Jancy
Programmer's Manual

Vladimir Gladkov

June 4, 2016

Contents

Chapter 1

Introduction

1.1 Abstract

Jancy is a scripting programming language with LLVM back-end. Jancy offers a lot of convenient features for low-level IO (input-output) and UI (user-interface) programming which are not found in mainstream languages (and sometimes, nowhere else). This includes safe pointer arithmetics, high level of source-level and ABI compatibility with C/C++, reactive programming, built-in generator of incremental lexers/scanners and a lot more.

1.2 Motivation

Why create yet another programming language? Like, there isn't enough of them already?

I have asked myself these questions hundreds of times, over and over again. I can name dozens of possible arguments against creating a new programming language. I understand all the difficulties a language creator is doomed to face before he can gather any meaningful number of users of the new language. And I still believe that creation of Jancy was justified. The truth is, Jancy was not created just to fix the infamous fatal flaw of other languages (aka *they didn't write it*). Of course, the passion to invent was a significant driving force, but there was *practival* reasoning besides that.

During development of a product called IO Ninja (a universal all-in-one low-level IO debugger) we were looking for a scripting language with the support for safe pointers and safe pointer arithmetics. After not finding one, we basically had a simple choice: either we settle for some existing scripting language without safe pointer arithmetics (say, embedded Python) or create a new scripting language – tailor-suited for this task. Create it both for ourselves, and for other developers who could be in need for a scripting language really good at handling binary data. Needless to say, we have chosen the later option – otherwise, you would not be reading this manual.

Of course, besides featuring safe pointers and safe pointer arithmetics, Jancy offers a long list of other useful but rarely found facilities. So let's outline the distinguishing features of Jancy below. And for those of you wondering *what's in a name*, Jancy is an acronym: [in-between] Java-and-C.

1.3 Design Principles

- Object-oriented scripting language for IO and UI programming with C-family syntax
- ABI (application-binary-interface) compatibility with C
- Automatic memory management via accurate GC (garbage collection)
- LLVM (Low Level Virtual Machine) as a back-end

1.4 Key Features

- Safe pointers and pointer arithmetic
- Unprecedented for scripting languages source-level compatibility with C
- Built-in Reactive Programming support
- Built-in regexp-based generator of incremental lexers/scanners

1.5 Other Notable Features

- Exception-style syntax over error code checks
- Properties – the most comprehensive implementation thereof
- Multicasts and events (including weak events, which do not require to unsubscribe)
- Multiple inheritance
- Const-correctness
- Thread local storage
- Weak pointers (do not retain objects)
- Partial application for functions and properties
- Scheduled function pointers
- Bitflag enums
- Perl-style formatting
- Hexadimal, binary and multi-line literals

Chapter 2

Jancy Language

2.1 Hello World

Tradition dictates to start any language book with a *"Hello World"* program, so I'm not going to re-invent the wheel in this paricular case:

```
int main ()
{
    printf ("hello world!\n");
    return 0;
}
```

No explanation is needed, I suppose? Keeping Jancy language syntax as close to C/C++/Java as possible was imperative, and I believe a very pleasant level of source-level compatibility was eventually achieved. Any experienced C, C++ or Java programmer should be able to read and understand most of Jancy code from the get-go, without any special training. Moreover, often times it is possible to simply copy-paste C/C++/Java code snippets into Jancy source files and compile them with little to no modifications.

2.2 Primitive Types

System of primitive types in Jancy is a lot like the one in C/C++ with one exception: there is native support for integers with reversed byte-order (aka *bigendians*).

Bigendians are widely used throughout most network protocol stacks and everyone who was ever involved in network programming knows two things: number one, it kind of clutters the programs to constantly reverse byte order manually with all these *ntohs/htons/ntohl/htonl* calls. Even more importantly, number two – it's *sooo* easy to forget to reverse byte order, thus getting a program which compiles just fine but has a logical bug in it! Since bigendians are so abundant in networking, and Jancy is intended for low-level IO programmers, bigendians got native support in Jancy.

Below is a list of all primitive types in Jancy.

