Inca User Manual

Bernhard Liebl

January 31, 2003

Contents

1 Introduction **5**
1.1 Legal 5
1.2 Copyright 5
1.3 License 5
1.4 Motivation 6
1.5 Caveats 6
1.6 Shareware 7
1.7 Features 7
1.8 First Steps 8
1.9 Language Basics 9

2 Language Reference **12**
2.1 About this Reference 12
2.2 Statements 12
2.2.1 If 12
2.2.2 Switch 12
2.2.3 Return 13
2.2.4 While 13
2.2.5 For 13
2.2.6 Loop Control 13
2.3 Types 13
2.3.1 Basic Types 13
2.3.2 The const Modifier 14
2.3.3 The static Modifier 15
2.3.4 The typedef Modifier 15
2.3.5 Naming Schemes 16
2.3.6 Pointers and Arrays 17
2.3.7 References 17
2.3.8 Enums 18
2.4 Classes 19
2.4.1 Structures 19

2.4.2 Classes 20
2.4.3 Type Casting 21
2.4.4 Constructors and Destructors 21
2.4.5 Allocating Classes and Memory 24
2.4.6 Operator Overloading 25
2.4.7 Templates 26
2.5 Scopes 27
2.5.1 Scopes 27
2.5.2 Function Overloading 29
2.5.3 Includes 29
2.5.4 Modules and Namespaces 30

3 Library Reference **32**
3.1 System Library 33
3.1.1 Introduction 33
3.1.2 Overview 34
3.1.3 Date and Time Functions 35
3.1.4 System Functions 37
3.1.5 Cursor Functions 39
3.1.6 Miscellaneous Functions 40
3.1.7 Dialog Classes 41
3.2 BigInt Library 43
3.2.1 Introduction 43
3.2.2 Overview 45
3.2.3 BigInt Function Reference 46
3.3 String Library 54
3.3.1 Introduction 54
3.3.2 Overview 56
3.3.3 String Function Reference 57
3.4 General Math Library 69
3.4.1 Introduction 69
3.4.2 Caveats 69
3.4.3 The Complex Type 71
3.4.4 Constants 72
3.4.5 Overview 73
3.4.6 Math Function Reference 76
3.4.7 Complex Function Reference 107
3.5 IO Library 109
3.5.1 Overview 117
3.5.2 File System Services Reference 120
3.5.3 FileIterator Methods 124

3.5.4 File Method Reference 125
3.5.5 InputStream Method Reference 128
3.5.6 OutputStream Method Reference 131
3.5.7 Keyboard and Mouse Function Reference 135
3.5.8 InflaterStream Method Reference 140
3.5.9 DeflaterStream Method Reference 141
3.5.10 Print Functions 142
3.6 Container Library 145
3.6.1 Introduction 145
3.6.2 Sorting 146
3.6.3 Array Method Reference 148
3.7 Thread Library 154
3.7.1 Introduction 154
3.7.2 Threads 154
3.7.3 Mutexes 155
3.7.4 Signals 156
3.7.5 Queues 156
3.7.6 Overview 157
3.7.7 Thread Methods 158
3.7.8 Mutex Methods 161
3.7.9 Signal Methods 162
3.7.10 Queue Methods 164
3.7.11 Global Functions 168
3.8 3d Support Library 171
3.8.1 Introduction 171
3.8.2 Special Color Constructors 172
3.8.3 Overview 174
3.8.4 Vector Function Reference 175
3.8.5 Noise Function Reference 178
3.9 OpenGL Library 180
3.9.1 Three-Dimensional Graphics 180
3.9.2 Two-Dimensional Graphics 181
3.9.3 OpenGL Function Overview 183
3.9.4 Overview 192
3.9.5 Console Methods 193
3.10 Simple Graphics Library 197
3.10.1 Introduction 197
3.10.2 Setting the Color 197
3.10.3 Drawing Lines 197
3.10.4 Drawing Shapes 198
3.10.5 Drawing Text 198

3.10.6 Overview 200
3.10.7 SGL Function Reference 201
3.11 2d Graphics Library 209
3.11.1 Introduction 209
3.11.2 2d Graphics 209
3.11.3 The Pixmap Class 211
3.11.4 Mixing OpenGL and Pixmaps 212
3.11.5 Overview 214
3.11.6 Pixmap Method Reference 216
3.11.7 General Function Reference 219

Chapter 1

Introduction

1.1 Legal

All trademarks within this document belong to their respective owners.

1.2 Copyright

Inca , its program code, the accompanying data files, sample code and this manual are copyrighted. Neither these documents nor the Inca program code may be copied, redistributed, sold, lended or rented in whole or partially.

1.3 License

By installing and/or using this software you accept the terms and conditions of the following agreement.

1. The author does not make any representations concerning the suitablity for any purpose of this software. This software is provided "as is" without express or implied warranties, including but not limited to warranties of merchantability and fitness for particular purpose.

2. You use the this software completely at your own risk. Under no circumstances shall the author be liable for any direct, indirect, collateral, exemplary, special, incidental or consequential damage (including without limitation loss of use, profits, goodwill or savings, or loss of data, data files or programs) suffered by you or any third party as a result of the use of this software.

3. This software has not undergone testing and may contain errors. It may be incomplete and it may not function properly. You are solely respon-

sible for determining the appropriateness of using this Software.

4. This software may not be used for life critical applications.

5. Should any part of this license be unenforcable, invalid or violating applicable law, this part shall be deemed stricken and shall not affect the enforceability of any other part of this agreement.

1.4 Motivation

Often software engineers, scientists, researchers or hobby programmers want to get a better understanding of ideas, concepts or algorithms without having to code huge programs to implement them. Other people need to write small tools to accomplish simple tasks.

Inca was designed as a small tool for writing small programs that can be tested with minimal effort.

Of course there are plenty of programming languages out there to do those things, like Perl, C, C++, Java, or the embedded languages in mathematics packages like Maple or Mathematica. However, I felt that each of these languages had certain drawbacks for implementing the kinds of problems I usually run across. Sometimes this was the lack of a simple framework, then the lack of a graphical framework, then the lack of object orientation, and sometimes I just didn't like syntax of the language.

So I implemented Inca , which is by no means any better than any of the languages just mentioned, yet it offers a new mix of features. Inca is powerful enough to write quite complex tools and offers a lot of standard functionality in a way that's very simple to access.

1.5 Caveats

Although I've put quite a lot of work into Inca , it's by no means finished or bugfree. I simply do not have the time to take care of all the open ends. Please also note that my ability to provide support for Inca in the future is very limited.

The documentation you're just reading is anything but complete and as some might find, the overall structure is sometimes misleading. Generally, the documentation should be ok to refer to for people with good understanding of C++ and Java. This is especially true as many concepts are stolen from one of these two languages.

On the other hand, this text is definitely no introduction to programming generally and I'm not especially good at writing manuals. Furthermore, many concepts like classes, threads or OpenGL are assumed to be known and understood by the reader. If you're not familiar with these concepts, this text is pretty useless in most cases. Sometimes, studying the sample source code provided with Inca might be a good starting point.

I hope that the package in its current form can still be of utility or help for some people.

1.6 Shareware

Inca is distributed as shareware. This means, if you use it regularly, I ask you to pay a small amount of money to buy a license. Inca does not have a restricted feature set if you don't pay, yet I hope some people will see the effort that's in there.

Inca may be used freely - without paying the shareware fee - for educational and research purposes. The author however encourages users who use Inca in educational group settings to acquire one license.

1.7 Features

So what has Inca to offer? Some of the more interesting features are listed below.

- Fully integrated IDE

  In Inca , to develop programs you don't need any other programs, editors or tools. Inca provides everything you need, from the source code editor, the compiler and debugger, up to the virtual machine that will run your programs.

- An editor providing syntax coloring, automatic formatting, automatic identation and block collapsing

  The editor in the Inca environment provides syntax coloring and automatic formatting and identation support for your source code. It helps you spend more time on writing code and less time on formatting it.

- Inca is an easy to use language

  The Inca language can be nearly as easy to use as BASIC for many purposes, yet it has many of the powerful features of Java™ and C++. If you're looking for a language that has built-in character strings, print and input facilities as well as pointers, references, multiple inheritance, function overloading and namespaces, you're right here.

- Easy to port code to/from C++ or Java

  Since the syntax of Inca is quite similar to C++ and Java, it's very easy to port programs from these language to and from Inca .

- Fast one-pass compiler with automatic look ahead

  Different than C++, the Inca language won't care whether a class has been declared before its first usage or after it. Therefore you don't need header file in Inca (no manual forward declarations of classes or types).

- Versatile library of standard functions

  No matter if you're doing text processing, graphics, mathematics or number crunching, Inca provides you with a robust library of helpful functions, featuring for example a type `BigInt` for arbitrary big integer values, a `Complex` type for complex numbers, a support library for three dimension graphics with vector, point and noise functions, and finally full support for OpenGL.

- Real-time capable thread system

  Inca provides you with a library for threads, mutexes, mutexes, signals and queues that is able to handle microsecond granularity. Using a special real-time mode, Inca is able to run highly time critical tasks with high precision.

- Simple debugger

  Inca provides a simple debugger that helps you track what happens in your program and makes searching bugs easier. The debugger is an experimental feature however and doesn't work completely the way it should.

1.8 First Steps

After you launch Inca , you will be presented with a blank editor window waiting for you to type in text. This is the environment where your programs will be edited and debugged. In the toolbar above the editor window you will find symbols for creating a new document, opening a document and saving a document.

In the second toolbar partition from left, you will find buttons for running the current program, stopping a running program and pausing the currently running program (the latter two are disabled, since no program is currently running). You can also call these commands from the `program` menu. For the moment that's all you need to know.

Let's write a simple hello world program. To do this, we first of all need a `main` function, i.e. the same thing as in C. This function defines where your program starts and ends. If you're not familiar with this or the concepts introduced later on, any basic text on C should cover this. Let's try and type the following

```
void main()
{
    print << "hello world!";
}
```

As you type you might find the editor readjusting the identation depth of lines after you leave them with the cursor. This is due to the automatic source code formatting.

Select "run" in the `program` menu. If the compiler sees an error, a text in the editor window's status bar will show up telling you what's wrong. If everything works a new console text window should open and it should display *hello world!*. You can close the console window to terminate the program (by default, Inca will not terminate programs that did output text automatically).

1.9 Language Basics

The language used in Inca pretty much resembles C in nearly all basic aspects. Let's write a small program, that will print the numbers from 1 to 10 on the screen. To do this, let's declare an integer variable, making it count from 1 to 10, and print every number. Here we go:

```
void main()
{
    for( int i = 1; i <= 10; i++ )
        print << i;
}
```

If you start this program, you will see

```
12345678910
```

on the screen, not exactly what was intended. Obviously we should tell the print command to start a new line after every number. We can achieve this by using the *endl* tag like this:

```
void main()
{
    for( int i = 1; i <= 10; i++ )
        print << i << endl;
}
```

Now the output should be correct. Generally if we write several parameters after the print command, with `<<` between, all the parameters will be printed in the order we specified them. Let's take a look at a similar example. We want to output the squares from 1 to 5. Here's how we could do it:

```
void main()
{
    int i = 1;

    while( i <= 5 )
    {
        print << i << "*" << i << "=" << i * i << endl;
        i = i + 1;
    }
}
```

This time we used a different loop, the while loop. We could've also used the do-while loop that's available in C. The print statement might look pretty confusing at first, but it should be clear what it does when we look at the output:

```
1*1=1
2*2=4
3*3=9
4*4=16
5*5=25
```

Let's do something interactive. How about we write a program that computes the are of a circle, given its radius. We could do that this way:

```
void main()
{
    float r; // the radius of the circle

    print << "please enter the circle's radius: ";
    input >> r;
    print << "the circle's area is " << r * r * pi;
}
```

Here several things can be seen. The double slash after the declaration of r in the third line is a comment. It says that the rest of the line should be ignored by the compiler. The input command on the sixth line is the inverse of the print command. It reads an input typed in by the user into the text console. Note that the arrows point in the other direction than for the print command. The formula r^2pi on the last but one line is the formula for the area of a circle. The constant pi is predefined.

Chapter 2

Language Reference

2.1 About this Reference

As the title already points out, this chapter is written as a reference for people already familiar with C or C++ and is *not* meant as an introductory text. Actually the vast majority of the concepts is taken over from C or C++ with no or only minor variations. If you're not familiar with C and C++ it will probably be helpful to consult a C++ introductory text.

2.2 Statements

Items in `[ ]` brackets are optional.

2.2.1 If

```
if( expression )
    statement

if( expression )
    statement
else
    statement
```

2.2.2 Switch

```
switch( integer expression )
{
    case integer value one:
```

```
        ...

    case integer value two:
        ...

    default:
        ...
}
```

2.2.3 Return

```
return [return value]
```

2.2.4 While

```
while( expression )
    statement

do
    statement
while( expression )
```

2.2.5 For

```
for( [declaration]; [condition]; [iterator] )
    statement
```

2.2.6 Loop Control

```
break

continue
```

2.3 Types

2.3.1 Basic Types

Inca provides a number of basic, built-in types. The integer types come as signed and unsigned flavors, just as in C and C++.

The `char` type is intended to hold characters, whereas the other integer types `byte`, `short`, `int`, `long` are intended for integer valued computations.

Type	Bits	Default Sign	Default Range
bool	8		{true,false}
char	8	signed	[-128,127]
byte	8	unsigned	[0,255]
short	16	signed	[-32768,32767]
int	32	signed	[-2147483648,2147483647]
long	64	signed	$[-2^{63}, 2^{63}-1]$
float	32		
double	64		

The default sign for the integer types is the sign that's used if no explicit sign is given in by the programmer in a type declaration. Note that, different than in C and C++, the `long` type in Inca is 64 bit wide.

A variable declaration consists of a type name followed by a variable name. For example `int a;` is a valid declaration and declares a variable of type int, that has the name a. Note that variable names always have to start with a lower case letter. The reason for this is explained further below in the section "Naming Schemes". In a function, variables are valid from the point of their declaration, but not above it. For example in the program

```
void main()
{
    int a;
    a = 1;

    b = 2; // invalid
    int b;
}
```

the access to b is invalid, since b is used before it was even declared.

2.3.2 The const Modifier

There are several modifiers that can be added to a type to specify certain characteristics. The `const` modifier tells Inca that the variable declared under a certain type may not be changed after its declaration. In

```
void main()
{
   const int a = 5;
   a = 3; // invalid
}
```

the access to a in line 4 is invalid, since it was declared `const`. The initial assignment of the value 5 in the declaration line is allowed however.

2.3.3 The static Modifier

The `static` modifier tells Inca that a variable is static, which means that it is only initialized once at program start and held in a nonvolatile frame that lives across function calls. It's pretty similar to a global variable in many aspects, only that you can access a static variable only within the scope you declared it.

```
void f()
{
    static int z = 0;
    print << z << endl;
    z++;
}

void main()
{
    for( int i = 0; i < 10; i++ )
        f();
}
```

The program above will print the numbers 1 to 10. The variable z is initialized by Inca to 0 at the program start. Since z is not affected by the function context, it acts as if it was global.

2.3.4 The typedef Modifier

Using the `typedef` modifier you can create aliases of types. In many situations it's quite handy to have a type renamed to make a certain design principle more obvious or enhance abstraction. `typedef` in Inca works like in C++ in many aspects.

```
typedef int MyType;

void main()
{
    MyType a = 12345;
    print << a;
}
```

There are two differences to the `typedef` in C++. First, Inca expects that every type you declare starts with an upper case letter. The reason for this is explained in detail in the next section. The second difference is that Inca does not expect the typedef to actually occur before the first usage of the newly defined type, like C++ does. This means that

```
void main()
{
    MyType a = 12345;
    print << a;
}

typedef int MyType;
```

is a completely valid variant of the example below. The two peculiarities just mentioned also apply to class declarations by the way, but that's a later section.

2.3.5 Naming Schemes

Inca expects that every type you declare starts with an uppercase letter and every variable and function name starts with a lower case letter. Declarations like

```
typedef int my_type_t;
float Result;
```

are *not* valid. The reason for this is that Inca is a one pass compiler. The first time it encounters some name, it doesn't know whether it deals with a type name, a variable name, or something else. The problem does not seem to be solvable with one pass. A second pass however would increase compile time, which is not tolerable for a rapid prototyping system of this kind. So Inca needs some hint what it's actually dealing with. That's where this rule was brought in. The designer hopes it will not pose serious problems, since most programmers tend to have different naming schemes for variables, functions and classes anyway, and the scheme enforced here pretty much resembles the one commonly used in Java™ , which seems to be widely accepted.

If you declare something that Inca thinks is not corresponding to the naming scheme needed, it makes a remark to please rename some identifier or type and makes a suggestion how to do so.

2.3.6 Pointers and Arrays

Inca deals with pointers and arrays pretty much the way C and C++ does. A pointer is a typed address to a location in memory. The item the pointer points to can be read and written using the dereference operator `*`. The address of an item can be taken using the `&` operator. For example

```
void f( int *x )
{
    *x = *x + 1;
}

void main()
{
    int a = 1;
    f( &a );
    print << a;
}
```

will output 2. Here a is initially set to 1, then the address of a is taken and handed over to the function f. The function f reads the current value of x using the `*` operator, adds one, and writes it again at the same location, which is actually the location of a in the main function. That's why, when we return to main, a will have the value 2.

Arrays are declared and accessed the same way as in C and C++. Multiple dimensions are supported. The program

```
void main()
{
    int a[ 100 ];

    for( int i = 0; i < 100; i++ )
        a[ i ] = i;
}
```

fills an array of 100 integers with the values from 0 to 99.

2.3.7 References

Inca supports references in the style of C++ references. Basically references can be used to symbolically link between variables without having to use the dereferencing involved when using pointers. As an example

```
void f( int &x )
{
    x = x + 1;
}

void main()
{
    int a = 1;
    f( a );
    print << a;
}
```

will do the same thing, as the sample program from the last section, which is to output 2, only that the code looks much more cleaner. A reference is declared via the `&` operator in the parameter declaration of the function x. The `int &x` declaration tells Inca to hand over this parameter not by value, but by reference, which means that every operation that's performed on x, is also performed on a, and vice versa.

2.3.8 Enums

As in C++, `enum`s are a way of defining constant integer values. They can be either typed (by providing a name with the `enum` declaration) or untyped. The enumerator values inside the `enum` block have to fulfill the conventions of variable and function names (i.e. start with a lower case letter).

```
enum Ingredient {
    flour,
    apples,
    cinnamon
}

enum {
    teaspoon = 10,
    alot = 100
}

void addToRecipe( Ingredient ingredient, int amount )
{
    ...code to add ingredient...
}
```

```
void main()
{
    addToRecipe( flour, alot );
    addToRecipe( apples, 50 );
    addToRecipe( cinnamon, teaspoon );
}
```

Like in C++, `enum` values will automatically increment in steps of one relative to the previous enumerator value if you don't specify an explicit value via `=`.

2.4 Classes

2.4.1 Structures

A structure is a compound type consisting of several variables. For example, a record for a star database storage system might look like this

```
struct Star {
    String name;
    float distance;
    float brightness;
    int numberOfPlanets;
}
```

Structures can be used in declarations the way basic types like `int` and `float` are used. The program

```
void readStars( Star* stars, int count )
{
    ...read stars into array...
}

void main()
{
    Star myStars[ 200 ];

    readStars( myStars, 200 );
    for( int i = 0; i < 200; i++ )
        print << myStars[ i ].name << endl;
}
```

reads in 200 star records using a function `readStars` and then displays the name of each one of them using the `.` structure access operator, which works the same way like in C. If dealing with pointers to structures, structures can also be accessed using the `->` operator, like in

```
void initializeStar( Star* star )
{
    star->name = "untitled";
    ...
}
```

2.4.2 Classes

A `class` is like a structure, but it additionally contains methods and access privilege modifiers. A `class` is declared using the `class` statement like in the following example modelling a geometric primitive.

```
class Primitive {
public:
    void setLocation( int x, int y )
    {
        mX = x;
        mY = y;
    }

    abstract void draw();

private:
    int mX;
    int mY;
}
```

The `public:`, `protected:` and `private:` have the same meaning as in C++. Functions are always specified inside the class declaration as in Java™ . The `abstract` modifier before the function `draw` tells Inca that this function has no implementation. Classes deriving from `Primitive` must provide it. Classes can be derived using the `extends` keyword. For example a class representing a sphere could be defined as

```
class Sphere extends Primitive {
public:
    void draw()
```

```
    {
        // ...draw the sphere...
    }
}
```

Different than Java™ , Inca supports multiple inheritance. The classes which should be base classes are just provided as a list separated by commas, like in

```
class MyComplexType extends TypeOne,
    TypeTwo, TypeThree, ... { ... }
```

There is no `virtual` modifier in Inca since every method is automatically virtual, which means that every method can be overloaded by classes deriving the method's class.

2.4.3 Type Casting

Type casting works like in C++. Inca provides two semantically equivalent syntax variants, namely

```
(Type)expression
cast<Type>( expression )
```

The second variant resembles the `static_cast` and `dynamic_cast` modifiers of C++. Independent of what syntax you use, Inca will always typecast the way that a C++ compiler does it when using `dynamic_cast`. That means, that if pointers to classes are involved, Inca will always check if the typecasted class is really of the expected type at runtime and return `null` if not.

2.4.4 Constructors and Destructors

Inca provides a mechanism that is similar to what C++ calls constructors and destructors. In Inca every class can have one or two functions called `create` and `destroy`. The `create` function is called automatically whenever an object of this class is instantiated. The `destroy` method is called whenever an object of this class is destroyed. For example the program

```
class MyClass {
public:
    void create()
```

```
    {
        print << "created instance " << this << endl;
    }

    void destroy()
    {
        print << "destroyed instance " << this << endl;
    }
}

void main()
{
    MyClass a;
    print << "hello world" << endl;
}
```

will generate output that will look something like this

```
created instance 0x17a20d78
hello world
destroyed instance 0x17a2d078
```

Since "a" lives within the local frame of the `main` function, it is automatically destructed as we leave the function. Generally the `construct` function is called as soon as the corresponding variable can be accessed, and the `destruct` function is called as soon as the corresponding variable can no longer be accessed. As another, sort of very technical example, consider the function

```
void main()
{
    print << "entering main" << endl;

    MyClass a;

    if( true )
    {
        print << "entering if" << endl;
        MyClass b;
        print << "leaving if" << endl;
    }
```

```
    print << "leaving main" << endl;
}
```

If we write "a" and "b" instead of the actual addresses for clarity, the output looks like this:

```
entering main
created instance a
entering if
created instance b
leaving if
destroyed instance b
leaving main
destroyed instance a
```

Derived Classes

`create` and `destroy` also work in derived classes. Let's look at another example.

```
class MyBaseType {
public:
    void destroy() {
        print << "closing down base type" << endl;
    }
}

class MyDerivedType extends MyBaseType {
public:
    void destroy() {
        print << "closing down derived type" << endl;
    }
}

void main()
{
   MyDerivedType type;
}
```

If we run this example, it will output

```
entering main
closing down derived type
closing down base type
```

We get both print outs because the `destroy` method of `MyBaseType` is called automatically, no matter whether you specify a `destroy` method in `MyDerivedType` or not.

