retorna um objeto da classe TStyle com os dados do Estilo do componente.

Definição da classe TStyle:

```

class TStyle
{
    string ID(); // Retorna o nome ID do estilo e o tipo no formato "ID : Tipo"
    int Count(); // Retorna o número de tipos definidos no estilo
    string SetType(int Index); // Ajusta o estilo para acessar as propriedades do tipo indicado

    // Métodos para obter valores das propriedades
    bool GetBoolean(string Propriedade); // Retorna o valor booleano de uma propriedade
    int GetInt(string Propriedade); // Retorna o valor inteiro de uma propriedade
    double GetDouble(string Propriedade); // Retorna o valor double de uma propriedade
    string GetString(string Propriedade); // Retorna o valor string de uma propriedade
}

```


Cada Estilo de componentes poderá possuir diversos tipos definidos. Por exemplo, um estilo de janela poderá ter vários tamanhos diferentes. Utilize a função `SetStyle` para ajustar o estilo ao tipo desejado.

```
bool TObject::SetStyle(TStyle@ Estilo);
```

Modifica o estilo do objeto para o estilo indicado, caso aplicável. O estilo deverá possuir a mesma categoria do estilo atualmente definido.

```
bool TObject:: SetStyle (string EstiloID);
```

Modifica o estilo do objeto para o nome ID indicado, caso aplicável. O estilo deverá possuir a mesma categoria do estilo atualmente definido.

Objetos BIM anexados

Os objetos BIM podem ter uma lista objetos anexados ao objeto principal. Utilize os seguintes métodos para acessar estes objetos:

```
int TObject::Count();
```

Retorna o número de objetos anexados.

```
TObject@ TObject:: Item (int Index);
```

Obtém o objeto indicado pela variável `Index`. O valor de `Index` deverá ser entre **0** e o valor retornado por **Count() - 1**, onde **0** é o primeiro objeto da lista.

```
bool TObject:: Add (TObject@ Objeto);
```

Adiciona um objetos anexo na lista do objeto principal.

```
bool TObject:: Insert (int Index, TObject@ Objeto);
```

Inserir um objetos anexo na posição indicada na lista do objeto principal.

```
bool TObject:: Delete (int Index);
```

Apaga e exclui da lista de anexos o objeto na posição indicada.

```
bool TObject:: DeleteAll ();
```

Apaga e exclui todos os objetos anexados.

```
bool TObject:: OrderBy (string Propriedade);
```

Ordena os objetos da lista de acordo com os valores da propriedade indicada.

Componentes ASX

Os componentes ASX também poderão possuir Estilo associado, e o acesso aos dados parametrizados do Estilo ou instanciados no componente podem ser feitos da mesma forma que os objetos BIM.

As propriedades definidas no **Editor ASX** podem ser acessadas diretamente pelo script como variáveis definidas no próprio código. Isso já foi demonstrado no **Capítulo 6 – Componentes ASX**.

Acessando as tabelas

Objetos da tabela Layers (Camadas)

Objeto	LAYER (Camada)	
Tipo	Propriedade	Descrição
string	Name	Nome da camada
int	Color	Cor com que a camada é desenhada na tela
int	Printer_Color	Cor com que a camada é impressa
double	Printer_Width	Espessura do traço para impressão (Pena)
int	DXF_Code	Código de cor DXF
bool	Visible	Indicador de visibilidade da camada
bool	Frozen	Indica quando a camada esta congelada

Exemplo: Este script mostra a quantidade e o nome de todas as camadas definidas no desenho, acessando a lista **Drawing.Layers**.

```
void main()
{
    //Mostra quantas camadas existem no desenho
    Prompt("Existem "+Drawing.Layers.Count()+" camadas no desenho.");

    //Camada por camada
    for(int a=0; a<Drawing.Layers.Count(); a++)
    {
        //Pega a camada
        TObject @Object=Drawing.Layers.Items(a);

        //Pega o nome na propriedade "NAME"
        string Nome=Object.GetString("NAME");

        //Mostra o nome
        Prompt("-> "+Nome);
    }
}
```

Objetos da tabela Materials

Objeto	MATERIAL (Material)	
Tipo	Propriedade	Descrição
string	Name	Nome do material