- void
- bool

- int8 (char)
- unsigned int8 (uint8_t, byte_t)
- int16 (short)
- unsigned int16 (uint16_t)
- bigendian int16
- bigendian unsigned int16
- int32 (int)
- unsigned int32 (uint32_t)
- bigendian int32
- bigendian unsigned int32
- int64
- unsigned int64 (uint64_t)
- bigendian int64
- bigendian unsigned int64
- intptr (int32/int64 – depending on platform)
- unsigned intptr (uint32_t/uint64_t – depending on platform)
- float
- double

2.3 Simple Declarations

Syntax of declarations in Jancy is common for all the C-family languages and can be expressed with the following formula: *specifier(s)-declarator(s)*.

```
int a;
```

In example above *int* is a type specifier and *a* is a declarators. This declaration creates a new integer variable (or field) named *a* – just like you would expect in C-world. Unlike C/C++ language though, any uninitialized variable in Jancy is *zero-initialized*, so *a* will be holding *zero*.

Declaring *functions* in Jancy is also similar to any C-family language:

```
void foo (
    int a,
    double b
    )
{
    // ...
}
```

This defines a function which return no value and takes two arguments of types *int* and *double* – again, nothing unexpected here.

It's important to say a couple of words on *arrays*, however. In Java, C#, D and other modern languages arrays are *dynamically-sized*; in C/C++ compiler-generated arrays are *fixed-sized* while dynamically-sized arrays are implemented as classes. Since being able to copy-paste C/C++ declarations of network protocol headers was crucial, Jancy adopts C/C++ model:

```
int a [10] [20];
```

This declares a two-dimensional array of integers (10 rows and 20 columns) named *a*. Now let's proceed to something more complex.

2.4 Advanced Declarations

Just like C/C++ (and even more so) Jancy allows declaring quite sophisticated entities in one go; therefore, declarations in Jancy can be rather complex, and both specifier and declarator can be composite.

Specifiers can contain:

- Access specifiers (*public* or *protected*)
- Storage specifiers (e.g. *static*, *typedef*, *virtual* etc)
- Type specifiers (e.g. *char*, *int*, *double* etc)
- Type modifiers (e.g. *const*, *volatile*, *unsigned* etc)

Declarators can contain:

- Type modifiers (e.g. *const*, *volatile*, *unsigned* etc)
- Pointer prefixes (*)
- Declarator identifiers (e.g. *myVariable*)
- Declarator special function kind (e.g. *construct*, *get*, *set* etc)
- Array suffixes (e.g. *[10]*)
- Function suffixes (e.g. *(int, double)*)
- Bitfied suffixes (e.g. *:10*)
- Initializers (e.g. *= 0*)
- Function bodies (e.g. { *return 0;* })

A realistic example of a complex declaration could look like:

```
static int const* function* a [2] () = { foo, bar };
```

Here *static* is a storage specifier, *int* is a type specifier, *const* and *function* are type modifiers, two *asterisks* are pointer prefixes, *a* is a declarator name, *[2]* is a array suffix, *()* is a function suffix and = { *foo, bar* } is an initializer. This line declares a static variable *a* as an array of two elements of type "pointer to a function which takes no arguments and returns a const-pointer to int" and initializes it with pointers to functions *foo* and *bar*. Wheh! Seriously, it's much easier to read the code than its explanation.

C/C++ equivalent of the above example would look like:

```
static int const* (*a [2]) () = { foo, bar };
```

Now, if we add one extra layer of function pointers, C/C++ falls short of declaring it in one go (you will receive *function returns function* error); Jancy syntax still allows to do so (not like that could be crucial in any realistic scenario; just a small demonstration of flexibility):

```
static int const* function* function* a [2] () (int);
```

There are no nested declarators in Jancy. Nested declarators in C/C++ emerged as a solution (and in my personal opinion, not an elegant one) to the problem of resolving ambiguities in complex pointer-to-function declarations. Like you just saw, Jancy uses a different approach with type modifiers *function*, *property*, *array*:

```
int property* function* array* a [2] [3] ();
```

Here *a* is an array of three elements of type *pointer to array of two elements of type pointer to a function taking no arguments and returning a pointer to int property*. Speaking formally, the rules for reading Jancy declaration are as follows. First, you start unrolling declarator's pointer prefixes right-to-left. If type modifiers of a pointer prefix requires a suffix, you unroll the first suffix. After all pointer prefixes are unrolled, you unroll the remaining suffixes.