2.4.5 Allocating Classes and Memory

Classes and other types can be allocated on the heap dynamically using the `new` and `delete` commands. Basically they work the same way as they do in C++. For example the program

```
void main()
{
    MyClass *c = new MyClass;
    c->doSomething();
    delete c;
}
```

dynamically creates a class of type `MyClass` in the variable c, calls the class method `doSomething`, and then deletes the class again using the `delete` command.

`new` and `delete` can also be used to allocate arrays. As an example the program

```
void main()
{
    int *a = new int[ 200 ];
    doSomething( a );
    delete a;
}
```

creates an array of 200 `int` values, performs some action on the array in the function `doSomething` and then deletes the array again. Note that in Inca there's no `delete[ ]` command like in C++. Everything allocated via `new`, no matter if it's a singular type or an array is disposed of via the same `delete` command in Inca .

When dealing with array of classes, Inca takes care of calling each class instance's constructor upon creation of the array, and each class instance's destructor upon destruction of the array. Note that this might take quite some time for large arrays.

2.4.6 Operator Overloading

Inca gives you the opportunity to defines new types using classes and even to define new operator functionality by overloading operators. For example, assume you have a type `Money` that is defined as

```
class Money {
public:
    int dollars;
    int cents;
}
```

Now, you would like to use instances of the `Money` class just like you use instances of int or float, for example

```
Money a, b, c;
...
c = a + b;
```

What you need to do here is to define a new + operator for `Money`. You can do this in two ways. The first way is to define a global function:

```
Money operator+( Money a, Money b )
{
    Money c;
    int newcents;
    c.dollars = a.dollars + b.dollars;
    newcents = a.cents + b.cents;
    c.dollars += newcents / 100;
    c.cents = newcents % 100;
    return c;
}
```

Using the function above, you can now use the + operator on `Money` objects as indicated above. There's a second possibility of defining a custom + operator: by declaring a member function inside `Money` like this:

```
class Money {
public:
    int dollars;
    int cents;
    Money operator+( Money b )
```

```
{
    Money c;
    int newcents;
        c.dollars = dollars + b.dollars;
        newcents = cents + b.cents;
        c.dollars += newcents / 100;
        c.cents = newcents \% 100;
        return c;
}
}
```

The both ways of declaring operator functions are mostly equivalent. Declaring operator functions inside a class gives you the opportunity to define access constraints (public, private, protected) on the operator, when needed.

Note that for unary operator (i.e. operators taking only one parameter), the syntax is analogous, for example

```
MyClass operator^( MyClass& x )
{ ... }
```

and

```
class MyClass {
    MyClass operator^()
        { ... }
}
```

2.4.7 Templates

Inca provides support for simple template definitions of functions. For example

```
template T void swap( T& a, T& b )
{
    T c = a;
    a = b;
    b = c;
}
```

will declare a `swap` function that can be called with whatever type you want, i.e.

```
int a = 2, b = 5;
float c = -1, d = 10.5;
MyClass e, f;
swap( a, b );
swap( c, d );
swap( e, f );
```

are all valid. Inca will generate the code needed for the functions on demand. You can also declare template functions of several free parameters as in

```
template A, B void myfunc( A a, B b )
```

Note that currently, although Inca supports template functions in classes, Inca does *not* support template classes.

2.5 Scopes

2.5.1 Scopes

When accessing variables, functions, types or constants, there's a certain way, in which Inca tries to find out which item you actually meant. Consider the really evil example

```
int a;

class MyBaseClass {
public:
    int a;
}

class MyClass {
public:
    void f()
    {
        int a;
        print << a;
    }

    int a;
}
```

In the current state, the `print` command in the function `f` in class `MyClass` will print the variable a declared locally in function `f`. Suppose we skip this declaration. What happens now is that Inca recognizes, that there's no local variable, so it looks in the class scope. There it finds the member variable a of the class `MyClass`. Now suppose we skip this declaration too. What happens now is that there are two options, namely the variable a declared in global scope and the one declared in the base class of `MyClass`, namely `MyBaseClass`. /akla always prefers the base class in these situations. The resolution scheme used by Inca is pretty similar to that employed by C++ in the vast majority of cases.

A special case occurs, if more than one base class could supply a member variable of the same name like in

```
class MyFirstBaseClass {
public:
    int a;
}

class MyOtherBaseClass {
public:
    int a;
}

class MyClass extends MyFirstBaseClass, MyOtherBaseClass {
public:
    void f()
    {
        print << a;
    }
}
```

In this situation Inca will issue an error, since it's not really clear whether the programmer referred to the member variable a in `MyFirstBaseClass` or the one in `MyOtherBaseClass`. To resolve the ambiguity in these situations, the `::` operator can be used like this

```
class MyClass extends MyFirstBaseClass, MyOtherBaseClass {
public:
    void f()
    {
        print << MyFirstBaseClass::a;
```

```
    }
}
```

Now Inca knows that the member variable of the class `MyFirstBaseClass` is meant. The mechanism is also applicable to functions and other items.

2.5.2 Function Overloading

Sometimes situations occur where different functions with the same name but different parameter types need to be defined. A very typical example is taken from the Inca Math Libraries:

```
double abs( double x );
long abs( long x );
```

Here the function `abs` is defined for `double` and `long` types, though with presumably different implementations. When calling the function, Inca will check which types you have used in your call and then pick one function that seems to be the one you intended. Therefore `abs( 3 )` will map to the `long` variant, whereas `abs( 3.5 )` will map to the `double` variant. There are cases, in which Inca is not sure which function you intended to use. In these cases, an error will be issued, listing all the possibilitied that Inca can choose from in the current context.

2.5.3 Includes

Inca provides a way to share code between several source code files using `includes`. They are quite similiar to including headers in C or C++. To include another file, write

```
include "<file path>"
```

where `<file path>` is the path of the include file relative to the Inca application. Press return. The Inca editor will now check if it can find the file. If yes, it will append a `<...>` behind the include statement, if not, it will append a `file not found` message. In the latter case, delete the line and try again.

`Includes` are loaded the second you type the declaration in the editor or the moment a source code file is loaded that contains a module. Inca will not track changes in the `include` file right now. If you change something in an `include` file, make sure you close and reopen all the source code files using it.

Once written down, you can change the path of a `include` directive in the editor by pression option-e. The cursor will jump to the path where you can edit it.

2.5.4 Modules and Namespaces

`Modules` in Inca are a way to partition the source code for better readability. A `module` does not have any practical use for the language, it is ignored by the compiler. It is merely to bring structure to the code. See the example source code for how to use `modules`.

Furthermore, Inca has support for `namespaces`. The semantic is the same as in C++. `namespaces` give you the opportunity to segment your code into different areas, that might declare identifiers of same name, like this:

```
namespace MyFirstNamespace {
    void f( int x ) { ... }
}

namespace MyOtherNamespace {
    void f( int x ) { ... }
}

void main()
{
    MyOtherNamespace::f( 10 );
    MyFirstNamespace::f( 5 );
    f( 10 ); // error - ambigouus
}
```

Using namespaces to segment your code, you might want to indicate sometimes that a certain namespace should be searched without having to explicitly qualify it. For example in

```
namespace MyHighlyComplicatedNamespace {
    void f( int x ) {...}
}

void main()
{
    f( x );
}
```

it would be nice, if you could tell Inca that `MyHighlyComplicatedNamespace` is to be searched by default. You can achieve this by means of the `using` statement, which works pretty the same way as in C++:

```
namespace MyHighlyComplicatedNamespace {
    void f( int x ) {...}
}

using namespace MyHighlyComplicatedNamespace;

void main()
{
    f( x );
}
```

Chapter 3

Library Reference

3.1 System Library

3.1.1 Introduction

The system library in Inca is a pool for all the functions that didn't make it to any of the other libraries. That's why you find a set of rather different functions here.

3.1.2 Overview

Date and Time Functions

```
String date()
String time()
long millis()
long micros()
```

System Functions

```
void exit( int code )
void fatal( String error )
void systemTask()
```

Cursor Functions

```
void showCursor()
void hideCursor()
```

Miscellaneous Functions

```
void bmove( const void* from, void* to, int size )
void exec( String path, StringArray params )
void debugger()
```

Dialog Classes

```
ColorChooser
OpenFileChooser
SaveFileChooser
```

3.1.3 Date and Time Functions

`date`

SYNTAX

```
void date()
```

PURPOSE

The `date` functions returns the current date as a string with the format DD/MM/YYYY.

EXAMPLE

```
print << date();
```

OUTPUT

```
01/31/2003
```

`time`

SYNTAX

```
void time()
```

PURPOSE

The `time` functions returns the current time as a string with the format HH:MM:SS.

EXAMPLE

```
print << time();
```

OUTPUT

```
14:20:53
```

```
millis
```

SYNTAX

```
long millis()
```

PURPOSE

The `millis` functions returns the number of milliseconds since the start of the system.

```
micros
```

SYNTAX

```
long micros()
```

PURPOSE

The `micros` functions returns the number of microseconds since the start of the system.

3.1.4 System Functions

`exit`

SYNTAX

```
void exit( int code )
```

PURPOSE

The `exit` functions immediately terminates the running program with the given exit code `code`. Usually you should use either `EXIT_SUCCESS`, if the program terminates correctly and without an error, or `EXIT_FAILURE` if the program terminates due to an error.

`fatal`

SYNTAX

```
void fatal( String error )
```

PURPOSE

By calling the `fatal` function you can indicate that a fatal error has occured and your program can't continue execution. The `String` error should contain a description of the fatal error. If you're running in editor mode, Inca will break at the location you call `fatal` and display the error description you give in the status line of the editor window.

In many cases, it might be a better idea to call `fatal` instead of `exit( EXIT_FAILURE )`, since you can provide an error text here.

`systemTask`

SYNTAX

```
void systemTask()
```

PURPOSE

The `systemTask` function gives time to Windows. For example, if you need to update mouse position, status of the mouse keys, keyboard inputs (generally anything that is related to Windows events), you should call `systemTask`.

3.1.5 Cursor Functions

`showCursor`

SYNTAX

```
void showCursor()
```

PURPOSE

The `showCursor` shows the mouse cursor.

`hideCursor`

SYNTAX

```
void hideCursor()
```

PURPOSE

The `hideCursor` hides the mouse cursor.

3.1.6 Miscellaneous Functions

`bmove`

SYNTAX

```
void bmove( const void* from, void* to, int size )
```

PURPOSE

The `bmove` functions copies a block of memory from one location to another. Specifically, a block of `size` bytes is copied from the address `from` to the address `to`.

`exec`

SYNTAX

```
void exec( String path, StringArray params )
```

PURPOSE

The `exec` function executes a given program. `path` should contain a valid path to the program being executed. `params` should be an array of parameters that are passed to the program. `params` can of course be an empty array.

`debugger`

SYNTAX

```
void debugger()
```

PURPOSE

The `debugger` function calls the debugger and makes the program execution stop at the point of this statement.

3.1.7 Dialog Classes

`ColorChooser`

SYNTAX

```
bool choose( Color& color )
```

PURPOSE

The `ColorChooser` class provides you with a dialog that lets users choose a color. The `choose` method presents the color dialog and returns `true`, if the user chose a color, and `false`, if the user cancelled the dialog. If `true` was returned, `color` contains the chosen color.

EXAMPLE

```
ColorChooser myChooser;
Color myColor;
if( myChooser.choose( myColor ) )
print << "you chose " << myColor;
else
print << "you chose nothing";
```

`OpenFileChooser`

SYNTAX

```
bool choose( String& path )
```

PURPOSE

The `OpenFileChooser` class provides you with a dialog that lets users choose a file to open. The `choose` method presents the open file dialog and returns `true`, if the user chose a file, and `false`, if the user cancelled the dialog. If `true` was returned, `path` contains the path of the chosen file.

EXAMPLE

```
OpenFileChooser myChooser;
String myPath;
if( myChooser.choose( myPath ) )
print << "you chose " << myPath;
else
print << "you chose nothing";
```

SaveFileChooser

SYNTAX

```
bool choose( String& path )
```

PURPOSE

The `SaveFileChooser` class provides you with a dialog that lets users choose a file to save to. The `choose` method presents the save file dialog and returns `true`, if the user chose a file, and `false`, if the user cancelled the dialog. If `true` was returned, `path` contains the path of the chosen file.

EXAMPLE

```
SaveFileChooser myChooser;
String myPath;
if( myChooser.choose( myPath ) )
print << "you chose " << myPath;
else
print << "you chose nothing";
```

3.2 BigInt Library

3.2.1 Introduction

The BigInt Library provides you with a way to do computations with big integers (meaning they don't fit in 32 or 64 bits). This type might be interesting for applications in number theory, prime number testing, cryptology or the like.

A `BigInt` is a built-in type in Inca . It represents a signed integer value and can grow arbitrarily large. The number of bits it uses is allocated dynamicially by Inca . The only limit is the amount of available memory. Note that the computation time to work with these numbers will grow at least linearily with their size, depending on what operations you perform on them.

In most cases, you can use `BigInts` just like normal ints and longs. You can also mix the usage of `BigInts` and regular ints in statements, for example:

```
BigInt a, b;

a = 3;
b = pow( a, 40 );

if( even( b - 17 ) )
    print << "hooray, ( 3 ^ 40 ) - 17 is even" << endl;

a = sqrt( b ) + 3 * a;

print << a;
```

This rather senseless code snippet shows some basic arithmetic operations you can perform on `BigInts`. In line 3, we assign 3 to a. In the next line, we compute a^{40} and assign it to b. Then we check if $b - 17$ is an even value and print a message, if so. Finally, we compute the square root of b and add 3 times a and assign the result to a again, just to print it in the next line. If you're looking for an example with more real world value, you might find the Rabin-Miller probability prime test implementation in the examples folder more interesting.

You can compare `BigInts` using the standard six comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=` just the way you would use them on other arithmetic types. Other allowed operations include addition, subtraction, multiplication, division, modulo and bit shifting via `<<` and `>>`. Currently not supported are bitwise and (`&`), or (`|`), xor (`^`) and bitwise negation (`~`) operators.

In this section most of the examples were omitted, since many of the functions presented here are completely analogous to their pendants in the General Math Library.

3.2.2 Overview

Math Functions

```
BigInt abs( BigInt num )
BigInt bigpow( unsigned long a, unsigned long b )
BigInt modinv( BigInt x, BigInt n )
BigInt pow( BigInt a, BigInt b )
BigInt powmod( BigInt a, BigInt b, BigInt n )
int sgn( BigInt num )
BigInt sqr( BigInt num )
BigInt sqrt( BigInt num )
```

Bit Functions

```
BigInt bchg( BigInt num, int n )
BigInt bclr( BigInt num, int n )
BigInt bset( BigInt num, int n )
bool btst( BigInt num, int n )
```

Divisor Functions

```
BigInt egcd( BigInt a, BigInt b, BigInt& x, BigInt& y )
bool even( BigInt num )
BigInt gcd( BigInt a, BigInt b )
BigInt lcm( BigInt a, BigInt b )
bool odd( BigInt num )
```

String Functions

```
String bin( BigInt num )
String hex( BigInt num )
String oct( BigInt num )
String str( BigInt num )
```

Conversion Functions

```
int bitlen( BigInt num )
long val( BigInt num )
```

Random Functions

```
BigInt random( BigInt n )
```

3.2.3 BigInt Function Reference

`abs`

SYNTAX

```
BigInt abs( BigInt num )
```

PURPOSE

The `abs` function returns the absolute value of a `BigInt`, which is num if num `>=` 0, and -num otherwise.

`bchg`

SYNTAX

```
BigInt bchg( BigInt num, int n )
```

PURPOSE

The `bchg` function returns the given `BigInt` num with the n-th bit flipped, i.e. the bit is cleared if it was set in num, and it's set if it was cleared in num. Bit numbering starts with zero for the least significant bit. Note that this operation ignores the sign of the given `BigInt` num. Different than int and long types, negative `BigInt`s are not represented as two-complement numbers.

`bclr`

SYNTAX

```
BigInt bclr( BigInt num, int n )
```

PURPOSE

The `bchg` function returns the given `BigInt` num with the n-th bit cleared. Bit numbering starts with zero for the least significant bit. Note that this operation ignores the sign of the given `BigInt` num. Different than int and long types, negative `BigInt`s are not represented as two-complement numbers.

`bigpow`

SYNTAX

```
BigInt bigpow( unsigned long a, unsigned long b )
```

PURPOSE

The `bigpow`function computes a^b and returns the result as a `BigInt`.

`bin`

SYNTAX

```
String bin( BigInt num )
```

PURPOSE

The `bin` function generates a signed binary string representation of BigNum. Except for the handling of the sign, this function is analogous to the General Math Library's `bin` function.

`bitlen`

SYNTAX

```
int bitlen( BigInt num )
```

PURPOSE

The `bitlen` function returns the minimum number of bits that are needed to express the absolute value of the given `BigInt` num.

`bset`

SYNTAX

```
BigInt bset( BigInt num, int n )
```

PURPOSE

The `bchg` function returns the given `BigInt` num with the n-th bit set. Bit numbering starts with zero for the least significant bit. Note that this operation ignores the sign of the given `BigInt` num. Different than int and long types, negative `BigInt`s are not represented as two-complement numbers.

`btst`

SYNTAX

```
bool btst( BigInt num, int n )
```

PURPOSE

The `btst` function returns true if the n-th bit is set in the given `BigInt` num and false otherwise. Bit numbering starts with zero for the least significant bit. Note that this operation ignores the sign of the given `BigInt` num. Other than int and long types, negative `BigInt`s are not represented as two-complement numbers.

`egcd`

SYNTAX

```
BigInt egcd( BigInt a, BigInt b,
BigInt& x, BigInt& y )
```

PURPOSE

The `egcd` function (extended gcd) computes the greatest common divisor of a and b, returns the result z, and stores integers (positive, zero or negative) in x and y, such that $a \cdot x + b \cdot y = z$.

`even`

SYNTAX

```
bool even( BigInt num )
```

PURPOSE

The `even` function returns true if num is even, and false otherwise.

`gcd`

SYNTAX

```
BigInt gcd( BigInt a, BigInt b )
```

PURPOSE

The `gcd` function computes the greatest common divisor of a and b and returns the result as a `BigInt`. Special cases are handled the same way as in the `gcd` function of the General Math Library.

`hex`

SYNTAX

```
String hex( BigInt num )
```

PURPOSE

The `hex` function generates a signed hexadecimal string representation of BigNum. Except for the handling of the sign, this function is analogous to the General Math Library's `hex` function.

`lcm`

SYNTAX

```
BigInt lcm( BigInt a, BigInt b )
```

PURPOSE

The `lcm` function returns the least common multiple of a and b.

`modinv`

SYNTAX

```
BigInt modinv( BigInt x, BigInt n )
```

PURPOSE

The `modinv` function returns the inverse to a given number x modulo n. In this group theoretic context, the inverse is defined to be an element y, for which $y \cdot x = 1$ (modulo n), with 1 being the neutral element. If gcd(x, n) is not one, there is no inverse to x and the function will issue an error.

`oct`

SYNTAX

```
String oct( BigInt num )
```

PURPOSE

The `oct` function generates a signed octal string representation of BigNum. Except for the handling of the sign, this function is analogous to the General Math Library's `oct` function.

`odd`

SYNTAX

```
bool odd( BigInt num )
```

PURPOSE

The `odd` function returns true if num is odd, and false otherwise.

`pow`

SYNTAX

```
BigInt pow( BigInt a, BigInt b )
```

PURPOSE

The `pow` function computes a^b and returns the result in a `BigInt`. Note that powers can get very large quite easily.

`powmod`

SYNTAX

```
BigInt powmod( BigInt a, BigInt b, BigInt n )
```

PURPOSE

The `powmod` function computes a^b modulo n and returns the result in a `BigInt`. The advantage of using `pow`(a, b, n) instead of using `pow`(a, b)`%` n is that the result of `pow`(a, b) might be very large, it might take a very long time to compute it and so you might not even get to the point of taking the modulo. The powmod function on the other hand takes care that the intermediate results of the power calculation stay small (smaller than n) during all times.

`random`

SYNTAX

```
BigInt random( BigInt n )
```

PURPOSE

The `random` function returns a nonnegative `BigInt` pseudorandom number in the range [0,n-1]. You can think of this call as returning a "nonnegative pseudorandom number modulo n".

`sgn`

SYNTAX

```
int sgn( BigInt num )
```

PURPOSE

The `sgn` function returns the sign of a Bignum, which is 1, -1 or 0, depending on whether num is `<` 0, `>` 0, or `=` 0.

`str`

SYNTAX

```
String str( BigInt num )
```

PURPOSE

The `str` function returns a decimal string representation of BigNum. This function is analogous to the General Math Library's `str` function.

`sqr`

SYNTAX

```
BigInt sqr( BigInt num )
```

PURPOSE

The `sqr` function returns the square of the given `BigInt` num, i.e. $num \cdot num$.

`sqrt`

SYNTAX

```
BigInt sqrt( BigInt num )
```

PURPOSE

The `sqrt` function returns the integer square root of the given `BigInt` num as `BigInt`.

`val`

SYNTAX

```
long val( BigInt num )
```

PURPOSE

The `val` function tries to return the signed value of the given `BigInt` in a long. If the `BigInt` is small enough, no problems can occur. If the `BigInt` is too large for a long, the maximum negative or positive value representable in a long will be returned, depending on which one of both approximates the actual `BigInt` value better.