Objetos da tabela Blocks

Objeto	BLOCK (Bloco)	
--------	---------------	--

Tipo	Propriedade	Descrição
string	Name	Nome do bloco

Exemplo: Similar ao script anterior, mas agora mostra a quantidade e nomes dos materiais e blocos definidos no desenho acessando as listas **Drawing.Materials** e **Drawing.Blocks**.

```
void main()
{
    Prompt("Existem "+Drawing.Materials.Count()+" materiais no desenho.");

    for(int a=0; a<Drawing.Materials.Count(); a++)
    {
        TObject @Object=Drawing.Materials.Items(a);
        string Nome=Object.GetString("Name");
        Prompt("      -> "+Nome);
    }

    Prompt("Existem "+Drawing.Blocks.Count()+" blocos definidos no
desenho.");

    for(int a=0; a<Drawing.Blocks.Count(); a++)
    {
        TObject @Object=Drawing.Blocks.Items(a);
        string Nome=Object.GetString("Name");
        Prompt("      -> "+Nome);
    }
}
```

CAPÍTULO 13

Selecione objetos

A partir do ArchiStation 2013 foram adicionados novos comandos e recursos aos scripts ASX. Um destes aprimoramentos permite a interação do script com objetos já criados no desenho. Para isso, foram criadas as seguintes funções:

Funções de seleção

```
void ssGet();
```

Solicita ao usuário que selecione objetos por qualquer meio disponível.

```
bool ssGet(point@ p1);
```

Selecione o objeto encontrado no ponto indicado.

Retorna **true** se encontrou algum objeto, senão retorna **false**.

```
bool ssGet(point@ p1, point@ p2);
```

Seleção por janela. Seleciona todos objetos inteiramente contidos dentro da janela determinada pelos pontos p1 e p2.

```
bool ssGetc(point@ p1, point@ p2);
```

Seleção por cruzamento. Seleciona todos objetos inteiramente contidos ou que cruzarem com as bordas da janela determinada pelos pontos p1 e p2.

Obtendo acesso aos objetos selecionados

A lista global que armazena os objetos selecionados é a **Selection**, e pode ser usada de maneira similar a lista de objetos do projeto **Drawing.Objects**.

Exemplo:

```
void main()
{
    //Pede ao usuário que selecione alguns objetos
    Prompt("Selecione objetos:");
    ssGet();

    //Mostra a quantidade de objetos selecionados
    Prompt("Você selecionou "+Selection.Count()+" objetos.");
}
```

CAPÍTULO 14

Instalações

Traça o caminho de um objeto a outro

```
bool TObject::SetInt(string Propriedade, int Valor);  
Ajusta o valor de uma propriedade int do objeto.
```

```
double TObject::GetDouble(string Propriedade);  
Pega o valor de uma propriedade double do objeto.
```

```
bool TObject::SetDouble(string Propriedade, double Valor);  
Ajusta o valor de uma propriedade double do objeto.
```

```
string TObject::GetString(string Propriedade);  
Pega o valor de uma propriedade string do objeto.
```

```
bool TObject::SetString(string Propriedade, string Valor);  
Ajusta o valor de uma propriedade string do objeto.
```

APÊNDICE A

Lista de Funções

Funções para Desenhar

```
bool Line(point@ p1, point@ p2);
```

Desenha uma linha entre os pontos p1 ao p2.

```
bool Line(double x0, double y0, double z0, double x1, double y1, double z1);
```

Desenha uma linha entre os pontos p1 ao p2.

```
bool Circle(point@ Centro, double Raio);
```

Desenha um círculo com centro e raio indicados.

```
bool Arc(point@ Centro, double Raio, double AnguloInicial, double AnguloFinal);
```

Desenha um arco definido por centro, raio, ângulos inicial e final.

```
bool Ellipse(point@ Centro, double RaioX, double RaioY, double AnguloInicial, double AnguloFinal, double AnguloRotacao);
```

Cria uma elipse definida por Centro, Raios no eixo X e Y, ângulos inicial, final e de rotação.

```
bool Face(point@ p1, point@ p2, point@ p3, point@ p4);
```

Desenha uma face definida pelos quatro pontos indicados.

```
bool Face(point@ p1, point@ p2, point@ p3, point@ p4, bool edge1, bool edge2, bool edge3, bool edge4);
```