In reality, however, you are unlikely going to need mind-boggling declarations like the one above. It's always possible to split an overcomplicated declaration into two (or more) using good-old typedefs: just like in C/C++, Jancy declarations with *typedef* storage specifier result in creation of a type alias:

```
typedef double DoubleBinaryFunc (double, double);
DoubleBinaryFunc* funcPtr; // use new typedef to declare a variable

typedef int IntArray [10] [20];
IntArray foo ();  // use new typedef as a retval type
```

There are other important differences with C/C++. In Jancy named type declaration is *not* a type specifier. The following code, perfectly valid in C/C++ will produce an error in Jancy:

```
struct Point
{
    int m_x;
    int m_y;
} point;
```

In Jancy you cannot declare a named type and immediatly use it to declare a variable or a field. Therefore, to fix previous example, we need to simply split a single declaration into two:

```
struct Point
{
    int m_x;
    int m_y;
}

Point point;
```

Note that declaration of a named type does not need to end with a semicolon (needless to say, it will also compile should you add a semicolon).

Jancy does not require *declaration-before-usage* at global scope. Therefore, there is no need to create so-called *forward declarations* of functions or types, so the following example will compile in Jancy, but not in C/C++:

```
void foo ()
{
    A a;
    B b;
}

struct A
{
    B* m_b;
}

struct B
{
    A* m_a;
}
```

It is allowed to omit type specifier; *void* type is assumed in this case. This is done to unify rules applied to declaration of normal functions and *special* functions like *constructors*, *destructors*, *setters* etc. In Jancy the following two declarations are equivalent:

```
void foo ()
{
    //...
}

foo ()
{
    //...
}
```

2.5 Curly Initializers

Jancy supports a convenient method of assigning aggregate values with curly initializers:

Classic C-style curly-intializers:

```
int a [] = { 1, 2, 3 };
```

It's OK to skip elements leaving them zero-initialized:

```
int b [10] = { ,, 3, 4,,, 7 };
```

You can use both index- and name-based addressing:

```
struct Point
{
    int m_x;
    int m_y;
    int m_z;
}

Point point = { 10, m_z = 30 };
```

You can also use curly-initializers in assignment operator after declaration:

```
point = { , 200, 300 };
```

...or in a new operator:

```
Point* point2 = new Point { m_y = 2000, m_z = 3000 };
```

2.6 Storage Specifiers

Both global and local variables can have *static* or *thread* storage specifiers. *static* means that the memory for variable is allocated at the program start and it stays there until the program terminates. *thread*, on the other hand, means that variable is allocated from the thread local storage – each thread in program has its own copy of such variable. To avoid unnecessary performance penalties during thread start, two limitations are imposed on *thread* variables. The first limitation is: thread variables cannot have initializers; the second – thread variables cannot be aggregate (e.g. class, struct, array etc).

Now, what if you *do* need an aggregate thead variable and/or a thread variable which needs to be initilized? The answer is simple – use pointers:

```
thread MyClass* threadClass;

// ...

if (!threadClass)
    threadClass = new MyClass (10, 20, 30);
```

There is one more storage class, which actually cannot be explicitly expressed with storage specifier – *local* storage. The actual memory for local storage can come from *stack* or from *GC-heap*, but the most important thing about local storage is this: every time intruction pointer goes through variable declaration, that will be a *new* copy of local variable.

If storage specifier is omitted, then global variables get assigned *static* storage class and local variables – *local* storage class.

Member fields of classes, structs and unions can have both *static* and *thread* storage specifiers – the meaning of these are identical to what we have just discussed. Member fields can also have *mutable* storage specifier, which means that this member field can be modified using *const* pointer to the parent class/struct/union – just like in C/C++.