3.3 String Library

3.3.1 Introduction

The String Library provides you with a way of easily manipulating strings of characters. A String is a sequence of characters, each character is indexed by its position in the string starting with zero. You can access the characters in a String by using the `[ ]` operator, like you would do it for arrays. The program

```
void main()
{
    String a;

    a = "Richard II";
    print << a[ 0 ] << endl;
    print << a[ 1 ] << endl;

    a[ 9 ] = 'V';
    print << a << endl;
}
```

will output

```
R
i
Richard IV
```

If you try to access a character that does not exist, Inca will come up with an `invalid string index` error. You can concatenate `String`s using the `+` operator. The program

```
void main()
{
    String a;

    a = "Rich";
    a = a + "ard " + "II";
    a += "I";
    print << a;
}
```

will output

```
Richard III
```

`String`s can be compared using the common six comparison operators `==`, `!=`, `<`, `>`, `<=`, `>=`. The comparison works by comparing the ASCII codes of characters at corresponding locations in the involved `String`s. For example to find out the lexigraphical order in which the names Lindgren and Lundgren would be listed in an alphabetical dictionary we could write the program

```
void main()
{
    if( "Lindgren" < "Lundgren" )
        print << "Lindgren, Lundgren";
    else
        print << "Lundgren, Lindgren";
}
```

3.3.2 Overview

Ascii Functions

```
int asc( String s )
String chr( int ascii )
```

Number Functions

```
String bin( unsigned long val )
String hex( unsigned long val )
String oct( unsigned long val )
String str( double val )
double val( String s )
int valn( String s )
```

Modifier Functions

```
String left( String s, [int n] )
String lower( String s )
String mid( String s, int n, [int l] )
String right( String s, [int n] )
String string( int n, String s )
String upper( String s )
```

Search and Query Functions

```
int instr( String s, String t, [int n] )
int grep( String s, String p, [int n], [byte flags] )
int len( String s )
```

3.3.3 String Function Reference

`asc`

SYNTAX

```
int asc( String s )
```

PURPOSE

The `asc` function returns the ASCII code of the first character in the given `String`. If there's no first character because the `String` is empty, 0 will be returned. It's the inverse to the `chr` function.

EXAMPLE

```
print << asc( "d" ) << endl;
print << asc( "dog" ) << endl;
```

OUTPUT

```
100
100
```

`bin`

SYNTAX

```
String bin( unsigned long val )
```

PURPOSE

The `bin` function generates a `String` of the unsigned binary representation of the given numerical value. The `bin` function returns a `String` and not a number.

EXAMPLE

```
print << bin( 1234 ) << endl;
print << bin( 1024 ) << endl;
```

OUTPUT

```
10011010010
10000000000
```

chr

SYNTAX

```
String chr( int ascii )
```

PURPOSE

The `chr` function generates a `String` that contains one character with the ASCII code specified by the ascii parameter. It's the inverse to the `asc` function.

EXAMPLE

```
print << chr( 65 ) << endl;
print << chr( 100 ) << endl;
```

OUTPUT

```
A
d
```

hex

SYNTAX

```
String hex( unsigned long val )
```

PURPOSE

The `hex` function generates a `String` of the unsigned hexadecimal representation of the given numerical value. The `hex` function returns a `String` and not a number.

EXAMPLE

```
print << hex( 1234 ) << endl;
print << hex( 1024 ) << endl;
```

OUTPUT

```
4D2
400
```

grep

SYNTAX

```
int grep( String s, String p,
[int n], [byte flags] )
```

PURPOSE

The `grep` function performs a grep pattern search on the `String` s using the pattern p. It starts at position n, which defaults to 0. Using the flags, the search can be chosen to be either case sensitive (flags is 0, which is default), or case insensitive (flags is 1).

For a description of the syntax of grep patterns, look in any Unix manual on the `grep` utility.

EXAMPLE

```
String a;
a = "and then 5 dogs ran across the docks";
print << grep( a, "[0-9]" );
print << grep( a, "do[abc]" );
```

OUTPUT

```
9
31
```

instr

SYNTAX

```
int instr( String s, String t, [int n] )
```

PURPOSE

The `instr` function look for an occurence of t inside s. More specifically it looks if the sequence of characters of t can be found in exactly the same order in s. If so, the function returns the index of the first character in s matching t. If no match is found, instr will return -1. The optional parameter n tells the instr function where to begin searching.

EXAMPLE

```
String a;
a = "an albatross albeit a small one";
print << instr( a, "alb" );
print << instr( a, "alb", 4 );
```

OUTPUT

```
3
```

```
12
```

left

SYNTAX

```
String left( String s, [int n] )
```

PURPOSE

The `left` function extracts the n leftmost characters of `String` s. If n is greater than the total length of the `String`, the whole `String` is returned. If n is omitted as parameter, only the one leftmost character is extracted, if available.

EXAMPLE

```
String a;
a = "operationalization";
print << left( a ) << endl;
print << left( a, 5 );
```

OUTPUT

```
o
opera
```

len

SYNTAX

```
int len( String s )
```

PURPOSE

The `len` function returns the length of a `String`, which is the number of characters s consists of.

EXAMPLE

```
String a;
a = "inverse square law";
print << len( a );
```

OUTPUT

```
18
```

`lower`

SYNTAX

```
String lower( String s )
```

PURPOSE

The `lower` function makes all characters in a `String` lower case. Actually it doesn't touch the original given `String`, but returns a new `String` that is altered in the according way.

EXAMPLE

```
String a;
a = "Past The Mission";
print << lower( a );
```

OUTPUT

```
past the mission
```

`mid`

SYNTAX

```
String mid( String s, int n, [int l] )
```

PURPOSE

The `mid` function extracts l consecutive characters out of `String` s, starting with the character with index n. If l is omitted, all the characters starting at n up to the end of s are extracted.

EXAMPLE

```
String a;
a = "operationalization";
print << mid( a, 5, 4 ) << endl;
print << mid( a, 10 );
```

OUTPUT

```
tion
lization
```

oct

SYNTAX

```
String oct( unsigned long val )
```

PURPOSE

The `oct` function generates a `String` of the unsigned octal representation of the given numerical value. The oct function returns a `String` and not a number.

EXAMPLE

```
print << oct( 1234 ) << endl;
print << oct( 1024 ) << endl;
```

OUTPUT

```
2322
2000
```

right

SYNTAX

```
String right( String s, [int n] )
```

PURPOSE

The `right` function extracts the n rightmost characters of `String` s. If n is greater than the total length of the `String`, the whole `String` is returned. If n is omitted as parameter, only the one rightmost character is extracted, if available.

EXAMPLE

```
String a;
a = "operationalization";
print << right( a ) << endl;
print << right( a, 5 );
```

OUTPUT

```
n
ation
```

str

SYNTAX

```
String str( double val )
```

PURPOSE

The `str` function generates a `String` that contains the signed decimal representation of the given numerical floating point value. Note that the `str` function returns a string and not a number.

EXAMPLE

```
print << right( str( 1234.5 ), 3 ) << endl;
print << "abc" + str( -310 ) << endl;
```

OUTPUT

```
4.5
abc-310
```

string

SYNTAX

```
String string( int n, String s )
```

PURPOSE

The `string` function sort of multiplies a given `String` s by returning a `String` that consists of n times s.

EXAMPLE

```
print << string( 5, "a" ) << endl;
print << string( 3, "xyzzyx" ) << endl;
```

OUTPUT

```
aaaaa
xyzzyxxyzzyxxyzzyxxyzzyxxyzzyx
```

upper

SYNTAX

```
String upper( String s )
```

PURPOSE

The `upper` function makes all characters in a `String` upper case. Actually it doesn't touch the original given `String`, but returns a new `String` that is altered in the according way.

EXAMPLE

```
String a;
a = "Past The Mission";
print << upper( a );
```

OUTPUT

```
PAST THE MISSION
```

val

SYNTAX

```
double val( String s )
```

PURPOSE

The `val` function returns the numerical value a `String` represents, if any. If the `String` represents no numerical value, 0 is returned. If the `String` represents a numerical value up to a certain point, that value is returned.

EXAMPLE

```
print << val( "-1234.5" ) << endl;
print << val( "reductio ad absurdum" ) << endl;
print << val( "492 pixies set out for 7 moons" ) << endl;
```

OUTPUT

```
-1234.5
0
492
```

valn

SYNTAX

```
int valn( String s )
```

PURPOSE

The `valn` function returns the number of characters in a given `String` that represent a numerical value, starting at the first character. If 0 is returned, this indicates that the string contains no numerical value at the beginning, if a different number is returned, it's the number of characters from the beginning of the `String` that represent a value.

EXAMPLE

```
print << valn( "-1234.5" ) << endl;
print << valn( "reductio ad absurdum" ) << endl;
print << valn( "492 pixies set out for 7 moons" ) << endl;
```

OUTPUT

```
7
0
3
```

3.4 General Math Library

3.4.1 Introduction

Every programming language needs some basic functions that can be found on every pocket calculator, like trigonometry, interpolation, integer bit manipulation or doing calculations with complex number. This chapter describes the library in Inca which provides such kinds of functions.

3.4.2 Caveats

Like the majority of computer software, the floating point math functions in Inca have limited precision. Before you dive into the details of the functions you might want to take a look at some of the caveats you should bear in mind when using them.

Consider the following example.

```
double a = 1e-20;
double b = 1e20;
double c;

c = a;
c += b;
c -= b;

print << a << endl;
print << b << endl;
print << c << endl;
```

The output Inca will give you is

```
1e-20
1e+20
0
```

which is obviously *not* correct. We initialize c with the value of a, and then add b and subtract it again, so c should have the value of a at the end. However it says zero in the print out. The reason for this is that the value of floating point values is representes with limited precision. The moment c holds a + b, i.e. 1e20 + 1e-20, the 1e-20 is so small compared to the dominating 1e20 that it vanishes (to avoid this we would have to have a number precision of 1e-40). This problem comes in many flavours.

Evaluating high degree polynomials or calculating the sinus or cosinus of large angles are two examples. If you're not familiar with these issues, there are quite a number of good texts out there on the numerical background behind floating point number computations in computer systems.

If something like a = b is stated, or something like f(g(a)) = a, where f and g are functions, then what is actually meant is that this identity is fulfilled from a purely mathematical viewpoint (at least for all the valid parameter range of a). Since the same restrictions just mentioned apply here, you should actually read the above statements as a $\approx$ b, and f(g(a)) $\approx$ a. The error here can get very large, actually it can get as large as you want and therefore spoil every result. This is just to say that these functions don't behave as nearly as perfectly as they should with regards to numerical precision, and that calculations can end up producing totally wrong results if no care is taken to avoid precision problems.

3.4.3 The Complex Type

Inca offers a type `Complex` that makes the handling of complex numbers rather easy. You can use it like simple int and float types in most cases. Complex number can be created using the `complex` function. The real and imaginary parts can be read using the `real` and `imag` functions. For example

```
Complex a, b;

a = complex( 1, 2 );
b = complex( -5, 3 );

print << real( a + 2.5 ) << endl;
print << imag( conjugate( a * b ) ) << endl;
print << sqrt( a / b ) << endl;
```

will output

```
3.5
7
(0.454365,-0.420756)
```

Refer to the `Complex` type reference section below for further details.

3.4.4 Constants

Inca predefines the following mathematical constants. π is the ratio of the circumference of a circle to its diameter. e is the euler constant.

Constant Name	Symbolic Value	Arithmetic Value
pi	π	3.14159265358979323 85
pi2	2π	6.28318530717958647 70
pih	$\frac{\pi}{2}$	1.57079632679489661 92
eul	e	2.71828182845904523 54

3.4.5 Overview

General Functions

```
double abs( double x )
double ceil( double x )
double exp( double x )
double floor( double x )
double fract( double x )
double log( double x, [double base] )
double log10( double x )
double pow( double x, double y )
double round( double x, [int n] )
double sgn( double num )
double sqr( double x )
double sqrt( double x )
double toint( double x )
double trunc( double x )
```

Trigonometric Functions

```
double acos( double x )
double asin( double x )
double atan( double x )
double atan2( double y, double x )
double cos( double phi )
double deg( double x )
double rad( double x )
double sin( double phi )
double tan( double phi )
```

Bit Functions

```
long bchg( long x, int n )
long bclr( long x, int n )
long bset( long x, int n )
long btst( long x, int n )
byte rol( byte x, int n )
byte ror( byte x, int n )
short rol( short x, int n )
short ror( short x, int n )
int rol( int x, int n )
```

```
int ror( int x, int n )
long rol( long x, int n )
long ror( long x, int n )
```

Clamp Functions

```
double clamp( double x, double a, double b )
double max( double a, double b )
double min( double a, double b )
```

Interpolation Functions

```
double boxstep( double a, double b, double x )
double cosip( double t, double a, double b )
double cubip( double t, double la, double a, double b, double rb )
double lerp( double t, double a, double b )
double smoothstep( double a, double b, double x )
double spline( float t, int n, const float* knots )
Point spline( float t, int n, const Point* knots )
```

Integer Functions

```
unsigned long comb( int n, int k )
bool even( long x )
long gcd( long x, long y )
bool odd( long x )
```

Random Functions

```
void randomize( long seed )
float rnd()
int random( int n )
```

Hash Functions

```
int hash( float x )
int hash( float x, float y )
int hash( Point p )
int hash( int x, int y )
int hash( int x, int y, int z )
int hash( const void* block, int n )
int hash( const String& s )
```

Complex Functions

```
double abs( Complex )
Complex conjugate( Complex )
Complex imag( Complex )
Complex inverse( Complex )
Complex real( Complex )
Complex sqrt( Complex )
```

3.4.6 Math Function Reference

`abs`

SYNTAX

```
double abs( double x )
```

PURPOSE

The `abs` function returns the absolute value to the given number. If the given number if positive, the same number will be returned. If x is negative, -x will be returned. If x is zero, zero will be returned.

EXAMPLE

```
print << abs( 3.5 ) << endl;
print << abs( -10 ) << endl;
print << abs( 0 ) << endl;
```

OUTPUT

```
3.5
10
0
```

`acos`

SYNTAX

```
double acos( double x )
```

PURPOSE

The `acos` function computes the arcus cosinus of a given value. It is the inverse to the `cos` function. It gives you the angle, that you have to use `cos` on to get x, i.e. `cos( acos( x ) )`

= x. The angle returned is in radians. The call does only make sense for values of x that cos can actually produce, i.e. x in $[-1, 1]$.

EXAMPLE

```
print << acos( 1 ) << endl;
print << acos( -0.1 ) << endl;
print << acos( 0 ) << endl;
```

OUTPUT

```
0
1.67096
1.5708
```

asin

SYNTAX

```
double asin( double x )
```

PURPOSE

The `asin` function computes the arcus sinus of a given value. It is the inverse to the `sin` function. It gives you the angle, that you have to use `sin` on to get x, i.e. `sin`(`asin`(x)) = x. The angle returned is in radians. The call does only make sense for values of x that sin can actually produce, i.e. x in $[-1, 1]$.

EXAMPLE

```
print << asin( 1 ) << endl;
print << asin( -0.1 ) << endl;
print << asin( 0 ) << endl;
```

OUTPUT

```
1.5708
-0.100167
0
```

`atan`

SYNTAX

```
double atan( double x )
```

PURPOSE

The `atan` function computes the arcus tangens of a given value. It is the inverse to the `tan` function. It gives you the angle, that you have to use `tan` on to get x, i.e. `tan( atan( x ) )` = x. The angle returned is in radians and in the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

EXAMPLE

```
print << atan( 1 ) << endl;
print << atan( -0.1 ) << endl;
print << atan( 0 ) << endl;
```

OUTPUT

```
0.785398
-0.0996687
0
```

`atan2`

SYNTAX

```
double atan2( double y, double x )
```

PURPOSE

The `atan2` function computes the arcus tangens of a given location. It is the inverse to the `tan` function, i.e. tan(atan2(y, x)) $= \frac{y}{x}$. Note that different than in the `atan` function with one parameter, the concept of infinity can be expressed safely here by setting x to zero. The other advantage over the one parameter version is that the `atan2` function has the signs of x and y separately. With this additional information, `atan2` can return an angle in the full range $[-\pi, \pi]$.

EXAMPLE

```
print << atan2( 1, -1 ) << endl;
print << atan2( 0, 1 ) << endl;
print << atan2( 1, 0 ) << endl;
print << atan2( -1, 0 ) << endl;
```

OUTPUT

```
2.35619
0
1.5708
-1.5708
```

bchg

SYNTAX

```
long bchg( long x, int n )
```

PURPOSE

The `bchg` function sets bit n in the integer number x and returns the result. Bit numbering starts at zero for the least significant bit.

EXAMPLE

```
print << bchg( 2, 0 ) << endl;
print << bchg( 256, 8 ) << endl;
print << bchg( 1023, 5 ) << endl;
```

OUTPUT

```
3
0
991
```

bclr

SYNTAX

```
long bclr( long x, int n )
```

PURPOSE

The `bclr` function clears bit n in the integer number x and returns the result. Bit numbering starts at zero for the least significant bit.

EXAMPLE

```
print << bclr( 3, 0 ) << endl;
print << bclr( 3, 1 ) << endl;
print << bclr( -1, 0 ) << endl;
```

OUTPUT

```
2
1
-2
```

boxstep

SYNTAX

```
double boxstep( double a, double b, double x )
```

PURPOSE

The `boxstep` function will map the range [a,b] to the range [0,1]. If x is smaller than a, 0 is returned. If x is bigger than b, 1 is returned. If x is in between, $\frac{x-a}{b-a}$ is returned. Note that, different than in the `clamp` function, the interval is specified first here.

EXAMPLE

```
print << boxstep( 0, 10, 7 ) << endl;
print << boxstep( 50, 60, 3 ) << endl;
print << boxstep( -100, 100, -5 ) << endl;
```

OUTPUT

```
0.7
0
0.475
```

bset

SYNTAX

```
long bset( long x, int n )
```

PURPOSE

The `bset` function sets bit n in the integer number x and returns the result. Bit numbering starts at zero for the least significant bit.

EXAMPLE

```
print << bset( 2, 0 ) << endl;
print << bset( 0, 8 ) << endl;
print << bset( 1023, 5 ) << endl;
```

OUTPUT

```
3
256
1023
```

btst

SYNTAX

```
long btst( long x, int n )
```

PURPOSE

The `btst` function tests whether bit n is set in the integer number x, and returns true in case it's set and false otherweise. Bit numbering starts at zero for the least significant bit.

EXAMPLE

```
print << btst( 2, 1 ) << endl;
print << btst( 0, 8 ) << endl;
print << btst( 1023, 5 ) << endl;
```

OUTPUT

```
true
false
true
```

ceil

SYNTAX

```
double ceil( double x )
```

PURPOSE

The `ceil` function returns the smallest integer value, that is not smaller than x.

EXAMPLE

```
print << ceil( 1.5 ) << endl;
print << ceil( -2.3 ) << endl;
print << ceil( 10 ) << endl;
```

OUTPUT

```
2
-2
10
```

clamp

SYNTAX

```
double clamp( double x, double a, double b )
```

PURPOSE

The `clamp` function clamps a given value x to a range [a,b]. If x is bigger than b, b is returned. If x is smaller than a, a is returned. Otherwise, x is returned.

EXAMPLE

```
print << clamp( 0.5, 0, 1 ) << endl;
print << clamp( -123, 2.5, 3 ) << endl;
print << clamp( 1000, 100, 500 ) << endl;
print << clamp( 100.1, 100, 500 ) << endl;
```

OUTPUT

```
0.5
2.5
500
100.1
```

comb

SYNTAX

```
unsigned long comb( int n, int k )
```

PURPOSE

The `comb` function computes the number of possibilites to choose k items from a set of n items. `comb`(n, k) = $\binom{n}{k}$.

EXAMPLE

```
print << comb( 5, 1 ) << endl;
print << comb( 10, 3 ) << endl;
print << comb( 10, 7 ) << endl;
```

OUTPUT

```
5
120
120
```

`cos`

SYNTAX

```
double cos( double phi )
```

PURPOSE

The `cos` function returns the cosinus of the given angle phi. It is the inverse to the `acos` function. The angle phi is supposed to be in radians. However, phi does *not* have to be inside of the range $[0, 2\pi]$.

EXAMPLE

```
print << cos( -pi ) << endl;
print << cos( pih ) << endl;
print << cos( pi2 ) << endl;
print << cos( 1 ) << endl;
```

OUTPUT

```
-1
6.12323e-17
1
0.540302
```

`cosip`

SYNTAX

```
double cosip( double t, double a, double b )
```

PURPOSE

The `cosip` function computes a cosinus interpolation between two values a and b, using the interpolation parameter t. More specifically, the domain [a,b] is mapped to the [0,1] range of t. If t is 0, a is returned. If t is 1, b is returned. If t is in between, $a \cdot (1 - \theta) + b \cdot \theta$ is returned, with $\theta = (1 - cos(x \cdot \pi)) \cdot 0.5$.

EXAMPLE

```
print << cosip( 0.1, 0, 10 ) << endl;
print << cosip( 0.85, 50, 60 ) << endl;
print << cosip( 0.25, 50, 60 ) << endl;
```

OUTPUT

```
0.244717
59.455
51.4645
```

`cubip`

SYNTAX

```
double cubip( double t, double la, double a, double
b, double rb )
```

PURPOSE

The `cubip` function computes a cubic interpolation between two values a and b, using the interpolation parameter t. An additional value left of a and right of b are needed for interpolation. The domain [a,b] is mapped to the [0,1] range of t. If t is 0, a is returned. If t is 1, b is returned. If t is in between, the cubic interpolation is returned. If t is outside [0,1] extrapolation takes place.

EXAMPLE

```
print << cubip( 0.5, 0, 10, 50, 100 ) << endl;
print << cubip( 0.95, 0, 10, 50, 100 ) << endl;
```

```
print << cubip( 1.5, 0, 10, 50, 100 ) << endl;
```

OUTPUT

```
25
33.4375
130
```

deg

SYNTAX

```
double deg( double x )
```

PURPOSE

The `deg` function converts an angle from radians to degrees. It's a short form for $\frac{x \cdot 180}{\pi}$.