Desenha uma face definida pelos quatro pontos indicados, sendo que as variáveis **edge** definem quais das arestas serão visíveis ou invisíveis.

```
bool Face(point@ p1, point@ p2, point@ p3, point@ p4, bool edge1, bool edge2, bool edge3, bool edge4, double tx1, double ty1, double tx2, double ty2, double tx3, double ty3, double tx4, double ty4);
```

Desenha uma face definida pelos quatro pontos indicados, sendo que as variáveis **edge** definem quais das arestas serão visíveis ou invisíveis, e associa as coordenadas de mapeamento de textura tx1, ty1, tx2, ty2, tx3, ty3, tx4 e ty4 para cada um dos pontos. As coordenadas de mapeamento padrão são (0,0), (0,1), (1,1) e (1,0).

```
bool Text(point@ Posicao, string Texto, double Tamanho, double Rotacao);
```

Desenha um texto na posição, tamanho e rotação indicados.

```
bool Text(point@ Posicao, string Texto, double Tamanho, double Rotacao, int Justificacao );
```

Desenha um texto na posição, tamanho, rotação e justificação indicados.

Alterarando estados e tabelas

Espessura e altura

`bool SetWidth(double width);`
Especifica a **Espessura** corrente.

`bool SetHeight(double height);`
Especifica a **Altura** corrente.

Camadas

`bool AddLayer(string Nome, int CorRGB, string LType);`
Adiciona uma nova camada ao desenho.

`void AddLayer(string Nome, int Cor, string LType, string Material);`
Cria uma camada com nome, cor, estilo de traço e material especificados (ASX 2.0).

`bool SetLayer(string Nome);`
Define a camada indicada como corrente. Se a camada não existir ou não for possível a tornar corrente, a função retorna **false**.

`string GetLayer();`
Retorna na variável o nome da camada corrente.

Grupos

`void SetGroup(int group);`
Especifica o número do grupo em que os próximos objetos serão desenhados.
`SetGroup(0)` especifica que os objetos não serão agrupados.

`int SetNewGroup();`
Especifica que os próximos objetos serão criados em um novo agrupamento retornando seu número.

Pavimentos

`void SetFloor(string Pavimento);`
Torna corrente o pavimento indicado.

`string GetFloor ();`
Retorna o nome do pavimento corrente.

`string GetFloorHeigth();`
Retorna a altura do pavimento corrente.

Funções de controle

`bool Redraw();`
Redesenha a tela, torna os objetos criados pelo script visíveis.
Obs.: A função **Redraw** é chamada ao término da execução do script.

```
void Prompt(string Mensagem);
```

Exibe uma mensagem na linha de comando.

```
void SetHelp (string Arquivo, string Tópico);
```

Define o arquivo **.chm** (*Microsoft Compiled HTML Help*) e o nome do tópico que será exibido se o usuário chamar o **Ajuda do ArchiStation** pressionando a tecla **<F1>**. Se o nome do arquivo for fornecido sem a pasta, o ArchiStation irá procurá-lo na pasta Plugins.

Exemplos:

```
SetHelp ("MeuHelp.chm", "Estrutura");
SetHelp ("D:\\MeuHelp.chm", "Estrutura");
SetHelp (PluginsDir ()+"Aplicativo\\MeuHelp.chm", "Estrutura");
```

```
void Help (string Arquivo, string Tópico);
void Help ();
```

Mostra o arquivo de ajuda indicado. Se nenhum parâmetro for passado na chamada da função, o sistema exibirá o arquivo e tópico da última configuração.

Entrada de dados

Os valores dos comandos de entrada de dados são retornados nas variáveis indicadas por **"in"**.

```
bool GetPoint(point@ in);
```

Solicita a entrada de um ponto. O Ponto pode ser indicado no desenho, ou digitado no **Quadro de comando**.

```
bool GetPoint(point@ in, point@ p1);
```

Solicita a entrada de um ponto usando como referência o ponto p1.

```
bool GetCorner(point@ in, point@ p1);
```

Solicita a entrada de um ponto usando como referência a área retangular criada a partir do ponto p1.

```
void GetDist(double &in);
```

Solicita a entrada de uma distância. Se o usuário indicar um ponto no desenho, será solicitada a entrada de mais um ponto e retornada a distância entre os dois.

```
void GetDist(double &in, point@ p1);
```