Now let's talk about functions. Global functions always have *static* storage class. Member functions, on the other hand, can have the following storage specifiers:

- static
- virtual
- abstract
- override

static in respect to member functions mean that this function does not accept implict *this* argument. Therefore, static functions cannot access any non-static member fileds. *virtual*, *abstract* and *override* specifiers allow programmer to create virtual functions – i.e. the functions which can be overriden in derived classes. Note that neither structs nor unions are not allowed to have virtual functions, only *classes* do.

2.7 Properties

In the context of a programming language, property is an entity that looks like a variable/field but allows performing actions on read or write. Functions implementing these actions are called *accessors*. The read accessor is called a *getter*, and the write accessor is called a *setter*. Each property has a single getter and optionally one or more setters.

If a setter is overloaded then the selection of particular setter function will be performed according to the same rules that apply to regular overloaded functions. If a property has no setters then it is a *const* property (*read-only* term has a special meaning in Jancy; more on that in *Dual Modifiers*).

Jancy provides two methods of declaring a property: simple and full.

2.7.1 Simple Declaration

Jancy supports what I believe to be the most natural syntax for declaring properties:

```
int property g_simpleProp;
```

This syntax is ideal for declaring interfaces or when the developer prefers to follow the C++-style of placing definitions outside of a class:

```
int g_simpleProp.get ()
{
    return rand () % 3;
}

g_simpleProp.set (int x)
{
    // set property value
}
```

Const properties can use a simple declaration syntax:

```
int const property g_simpleReadOnlyProp;

int g_simpleReadOnlyProp.get ()
{
    return rand () % 3;
}
```

For obvious reasons, this simple syntax is only possible if a property has no overloaded setters, in which case you should use the second method: full property declaration.

2.7.2 Full Declaration

A full property declaration looks a lot like a declaration for a class. It implicitly opens a namespace and allows for overloaded setters, member fields, helper methods, constructors/destructors etc.

```
property g_prop
{
    int m_x = 5; // member field with in-place initializer

    int get ()
    {
```

```
        return m_x;
    }

    set (int x)
    {
        m_x = x;
        update ();
    }

    set (double x); // overloaded setter
    update (); // helper method
}
```

A body of a method can be placed on the right (Java-style) or outside (C++-style).

2.7.3 Indexed Properties

Jancy also supports *indexed* properties, which are properties with array semantics. Accessors for such properties accept additional index arguments. Unlike with real arrays, a property index doesn't have to be of the integer type, nor does it mean *index* exclusively – it is up to the developer how to use it.

Simple indexed property declaration syntax:

```
int g_x [2];

int indexed property g_simpleProp (unsigned i);

// here the index argument is really used as the array index

int g_simpleProp.get (unsigned i)
{
    return g_x [i];
}

g_simpleProp.set (
    unsigned i,
    int x
    )
{
    g_x [i] = x;
}
```

A similar property declared using full syntax:

```
property g_prop
{
    int m_x [2] [2];

    // more than one index argument could be used

    int get (
        unsigned i,
        unsigned j
        )
    {
        return m_x [i] [j];
    }

    set (
        unsigned i,
        unsigned j,
        int x
```

```
        )
    {
        m_x [i] [j] = x;
    }

    // setters of indexed property can be overloaded

    set (
        unsigned i,
        unsigned j,
        double x
        )
    {
        m_x [i] [j] = (int) x;
    }
}
```

Accessing indexed properties looks like accessing arrays

```
int indexed property g_prop (
    unsigned i,
    unsigned j
    );

foo ()
{
    int value = g_prop [10] [20];

    // ...

    g_prop [30] [40] = 100;

    // ...
}
```

2.7.4 Autoget Properties

In most cases a property getter is supposed to return a variable value or field, and all of the property logic is contained in the property setter. Jancy takes care of this common case by providing autoget properties. Such properties do not require a getter implementation: the compiler will access the data variable/field directly if possible, or otherwise generate a getter to access it.