EXAMPLE

```
print << deg( pi ) << endl;
print << deg( pih ) << endl;
print << deg( -pi / 4 ) << endl;
```

OUTPUT

```
180
90
-45
```

even

SYNTAX

```
bool even( long x )
```

PURPOSE

The `even` function returns true if the given number is even and false otherwise. Zero is regarded as even in this context.

EXAMPLE

```
print << even( 3 ) << endl;
print << even( 0 ) << endl;
print << even( -4 ) << endl;
```

OUTPUT

```
false
true
true
```

exp

SYNTAX

```
double exp( double x )
```

PURPOSE

The `exp` function returns e^x, with e being the euler constant. The `exp` function is the inverse to the `log` function calculating the natural logarithm. In theory, this relation satisfies `log( exp(` x `) )` = x.

EXAMPLE

```
print << exp( 0.5 ) << endl;
print << exp( -1.2 ) << endl;
print << exp( 3.1 ) << endl;
```

OUTPUT

```
1.64872
0.301194
22.198
```

floor

SYNTAX

```
double floor( double x )
```

PURPOSE

The `floor` function returns the biggest integer value, that is not bigger than x.

EXAMPLE

```
print << floor( 1.5 ) << endl;
print << floor( -2.3 ) << endl;
print << floor( 10 ) << endl;
```

OUTPUT

```
1
-3
10
```

fract

SYNTAX

```
double fract( double x )
```

PURPOSE

The `fract` function returns the fractional part of a number. The returned fraction will have the number's sign (except when the fraction was zero). The relation between the `fract` and the closely related `trunc` function is `fract`(x) = x - `trunc`(x).

EXAMPLE

```
print << fract( 3.5 ) << endl;
print << fract( -2.3 ) << endl;
print << fract( -5 ) << endl;
```

OUTPUT

```
0.5
-0.3
0
```

gcd

SYNTAX

```
long gcd( long x, long y )
```

PURPOSE

The `gcd` function computes the greatest common divisor of x and y. It tries to handle negative and zero parameters gracefully.

EXAMPLE

```
print << gcd( 14, 7 ) << endl;
print << gcd( 29, 51 ) << endl;
print << gcd( -100, 50 ) << endl;
print << gcd( 0, 1234 ) << endl;
print << gcd( 3456, 0 ) << endl;
```

OUTPUT

```
7
1
50
1234
3456
```

hash

SYNTAX

```
int hash( float x )
int hash( float x, float y )
int hash( Point p )
int hash( int x, int y )
int hash( int x, int y, int z )
int hash( const void* block, int n )
int hash( const String& s )
```

PURPOSE

The `hash` function returns an integer hash value for some entity. The first five prototypes hash one-, two- or three-dimensional points with floating point or integer coordinates.

The last two prototypes calculate a hash value for the given memory block having a size of n bytes or for a given `String`.

lerp

SYNTAX

```
double lerp( double t, double a, double b )
```

PURPOSE

The `lerp` function interpolates linearly between two values a and b, using the interpolation parameter t. More specifically, the domain [a,b] is mapped to the [0,1] range of t. If t is 0, a is returned. If t is 1, b is returned. If t is in between, $a + t \cdot (b - a)$ is returned. If t is outside [0,1] you're actually extrapolating.

EXAMPLE

```
print << lerp( 0.1, 0, 10 ) << endl;
print << lerp( 0.85, 50, 60 ) << endl;
print << lerp( -1, 50, 60 ) << endl;
```

OUTPUT

```
1
58.5
40
```

log

SYNTAX

```
double log( double x, [double base] )
```

PURPOSE

The `log` function computes the logarithm of x to the given base. If the base parameter is omitted, the natural logarithm to the base e is computed, where e is the euler constant. For the case that the base is e, this function is the inverse to the `exp` function.

EXAMPLE

```
print << log( 10, 27 ) << endl;
print << log( 8, 8 ) << endl;
print << log( 1.5 ) << endl;
print << log( eul ) << endl;
```

OUTPUT

```
0.698634
1
0.405465
1
```

`log10`

SYNTAX

```
double log10( double x )
```

PURPOSE

The `log10` function computes the logarithm to the base 10.

EXAMPLE

```
print << log10( 1.5 ) << endl;
print << log10( 10 ) << endl;
```

OUTPUT

```
0.176091
1
```

`max`

SYNTAX

```
long max( long a, long b )
double max( double a, double b )
```

PURPOSE

The `max` function returns the bigger of the given arguments, i.e. a if a ¿ b, and b if b ¿ a.

EXAMPLE

```
print << max( 10, 4 ) << endl;
print << max( -5.5, 100 ) << endl;
```

OUTPUT

```
10
100
```

`min`

SYNTAX

```
long min( long a, long b )
double min( double a, double b )
```

PURPOSE

The `min` function returns the smaller of the given arguments, i.e. a if a ¡ b, and b if b ¡ a.

EXAMPLE

```
print << min( 10, 4 ) << endl;
print << min( -5.5, 100 ) << endl;
```

OUTPUT

```
4
-5.5
```

`odd`

SYNTAX

```
bool odd( long x )
```

PURPOSE

The `odd` function returns true if the given number is odd and false otherwise. Zero is regarded as even in this context.

EXAMPLE

```
print << odd( 3 ) << endl;
print << odd( 0 ) << endl;
print << odd( -4 ) << endl;
```

OUTPUT

```
true
false
false
```

`pow`

SYNTAX

```
double pow( double x, double y )
```

PURPOSE

The `pow` function computes x^y and returns the result.

EXAMPLE

```
print << pow( 2, 8 ) << endl;
print << pow( 10, -0.5 ) << endl;
print << pow( eul, log( 1 ) ) << endl;
```

OUTPUT

```
256
0.316228
1
```

rad

SYNTAX

```
double rad( double x )
```

PURPOSE

The `rad` function converts an angle from degrees to radians. It's a short form for $\frac{x \cdot \pi}{180}$.

EXAMPLE

```
print << rad( 180 ) << endl;
print << rad( 90 ) << endl;
print << rad( -45 ) << endl;
```

OUTPUT

```
3.14159
1.5708
-0.785398
```

random

SYNTAX

```
int random( int n )
```

PURPOSE

The `random` returns an unsigned pseudorandom integer value in the range [0,n-1]. You can think of this call as returning a "nonnegative pseudorandom integer value modulo n".

`randomize`

SYNTAX

```
void randomize( long seed )
```

PURPOSE

The `randomize` initializes the random number generator with a given seed. You normally don't have to call this function since the random number generator is initialized automatically at every program start. However, if you want a certain sequence of random numbers to be generated again and again for debugging purposes for example, you can use `randomize` with a fixed seed value.

`rnd`

SYNTAX

```
float rnd
```

PURPOSE

The `rnd` returns a nonnegative pseudorandom floating point number in the range [0,1].

`rol`

SYNTAX

```
int rol( int x, int n )
```

PURPOSE

The `rol` function rolls the given integer value x by n bits to the left (towards higher significant bits). "Rolling" means that bits that disappear on one end of the bit string will appear on the other end (instead of supplying zero bits). The `rol` function is defined for the byte, short, int and long integer types. Accordingly it will roll a bit string that is 8, 16, 32 or 64 bit wide.

EXAMPLE

```
print << rol( 8, 2 ) << endl;
print << rol( 1 + 8, 1 ) << endl;
print << rol( 1 + 8, 4 ) << endl;
```

OUTPUT

```
32
18
144
```

ror

SYNTAX

```
int ror( int x, int n )
```

PURPOSE

The `ror` function rolls the given integer value x by n bits to the right (towards lower significant bits). "Rolling" means that bits that disappear on one end of the bit string will appear on the other end (instead of supplying zero bits). The `ror` function is defined for the byte, short, int and long integer types. Accordingly it will roll a bit string that is 8, 16, 32 or 64 bit wide.

EXAMPLE

```
print << hex( ror( 8, 2 ) ) << endl;
print << hex( ror( (long)( 1 + 8 ), 1 ) ) << endl;
print << hex( ror( (long)( 1 + 8 ), 4 ) ) << endl;
```

OUTPUT

```
2
8000000000000004
9000000000000000
```

round

SYNTAX

```
double round( double x, [int n] )
```

PURPOSE

The `round` function rounds a given value. The optional parameter n specifies to how many digits after the decimal point the value should be rounded.If no parameter n is specified the number is rounded to an integer value. If n is negative, the number is rounded to -n numbers before the decimal point.

EXAMPLE

```
print << round( 3.3 ) << endl;
print << round( 3.5 ) << endl;
print << round( -2.7 ) << endl;
print << round( 1.238, 2 ) << endl;
print << round( 12345, -2 ) << endl;
```

OUTPUT

```
3
```

```
4
-3
1.24
12300
```

sgn

SYNTAX

```
double sgn( double num )
```

PURPOSE

The `sgn` function returns one of the value (1,-1,0) to represent the sign of the given number. Specifically, if the given number if positive, sgn will return 1. If the number is negative, sgn will return 0. If the number is zero, sgn will return 0.

EXAMPLE

```
print << sgn( 3.5 ) << endl;
print << sgn( -10 ) << endl;
print << sgn( 0 ) << endl;
```

OUTPUT

```
1
-1
0
```

sin

SYNTAX

```
double sin( double phi )
```

PURPOSE

The `sin` function returns the sinus of the given angle phi. This function is the inverse to the `asin` function. The angle phi is supposed to be in radians. However, phi does *not* have to be inside of the range $[0, 2\pi]$. For small ϕ, $sin(\phi) \approx \phi$.

EXAMPLE

```
print << sin( -pi ) << endl;
print << sin( pih ) << endl;
print << sin( pi2 ) << endl;
print << sin( 1 ) << endl;
```

OUTPUT

```
-1.22465e-16
1
-1.3311e-15
0.841471
```

smoothstep

SYNTAX

```
double smoothstep( double a, double b, double x )
```

PURPOSE

The `smoothstep` function will map the range [a,b] to the range [0,1] using hermite interpolation. If x is smaller than a, 0 is returned. If x is bigger than b, 1 is returned. If x is in between, $y \cdot y \cdot (3 - (y + y))$ is returned, with y being the x normalized to the range [0,1], i.e. $y = \frac{x-a}{b-a}$. Note that, different than in the clamp function, the interval is specified first here.

EXAMPLE

```
print << smoothstep( 0, 10, 7 ) << endl;
print << smoothstep( 50, 60, 3 ) << endl;
print << smoothstep( -100, 100, -5 ) << endl;
```

OUTPUT

```
0.784
0
0.462531
```

spline

SYNTAX

```
double spline( float t, int n, const float* knots )
Point spline( float t, int n, const Point* knots )
```

PURPOSE

The `spline` function evaluates a Catmull-Rom spline that passes through n values specified in the knots array. The parameter t must be in the range [0,1], regardless how big n is. The minimum number of knots for a valid spline is 4, if you try to evaluate a spline with less knots an error will occur. If t is 0, knots[1] will be returned. If t is 1, knots[n - 2] will be returned.

The two prototypes given here operate on knots that are `float` or `Point` values and return the appropiate type.

EXAMPLE

```
static const float knots[ 5 ] = { -25, 10, 15, 50, 60 };
print << spline( 0.15, 5, knots ) << endl;
print << spline( 0.7, 5, knots ) << endl;
```

OUTPUT

```
12.76
28.04
```

sqr

SYNTAX

```
double sqr( double x )
```

PURPOSE

The `sqr` function calculates the square of x, that is $x \cdot x$.

EXAMPLE

```
print << sqr( 2 ) << endl;
print << sqr( -2 ) << endl;
print << sqr( 10.5 ) << endl;
```

OUTPUT

```
4
4
110.25
```

sqrt

SYNTAX

```
double sqrt( double x )
```

PURPOSE

The `sqrt` function returns the square root of the given number.

EXAMPLE

```
print << sqrt( 2 ) << endl;
print << sqrt( 100 ) << endl;
print << sqrt( 1234 ) << endl;
```

OUTPUT

```
1.41421
10
35.1283
```

`tan`

SYNTAX

```
double tan( double phi )
```

PURPOSE

The `tan` function returns the tangens of the given angle phi (ϕ). $tan(\phi) = \frac{sin(\phi)}{cos(\phi)}$. That's why $tan(\phi)$ will tend to go to infinity as $cos(\phi)$ goes to zero. This function is the inverse to the atan function. The angle phi is supposed to be in radians. However, phi does *not* have to be inside of the range $[0, 2\pi]$. For small ϕ, $tan(\phi) \approx \phi$.

EXAMPLE

```
print << tan( 0.1 ) << endl;
print << tan( 1 ) << endl;
```

OUTPUT

```
0.100335
1.55741
```

```
toint
```

SYNTAX

```
double toint( double x )
```

PURPOSE

The `toint` function returns an integer value near x. If x is nonnegative, the result is `trunc`(x). If x is negative, the number returned is `trunc`(x) - 1.

EXAMPLE

```
print << toint( 4 ) << endl;
print << toint( 3.5 ) << endl;
print << toint( -2.3 ) << endl;
print << toint( -5 ) << endl;
```

OUTPUT

```
4
3
-3
6
```

```
trunc
```

SYNTAX

```
double trunc( double x )
```

PURPOSE

The `trunc` function simply removes the fractional part from a number, ignoring the number's sign. Note that for negative parameters x, the resulting number is *not* smaller but bigger or equal compared to x. The closely related function `fract` returns the fraction of a number, and `fract`(x) = x - `trunc`(x).

EXAMPLE

```
print << trunc( 3.5 ) << endl;
print << trunc( -2.3 ) << endl;
print << trunc( -5 ) << endl;
```

OUTPUT

```
3
-2
-5
```

3.4.7 Complex Function Reference

`abs`

SYNTAX

```
double abs( Complex x )
```

PURPOSE

The `abs` function returns the absolute value of the given `Complex` number x.

`conjugate`

SYNTAX

```
Complex conjugate( Complex x )
```

PURPOSE

The `conjugate` function returns the conjugate to the given `Complex` number x.

`imag`

SYNTAX

```
double imag( Complex x )
```

PURPOSE

The `imag` function returns the imaginary part of the given `Complex` number x.

`inverse`

SYNTAX

```
Complex inverse( Complex x )
```

PURPOSE

The `inverse` function returns the inverse to the given `Complex` number x.

`real`

SYNTAX

```
double real( Complex x )
```

PURPOSE

The `real` function returns the real part of the given `Complex` number x.

`sqrt`

SYNTAX

```
Complex sqrt( Complex x )
```

PURPOSE

The `sqrt` function returns the square root of the given `Complex` number x.

3.5 IO Library

Print and Input

The IO library is one of the most fundamental libraries in Inca . It provides for input and output from and to the text console and files. As already seen in the introduction to this manual, the text console is accessed via the `print` and aclinput statements. Actually they're not statements but instances of classes.

`print` and aclinput work pretty straightforward. To print out an text, you simply send the text to the print console:

```
print << "hello world!";
```

To print more than one item, you simply concatenate the items you wish to print by `<<`:

```
int i = 7;
print << "currently, i = " << i << " !!!";
```

which prints

```
currently, i = 7 !!!
```

`print` is an instance of an `OutputStream`, broadcasting everything it receives to the text console. The types that can be sent via `<<` to an `OutputStream` are

```
void*
bool
char
int
long
float
double
String
BigInt
Complex
Point
Vector
Color
```

For example, to print a `Vector`, you simply send it to the `print` stream via

```
Vector v = vector( 1, 2, -1 );
print << v;
```

and obtain as output:

```
(1,2,-1)
```

On the other hand `input` is an instance of an `InputStream`, which reads data from the text console. The types which can be read via `>>` from an `InputStream` are

```
char
int
long
float
double
String
BigInt
```

For example, to read a `String` from the text console, you would use

```
String s;
input >> s;
```

Because `print` and `input` are instances of classes, you can actually call methods that are defined for these classes, like in

```
print.flush();
```

which flushes the text console output and will bring everything not currently displayed on the screen immediately. For more information on the methods of the `OutputStream` and `InputStream` classes, refer to the sections below.

Printing to the text console via `print` further offers several special features, like setting the printing color and positioning the cursor. These special commands are simply embedded into the printing stream like in

```
print << at( 5, 10 );
print << forecolor( color( "red" ) );
print << "hi there";
```

which prints a red colored `hi there` at column 5 and row 10.

Let's give a short overview over formatting functions. `crscol` and `crslin` return the column and row of the current print cursor location respectively. `htab` and `vtab` do exactly the opposite. They manipulate the column or row location of the cursor. The `at` function is a composite form for manipulation both the column and the row at once, and is embedded in the `print` stream. Finally, `forecolor` and `backcolor` can also be embedded into a `print` stream to change the foreground and background color of the text subsequently printed.

Files

`OutputStream`s and `InputStream`s are also used for binary input and output as in files. Actually a `File` in Inca is defined as

```
class File extends InputStream, OutputStream { ... }
```

Reading or writing binary data from a file is rather easy. For example the program

```
File myFile;

myFile.openForReading( "MyFiles/SomeTestFile.dat" );
print << myFile.readWord();
myFile.close();
```

will open the file "SomeTestFile.dat" in the directory "MyFiles", read a binary 16-bit signed word, output it as readable number to the text console, and then close the file again. Path specifications in Inca are relative to to the place where the source code file containing the program is located, except they start with an "/", which means they're absolute.

The names given to the binary data types in the stream classes below differ from the built-in types in Inca . This is to make it more clear, which specific type with which specific bit width is actually written to a stream. If a function is called something like `readInt`, it might be not clear to many people with background on several platforms, if this means to read a 16-bit or a 32-bit integer for example. That's why the corresponding function in Inca is called `readQuad`, which means a 32-bit integer - quad is intended to be a short form for quadruple, i.e. four bytes. The names used in the stream classes are listed in the table below for clarity. All the types are assumed to be signed by default. There are currently no separate reading or writing functions for unsigned types. You have to typecast them accordingly.

Stream Type Name	Bits	Language Type Name
char	8	char
bool	8	bool
byte	8	byte
word	16	short
quad	32	int
octa	64	long
float	32	float
double	64	double

The endianness of streams in Inca is always big endian no matter what system you're on. As long as you use the functions properly, you always receive binary data read from a stream, as if you were running on a big endian machine (even if you're actually running on a little endian machine), and the data you write to a stream will always look, as if it has been written by a big endian machine.

File System Services

Inca offers ways to query and manipulate the directory and file structure of the system you're running on.

First of all, there's a number of directory functions. You can get the current working directory via the `currentDir` function. You can set the current working directory via the `changeDir` function. Directories can be created and deleted via the `createDir` and `deleteDir` functions. To find out how much free space there's on a disk, you can use the `diskFree` function.

To iterate through all files that are located in a specifc directory, you can use the `FileIterator` class, which iterates through all files of the current working directory. For example

```
FileIterator it;
while( it.next() )
    print << it.name() << endl;
```

will print the names of all files in the current working directory.

Files can be copied or moved using the `copyFile` and `moveFile` functions. Files can be deleted via the `deleteFile` function. You can check if a file at a certain path exists using the `fileExists` function.

Keyboard and Mouse

In Inca , to get information of keyboard and mouse status, two groups of functions are available. The first group consists of:

```
int mousex()
int mousey()
int mousez()
bool mousek( [int k] )
int waitmouse()
String inkey()
String waitkey()
```

These functions allow you to access the location of the mouse relative to a graphics console window, check the status of mouse buttons, or retrieve keyboard input. For example, the BASIC-like `inkey` function returns a `String` containing the pressed key since the last check, if any. The `mousex` and `mousey` report the x and y location of the mouse cursor. The `mousek` function reports the status of a specific mouse button indexed by the parameter k (defaults to the left mouse button). The following example (which assumes that a console window is open!) shows how to use the functions:

```
// loop while the left mouse button is not pressed
while( mousek( 0 ) == false )
{
    print << "mouse is at (" << mousex() << ","
        << mousey() << ")" << endl;
    if( mousek( 1 ) )
    {
        print << "right mouse button is pressed!";
        print << endl;
    }
    String s = inkey();
    if( s != "" )
        print << "key was pressed - ASCII code: "
            << asc( s ) << endl;

    // since no console swap happens here, we
    // have to manually give time to the system
    // to update keyboard and mouse status
    systemTask();
}
```

For many purposes it makes sense to use the functions just described. They're internally implemented by checking Windows event messages on the console windows. Therefore, the status of mouse and keyboard these functions report only gets updated if a graphics console performs a `swap` operation (see the documentation there), or if you explicitly call `systemTask()`.

For purposes where time is critical and you need poll mouse or keyboard status with very high frequencies, without wanting to give control to the operating system each time, the following functions exist:

```
int diMousex()
int diMousey()
int diMousez()
bool diMousek( [int k] )
bool vkeyDown( int keyCode )
void diUpdate()
```

All these functions are implemented via DirectInput. They even work if Inca is running in real-time mode, and they provide similar functionality as the regular mouse and keyboard functions.

For real-time mode, the `diUpdate()` function exists, which updates the status these functions report with minimal overhead. For normal applications, you won't need to call the `diUpdate()` function.

Data Compression

Inca offers two special streams that provide services for lossless data compression and decompression, based on compression techniques very similar to those used in zip files. The streams are called `InflaterStream` and `DeflaterStream` and t are plugged "on top" of another stream and compress or decompress incoming or outgoing bit streams. `InflaterStream` is a special kind of `InputStream`, whereas `DeflaterStream` is derived from `OutputStream`.

To write compressed data into a file, for example, you open a `File` stream and plug an `DeflaterStream` on top of it:

```
File myFile;
DeflaterStream myDeflater;

myFile.openForWriting( "CompressedFile.dat" );
myDeflater.attach( &myFile );
myDeflater.writeString( "Hello World!" );
```

What happens now, if you write to the `DeflaterStream` via `writeString` is that, the `DeflaterStream` compresses the `String` and writes the compressed data to the stream that was `attached` to it (`myFile` in this case).

Decompressing data from a compressed stream that was written using a `DeflaterStream` works the other way round, this time using the `InflaterStream`:

```
File myFile;
InflaterStream myInflater;

myFile.openForReading( "CompressedFile.dat" );
myInflater.attach( &myFile, myFile.size() );
print << myInflater.readString();
```

The second argument to the `InflaterStream`'s `attach` function is the number of bytes that are available from the stream for decompression. Since no other data was saved in the `File` except the compressed data, this is the file size here.

Data compression services in Inca are implemented on top of the zlib library, Copyright (C) 1995-2002 Jean-loup Gailly and Mark Adler.