Solicita a entrada de uma distância. Se o usuário indicar um ponto no desenho, será retornada a distância deste ponto ao ponto p1.

```
void GetAngle(double &in);
```

Solicita a entrada de um valor angular. Se o usuário indicar um ponto no desenho, será solicitada a entrada de mais um ponto e retornado o ângulo do segmento formado com os dois pontos.

```
void GetAngle(double &in, point@ p1);
```

Solicita a entrada de um valor angular. Se o usuário indicar um ponto no desenho, será retornado o ângulo do segmento formado com o ponto indicado e o ponto p1.

```
void GetReal(double &in);
```

Solicita a entrada de um número real.


```
void GetInt(int &in);
```

Solicita a entrada de um número inteiro.

```
bool GetString(string &in);
```

Solicita a entrada de uma cadeia de caracteres.

Funções Matemáticas

Funções Aritméticas

```
int abs(int i);
```

Retorna o valor absoluto de um inteiro.

```
double fabs(double d);
```

Retorna o valor absoluto de um valor de ponto flutuante.

```
double sqrt(double d);
```

Retorna a raiz quadrada.

```
double exp(double d);
```

Retorna a constante de Euler elevada a um valor, e^d .

```
double pow(double da, double db);
```

Retorna 'da' elevado a potência 'db', da^{db} .

```
double log(double d);
```

Retorna o logaritmo natural, $\log_e d$.

Gerador de números aleatórios

```
int random(int num);
```

Retorna um número inteiro aleatório entre 0 e num-1.

```
void srand(int seed);
```

Inicializa o gerador de números aleatórios com a semente indicada pelo valor de *seed*.
Seed=1 é o valor inicial do gerador.

```
void randomize();
```

Inicializa o gerador de números aleatórios com uma semente de valor aleatório.

Funções Trigonométricas

```
double sin(double Ang);
```

Calcula o seno do ângulo fornecido em radianos.

```
double cos(double Ang);
```

Calcula o co-seno do ângulo fornecido em radianos.

```
double atan(double d);
```

Calcula o arco tangente em radianos de um valor double.

```
double radians(double degrees);
```

Converte o ângulo de graus para radianos.

```
double degrees(double radians);
```

Converte o ângulo de radianos para graus.

Funções de geometria

```
double Angle(point@ p1, point@ p2);
```

Retorna o ângulo em radianos de inclinação do segmento definido pelos pontos indicados.

```
double Distance(point@ p1, point@ p2);
```

Retorna a distância espacial entre os pontos indicados.

```
void Polar(point@ p1, double Ângulo, double Distância);
```

Atualiza *p1* com o ponto definido na *Distância* indicada seguindo na direção apontada pelo *Ângulo* (em radianos) especificado.

```
void PointAt(point@ *p, point@ *p1, point@ *p2, double Distancia);
```

Atualiza o ponto *p* para a posição em relação a *p1*, percorrida a distância indicada em direção a *p2*.

```
void Parallel(point@ *l1, point@ *l2, point@ *p1, point@ *p2, double Distancia);
```

Atualiza a posição dos pontos *l1* e *l2* para os valores paralelos ao segmento entre *p1* e *p2* a distância indicada.

```
bool Perpend(point@ p, point@ p1, point@ l1, point@ l2);
```

Atualiza o ponto *p* com o ponto que forma com *p1* a perpendicular com a reta que liga *l1* e *l2*. Retorna **true** se o ponto *p* pertence ao seguimento de reta formado por *l1* e *l2*.

```
void Intersection(point@ i, point@ l1, point@ l2, point@ p1, point@ p2);
```

Atualiza *i* com o ponto a interseção entre os segmentos *l1*, *l2* e *p1*, *p2*.

Funções de conversão

```
int atoi(string str);
```

Converte uma cadeia de caracteres em um valor inteiro **int**.

```
double atof(string str);
```

Converte uma cadeia de caracteres em um valor de ponto flutuante **double**.

```
int RGB(char Red, int Green, int Blue);
```

Retorna valor de referência de cor no padrão RGB.
Os valores para Red, Green e Blue devem estar compreendidos entre 0 e 255.

Funções de formatação

```
string format(double Número, int Precisão, bool RemoverZeros, bool Milhares);
```

Converte um valor double para uma string formatada com os parâmetros indicados.

Exemplo: `format(10000.234, 2, false, true)` vai retornar **10.000,23**.