Simple syntax for declaring autoget property:

```
int autoget property g_simpleProp;

g_simpleProp.set (int x)
{
    m_value = x; // name of compiler-generated field is 'm_value'
}
```

The same property declared using full syntax:

```
property g_prop
{
    int autoget m_x; // 'autoget' field implicitly makes property 'autoget'

    set (int x)
    {
        m_x = x;
    }
```

```
    // setters of autoget property can be overloaded

    set (double x)
    {
        m_x = (int) x;
    }
}
```

Autoget and indexed property modifiers are mutually exclusive.

2.7.5 Bindable Properties

Jancy supports bindable properties which automatically notify subscribers when they change. Bindable properties play crucial role in the reactive programming.

Simple bindable property declaration syntax:

```
int autoget bindable property g_simpleProp;

g_simpleProp.set (int x)
{
    if (x == m_value)
        return;

    m_value = x;
    m_onChanged (); // name of compiler-generated event is 'm_onChanged'
}

onPropChanged ()
{
    // ...
}

int main ()
{
    // ...

    bindingof (g_simpleProp) += onPropChanged;
    g_simpleProp = 100; // onPropChanged will get called
    g_simpleProp = 100; // onPropChanged will NOT get called

    // ...
}
```

Similar property declared using full syntax:

```
property g_prop
{
    autoget int m_x; // 'autoget' field implicitly makes property 'autoget'
    bindable event m_e (); //'bindable' event implicitly makes property 'bindable'

    set (int x)
    {
        m_x = x;
        m_e ();
    }
}
```

2.7.6 Bindable Data

Jancy offers fully compiler-generated properties: getter, setter, back-up field and on-change event are all generated by the Jancy compiler. These degenerate properties are designed to track data changes: they can be used as variables or fields that automatically notify subscribers when they change.

```
int bindable g_data;

onDataChanged ()
{
    // ...
}

int main ()
{
    // ...

    bindingof (g_data) += onDataChanged;
    g_data = 100; // onDataChanged will get called
    g_data = 100; // onDataChanged will NOT get called

    // ...
}
```

2.8 Global Namespaces

Just like in C++ and C#, Jancy features support for global namespaces. It is OK to open a nested namespace in one go; it is also OK to open and close the same namespace multiple times for adding new elements.

```
namespace a.b.c {
namespace d {

} // namespace a.b.c.d
} // namespace a.b.c

namespace a.b.c.d {
} // namespace a.b.c.d
```

2.9 Extension Namespaces

Jancy offers a way to extend the functionality of existing classes with extension namespaces. An extension namespace declares additional methods which have access to all the members of the class that they extend. There are certain limitations imposed on the extension methods. These ensure that if your code runs without extension namespaces, then it runs exactly the same with the introduction of any extension namespace(s):

```
class C1
{
    protected int m_x;

    construct (int x)
    {
        printf ("C1.construct (%d)\n", x);
        m_x = x;
    }

    foo ()
    {
```

```
        printf ("C1.foo () { m_x = %d }\n", m_x);
    }
}

extension ExtC1: C1
{
    bar ()
    {
        // extension method has access to protected data
        printf ("C1 (extend).bar () { m_x = %d }\n", m_x);
    }

    static baz ()
    {
        printf ("C1 (extend).baz ()\n");
    }

    // constructors cannot be part of extension namespace
    construct (double x); // error

    // operator methods cannot be part of extension namespace
    int operator += (int x); // error

    // virtual methods cannot be part of extension namespace
    virtual baz (); // error
}

// entry point

int main ()
{
    C1 c construct (100);
    c.foo ();
    c.bar ();  // bar () is extension method
    C1.baz (); // baz () is static extension method
    return 0;
}
```

2.10 Named Types

2.10.1 Structs/Unions

Jancy features structs and unions just the way C/C++ does with the following exceptions:

2.10.2 Classes

Classes are special types of data with ancillary header holding the type information, virtual table pointer, etc.

Member Access Control

There are only 2 access specifiers in Jancy: public and protected (read more about Jancy access control model here).