Memory Buffers

Inca offers one special stream that can be used to efficiently hold data that can grow dynamically in size. It's called `MemoryBuffer`, and you can write to and read from it:

```
class MemoryBuffer extends InputStream, OutputStream { ... }
```

A `MemoryBuffer` is a chunk of memory that dynamically grows in size as you write data to it. For example the piece of code

```
MemoryBuffer b;
b.writeShort( 1 );
b.writeInt( 10 );
for( int i = 0; i < 1000; i++ )
    b.writeDouble( i * 3 );
print << b.size();
```

will output

```
4006
```

indicating that the MemoryBuffer is 4006 bytes large at the time of printing. There's no maximum size limit for how large a MemoryBuffer can grow (except for the amount of available main memory of course).

You can also seek and read data from a MemoryBuffer. For example

```
MemoryBuffer b;
b.writeShort( 1 );
b.writeInt( 10 );
for( int i = 0; i < 1000; i++ )
    b.writeDouble( i * 3 );
b.seekAbsolute( 2 );
print << b.readInt();
```

will output

```
10
```

as this was the integer value, that was written at location two (the first short value written takes up two bytes).

MemoryBuffers provide a versatile way of storing and retrieving data that can grow dynamically in size, yet they are much faster than files.

3.5.1 Overview

Printing Functions

```
int crscol()
int crslin()
void htab( int column )
void vtab( int row )
String at( int column, int row )
String forecolor( Color color )
String backcolor( Color color )
```

InputStream Methods

```
char readChar()
bool readBool()
int readWord()
int readQuad()
long readOcta()
float readFloat()
double readDouble()
String readString()
void* readPointer()
void readBytes( byte* block, int count )
```

OutputStream Methods

```
void flush()
void writeChar( char c )
void writeBool( bool b )
void writeWord( int value )
void writeQuad( int value )
void writeOcta( long value )
void writeFloat( float value )
void writeDouble( double value )
void writeString( String s )
void writePointer( void* p )
void writeBytes( const byte* block, int count )
```

File Methods

```
void close()
```

```
void openForReading( String path )
void openForWriting( String path )
void seekAbsolute( long position )
void seekFromEnd( long position )
void seekRelative( long offset )
int size()
int tell()
```

File System Functions

```
void changeDir( String path )
void copyFile( String from, String to )
void createDir( String path )
String currentDir()
void deleteDir( String path )
long diskFree( String path )
void fileExists( String path )
void moveFile( String from, String to )
```

FileIterator Methods

```
void setFilter( String filter )
bool next()
String name()
```

InflaterStream Methods

```
void attach( InputStream* stream, int size )
```

DeflaterStream Methods

```
void attach( OutputStream* stream, [int level] )
```

Keyboard and Mouse Functions

```
bool diMousek( [int k] )
int diMousex()
int diMousex()
int diMousex()
void diUpdate()
String inkey()
bool mousek( [int k] )
```

```
int mousex()
int mousey()
int mousez()
bool vkeyDown( int keyCode )
String waitkey()
int waitmouse()
```

3.5.2 File System Services Reference

`changeDir`

SYNTAX

```
void changeDir( String path )
```

PURPOSE

The `changeDir` function changes the current working directory to the given path, which may be either absolute or relative.

`copyFile`

SYNTAX

```
void copyFile( String from, String to )
```

PURPOSE

The `copyFile` function copies the file that's located at the path specified by `from` to the path specified by `to`. If the function suceeds, there will be two instances of the file, one at the path given by `from`, and one at the path given by `to`. The paths given to this function can be either relative or absolute.

`createDir`

SYNTAX

```
void createDir( String path )
```

PURPOSE

The `createDir` function creates a new directory as specified by the given path, which may be either absolute or relative.

currentDir

SYNTAX

```
String currentDir()
```

PURPOSE

The `currentDir` returns the full absolute path of the current working directory (without the last slash).

EXAMPLE

```
print << currentDir();
```

OUTPUT

```
C:\PROGRAMS\MYAPP
```

deleteDir

SYNTAX

```
void deleteDir( String path )
```

PURPOSE

The `deleteDir` function deletes an existing directory as specified by the given path, which may be either absolute or relative.

deleteFile

SYNTAX

```
void deleteFile( String path )
```

PURPOSE

The `deleteFile` function deletes the file that's located at the path given by `path`. The path given to this function can be either relative or absolute.

EXAMPLE

```
deleteFile( "C:\file_to_be_deleted.txt" );
```

diskFree

SYNTAX

```
long diskFree( String path )
```

PURPOSE

The `diskFree` function returns the number of free bytes on the disk that's specified via `path`. Here, `path` should only consist of a volume name and a trailing backslash.

EXAMPLE

```
print << diskFree( "C:\\" ) /
    ( 1024 * 1024 ) << " MB";
```

OUTPUT

```
70401 MB
```

fileExists

SYNTAX

```
void fileExists( String path )
```

PURPOSE

The `fileExists` function checks whether a file exists that's located at the path given by `path`. Is a file exists there, true is returned. If no file exists there, false is returned. The path given to this function can be either relative or absolute.

EXAMPLE

```
print << fileExists( "C:\myfile.txt" );
```

OUTPUT

```
true
```

moveFile

SYNTAX

```
void moveFile( String from, String to )
```

PURPOSE

The `moveFile` function moves the file that's located at the path specified by `from` to the path specified by `to`. Note that the `moveFile` will only work, if the source and destination path are on the same volume. If the function suceeds, there will be one instance of the file at the path given by `to`. The paths given to this function can be either relative or absolute.

3.5.3 FileIterator Methods

`setFilter`

SYNTAX

```
void setFilter( String filter )
```

PURPOSE

The `setFilter` function sets a filter for a `FilterIterator`. You should call this function before the first call to `next`. The given filter `filter` is a filter (can be wildcarded) for the names of files. It is applied such that this `FileIterator` will only iterate over files which name matches the given filter `filter`.

`next`

SYNTAX

```
bool next()
```

PURPOSE

The `next` function iterates to the next file in the group of files this `FileIterator` iterates over. If there's no such file, `next` returns `false`. If there's a file left, `next` return `true`.

`name`

SYNTAX

```
String name()
```

PURPOSE

The `name` function returns the name of the file this `FileIterator` currently points to.

3.5.4 File Method Reference

The `File` class is derived from the `InputStream` and the `OutputStream` class. That's why the `File` class has all the methods of both of them plus the following ones listed in this section.

`close`

SYNTAX

```
void close()
```

PURPOSE

The `close` function closes the file currently opened by this `File` class. If no file is opened, nothing happens.

`openForReading`

SYNTAX

```
void openForReading( String path )
```

PURPOSE

The `openForReading` function opens the file at the location specified via `path` for reading. Subsequent operations on the `File` class will be performed on this file.

`openForUpdating`

SYNTAX

```
void openForWriting( String path )
```

PURPOSE

The `openForUpdating` function opens the file at the location specified via `path` for updating, which is reading and writing. Subsequent operations on the `File` class will be performed on this file.

`openForWriting`

SYNTAX

```
void openForWriting( String path )
```

PURPOSE

The `openForWriting` function opens the file at the location specified via `path` for writing. Subsequent operations on the `File` class will be performed on this file.

`seekAbsolute`

SYNTAX

```
void seekAbsolute( long position )
```

PURPOSE

The `seekAbsolute` function moves the file marker of the currently open file to aclposition bytes after the beginning of the file. A position of zero corresponds to the beginning of the file. The file marker is the position in the file where reading and writing takes place. If no file is opened, an error will occur.

`seekFromEnd`

SYNTAX

```
void seekFromEnd( long position )
```

PURPOSE

The `seekFromEnd` function moves the file marker of the currently open file to `position` bytes before the end of the file. A position of zero corresponds to the end of the file. The file marker is the position in the file where reading and writing takes place. If no file is opened, an error will occur.

`seekRelative`

SYNTAX

```
void seekRelative( long offset )
```

PURPOSE

The `seekRelative` function moves the file marker of the currently open file relative to its current position by `offset` bytes. An offset of zero has no offset. If no file is opened, an error will occur.

`size`

SYNTAX

```
int size()
```

PURPOSE

The `size` function returns the total size of the file currently held open by this `File` class in bytes. If no file is opened, an error will occur.

`tell`

SYNTAX

```
int tell()
```

PURPOSE

The `tell` function returns the absolute position of the file marker, where reading and writing takes place. If no file is opened, an error will occur.

3.5.5 InputStream Method Reference

`readChar`

SYNTAX

```
char readChar()
```

PURPOSE

The `readChar` function reads a binary 8-bit character from this `InputStream`.

`readBool`

SYNTAX

```
bool readBool()
```

PURPOSE

The `readBool` function reads a binary boolean value by reading a 8-bit byte from this `InputStream`. If the byte is 0, false is returned, otherwise true is returned.

`readWord`

SYNTAX

```
int readWord()
```

PURPOSE

The `readWord` function reads a binary 16-bit integer value from this `InputStream`.

`readQuad`

SYNTAX

```
int readQuad()
```

PURPOSE

The `readQuad` function reads a binary 32-bit integer value from this `InputStream`.

readOcta

SYNTAX

```
long readOcta()
```

PURPOSE

The `readOcta` function reads a binary 64-bit integer value from this `InputStream`.

readFloat

SYNTAX

```
float readFloat()
```

PURPOSE

The `readFloat` function reads a binary 32-bit IEEE float value from this `InputStream`.

readDouble

SYNTAX

```
double readDouble()
```

PURPOSE

The `readDouble` function reads a binary 64-bit IEEE float value from this `InputStream`.

`readString`

SYNTAX

```
String readString()
```

PURPOSE

The `readString` function reads a binary representation of an Inca `String` from this `InputStream` and returns it. It expects the `String` was written by the `OutputStream` class method `writeString`.

`readPointer`

SYNTAX

```
void* readPointer()
```

PURPOSE

The `readPointer` function reads a binary representation of the given pointer p from this `InputStream`, by reading a 64-bit integer value and then converting it to a pointer.

`readBytes`

SYNTAX

```
void readBytes( byte* block, int count )
```

PURPOSE

The `readBytes` function reads `count` bytes from this `InputStream`, storing them into the the given array `block`.

3.5.6 OutputStream Method Reference

`flush`

SYNTAX

```
void flush()
```

PURPOSE

The `flush` function flushed all pending output to the stream. After this call no caching buffers will hold back any data from the target medium.

`writeChar`

SYNTAX

```
void writeChar( char c )
```

PURPOSE

The `writeChar` function writes a binary 8-bit character to this `OutputStream`.

`writeBool`

SYNTAX

```
void writeBool( bool b )
```

PURPOSE

The `writeBool` function writes a binary boolean value as 8-bit byte containing 0 or 1 to this `OutputStream`.

`writeWord`

SYNTAX

```
void writeWord( int value )
```

PURPOSE

The `writeWord` function writes a binary 16-bit integer value to this `OutputStream`.

writeQuad

SYNTAX

```
void writeQuad( int value )
```

PURPOSE

The `writeQuad` function writes a binary 32-bit integer value to this `OutputStream`.

writeOcta

SYNTAX

```
void writeOcta( long value )
```

PURPOSE

The `writeOcta` function writes a binary 64-bit integer value to this `OutputStream`.

writeFloat

SYNTAX

```
void writeFloat( float value )
```

PURPOSE

The `writeFloat` function writes a binary 32-bit IEEE float value to this `OutputStream`.

`writeDouble`

SYNTAX

```
void writeDouble( double value )
```

PURPOSE

The `writeDouble` function writes a binary 64-bit IEEE float value to this `OutputStream`.

`writeString`

SYNTAX

```
void writeString( String s )
```

PURPOSE

The `writeString` function writes a binary representation of the given `String` s to this `OutputStream`.

`writePointer`

SYNTAX

```
void writePointer( void* p )
```

PURPOSE

The `writeString` function writes a binary representation of the given pointer p to this `OutputStream`. It writes the pointer as a 64-bit integer value.

`writeBytes`

SYNTAX

```
void writeBytes( const byte* block, int count )
```

PURPOSE

The `writeBytes` function writes the given binary block of `count` bytes to this `OutputStream`.

3.5.7 Keyboard and Mouse Function Reference

```
diMousek
```

SYNTAX

```
bool diMousek( [int k] )
```

PURPOSE

This function checks if a mouse button is currently pressed, just like `mousek` does, however `diMousek` is implemented via DirectInput and allows for high frequency polling.

```
diMousex
diMousex
diMousex
```

SYNTAX

```
int diMousex()
int diMousex()
int diMousex()
```

PURPOSE

These functions return the mouse location, just like `mousex`, `mousey` and `mousez` do, however these here are implemented via DirectInput and allow for high frequency polling.

```
diUpdate
```

SYNTAX

```
void diUpdate()
```

PURPOSE

This `diUpdate` function updates the mouse and keyboard states that the `diMousek`, `diMousex`, `diMousey`, `diMousez` and `vkeyDown` functions return. This function is automatically called on `systemTask()` calls and on graphic console `swaps` (except you're running in real-time mode), so usually it's not necessary to call it.

`inkey`

SYNTAX

```
String inkey()
```

PURPOSE

The `inkey` function reads a key from the keyboard. If no key was pressed, an empty `String`, i.e. "", is returned. If a key was pressed, a `String` containing the character pressed is returned. The function does not wait for a key to be pressed but returns immediately.

The `inkey` function generates two kinds of `String`s:

1. If a key is pressed that has an ASCII code, then `inkey` returns `chr( asciiCode )`, i.e. a `String` of one character representing the pressed key.
2. Is key a is pressed that has *no* ASCII code (like the arrow keys for example), `inkey` returns `chr( 0 ) + chr( specialKeyCode )`, with `specialKeyCode` being a unique (non-ASCII) code specifying the pressed key.

If you're interested in ASCII keys as well as in non-ASCII keys, a common way to handle the situation could look like this:

```
void handleAsciiKey( int asciiCode ) { ... }
void handleSpecialKey( int specialCode ) { ... }

...

String s = inkey();
int key = asc( s );
if( key != 0 )
    handleAsciiKey( key );
else if( len( s ) > 1 )
    handleSpecialKey( asc( mid( key, 1 ) );
```

Note that Inca does not monitor every key press that occurs, but only keys pressed while a console or graphics window is active and has the keyboard focus. So if you for example press a key in another application, `inkey` will not recognize that.

The `inkey` function won't work in real-time mode.

`mousek`

SYNTAX

```
bool mousek( [int k] )
```

PURPOSE

The `mousek` function returns true, if the mouse button with index k is currently pressed over a 2d or 3d graphics console window. If the button isn't pressed, or if no graphics console window is currently open, false is returned.

If the parameter k is omitted, it defaults to zero, which corresponds to the left mouse button. A value of $k = 1$ corresponds to the right mouse button.

This function won't work in real-time mode.

`mousex`
`mousey`
`mousez`

SYNTAX

```
int mousex()
int mousey()
int mousez()
```

PURPOSE

These `mouse` function return the x, y and z coordinates of the current mouse position relative to a 2d or 3d graphics console window, if available. If no graphics console window is currently open, -1 is returned.

Note that x and y correspond the the pixel location of the mouse, with (0,0) being the location of the top left pixel inside the console window. The z component of the location is only available for wheel mouses - turning the wheel by one step changes the z component by one.

This function won't work in real-time mode.

vkeyDown

SYNTAX

```
bool vkeyDown( int keyCode )
```

PURPOSE

The `vkeyDown` function checks if a key is currently pressed on the keyboard. Different than the `inkey` function, it doesn't care which context you're in, it's a low level hardware function directly checking a key's current status. If you need to run in a fast feedback mode, where time and keyboard input are essential, use this function.

Note that the keyCode parameter is not an ASCII code. It's an internal value that depends on the keyboard your computer is connected to. You can however use the `vkeyCode` function to get the virtual key code corresponding to an ascii character. There are also the following few predefined key code constants:

```
KEY_LEFT
KEY_RIGHT
KEY_UP
KEY_DOWN
KEY_SHIFT
KEY_SPACE
KEY_ESCAPE
KEY_TAB
KEY_BACKSPACE
KEY_DELETE
KEY_RETURN
```

This function internally uses DirectInput to get the key state and therefore can be used in real-time mode and for high frequency polling.

`waitkey`

SYNTAX

```
String waitkey()
```

PURPOSE

The `waitkey` function works the same way as the `inkey` function, only that it will not return until a key has been pressed. Therefore, `waitkey` never returns empty `String`s.

This function won't work in real-time mode.

`waitmouse`

SYNTAX

```
int waitmouse()
```

PURPOSE

The `waitmouse` function waits until a mouse button (doesn't matter which one) has been pressed and returns its index. It will not return until any mouse button has been pressed. A returned index of 0 indicates that the left mouse button has been pressed, whereas for example an index of 1 indicates that the right mouse button has been pressed.

This function won't work in real-time mode.

3.5.8 InflaterStream Method Reference

`attach`

SYNTAX

```
void attach( InputStream* stream, int size )
```

PURPOSE

The `attach` function attaches to an `InflaterStream` an arbitrary `InputStream` `stream`, that should hold data that was compressed using an `DeflaterStream`. Subsequent read operations on the `InflaterStream` (which is an `InputStream`) will result in reading necessary chunks of compressed data from the attached `InputStream` `stream`, decompressing them, and returning them. The second parameter `size` gives the number of bytes that is available in the given `InputStream` `stream` for decompression. This should be the number of writes that was originally written via an `DeflaterStream`.

3.5.9 DeflaterStream Method Reference

`attach`

SYNTAX

```
void attach( OutputStream* stream, [int level] )
```

PURPOSE

The `attach` function attaches to an `DeflaterStream` an arbitrary `OutputStream stream`, that will receive compressed data. Subsequent write operations on the `DeflaterStream` (which is an `OutputStream`) will result in compressing the written data and writing chunks of this compressed data to the attached `OutputStream stream`.

The second parameter `level` (which is optional), indicates how good the compression should be. Currently, Inca defines three values:

- `Z_NO_COMPRESSION` - no compression happens at all, i.e. the `DeflaterStream` passes all the data to the attached stream, without compressing or modifying it.
- `Z_BEST_SPEED` - the `DeflaterStream` compresses given data, however it chooses to do this as fast as possible, resulting in a less efficient compression in many cases.
- `Z_BEST_COMPRESSION` (default) - the `DeflaterStream` compresses given data, and will compress it as good as it can (which means leading to a minimal size of the compressed data), even if this might take long time.

Since the constants just given are integer values, one can choose a certain tradeoff between compression speed and compression quality by selecting values that lie between the extreme values `Z_BEST_SPEED` and `Z_BEST_COMPRESSION`.

3.5.10 Print Functions

`crscol`

SYNTAX

```
int crscol()
```

PURPOSE

The `crscol` function returns the column in which the cursor is located in the current text console.

`crslin`

SYNTAX

```
int crslin()
```

PURPOSE

The `crslin` function returns the row in which the cursor is located in the current text console.

`htab`

SYNTAX

```
void htab( int column )
```

PURPOSE

The `htab` function sets the cursor in the current text console to the given column `column`.

`vtab`

SYNTAX

```
void vtab( int row )
```

PURPOSE

The `vtab` function sets the cursor in the current text console to the given row `row`.

`at`

SYNTAX

```
String at( int column, int row )
```

PURPOSE

The `at` function returns a `String` for use in a `print` command. The `at` directive sets the cursor to the given column `column` and row `row`.

EXAMPLE

```
print << at( 5, 10 ) << "hello world";
```

`forecolor`

SYNTAX

```
String forecolor( Color color )
```

PURPOSE

The `forecolor` function returns a `String` for use in a `print` command. The `forecolor` directive sets the text foreground color to the console color that best matches the given color `color`. Note that since there are usually only eight colors in text console displays, the difference between given and realized color might be large. The `forecolor` function leaves the background text color untouched.

EXAMPLE

```
print << forecolor( color( "red" ) );
print << "this is red";
```

`backcolor`

SYNTAX

```
String backcolor( Color color )
```

PURPOSE

The `backcolor` function returns a `String` for use in a `print` command. The `forecolor` directive sets the text background color to the console color that best matches the given color `color`. Note that since there are usually only eight colors in text console displays, the difference between given and realized color might be large. The `backcolor` function leaves the foreground text color untouched.

EXAMPLE

```
print << backcolor( color( "green" ) );
print << "green background!";
```

3.6 Container Library

3.6.1 Introduction

Inca currently offers one dynamic container type to collect, store, retrieve and sort items. There exists a range of specialized containers for different types, for example an so called `IntArray` that can hold integer numbers. The most important difference to regular arrays (like one declared via `int a[10]`) is, that container `Arrays` grow dynamically, i.e. the number of elements they hold is not fixed. Currently, the following typed containers exist:

Array Name	Element Type
ByteArray	byte
IntArray	int
LongArray	long
FloatArray	float
DoubleArray	double
PointerArray	void*
StringArray	String

All these containers have certain common operations that can be applied to them. These operations include adding items, removing items or searching items. Read and write access to single elements in an `Array` happens via the `[ ]` operator, just like in regular arrays .Items can be added to the end (i.e. after the last position) of an `Array` using the `append` method. You can print whole arrays using the `print` command. The following code gives an example of the concepts so far:

```
IntArray a;
a.append( 1 );
a.append( 2 );
a.append( 3 );
print << a << endl;
a[ 1 ] = 5;
print << a << endl;
print << a[ 2 ] << endl;
```

This will produce the following output:

```
[1,2,3]
[1,5,3]
3
```

First, the integer numbers 1, 2 and 3 are sequentially added to the `Array`. The array is printed, which correctly results in the output `[1,2,3]`. Then we change the second element of the array to 5 (indices in container `Arrays` are - just like in regular arrays - zero-based). Finally we print the third item of the `Array`, which is 3.

The exact operations available on on each of the container `Arrays` listed in the table above can be seen on the next pages. Apart from this, the container library provides a mechanism to sort arrays (both regular and container) of types, using the `sort` function.