Caixas de mensagem e diálogo

Message() - Mensagem de texto

```
void Message(string Mensagem);
```

Exibe uma mensagem ao usuário.

```
void Message(string Mensagem, string Texto);
```

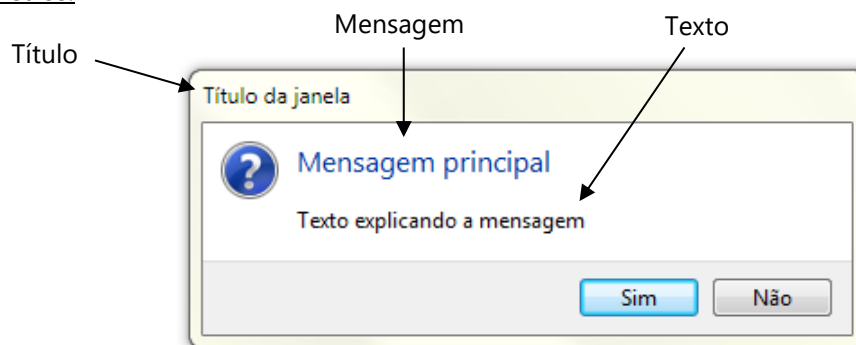
Exibe uma mensagem principal e um texto complementar.

MessageBox() - Caixa de mensagem de texto

```
int MessageBox(string Mensagem, string Texto, string Título, int Tipo);
```

Exibe uma caixa de mensagem com texto na tela e retorna a opção escolhida pelo usuário.

Parâmetros:



Caixa de mensagem de texto

Tipo:

- 0 = OK;
- 1 = OK e CANCELAR;
- 2 = REPETIR e CANCELAR;
- 3 = SIM e NÃO;
- 4 = SIM, NÃO e CANCELAR;
- 5 = ícone de exclamação;
- 6 = ícone de informação;
- 7 = ícone de asterisco;
- 8 = ícone de questão;

Códigos dos valores de retorno:

- 1 = OK;
- 2 = CANCELAR;
- 4 = REPETIR;
- 6 = SIM;
- 7 = NÃO;

GetFile() - Diálogo de seleção de nome de arquivo

```
string getFile(string Título, string Pasta, string Extensão);
```

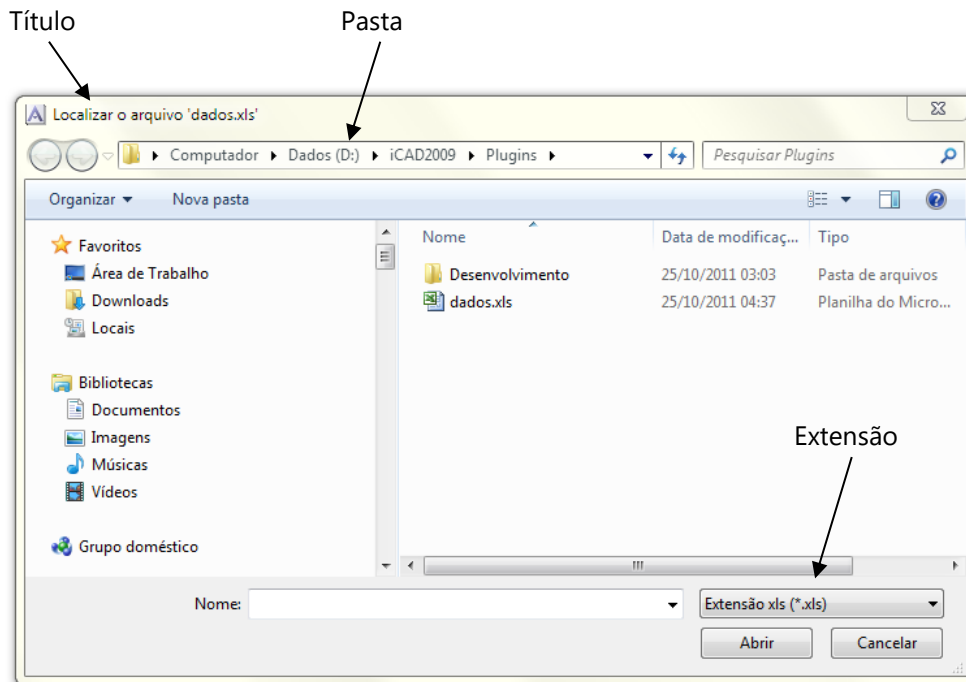
Exibe a caixa de diálogo padrão do Windows para seleção de nome de arquivo e retorna o nome completo incluindo a pasta do arquivo selecionado.