Both Java and C++ styles of declaring member access are supported. Contrary to most modern languages,

the default access mode is public.

```
class C1
{
    int m_x; // public by default

protected: // C++-style of access specification
    int m_y;

    public int m_z; // Java-style of access specification

    // ...
}
```

Method Body Placement

Jancy supports both Java and C++ styles of placing method bodies: it is up to developer to choose whether this will be *in-class* or *out-of-class* (i.e. in a different compilation unit).

```
class C1
{
    foo (); // C++-style

    bar () // Java-style
    {
        // ...
    }
}

// out-of-class method definition

C1.foo ()
{
    // ...
}
```

Construction/Destruction

Jancy supports in-place field initializers, constructors, destructors, static constructors, static destructors, and preconstructors. Constructors can be overloaded, the rest of construction methods must have no arguments.

Preconstructors are special methods that will be called before any of the overloaded constructors (similar to Java initializer blocks).

Constructor and destructor syntax is a bit different from most languages, as Jancy uses explicit keywords.

```
class C1
{
    int m_x = 100; // in-place initializer

    static construct ();
    static destruct ();

    preconstruct ();  // will be called before every overloaded constructor

    construct ();
    construct (int x);
    construct (double x);

    destruct ();
```

```
    // ...
}

// out-of-class method definitions

C1.static construct ()
{
    // ...
}

// ...
```

Jancy has pointers and, contrary to most managed languages, has no distinction between value-types and reference-types.

What is a pointer, must look like a pointer.

A type of a class variable or a field does not get implicitly converted to a class pointer type. Like in C++, the declaration of a class variable or field is an instruction to allocate a new object.

Member class fields get allocated on a parent memory block, global class variables get station allocation, local class variables are allocated on heap (unless explicitly specified otherwise).

```
class C1
{
    // ...
}

class C2
{
    // ...

    C1 m_classField; // allocated as part of C2 layout
}

C2 g_classVariable; // allocated statically

foo ()
{
    C1 a;         // allocated on heap (same as: C1* a = heap new C1;)
    stack C1 b;   // allocated on stack (same as: C1* b = stack new C1;)
    static C2 c; // allocated statically (same as: C1* c = static new C1;)
    thread C2 d; // error: thread-local variable cannot be of class type
    thread C2* e = new C2; // OK

    // ...
}

// ...
```

Jancy has a small syntactic difference with regard to calling a constructor of a class variable or field. This is to address an inherent ambiguity of the C/C++ constructor invocation syntax:

```
C1 a (); // is it a function 'a' which returns C1?
         //     or a construction of variable 'a' of type C1?
```

This ambiguity is even trickier to handle in Jancy given the fact that Jancy does not enforce the *declaration-before-usage* paradigm. To counter the ambiguity, Jancy introduces a slight syntax modification which fully resolves the issue:

```
class C1
{
```

```
    construct ();
    construct (int x);

    // ...
}

C1 g_a construct ();
C1 g_b construct (10);

// with operator new there is no ambiguity, so both versions of syntax are OK

C1* g_c = new C1 construct (20);
C1* g_d = new C1 (30);
```

Operator Overloading

Jancy supports operator overloading. Like in C++, any unary, binary, cast or call operators can be overloaded.

```
class C1
{
    operator += (int d) // overloaded '+=' operator
    {
        // ...
    }
}

foo ()
{
    C1 c;
    c += 10;

    // ...
}
```

Multiple inheritance

Jancy uses a simple multiple inheritance model (multiple instances of shared bases – if any). The infamous virtual multiple inheritance model of C++ is not and will not be supported.

Multiple inheritance is an extremely useful and unfairly abandoned tool, which allows the most natural sharing of interface implementation.

Virtual methods are declared using keywords *virtual*, *abstract*, and *override*.

```
class I1
{
    abstract foo ();
}

class C1: I1
{
    override foo ()
    {
        // ...
    }
}

class I2
{
```

```
    abstract bar ();

    abstract baz (
        int x,
        int y
        );
}

class C2: I2
{
    override baz (
        int x,
        int y
        )
    {
        // ...
    }
}

struct Point
{
    int m_x;
    int m_y;
}

class C3:
    C1,
    C2,
    Point // it's ok to inherit from structs and even unions
{
    override baz (
        int x,
        int y
        );
}

// it's ok to use storage specifier in out-of-class definition by the way.
// (must match the original one, of course)

override C3.baz (
    int x,
    int y
    )
{
    // ...
}
```