3.6.2 Sorting

The `sort` in Inca is defined as:

```
void sort( Type*, int count, [int *itab] )
```

Type in the definition above stands for any of the following types:

```
byte
int
long
float
double
void*
```

So the `sort` function takes a pointer to an array of elements and a number of elements and then sorts the given array. For example

```
int a[] = { 5, 3, 1, 2 };
sort( a, 4 );
print << a;
```

produces:

```
[1,2,3,5]
```

You can also apply `sort` to container `Arrays` by passing the address of a container `Array` element like this:

```
void sortIntArray( IntArray& a )
{
    sort( &a[ 0 ], a.count() );
}
```

If the optional `itab` parameter is given, all the swaps of elements, that are applied to the actual array to get it into the right order, are also applied to the `itab` integer array. Therefore the `itab` integer array is expected to have `count` elements. You can use this feature to find out how exactly `sort` changed the order of elements in the original array.

The `sort` function in Inca is implemented using heap sort and runs in $O(n \cdot log(n))$.

3.6.3 Array Method Reference

In the following descriptions, *Type* will be used generally for one of byte, int, long, float, double, void*, String.

```
append
```

SYNTAX

```
void append( Type x )
```

PURPOSE

The `append` method appends the elements to the end of the `Array`. This operation runs in $O(lgn)$ (amortized, n being the element count of the `Array`).

EXAMPLE

```
IntArray a;
for( int i = 0; i < 7; i++ )
    a.append( i );
a.append( -2 );
a.append( 10 );
print << a;
```

OUTPUT

```
[0,1,2,3,4,5,6,-2,10]
```

```
remove
```

SYNTAX

```
void remove( int index, [int amount] )
```

PURPOSE

The `remove` method removes items from the `Array`. If the `amount` parameter is omitted, the item at index `index` is removed. With the `amount` parameter given, the function removes the `amount` consecutive items starting at index `index`. This operation runs in $O(n)$ (n being the element count of the `Array`).

EXAMPLE

```
IntArray a;
for( int i = 0; i < 7; i++ )
    a.append( i );
a.remove( 1, 2 );
print << a;
```

OUTPUT

```
[0,3,4,5,6]
```

count

SYNTAX

```
int count()
```

PURPOSE

The `count` method returns the number of elements in an `Array`.

EXAMPLE

```
IntArray a;
for( int i = 0; i < 7; i++ )
    a.append( i );
print << a.count() << endl;
a.remove( 0 );
```

```
print << a.count() << endl;
```

OUTPUT

```
7
6
```

insert

SYNTAX

```
void insert( int index, Type x )
```

PURPOSE

The `insert` method inserts the element `x` at the specified index `index` in an `Array`, with $0 \leq$ `index` $\leq$ number of elements in `Array`.

This operation runs in $O(n)$ (n being the element count of the `Array`).

EXAMPLE

```
IntArray a;
for( int i = 0; i < 7; i++ )
    a.append( i );
print << a << endl;
insert( 0, -5 );
print << a << endl;
insert( 3, -10 );
print << a << endl;
```

OUTPUT

```
[0,1,2,3,4,5,6]
[-5,0,1,2,3,4,5,6]
```

```
[-5,0,1,-10,2,3,4,5,6]
```

search

SYNTAX

```
int search( Type x, [int index] )
```

PURPOSE

The `search` method performs a linear search on an `Array` to find a given item `x`. If the item is not found, -1 is returned. Otherwise, the index at which the item is located in the `Array` is returned.

The optional parameter `index` specifies, where the linear search should start (by default, linear search starts at index 0). The linear search then proceeds by examining items at indices `index`, `index` $+ 1$, and so on.

This operation runs in $O(n)$ (n being the element count of the `Array`).

EXAMPLE

```
IntArray a;
for( int i = 0; i < 7; i++ )
    a.append( i * 3 );
print << a << endl;
print << a.search( 12 ) << endl;
print << a.search( 10 ) << endl;
```

OUTPUT

```
[0,3,6,9,12,15,18]
4
-1
```

bsearch

SYNTAX

```
int bsearch( Type x )
```

PURPOSE

The `bsearch` method performs a binary search on an `Array` to find a given item `x`. If the item is not found, -1 is returned. Otherwise, the index at which the item is located in the `Array` is returned.

To use `bsearch`, you must ensure that the `Array` you use it on it sorted. If this is not the case, `bsearch` will not work correctly.

This operation runs in $O(log(n))$ (n being the element count of the `Array`).

EXAMPLE

```
IntArray a;
for( int i = 0; i < 7; i++ )
    a.append( i * 3 );
print << a << endl;
print << a.bsearch( 12 ) << endl;
print << a.bsearch( 10 ) << endl;
```

OUTPUT

```
[0,3,6,9,12,15,18]
4
-1
```

binsidx

SYNTAX

```
int binsidx( Type x )
```

PURPOSE

The `binsidx` method returns an index for the given item `x`, such that if `x` is inserted in the `Array` at that index, the `Array`will remain sorted (assuming that the `Array` was sorted beforehand).

You can use the `binsidx` to build sorted `Array`s of items, that can then be queried using the `bsearch` method.

The `binsidx` method will only work correctly on `Array`s that are sorted - of course, if they are not sorted, there's actually no point in inserting a new item via `binsidx`.

This operation runs in $O(log(n))$ (n being the element count of the `Array`).

EXAMPLE

```
IntArray a;
for( int i = 0; i < 7; i++ )
    a.append( i * 3 );
print << a << endl;
a.insert( a.binsidx( 10 ), 10 );
print << a << endl;
```

OUTPUT

```
[0,3,6,9,12,15,18]
[0,3,6,9,10,12,15,18]
```

3.7 Thread Library

3.7.1 Introduction

Inca offers a simple thread library with an underlying scheduler that has microsecond granularity. Inca does *not* use the standard Windows thread architecture, which only offers millisecond granularity (i.e. if you want to put a thread to sleep a certain time, you can only specify this time as milliseconds, but not as microseconds). Contrary to that, the thread system in Inca was designed for making it possible to simulate high-frequency, microsecond-granularity real-time effects of multiple threads of execution.

3.7.2 Threads

The basic primitive in this library is the `Thread` class. If you want to create a thread, you should derive a new class from the `Thread` class and overload its `run` method like this:

```
class MyThread extends Thread {
public:
    void run()
    {
        ...insert your thread code here...
    }
}
```

`MyThread` is now a definition of your own thread with the code being executed in the thread declared inside the `run` method. To actually *instantiate* and run the thread, you have to instantiate your class like this:

```
new MyThread;
```

After creating an instance from `MyThread`, it will start running as soon as the current thread of execution allows it to. A `Thread` will run until it exits its `run` methods or it's killed (usually by another `Thread`) by using `delete` on the `Thread` object.

The `Thread` class provides methods to make the thread sleep (using the `sleepMillis` and `sleepMicros` methods), `block` or `wake` the `Thread`. For example

```
aThread->sleepMillis( 500 );
```

will make the thread aThread sleep for 500 milliseconds (half a second). Alternatively, there are the two global functions sleepMillis and sleepMicros, which will make the *currently executing* thread sleep, as in

```
...do something...
sleepMillis( 500 ); // wait a while
...continue doing something...
```

Threads can have different priorities. You can get or set a Thread's priority using the setPriority and priority methods. Priorities can be used to tune the scheduling behaviour. If two or more thread are eligible to run at the same time, the thread with the highest priority will be chosen. When it comes down to assigning CPU time to the threads, Threads with higher priority will always be preferred over Threads with lower priority.

3.7.3 Mutexes

In Inca , running multiple Threads is preemptive, i.e. the execution of one thread might get interrupted at any time by the execution of another thread (in contrast to this, cooperative thread models assume that you call a special scheduling function for this to take place). For basic synchronization of several threads, Inca offers the Mutex class. A Mutex is a mutually exclusive semaphore. One Thread can obtain it by calling the Mutex's lock method and then holds it, until it calls unlock to release it again. While one Thread holds a Mutex no other Thread can hold the same Mutex. Specifically, if a Thread tries to lock a Mutex that was already locked by another Thread, the former Thread is blocked until the Mutexis unlocked again.

Typically, Mutexes are used when certain resources have to be protected or certain actions have to be made atomic, like in

```
Mutex gMutex;
int gCounter;

class MyThread extends Thread {
public:
    void run()
    {
        while( true )
        {
            gMutex.lock();
            gCounter += 1;
```

```
                gMutex.unlock();
            }
        }
    }

    void main()
    {
        MyThread* a = new MyThread;
        MyThread* b = new MyThread;
        ...
    }
```

where a global counter `gCounter` is accessed and updated from *two* `Threads` `a` and `b`. Without using a `Mutex` in such a setting, dirty reads of the counter value can occur (if a thread switch occurs just after one `Thread` has read the counter, but before writing the new updated value).

3.7.4 Signals

In many situations in multi-threaded environments, one or more `Threads` need to wait on a event that is triggered by yet another `Thread`. For handling these situations properly, Inca offers the `Signal` class.

A `Signal` is a signal that a `Thread` can wait on by calling the `Signal`'s `wait` method. The `Thread` blocks then until some other `Thread` calls the `Signal`'s `signal` method.

This mechanism is sometimes also referred to as event signalling or broadcasting (since one thread broadcasts the signal information to one or more other threads at the same time).

3.7.5 Queues

`Queue`'s are a simple way in Inca to exchange information between `Thread`'s. A `Queue` is a queue of items. Each item has to be a block of fixed memory size (you have to specify this size when creating the queue via `Queue::make`).

`Queue`'s in Inca are FIFO, i.e. "first in first out". A `Thread` can append items to the end of the queue by calling the `put` method. On the other hand, items can be dequeued from the head of the `Queue` by calling the `get` method.

The `Queue` class takes care of the necessary synchronization (i.e. internally uses `Mutexes` to ensure that there are no synchronization conflicts).

3.7.6 Overview

Thread Methods

```
void wake()
void block()
void sleepMillis( long millis )
void sleepMicros( long micros )
void setName( String name )
String name()
void setPriority( int priority )
int priority()
```

Mutex Methods

```
void lock()
void unlock()
```

Signal Methods

```
void signal()
void wait()
void waitFor( long millis )
void waitUntil( long millis )
```

Queue Methods

```
static Queue* make( int size, int capacity )
void put( const void* data )
void get( void* data )
bool putCond( const void* data )
bool getCond( void* data )
bool putUntil( const void* data, long millis )
bool getUntil( void* data, long millis )
```

Global Functions

```
void enterRealTime()
void leaveRealTime()
long systemLatency()
void resetSystemLatency()
void sleepMillis( long millis )
void sleepMicros( long millis )
```

3.7.7 Thread Methods

`wake`

SYNTAX

```
void wake()
```

PURPOSE

The `wake` method wakes up a `Thread` that has been blocked or put to sleep via `block`, `sleepMillis` or `sleepMicros`. In other words, the `Thread` is made to resume its execution. If the thread is not in a blocked or sleeping state, `wake` will have no effect.

`block`

SYNTAX

```
void block()
```

PURPOSE

The `block` method blocks a `Thread` for indefinite time. The `Thread` will suspend execution until it's either woke up using `wake`, or until new `sleep` command (having a definite wake up time) cancels the block.

`sleepMillis`

SYNTAX

```
void sleepMillis( long millis )
```

PURPOSE

The `sleepMillis` method makes a `Thread` sleep for `millis` milliseconds (thousands of seconds). In other words, the execution of the `Thread` is suspended for the next `millis` milliseconds. The `Thread` might wake earlier, if it gets woken up by `wake`, or if another `sleep` command cancels this sleep.

`sleepMicros`

SYNTAX

```
void sleepMicros( long micros )
```

PURPOSE

The `sleepMicros` method makes a `Thread` sleep for `micros` microseconds (millions of seconds). In other words, the execution of the `Thread` is suspended for the next `micros` microseconds. The `Thread` might wake earlier, if it gets woken up by `wake`, or if another `sleep` command cancels this sleep.

`setName`

SYNTAX

```
void setName( String name )
```

PURPOSE

The `setName` method sets the name of a `Thread` to `name`. You can query the name of a `Thread` using the `name` method.

`name`

SYNTAX

```
String name()
```

PURPOSE

The `name` method returns the name of a `Thread`. You can set the name of a `Thread` using the `setName` method.

`setPriority`

SYNTAX

```
void setPriority( int priority )
```

PURPOSE

The `setPriority` method sets the priority of a `Thread` to `priority`. The larger the priority of a `Thread` is, the more likely it gets processing time. If two or more `Threads` are eligible for processing time at the same moment, the `Thread` with the highest priority will actually run. You can query a `Thread`'s priority using the `priority` method.

priority

SYNTAX

```
int priority()
```

PURPOSE

The `priority` method returns the priority of a `Thread`. You can set a `Thread`'s priority using the `setPriority` method.

3.7.8 Mutex Methods

`lock`

SYNTAX

```
void lock()
```

PURPOSE

The `lock` method locks and acquires a `Mutex` for the currently executing `Thread`. If during this acquisition, other `Threads` call `lock` on this `Mutex`, they will block until the `Mutex` has been released again via a call to `unlock`.

`unlock`

SYNTAX

```
void unlock()
```

PURPOSE

The `unlock` method unlocks a `Mutex` that had previously been locked via `lock`. If there were `Threads` that called `lock` on this `Mutex` during its acquisition period, these threads will now be reconsidered for acquiring the `Mutex` lock.

3.7.9 Signal Methods

`signal`

SYNTAX

```
void signal()
```

PURPOSE

The `signal` method signals a `Signal` and by this, unblocks all `Threads` that are waiting on it at once.

`wait`

SYNTAX

```
void wait()
```

PURPOSE

The `wait` method waits for a `Signal` to get signaled (i.e. another `Thread` calling the `Signal`'s `signal` method). The currently executing `Thread` will suspend until the `Signal` has been signaled.

`waitFor`

SYNTAX

```
void waitFor( long millis )
```

PURPOSE

The `waitFor` method waits for a `Signal` to get signaled (i.e. another `Thread` calling the `Signal`'s `signal` method) before the given time passes. The currently executing `Thread` will suspend until either the `Signal` has been signaled or `millis` millisecond have passed without that happening.

`waitUntil`

SYNTAX

```
void waitUntil( long millis )
```

PURPOSE

The `waitUntil` method waits for a `Signal` to get signaled (i.e. another `Thread` calling the `Signal`'s `signal` method) before a certain time is reached. The currently executing `Thread` will suspend until either the `Signal` has been signaled or the millisecond time indicated by `millis` is reached. The current millisecond time can be obtained via the global `millis`.

3.7.10 Queue Methods

`make`

SYNTAX

```
Queue* make( int size, int capacity )
```

PURPOSE

The static `make` method creates a new `Queue`. The `Queue` created is initially empty.

The parameter `capacity` indicates how much elements the `Queue` can hold at maximum. If a `Thread` tries to put items into a `Queue` that has reached its maximum, the `Thread` will not succeed and block or return, depending on the mode of insertion into the `Queue`. If `capacity` is 0, the `Queue` has infinite capacity, i.e. not upper limit on the maximum number of elements it can hold.

The parameter `size` specifies the size of a `Queue` element's size in bytes. For example, if you want to store pointers, you should pass `sizeof( void* )` as `size`. If you want to store structures, say `MyStructure`, you should pass the size of the structure in bytes, for example `sizeof( MyStructure )`, here.

The method returns the newly created `Queue`. If you want to dispose of the `Queue`, you should use the `delete` operator.

`put`

SYNTAX

```
void put( const void* data )
```

PURPOSE

The `put` method appends a new element at the tail of the `Queue`. The parameter `data` is a pointer to a block of data, that should become the new element's data. `put` copies the data from there into an internal storage, i.e. you can safely dispose of the data at `data` after calling `put`. `put` expects the data at `data` to be the size that was specified in `Queue::make` when creating the `Queue`. If the `Queue` has reached its maximum capacity, `put` will wait until some other `Thread`has removed an element from the `Queue`, which will then make the insertion of the new element item possible.

`get`

SYNTAX

```
void get( void* data )
```

PURPOSE

The `get` method dequeues (i.e. retrieves and removes) an element from the head of the `Queue`. if there is no such element (i.e. the `Queue` is empty), `get` waits until another `Thread` has put an element into the `Queue`. `get` will copy the retrieved element into the memory block pointed to by `data`. `get` assumes that there is enough room at `data` to hold the element, i.e. the number of bytes specified in `Queue::make` when creating the `Queue`.

`putCond`

SYNTAX

```
bool putCond( const void* data )
```

PURPOSE

The `putCond` method works like the regular `put`method, only that it doesn't wait if the `Queue`'s maximum capacity has been reached, in which case it will return immediately and return `false`. Otherwise, it will insert the new element and return `true`.

`getCond`

SYNTAX

```
bool getCond( const void* data )
```

PURPOSE

The `getCond` method works like the regular `get`method, only that it doesn't wait if the `Queue` is empty, in which case it will return immediately and return `false`. Otherwise, it will dequeue an element and return `true`.

putUntil

SYNTAX

```
bool putUntil( const void* data, long millis )
```

PURPOSE

The `putUntil` method works like the regular `put`method, only that it will only wait a certain time if the `Queue`'s maximum capacity has been reached. In the latter case, it will wait no longer (yet maybe considerably less) than the millisecond time inidcated by `millis` has been reached. If by that time the `Queue` is still full, `putUntil` will return with `false`. On the other hand, if the `Queue` gets non-full before that time (or was already non-full at the time of calling `putUntil`), `putUntil` performs the insertion and returns `true`.

getUntil

SYNTAX

```
bool getUntil( const void* data, long millis )
```

PURPOSE

The `getUntil` method works like the regular `get`method, only that it will only wait a certain time if the `Queue` is empty. In the latter case, it will wait no longer (yet maybe considerably less) than the millisecond time inidcated by `millis` has been reached. If by that time the `Queue` is still empty, `getUntil` will return with `false`. On the other hand, if the `Queue` gets non-empty before that time (or was already non-empty at the time of calling `getUntil`), `getUntil` dequeues an element and returns `true`.

3.7.11 Global Functions

`enterRealTime`

SYNTAX

```
void enterRealTime()
```

PURPOSE

The `enterRealTime` enters a special real-time mode Inca provides. By calling this function, Inca will block all other Windows processes - even mouse, keyboard and io services. Use this function if you want to ensure, that your program is not interrupted by other Windows services (this might be helpful if you need real-time microsecond granularity).

Use this function with care. After calling it, you will not be able to interrupt your running program. You can terminate real-time mode again by calling `leaveRealTime`.

`leaveRealTime`

SYNTAX

```
void leaveRealTime()
```

PURPOSE

The `leaveRealTime` leaves the special real-time mode Inca provides. Call this function if you entered the real-time mode via `enterRealTime` and now want to leave it again.

`systemLatency`

SYNTAX

```
long systemLatency()
```

PURPOSE

The `systemLatency` function gives a measure (in microseconds) of how exact the thread scheduler inside Inca works. More specifically, the value returned gives the maximum error in number of microseconds that has occured since the last call to `resetSystemLatency`, that the system was *off* compared to an idealized scheduler.

To give an example, the most important measure here is how exact sleep requests are fulfilled. If a certain `Thread` calls `sleepMicros( n )` to sleep n microseconds, but it actually wakes after `n+e` microseconds, then an error of `e` is recorded. A number of similar measures are performed all the time. `systemLatency` returns the maximum of measured errors that occured since the last call to `resetSystemLatency`.

You can use `systemLatency` to measure how "real-time" your program runs.

`resetSystemLatency`

SYNTAX

```
void resetSystemLatency()
```

PURPOSE

The `resetSystemLatency` resets the measures of errors returned by `systemLatency`. Every time you therefore call `resetSystemLatency`, the system starts measuring the scheduler's latency from new, forgetting about all measures made before the call of `resetSystemLatency`.

`sleepMillis`

SYNTAX

```
void sleepMillis( long millis )
```

PURPOSE

The `sleepMillis` puts the currently executing `Thread` to sleep for `millis` milliseconds.

`sleepMicros`

SYNTAX

```
void sleepMicros( long micros )
```

PURPOSE

The `sleepMicros` puts the currently executing `Thread` to sleep for `micros` microseconds.

3.8 3d Support Library

3.8.1 Introduction

This library has functions that may be helpful if you're writing programs that deal with graphics or concepts in three dimensions. It provides basic point and vector functions. Also included in this library is a simple three-component color type.

A `Point` in Inca is a three-component tuple of floats. If you add or subtract two points, the add or subtract will be performed componentwise in each dimension. For `Vectors`, the same applies. Inca actually doesn't differentiate between `Points` and `Vectors` and so you can perform operations, that aren't defined from a mathematical point of view (like adding two `Points`).

`Vectors` can be scaled by a scalar floating point value. You can build the dot product of two `Vectors` using the `*` operator. You can compute the cross product of two `Vectors` using the `^` operator. Furthermore, there are functions to compute the length of `Vectors` and normalize them.

To create a `Point`, `Vector` or `Color`, you can use constructor-like functions, called `point`, `vector` and `color`. These take the parameters needed for their specific type and create it:

```
Vector v;

v = vector( 0, 0, 1 );
print << ( v ^ vector( 1, 0, 0 ) ) << endl;

print << v * 0.1 << endl;
print << 0.5 * v << endl;
print << v / 2 << endl;

print << point( 3, 4, 5 ) + v - 2 * vector( 1, 0, -1 );
```

The code above will generate the following output

```
(0,1,0)
(0,0,0.1)
(0,0,0.5)
(0,0,0.5)
(1,4,8)
```

Obviously, the cross product of (0,0,1) and (1,0,0) is (0,1,0). The following three `Vectors` are scaled variants of v. The last `Vector` is the result of the expression in the last print statement.

Another class of functions in the 3d Support Library are the so called noise functions, which implement Perlin Noise. They take a one, two or three dimensional parameter and return a noise value that's smoothly changing as you move through the n-dimensional parameter space.

In the following function specifications, the `Vector`, `Point` and `Color` types are used interchangeably. A function taking one of these types can also be called with any other of the three types. However, some of the functions might not make sense if called with different types than stated.

3.8.2 Special Color Constructors

Apart from constructing colors by specifying a three-component tuple as in `color( float, float, float )`, Inca provides two more ways of creating `Colors`.