Parâmetros:

Título: Título da janela;

Pasta: Define a pasta inicial de busca do arquivo;

Extensão: Extensão do nome do arquivo a ser selecionado sem ponto.



Caixa de diálogo de seleção de arquivo

Variáveis do sistema

```
string AppDir();
```

Retorna a pasta de instalação do ArchiStation.

```
string PluginsDir();
```

Retorna a pasta de Plugins do ArchiStation.

```
string FileDir();
```

Retorna a pasta do projeto que se encontra em edição. Se não houver arquivo aberto, o valor retornado será "".

```
string Date();
```

Retorna a data do sistema no formato DD/MM/AAAA.

```
string Time();
```

Retorna a hora do sistema no formato HH:MM 24 horas.

```
string Version();
```

Retorna a versão do ArchiStation.

Modificação da interface

Criando uma guia no menu principal

```
ToolbarPage@ CreateToolbarPage(string Título);
```

Cria uma nova guia no menu principal com o título especificado.

Criando um menu em uma guia

```
Toolbar@ ToolbarPage@.CreateToolbar(string Título);
```

Adiciona a guia um menu com o título especificado.

Adicionando um botão de comando a um menu

```
void Toolbar@.AddButton(string Título, string Imagem, string Comando, string Descrição);
```

Adiciona um botão de comando a um menu.

Título	Nome associado ao botão
Imagem	Se for uma string, deve referir-se a ao arquivo onde está a imagem que deve ser aplicada ao ícone do botão. Se for um inteiro (int) refere-se ao número da imagem na lista de imagens do container ASX.
Comando	Comando que deve ser chamado ao se clicar o botão. O comando poderá ser: <ul style="list-style-type: none"> • Um comando do ArchiStation, exemplo: _linha • Chamada a uma função definida no próprio script. O nome da função deve ser precedido por um duplo sinal de dois pontos (::). Exemplo: ::funcao() • Chamada a uma função de outro script localizado na pasta "/Plugins" da instalação do ArchiStation. Neste caso deve ser indicado o nome do arquivo seguido pelo sinal "::" e a função. Exemplo: arquivo::funcao()

Observação: As funções chamadas devem ser definidas como "void".

Descrição	Dica que aparece quando o usuário mantém o cursor do mouse sobre o botão por alguns instantes.
------------------	--

Obtendo dados de planilhas do MS Excel

`bool` ExcelOpenFile(`string` Titulo);

Abre um arquivo do Excel para manipulação. O MS Excel deve estar instalado no seu computador. Retorna **true** se obteve êxito, caso contrário retorna **false**.

`bool` ExcelSetSheet(`string` Sheet);

Seleciona qual planilha do arquivo será manipulada.
Retorna **true** se obteve êxito, caso contrário retorna **false**.

`string` ExcelGetItem(`int` Col, `int` Line);

Retorna uma string com o valor do item na coluna e linha indicados.
Observação: para utilizar como valores numéricos, use as funções de conversão **atoi()** ou **atof()**.

`bool` ExcelCloseFile();

Fecha o arquivo.
Retorna **true** se obteve êxito, caso contrário retorna **false**.

Tipos de variáveis do AngelScript

Valores booleanos

bool é um tipo de dados booleano que pode assumir apenas dois valores: **true** (verdadeiro) ou **false** (falso). As palavras chave **true** e **false** são constantes que podem ser utilizadas em expressões.

Números inteiros

Tipo	Valor mínimo	Valor máximo
int8	-128	127
int16	-32,768	32,767
int	-2,147,483,648	2,147,483,647
int64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
uint8	0	255
uint16	0	65,535
uint	0	4,294,967,295
uint64	0	18,446,744,073,709,551,615

Números reais

Tipo	Intervalo de valores	Menor valor positivo	Dígitos
float	+/- 3.402823466e+38	1.175494351e-38	6
double	+/- 1.7976931348623158e+308	2.2250738585072014e-308	15

Cadeia de caracteres

O tipo **string** é utilizado para armazenamento e manipulação de cadeias de caracteres.