The second one is using the index constructor `color( int )`. It takes an integer index and returns a `Color` from an internal palette. This functions comes in handy if you just need a bunch of different colors:

```
// print 20 different colors
for( int i = 0; i < 20; i++ )
print << color( i ) << endl;
```

The third `Color` constructor `color( const String& )` takes the name of a color and returns an RGB `Color`. Currently, Inca knows the following color names:

```
white
black
yellow
cyan
blue
green
red
aqua
magenta
grey
gray
orange
```

```
violet
azure
crimson
turquoise
lavender
tan
beige
lawngreen
silver
orchid
quartz
salmon
gold
khaki
bronze
linen
plum
aquamarine
wheat
copper
pink
moccasin
snow
ivory
sienna
peru
tomato
purple
brown
maroon
bisque
honeydew
mintcream
midnightblue
navyblue
steelblue
chartreuse
indianred
burlywood
coral
```

3.8.3 Overview

Vector Functions

```
float distance( Point a, Point b )
float length( Vector v )
Vector normalize( Vector v )
Point rotate( Point p, float angle, Point a, Point b )
float xcomp( Vector v )
float ycomp( Vector v )
float zcomp( Vector v )
```

Noise Functions

```
float noise( float x )
float noise( float x, float y )
float noise( Point p )
float snoise( float x )
float snoise( float x, float y )
float snoise( Point p )
Point pnoise( Point p )
Point psnoise( Point p )
```

Miscellaneous Functions

```
Color mix( Color a, Color b, float t )
```

Constructor Functions

```
Point point( float x, float y, float z )
Vector vector( float x, float y, float z )
Color color( float r, float g, float b )
Color color( int index )
Color color( const String\& name )
```

3.8.4 Vector Function Reference

`distance`

SYNTAX

```
float distance( Point a, Point b )
```

PURPOSE

The `distance` function returns the distance between two point a and b. If $c = a - b$ and and $c = (x, y, z)$, this function returns $\sqrt{x^2 + y^2 + z2}$.

`length`

SYNTAX

```
float length( Vector v )
```

PURPOSE

The `length` function returns the length of a vector. If $v = (x, y, z)$, then this function returns $\sqrt{x^2 + y^2 + z2}$.

`mix`

SYNTAX

```
Color mix( Color a, Color b, float t )
```

PURPOSE

The `mix` function computes the linear interpolation between a and b with t in the range [0,1]. If t is 0, a is returned. If t is 1, b is returned.

`normalize`

SYNTAX

```
Vector normalize( Vector v )
```

PURPOSE

The `normalize` function returns the normalized vector v. If $v = (x, y, z)$, then this function returns $\frac{1}{\sqrt{x^2+y^2+z2}}v$.

`rotate`

SYNTAX

```
Point rotate( Point p, float angle, Point a, Point b
)
```

PURPOSE

The `rotate` function rotates the Point p around the axis through a and b for the amount specified by angle. The given angle is expected to be in radians. Note that a is regarded as the origin of the rotation. Returns the rotated point.

`xcomp`

SYNTAX

```
float xcomp( Vector v )
```

PURPOSE

The `xcomp` function returns the x component of the given vector v. If $v = (x, y, z)$, then this function returns x.

`ycomp`

SYNTAX

```
float ycomp( Vector v )
```

PURPOSE

The `ycomp` function returns the y component of the given vector v. If $v = (x, y, z)$, then this function returns y.

zcomp

SYNTAX

```
float zcomp( Vector v )
```

PURPOSE

The `zcomp` function returns the z component of the given vector v. If $v = (x, y, z)$, then this function returns z.

3.8.5 Noise Function Reference

`noise`

SYNTAX

```
float noise( float x )
float noise( float x, float y )
float noise( Point p )
```

PURPOSE

The `snoise` function returns an unsigned gradient noise in the range [0,1]. Depending on the variant you use, you have a one-, two- or three-dimensional parameter space.

`pnoise`

SYNTAX

```
Point pnoise( Point p )
```

PURPOSE

The `pnoise` function returns an unsigned three dimensional gradient noise vector given a point in three dimensional parameter space.

`psnoise`

SYNTAX

```
Point psnoise( Point p )
```

PURPOSE

The `psnoise` function returns a signed three dimensional gradient noise vector given a point in three dimensional parameter space.

`snoise`

SYNTAX

```
float snoise( float x )
float snoise( float x, float y )
float snoise( Point p )
```

PURPOSE

The `snoise` function returns a signed gradient noise in the range [-1,1]. Depending on the variant you use, you have a one-, two- or three-dimensional parameter space.

3.9 OpenGL Library

3.9.1 Three-Dimensional Graphics

The Inca OpenGL library is basically just a binding library to the default OpenGL implementation of your system. There are however two support classes to open up windows that you can draw into using OpenGL. In Inca , these windows are called *consoles*.

Most typically, one uses the `G3Console` class and its `open` method to open up a window that's capable of OpenGL drawing. By default, a `G3Console` will open a double buffered OpenGL context. This means your drawing commands will not be visible immediately on the screen, but only after you call the `G3Console`'s `swap` method (for more information on double buffering, consult a text on OpenGL). A typical program might look like this:

```
G3Console console;
console.open();

...do your first time setup for OpenGL here...

while( as long as you want to run )
{
    if( console.resized() )
        { ... setup viewport here... }

    ...draw your frame here...

    console.swap();
}
```

First, a console window is opened via `open` (the call to this method opens the window). After the `open` call, you have a valid OpenGL context and you can issue OpenGL commands that are sent to the `G3Console` console (before calling `open` you should not issue any OpenGL commands, since there would be context for them). The main loop that follows takes care of setting up a proper viewport, drawing the frame that should be displayed via a series of OpenGL commands, and then, finally, displaying the frame on the screen by calling `swap`.

The `G3Console's resized` method checks whether the console window has been resized since the last call to `resized`. Furthermore, `resized` will always return true for the first time it's called after the console window is opened. Therefore, the `resized` method gives you a simple way of determining of when to set up or update your OpenGL viewport. A typical example of how to set up a perspective viewport could look like this

```
float width = console.width();
float height = console.height();

glViewport( 0, 0, width, height );
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
gluPerspective( 60, width / height, 1.0, 30.0 );
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
```

The only thing specific to Inca in the code above are the calls to `G3Console`'s `width` and `height` functions to acquire the console window's current width and height in pixels.

There are a number of different possibilities. For example, you can use the `G3Console`'s `setDoubleBuffer` method disable double buffering and therefore drawing directly to the screen, without having to call `swap`, like this:

```
G3Console console;
console.setDoubleBuffer( false );
console.open();

...do your first time setup for OpenGL here...

while( as long as you want to run )
{
    if( console.resized() )
        { ... setup viewport here... }

    ...draw here...
}
```

You can use `G3Console`'s `setFullscreen` method to open up a window in full screen mode.

3.9.2 Two-Dimensional Graphics

Inca offers a special console type making the drawing of strictly two dimensional graphics more similar to using a classic pixel-based graphics service (like Quickdraw on the Macintosh, for example). The special console is called `G2Console` and it basically works like `G3Console`, as it provides the same methods as `open`, `swap`, and so on. However, `G2Console` doesn't allocate a depth buffer. Also, by default, it sets up an orthogonal viewport, that allows you to use window coordinates (using (0,0) as the top left corner of the window, and (*width*,*height*) as the bottom right corner, with *width* and *height* being the size of the window in pixels (this viewport is set automatically by `G2Console`, so you don't even have to bother calling `resized` and setting up a viewport on your own). For example,

```
G2Console console;
console.open();

while( mousek() == false )
{
    sglSetColor( "white" );
    sglClear();
    sglSetColor( "red" );
    sglFillRect( 10, 20, 200, 300 );
    console.swap();
}
```

will use Inca 's SGL library (see the documentation there for further details) to draw a filled rect with its top left corner at pixel (10,20) and its bottom right corner at pixel (200,300).

The `G2Console` and the SGL library were designed to make it especially easy to draw two dimensional graphics in Inca .

3.9.3 OpenGL Function Overview

This is a list of all the OpenGL functions you can call and access from within Inca .

```
glAccum
glAlphaFunc
glAreTexturesResident
glArrayElement
glBegin
glBindTexture
glBitmap
glBlendFunc
glCallList
glCallLists
glClear
glClearAccum
glClearColor
glClearDepth
glClearIndex
glClearStencil
glClipPlane
glColor3b
glColor3bv
glColor3d
glColor3dv
glColor3f
glColor3fv
glColor3i
glColor3iv
glColor3s
glColor3sv
glColor3ub
glColor3ubv
glColor3ui
glColor3uiv
glColor3us
glColor3usv
glColor4b
glColor4bv
```

glColor4d
glColor4dv
glColor4f
glColor4fv
glColor4i
glColor4iv
glColor4s
glColor4sv
glColor4ub
glColor4ubv
glColor4ui
glColor4uiv
glColor4us
glColor4usv
glColorMask
glColorMaterial
glColorPointer
glCopyPixels
glCopyTexImage1D
glCopyTexImage2D
glCopyTexSubImage1D
glCopyTexSubImage2D
glCullFace
glDeleteLists
glDeleteTextures
glDepthFunc
glDepthMask
glDepthRange
glDisable
glDisableClientState
glDrawArrays
glDrawBuffer
glDrawElements
glDrawPixels
glEdgeFlag
glEdgeFlagPointer
glEdgeFlagv
glEnable
glEnableClientState
glEnd
glEndList

glEvalCoord1d
glEvalCoord1dv
glEvalCoord1f
glEvalCoord1fv
glEvalCoord2d
glEvalCoord2dv
glEvalCoord2f
glEvalCoord2fv
glEvalMesh1
glEvalMesh2
glEvalPoint1
glEvalPoint2
glFeedbackBuffer
glFinish
glFlush
glFogf
glFogfv
glFogi
glFogiv
glFrontFace
glFrustum
glGenLists
glGenTextures
glGetBooleanv
glGetClipPlane
glGetDoublev
glGetError
glGetFloatv
glGetIntegerv
glGetLightfv
glGetLightiv
glGetMapdv
glGetMapfv
glGetMapiv
glGetMaterialfv
glGetMaterialiv
glGetPixelMapfv
glGetPixelMapuiv
glGetPixelMapusv
glGetPointerv
glGetPolygonStipple

glGetString
glGetTexEnvfv
glGetTexEnviv
glGetTexGendv
glGetTexGenfv
glGetTexGeniv
glGetTexImage
glGetTexLevelParameterfv
glGetTexLevelParameteriv
glGetTexParameterfv
glGetTexParameteriv
glHint
glIndexMask
glIndexPointer
glIndexd
glIndexdv
glIndexf
glIndexfv
glIndexi
glIndexiv
glIndexs
glIndexsv
glIndexub
glIndexubv
glInitNames
glInterleavedArrays
glIsEnabled
glIsList
glIsTexture
glLightModelf
glLightModelfv
glLightModeli
glLightModeliv
glLightf
glLightfv
glLighti
glLightiv
glLineStipple
glLineWidth
glListBase
glLoadIdentity

glLoadMatrixd
glLoadMatrixf
glLoadName
glLogicOp
glMap1d
glMap1f
glMap2d
glMap2f
glMapGrid1d
glMapGrid1f
glMapGrid2d
glMapGrid2f
glMaterialf
glMaterialfv
glMateriali
glMaterialiv
glMatrixMode
glMultMatrixd
glMultMatrixf
glNewList
glNormal3b
glNormal3bv
glNormal3d
glNormal3dv
glNormal3f
glNormal3fv
glNormal3i
glNormal3iv
glNormal3s
glNormal3sv
glNormalPointer
glOrtho
glPassThrough
glPixelMapfv
glPixelMapuiv
glPixelMapusv
glPixelStoref
glPixelStorei
glPixelTransferf
glPixelTransferi
glPixelZoom

glPointSize
glPolygonMode
glPolygonOffset
glPolygonStipple
glPopAttrib
glPopClientAttrib
glPopMatrix
glPopName
glPrioritizeTextures
glPushAttrib
glPushClientAttrib
glPushMatrix
glPushName
glRasterPos2d
glRasterPos2dv
glRasterPos2f
glRasterPos2fv
glRasterPos2i
glRasterPos2iv
glRasterPos2s
glRasterPos2sv
glRasterPos3d
glRasterPos3dv
glRasterPos3f
glRasterPos3fv
glRasterPos3i
glRasterPos3iv
glRasterPos3s
glRasterPos3sv
glRasterPos4d
glRasterPos4dv
glRasterPos4f
glRasterPos4fv
glRasterPos4i
glRasterPos4iv
glRasterPos4s
glRasterPos4sv
glReadBuffer
glReadPixels
glRectd
glRectdv

glRectf
glRectfv
glRecti
glRectiv
glRects
glRectsv
glRenderMode
glRotated
glRotatef
glScaled
glScalef
glScissor
glSelectBuffer
glShadeModel
glStencilFunc
glStencilMask
glStencilOp
glTexCoord1d
glTexCoord1dv
glTexCoord1f
glTexCoord1fv
glTexCoord1i
glTexCoord1iv
glTexCoord1s
glTexCoord1sv
glTexCoord2d
glTexCoord2dv
glTexCoord2f
glTexCoord2fv
glTexCoord2i
glTexCoord2iv
glTexCoord2s
glTexCoord2sv
glTexCoord3d
glTexCoord3dv
glTexCoord3f
glTexCoord3fv
glTexCoord3i
glTexCoord3iv
glTexCoord3s
glTexCoord3sv

```
glTexCoord4d
glTexCoord4dv
glTexCoord4f
glTexCoord4fv
glTexCoord4i
glTexCoord4iv
glTexCoord4s
glTexCoord4sv
glTexCoordPointer
glTexEnvf
glTexEnvfv
glTexEnvi
glTexEnviv
glTexGend
glTexGendv
glTexGenf
glTexGenfv
glTexGeni
glTexGeniv
glTexImage1D
glTexImage2D
glTexParameterf
glTexParameterfv
glTexParameteri
glTexParameteriv
glTexSubImage1D
glTexSubImage2D
glTranslated
glTranslatef
glVertex2d
glVertex2dv
glVertex2f
glVertex2fv
glVertex2i
glVertex2iv
glVertex2s
glVertex2sv
glVertex3d
glVertex3dv
glVertex3f
glVertex3fv
```

```
glVertex3i
glVertex3iv
glVertex3s
glVertex3sv
glVertex4d
glVertex4dv
glVertex4f
glVertex4fv
glVertex4i
glVertex4iv
glVertex4s
glVertex4sv
glVertexPointer
glViewport
```

3.9.4 Overview

Console Methods

```
float fps()
int height()
void makeCurrent()
void open()
bool resized()
void setDoubleBuffer( bool doublebuffer )
void setFullscreen( bool fullscreen )
void setResolution( int width, int height )
void setTitle( String title )
void swap()
int width()
```

3.9.5 Console Methods

`fps`

SYNTAX

```
float fps()
```

PURPOSE

The `fps` method returns the number of frames per second this console window currently displays. This value is measured by checking how often you call the console's `swap` method. Therefore, this method will only work for double buffered OpenGL contexts.

`height`

SYNTAX

```
int height()
```

PURPOSE

The `height` method returns the current height of this console's window in pixels.

`makeCurrent`

SYNTAX

```
void makeCurrent()
```

PURPOSE

The `makeCurrent` method makes this console window the current OpenGL context. You will only need this function if you have several console windows open, and want to specify to which one subsequent OpenGL calls will be sent.

`open`

SYNTAX

```
void open()
```

PURPOSE

The `open` method opens up the console window and makes it the current OpenGL context. After calling `open` you can start to issue OpenGL commands to that console.

`resized`

SYNTAX

```
bool resized()
```

PURPOSE

The `resized` method returns true, if the console window has been manually resized (i.e. its width or height has changed) since the last call to `resized`. If the window has remained constant in size, false is returned. Usually, this method is polled from within an loop to regularly update the OpenGL context to a new window size if necessary. Note that `resized` will always return true for the first time it's called after the console window is opened (i.e. you can also use it for your initial setup of the OpenGL viewport).

`setDoubleBuffer`

SYNTAX

```
void setDoubleBuffer( bool doublebuffer )
```

PURPOSE

The `setDoubleBuffer` method toggles the existence or absence of a double buffer in subsequently opened window. Calling the `setDoubleBuffer` will only have an effect and make sense *before* calling the `open` method. If the `doublebuffer` parameter is set to true, Inca will allocate an double buffer for the OpenGL context. If the parameter is false, Inca will allocate a single buffer OpenGL context. The default is the double buffer configuration.

`setFullscreen`

SYNTAX

```
void setFullscreen( bool fullscreen )
```

PURPOSE

The `setFullscreen` method toggles the fullscreen property of subsequently opened windows. Calling the `setFullscreen` will only have an effect and make sense *before* calling the `open` method. If the `fullscreen` parameter is set to true, Inca will open the console window fullscreen. If the parameter is false, Inca will open the console as regular window. The default is the latter configuration.

`setResolution`

SYNTAX

```
void setResolution( int width, int height )
```

PURPOSE

The `setResolution` method method sets the resolution (width and height in pixels) of subsequently opened windows. Calling the `setResolution` will only have an effect and make sense *before* calling the `open` method. Calling it after `open` will have no effect currently.

`setTitle`

SYNTAX

```
void setTitle( String title )
```

PURPOSE

The `setTitle` method sets the console window's title to the given String `title`.

`swap`

SYNTAX

```
void swap()
```

PURPOSE

The `swap` method swaps the OpenGL back buffer with the front buffer and by this displays the picture that has been drawn using previously issued OpenGL commands. Note that this method only makes sense for consoles that are double buffered.

`width`

SYNTAX

```
int width()
```

PURPOSE

The `width` method returns the current width of this console's window in pixels.

3.10 Simple Graphics Library

3.10.1 Introduction

The Simple Graphics Library (SGL) is a library on top of OpenGL, which makes drawing of two-dimensional shapes a bit easier and provides simple functions for drawing text. Since SGL uses OpenGL commands to accomplish its actions, you can use SGL whereever you can use OpenGL.

Please note that if you want to mix OpenGL and SGL calls, you should always call `sglFlush` after a series of SGL calls, before you issue an OpenGL call. If you don't do this, the graphics might mess up and you might get wrong visual results.

3.10.2 Setting the Color

There are four functions in SGL to set the current drawing color, and they are:

```
void sglSetColor( const Color& )
void sglSetColor( float, float, float )
void sglSetColor( const String& )
void sglSetIndexedColor( int index )
```

The first two variants take a color as three-component value. The third variant takes the name of a color. For a list of supported names, see the documentation of the `Color` type. The fourth and final version takes an index into an internal palette of colors.

All lines, shapes and text in SGL is drawn with the drawing color that was most recently set using one of these functions.

3.10.3 Drawing Lines

Using the SGL line functions `sglMoveTo` and `sglLineTo` you can draw lines or polylines. With `sglMoveTo` the drawing cursor is moved to a certain location, with `sglLineTo` a line is drawn from the current cursor location to the location given in `sglLineTo`, then the cursor is moved to the new location. For example, to draw an triangle, you could use:

```
sglMoveTo( 200, 200 );
sglLineTo( 400, 200 );
sglLineTo( 300, 100 );
sglLineTo( 200, 200 );
```

Note that for large number of lines you should better use OpenGL directly (since it's a *lot* faster), more specifically the GL_LINES and GL_LINE_STRIP drawing modes.

3.10.4 Drawing Shapes

SGL currently supports drawing of filled and framed rectangles and ovals. The calls are `sglFillRect`, `sglFrameRect`, `sglFillOval` and `sglFrameOval`. Each call takes four coordinates x0,y0,x1,y1, with (x0,y0) specifying the top left corner and (x1,y1) specifying the bottom right corner of the shape. So, for example,

```
sglFrameRect( 10, 15, 200, 300 );
```

draws an framed rectangle with its top left corner at (10,15) and its bottom right corner at (200,300).

3.10.5 Drawing Text

SGL makes drawing text quite easy. Before drawing text in an OpenGL context (that might be a `G2Console` or `G3Console` window for example), you have to load the fonts. SGL differentiates between two kinds of fonts:

1. Bitmap Fonts. These fonts are directly rastered into the OpenGL frame buffer. They have fixed size (you have to specify the size when you load them), they cannot be scaled or rotated. However, their visual quality is excellent. Bitmap fonts are generated from Windows fonts, i.e. bitmap font names refer to fonts that are installed on your Windows system.

2. Texture Fonts. These fonts are rendered using OpenGL's texture capabilities. These fonts have variable size, i.e. once you load a texture font, you can draw it in any size you like. Texture fonts can be scaled and rotated. However, the visual quality for texture fonts might be quite bad sometimes. Texture fonts are loaded from .txf texture font files. The format and implementation Inca uses for texture fonts is based on Mark J. Kilgard's `TexFont` package. See documentation on the Web on that package on how to obtain and generate .txf files.

Which type of font you choose depends on your application. If visual quality is not critical, texture fonts usually provide the most versatile way to draw text in Inca .

Once a font has been loaded via the `sglLoadBitmapFont` or via the `sglLoadTextureFont` function, it can be activated via the `sglSetFont` function. Note that you have to load fonts for every OpenGL context you open separately, i.e. if you open two `G3Console` windows, you have to load the needed fonts for each of them separately.

All that's now left to do, before drawing text, is to choose a font size via the `sglSetFontSize` function (for bitmap fonts, this has to be a size, that was explicitly specified in `sglLoadBitmapFont`). Now you can finally draw text using the `sglDrawText` function. This functions draws the text at the location, that the SGL graphics cursor was set to (you can set the SGL graphics cursor using `sglMoveTo`). Note that text is always drawn in the current drawing color.

You can gather further information on certain attributes like height and width of the currently activated font using the query functions `sglTextWidth`, `sglGetFontSize`, `sglGetFontAscent` and `sglGetFontDescent`.

The following code loads the `Helvetica` bitmap font with a specified size and draws a line of text at location (10,20):

```
sglLoadBitmapFont( "Helvetica", 12 );
sglSetFont( "Helvetica" );
sglSetFontSize( 12 );
sglMoveTo( 10, 20 );
sglDrawText( "Hello World!" );
```

The next example loads a texture font with the name "Texture" and draws two lines of text of different size.

```
sglLoadTextureFont( "Helvetica" );
sglSetFont( "Helvetica" );
sglSetFontSize( 12 );
sglMoveTo( 10, 20 );
sglDrawText( "Hello World!" );
sglSetFontSize( 20 );
sglMoveTo( 10, 80 );
sglDrawText( "Hello World!" );
```

3.10.6 Overview

Miscellaneous Functions

```
void sglClear()
void sglFlush()
```

Color Functions

```
void sglSetColor( const Color& )
void sglSetColor( float, float, float )
void sglSetColor( const String& )
void sglSetIndexedColor( int index )
```

Line and Cursor Functions

```
void sglMoveTo( float x, float y )
void sglLineTo( float x, float y )
```

Shape Functions

```
void sglFrameRect( float x1, float y1, float x2, float y2 )
void sglFillRect( float x1, float y1, float x2, float y2 )
void sglFrameOval( float x1, float y1, float x2, float y2 )
void sglFillOval( float x1, float y1, float x2, float y2 )
```

Text Functions

```
void sglSetFont( const String& fontName )
void sglSetFontSize( float size )
float sglGetFontSize()
float sglGetFontAscent()
float sglGetFontDescent()
float sglTextWidth( const String& text )
void sglDrawText( const String& text )
void sglLoadBitmapFont( const String& fontName, int size )
void sglLoadTextureFont( const String& fontName )
```

3.10.7 SGL Function Reference

`sglClear`

SYNTAX

```
void sglClear()
```

PURPOSE

The `sglClear` function clears the whole drawing area with the current drawing color. It has the same effect like calling `sglFillRect` with the bounds of the screen as parameters (however it uses `glClear` internally and might be more efficient therefore).

`sglDrawText`

SYNTAX

```
void sglDrawText( const String& text )
```

PURPOSE

The `sglDrawText` function draws the given text at the current drawing cursor location and then updates the drawing cursor's horizontal component to mark the end of the drawn text (this enables you to make subsequent calls to `sglDrawText`, each appending text to the previously drawn text).

The current drawing cursor location's vertical component is taken to be the location of the font's *baseline*, and *not* the bottom or top line. If you want to draw text, that is vertically aligned to the top or bottom line, you have to calculate that location using `getFontAscent` or `getFontDescent`.

`sglFlush`

SYNTAX

```
void sglFlush()
```

PURPOSE

The `sglFlush` function flushes the SGL pipeline and finishes all pending drawing operations on besides of SGL immediately. Call this function whenever you called one or more SGL functions and want to resume by calling one or more OpenGL functions. SGL functions and OpenGL functions should always be separated by `sglFlush` calls.

sglFillOval

SYNTAX

```
void sglFillOval( float x1, float y1,
float x2, float y2 )
```

PURPOSE

The `sglFillOval` function draws a filled oval in the current drawing color at the given coordinates, with (x1,y1) being the top left corner and (x2,y2) being the bottom right corner.

sglFrameOval

SYNTAX

```
void sglFrameOval( float x1, float y1,
float x2, float y2 )
```

PURPOSE

The `sglFrameOval` function draws a framed oval in the current drawing color at the given coordinates, with (x1,y1) being the top left corner and (x2,y2) being the bottom right corner. `sglFrameOval` does not fill the interior of the oval.

sglFillRect

SYNTAX

```
void sglFillRect( float x1, float y1,
float x2, float y2 )
```

PURPOSE

The `sglFillRect` function draws a filled rectangle in the current drawing color at the given coordinates, with (x1,y1) being the top left corner and (x2,y2) being the bottom right corner.

sglFrameRect

SYNTAX

```
void sglFrameRect( float x1, float y1,
float x2, float y2 )
```

PURPOSE

The `sglFrameRect` function draws a framed rectangle in the current drawing color at the given coordinates, with (x1,y1) being the top left corner and (x2,y2) being the bottom right corner. `sglFrameRect` does not fill the interior of the rectangle.

sglGetFontAscent

SYNTAX

```
float sglGetFontAscent()
```

PURPOSE

The `sglGetFontAscent` function returns the ascent (distance from the font's baseline to the font's top line) of the currently selected font.

sglGetFontDescent

SYNTAX

```
float sglGetFontDescent()
```

PURPOSE

The `sglGetFontDescent` function returns the descent (distance from the font's baseline to the font's bottom line) of the currently selected font.

sglGetFontSize

SYNTAX

```
float sglGetFontSize()
```

PURPOSE

The `sglGetFontSize` function returns the current font size.

sglLineTo

SYNTAX

```
void sglLineTo( float x, float y )
```

PURPOSE

The `sglLineTo` function draws a line from the current SGL drawing cursor location to (x,y). Then it sets the new drawing cursor location to (x,y).

sglLoadBitmapFont

SYNTAX

```
void sglLoadBitmapFont( const String& fontName,
int size )
```

PURPOSE

The `sglLoadBitmapFont` function loads the bitmap font with the given name and the given size (in pixels). The given `fontName` must correspond to the name of a Windows system font.

Note that the font will only be loaded for the current OpenGL context. If you want to use this font in another OpenGL context too, you have to load it there separately.

sglLoadTextureFont

SYNTAX

```
void sglLoadTextureFont( const String& fontName )
```

PURPOSE

The `sglLoadBitmapFont` function loads the texture font with the given name. Inca looks for a file named "*fontName*.txf" that contains the texture font data. For more information on these font files and how to generate them, see Mark J. Kilgard's `TexFont` package.

Note that the font will only be loaded for the current OpenGL context. If you want to use this font in another OpenGL context too, you have to load it there separately.

sglMoveTo

SYNTAX

```
void sglMoveTo( float x, float y )
```

PURPOSE

The `sglMoveTo` function moves the SGL drawing cursor to the location (x,y).

sglSetColor

SYNTAX

```
void sglSetColor( const Color& color )
```

PURPOSE

The `sglSetColor` function sets the current drawing color to the `Color` color.

sglSetColor

SYNTAX

```
void sglSetColor( float r, float g, float b )
```

PURPOSE

The `sglSetColor` function sets the current drawing color to the RGB color (r,g,b).

sglSetColor

SYNTAX

```
void sglSetColor( const String& name )
```

PURPOSE

The `sglSetColor` function sets the current drawing color to the color with the given name. For a list of color names Inca supports, see the documentation for the `Color` type.

sglSetFont

SYNTAX

```
void sglSetFont( const String& fontName )
```

PURPOSE

The `sglSetFont` function sets the current font to the font with the given name. The font must have been loaded previously in this specific OpenGL context using `sglLoadBitmapFont` or `sglLoadTextureFont`. After calling this function, all text functions will work with the font specified here.

`sglSetFontSize`

SYNTAX

```
void sglSetFontSize( float size )
```

PURPOSE

The `sglSetFontSize` function sets the current font size to the given size. Note that the current font (which was set via `sglSetFont` has to support the given size.

`sglSetIndexedColor`

SYNTAX

```
void sglSetIndexedColor( int index )
```

PURPOSE

The `sglSetIndexedColor` function sets the current drawing color to the color with the given index, using an internal table of colors. This function is mainly thought to have an easy way to iterate through an amount of different colors, without caring *which* colors these actually are. If you want to use a table of specific colors, you should implement this yourself.

`sglTextWidth`

SYNTAX

```
float sglTextWidth( const String& text )
```

PURPOSE

The `sglTextWidth` function returns the width (in pixels) it would take to draw the given text with the current font in its current setting.

3.11 2d Graphics Library

3.11.1 Introduction

Inca offers two principal ways of dealing with graphics. First there's OpenGL. OpenGL is covered in the OpenGL chapter in this documentation. Second, and this is what this chapter about, there's rasterized two dimensional graphics. Inca provides a primitive class, called a `Pixmap`, that represents a two-dimensional image with a certain resolution. Inca further offers functions to draw into these `Pixmap`s using either special functions provided by Inca , or using OpenGL. The point here is that whenever you want to work on some kind of bitmap presentation of images (as for loading textures into OpenGL, or saving images rendered using OpenGL to disk), you will need to use the `Pixmap` class describes in this chapter.

3.11.2 2d Graphics

Inca offers a set of primitive functions to draw two dimensional graphics. The easiest way to draw something is to open up a graphics console window and start drawing. You can open this window by using the `PixmapConsole` class. Trying to draw something prior to opening a console using the `PixmapConsole` class will result in an error.

The methods of the `PixmapConsole` are very similar to the OpenGL console classes found in the OpenGL section of this documentation. The following example opens a console window and draws some graphics primitives:

```
PixmapConsole console;
console.setResolution( 640, 480 );
console.open();

setColor( 0, 0, 0 );
moveTo( 10, 10 );
lineTo( 200, 100 );

setColor( 255, 0, 0 );
fillRect( 10, 10, 20, 20 );

setColor( 0, 255, 0 );
frameOval( 50, 100, 300, 200 );

console.swap();
```

Note that calling the `swap` method in the last line is strictly necessary to display the graphics drawn on the screen.

The code above will draw a black line, a red filled rectangle, and a green framed oval. For details how the functions work, take a look in the function documentation below. Basically, at every moment there's one active Pixmap you're drawing in, so every drawing call like setColor or moveTo will be performed on that specific Pixmap. For the moment this doesn't matter, since there's only the graphics console window. However, you can also draw in different Pixmaps. Anyways, every graphics Pixmap you draw into (in this example it's the console window), keeps track of a current color and a pen location. You can change the current color using the setColor function, you can move the pen using the moveTo function. You can draw a line from the current pen location to a new location using the lineTo function. The rectangle and oval drawing function don't care about the pen location. Using a bunch of functions of this kind, you can draw images.

A `Pixmap` is the collection of pixels that make up a two dimensional image. The coordinate system of a `Pixmap` has its origin in the top left corner and extends to the right bottom. So, for example, in the graphics console in the example above, the top left most pixel has coordinates (0,0) and the pixel in the far right down bottom has coordinates (639,479).

There are two classes of 2d drawing functions in Inca . Functions that draw jaggy graphic primitives, and functions that try to antialias them. For example there's the lineTo function, that draws jaggy lines, and the smoothLineTo function, that draws antialiased lines. The smooth functions take floating point coordinates and can draw with brushes of varying thickness. They are however incredibly slower than their jaggy colleagues. Note that the pen locations for jaggy and smooth functions are separate, so you can actually imagine having two pens, one for jaggy and one for smooth drawing.

Note that the 2d drawing functions just describes are *not* OpenGL functions. They're implemented directly on the pixel represenation of the data and do not call OpenGL functions. You may however - as will be seen later in this text - draw using OpenGL functions.

3.11.3 The Pixmap Class

If you don't want to draw to the graphics console but to an offscreen image, or if you want to load an image from a file to work with it you will have to use the Pixmap class. It holds a pixel image of a certain size. Several so called pixel formats are supported. They define how the pixel colors are ordered in memory. The code

```
Pixmap myPixmap;
myPixmap.allocate( 256, 256, PIXEL_FORMAT_24_RGB );
```

allocates a pixmap, that is 256 by 256 pixels big and uses 24 bit color depth, with the color components red, green and blue placed in this order in consecutive bytes. Currently the following pixels formats are supported by Inca

```
PIXEL_FORMAT_8_INDEXED_GRAY
PIXEL_FORMAT_8_INDEXED_COLOR
PIXEL_FORMAT_24_RGB
PIXEL_FORMAT_24_GBR
PIXEL_FORMAT_24_BRG
PIXEL_FORMAT_24_BGR
PIXEL_FORMAT_24_GRB
PIXEL_FORMAT_24_RBG
PIXEL_FORMAT_32_RGBA
```

Drawing

To draw something into a Pixmap, you have to use the drawToPixmap function. This tells Inca that subsequent drawing commands should go to the Pixmap handed down there. For example

```
Pixmap myPixmap;

myPixmap.allocate( 256, 256, PIXEL_FORMAT_24_RGB );

drawToPixmap( &myPixmap );
setColor( 0, 0, 255 );
fillRect( 64, 64, 128, 128 );
```

will draw a blue rectangle in the Pixmap myPixmap just created. To switch back the context to the graphics window again, use the drawToConsole function. You can load an image from a file using the load method of the Pixmap class. Currently, only uncompressed tga is supported. Using the convert function you can make sure that a Pixmap is converted to a certain pixel format. For example a common code for loading an OpenGLtexture could be

```
Pixmap myPixmap;

// load the texture from a tga file
myPixmap.load( "images/textures/coolTexture.tga" );

// make sure the image is in RGBA format
myPixmap.convert( PIXEL_FORMAT_32_RGBA );

byte* pixels = myPixmap.pixels();

// now we could set up an OpenGL texture, with pixels
// being the address of the first top left pixel
```

Pixel Packing

The Pixmap class allocates images of pixels in such a way, that no interleaving and no additional space at the end of rows occurs. Some libraries optimize image pixel accesses, by making the row width in bytes a multiple of 4 or 8. The Pixmap class doesn't do that. You can at all times expect, that each pixel is tightly packed behind the one preceding it.

3.11.4 Mixing OpenGL and Pixmaps

Sometimes it might get necessary for you to mix graphics that are drawn with OpenGL and `Pixmaps`. Inca offers two classes for this purpose, and they're called `G2Pixmap` and `G3Pixmap`. Both `G2Pixmap` and `G3Pixmap` are extensions from `Pixmap`, so they have all the methods and properties just described for `Pixmaps`.

On the other hand, `G2Pixmap` and `G3Pixmap` are also OpenGL contexts, which means you can issue OpenGL commands to them and OpenGL will draw into the `Pixmap`. Stated differently, `G2Pixmap` and `G3Pixmap` offer a way to do offscreen drawing of OpenGL in Inca .

A common example of when you might want to use `G2Pixmap` or `G3Pixmap` is when you want to write an image to disk that was rendered using OpenGL. You might proceed like this:

```
G3Pixmap myPixmap;

myPixmap.allocate( 500, 500 );

...do your OpenGL drawing here...

glFinish();
myPixmap.save( "myImage.bmp" );
```

After calling `allocate` on the `Pixmap` you have a valid OpenGL context you can issue commands to. They will be drawin into the `Pixmap` `myPixmap`. When you're done, you can just save the pixmap to disk by calling the `Pixmap`'s save method. Note that prior to doing this, we call `glFinish` in the example. You should do this also, as we have to make sure that OpenGL has finished drawing when we save the image to disk.

The difference between `G2Pixmap` and `G3Pixmap` resembles the difference between `G2Console` and `G3Console`. See the explanations in the chapter about OpenGL for more details.

3.11.5 Overview

Context Functions

```
void setPixmapPort( Pixmap* pixmap )
Pixmap* getPixmapPort()
```

Color Functions

```
void setColor( int index )
void setColor( int r, int g, int b, [int a] )
```

Jaggy Drawing Functions

```
unsigned int getPixel( int x, int y )
void fillOval( int x1, int y1, int x2, int y2 )
void fillRect( int x1, int y1, int x2, int y2 )
void frameRect( int x1, int y1, int x2, int y2 )
void lineTo( int x, int y )
void moveTo( int x, int y )
void putPixel( int x, int y )
void blit( Pixmap* pixmap, int sx, int sy,
    int dx, int dy, int w, int h )
```

Smooth Drawing Function

```
void smoothFrameOval( float x1, float y1, float x2, float y2 )
void smoothFrameRect( float x1, float y1, float x2, float y2 )
void smoothLineTo( float x, float y )
void smoothMoveTo( float x, float y )
```

PixmapConsole Methods

```
void setResolution( int width, int height )
void open()
void swap()
int width()
int height()
void makePixmapPort()
```

Pixmap Methods

```
void allocate( int width, int height, PixelFormat format )
```

```
void convert( PixelFormat format )
int height()
void load( String path )
void save( String path )
byte* pixels()
int pitch()
void release()
int width()
```

3.11.6 Pixmap Method Reference

`allocate`

SYNTAX

```
void allocate( int width, int height,
PixelFormat format )
```

PURPOSE

The `allocate` method allocates an pixel image width pixels wide and height pixels height using the given `PixelFormat` format.

`convert`

SYNTAX

```
void convert( PixelFormat format )
```

PURPOSE

The `convert` method converts the pixels of a pixmap to the given PixelFormat. If the pixmap is already in that format, nothing will happen.

`height`

SYNTAX

```
int height()
```

PURPOSE

The `height` method returns the height of this pixmap in pixels.

`load`

SYNTAX

```
void load( String path )
```

PURPOSE

The `load` method loads an image from a file specified in the path parameter into this pixmap. Any image currently allocated by this pixmap will be deleted. Currently only uncompressed tga images are supported.

`pitch`

SYNTAX

```
int pitch()
```

PURPOSE

The `pitch` method returns the number of bytes in a row of a pixmap. Due to the tight pixel packing of the `Pixmap` class under all circumstances, it will always return width $\cdot$ (size of one pixel in bytes).

`pixels`

SYNTAX

```
byte* pixels()
```

PURPOSE

The `pixels` method returns a pointer to the first, top left pixel of the `Pixmap` image.

`release`

SYNTAX

```
void release()
```

PURPOSE

The `release` method releases any memory currently occupied by this Pixmap if any. After calling this function, you may no longer draw to this Pixmap, since its image data has been deleted.

`save`

SYNTAX

```
void save( String path )
```

PURPOSE

The `save` method save an image to the file specified in the `path` parameter. The image format in which this image should be saved is determined from `path`'s extension. Currently only uncompressed tga and bmp images are supported (i.e. ".tga" and ".bmp" extensions).

`width`

SYNTAX

```
int width()
```

PURPOSE

The `width` method returns the width of this pixmap in pixels.

3.11.7 General Function Reference

`blit`

SYNTAX

```
void blit( Pixmap* pixmap,
int sx, int sy,
int dx, int dy,
int w, int h )
```

PURPOSE

The `blit` function copies a rectangular block of pixels from the current pixmap as source to the given `Pixmap` pixmap as destination. The rectangular block has its top left corner at (sx,sy) in the source pixmap, and at (dx,dy) at the destination pixmap. It is w pixels wide and h pixels high. If the source and destination pixmap have different pixel format, automatic pixel format conversion will take place.

`getPixel`

SYNTAX

```
unsigned int getPixel( int x, int y )
```

PURPOSE

The `getPixel` function obtains the color of the pixel at the location (x,y) in the current pixmap and returns it.

`getPixmapPort`

SYNTAX

```
Pixmap* getPixmapPort()
```

PURPOSE

The `getPixmapPort` function returns the `Pixmap` to which currently 2d drawing commands are sent, i.e. the `Pixmap` that was set using a call to `setPixmapPort`.

`fillOval`

SYNTAX

```
void fillOval( int x1, int y1, int x2, int y2 )
```

PURPOSE

The `fillOval` function draws a jaggy filled oval with one corner being (x1,y1) and the other corner being (x2,y2) using the current color. Actually, the oval is drawn inside the (imagined) rectangle you specify. A suggested convention could be, that (x1,y1) is be the top left corner, however fillOval will also draw the oval, if this is not the case.

`fillRect`

SYNTAX

```
void fillRect( int x1, int y1, int x2, int y2 )
```

PURPOSE

The `fillRect` function draws a jaggy filled rectangle with one corner being (x1,y1) and the other corner being (x2,y2) using the current color. A suggested convention could be, that (x1,y1) is be the top left corner, however fillOval will also draw the rectangle, if this is not the case.

`frameOval`

SYNTAX

```
void frameOval( int x1, int y1, int x2, int y2 )
```

PURPOSE

The `frameOval` function draws a jaggy framed oval with one corner being (x1,y1) and the other corner being (x2,y2) using the current color. Actually, the oval is drawn inside the (imagined) rectangle you specify. A suggested convention could be, that (x1,y1) is be the top left corner, however frameOval will also draw the oval, if this is not the case.

`frameRect`

SYNTAX

```
void frameRect( int x1, int y1, int x2, int y2 )
```

PURPOSE

The `frameRect` function draws a jaggy framed rectangle with one corner being (x1,y1) and the other corner being (x2,y2) using the current color. A suggested convention could be, that (x1,y1) is be the top left corner, however frameRect will also draw the rectangle, if this is not the case.

`lineTo`

SYNTAX

```
void lineTo( int x, int y )
```

PURPOSE

The `lineTo` function draws a jaggy line beginning at the current jaggy pen location and ending at (x,y) using the current color. Moves the jaggy pen to location (x,y) afterwards.

`moveTo`

SYNTAX

```
void moveTo( int x, int y )
```

PURPOSE

The `moveTo` function moves the jaggy pen to location (x,y).

`putPixel`

SYNTAX

```
void putPixel( int x, int y )
```

PURPOSE

The `putPixel` function draws one pixel at the location (x,y) using the current color.

`setColor`

SYNTAX

```
void setColor( int index )
void setColor( int r, int g, int b, [int a] )
```

PURPOSE

The `setColor` function sets the drawing color for the current `Pixmap`. The one parameter variant makes sense for `Pixmap`s that are paletted, i.e. each pixels consists of an index into a color table. The second version should be used for all other types of `Pixmap`s. The first three parameters of the second variant specify red, green and blue components of a color as integers in the range [0,255]. Black in this notation is (0,0,0), white is (255,255,255). The last parameter of the second variant specifies an alpha component, which makes sense if the `Pixmap` pixel format supports it. It's especially useful if you want to use Pixmaps as textures for OpenGLwhich is generally a good idea.

`setPixmapPort`

SYNTAX

```
void setPixmapPort( Pixmap* pixmap )
```

PURPOSE

The `setPixmapPort` function tells Inca that subsequent drawing operations should go to the `Pixmap` specified as parameter here. In other words, it makes the given `Pixmap` active and eligible for drawing commands.

smoothFrameOval

SYNTAX

```
void smoothFrameOval( float x1, float y1, float x2,
float y2 )
```

PURPOSE

The `smoothFrameOval` function draws an antialiased framed oval with one corner being (x1,y1) and the other corner being (x2,y2) using the current color. A suggested convention could be, that (x1,y1) is be the top left corner, however smoothFrameOval will also draw the oval, if this is not the case.

smoothFrameRect

SYNTAX

```
void smoothFrameRect( float x1, float y1, float x2,
float y2 )
```

PURPOSE

The `smoothFrameRect` function draws an antialiased framed rectangle with one corner being (x1,y1) and the other corner being (x2,y2) using the current color. A suggested convention could be, that (x1,y1) is be the top left corner, however smoothFrameRect will also draw the rectangle, if this is not the case.

`smoothLineTo`

SYNTAX

```
void smoothLineTo( float x, float y )
```

PURPOSE

The `smoothLineTo` function draws an antialiased line beginning at the current antialiased pen location and ending at (x,y) using the current color. It moves the antialiased pen to location (x,y) afterwards.

`smoothMoveTo`

SYNTAX

```
void smoothMoveTo( float x, float y )
```

PURPOSE

The `smoothMoveTo` function moves to antialiasing pen to location (x,y).