Jancy provides keywords *basetype* and *basetype1..basetype9* to conveniently reference base types for construction or namespace resolution.

```
class Base1
{
    construct (
        int x,
        int y
        );

    foo ();
}

class Base2
{
    construct (int x);

    foo ();
}
```

```
class Derived:
    Base1,
    Base2
{
    construct (
        int x,
        int y,
        int z
        )
    {
        basetype1.construct (x, y);
        basetype2.construct (z);

        // ...
    }

    foo ()
    {
        basetype1.foo ();

        // ...
    }
}
```

2.10.3 Enums

Jancy brings a couple of enhancements to the enumeration types as well.

In Jancy, traditional enums conceal member identifiers within their enum namespaces to prevent namespace pollution. Plus, Jancy enums can be derived from an integer type. This comes handy when declaring fields of protocol headers.

```
enum IcmpType: uint8_t
{
    EchoReply               = 0,
    DestinationUnreachable  = 3,
    SourceQuench            = 4,
    Redirect                = 5,
    Echo                    = 8,
    RouterAdvertisement     = 9,
    RouterSelection         = 10,
    TimeExceeded            = 11,
    ParameterProblem        = 12,
    TimestampRequest        = 13,
    TimestampReply          = 14,
    InformationRequest      = 15,
    InformationReply        = 16,
    AddressMaskRequest      = 17,
    AddressMaskReply        = 18,
    TraceRoute              = 30,
}
```

To simplify porting existing C/C++ code into Jancy we offer an *exposed enum* variation, which behaves like a traditional C/C++ enum, i.e. it exposes the member identifier into the parent namespace.

```
exposed enum State
{
    State_Idle, // = 0
    State_Connecting,
    State_Connected,
    State_Disconnecting,
}
```

```
foo ()
{
    State state = State_Connecting; // State.State_Connecting is also ok
    state = 100; // error: cast int->enum must be explicit
}
```

Jancy also features bitflag enums, which are enumerations dedicated to describing a set of bitflags. A *bitflag enum* differs from a regular *enum* in the following aspects:

It's automatic value assignment sequence is 1, 2, 4, 8,... thus describing bit positions Bitwise OR operator — on two operands of matching *bitflag enum* types yields the same *bitflag enum* type Bitwise AND operator *&* on *bitflag enum* and integer yields the same *bitflag enum* type It's OK to assign *0* to a *bitflag enum* Like Jancy enums, bitflag enums do not pollute the parent namespace.

```
bitflag enum OpenFlags
{
    ReadOnly,       // = 0x01
    Exclusive          = 0x20,
    DeleteOnClose, // = 0x40
}

foo ()
{
    OpenFlags flags = 0; // 0 is ok to assign to 'bitflag enum'

    flags = OpenFlags.ReadOnly | OpenFlags.Exclusive | OpenFlags.DeleteOnClose;
    flags &= ~OpenFlags.Exclusive;
    flags = 200; // error: cast int->bitflag enum must be explicit
}
```

2.11 Pointer Types

Unsafe as they are, pointers are not something we can live without. Even languages without pointers (like Basic or Java) have pointers to classes and interfaces.

Pointers has always been considered an unsafe tool that could easily cause a program to crash or (worse!) silently corrupt user data.

Even if we limit the number of pointer kinds and pointer operations available to developers, we still won't be able to perform the complete analysis of pointer-related correctness at compile time.

For the purpose of the discussion that follows let's define *pointer safety*.

We will call a pointer *safe* if it's impossible to either crash a program or corrupt user data by accessing this pointer. This means that any invalid pointer access will be caught and handled by the language runtime.

2.11.1 Data Pointers

Do we even need data pointers? In C/C++ world that's not much of a question: using pointers is the one and only way of working with dynamic memory. What about the managed world, do we need data pointers there?

Most managed language designers believe that the answer is NO. This is largely because data pointers fall into the disadvantageous area on the risk/reward chart for most programming tasks. There is, however, one

programming area where the use of data pointers truly shines: working with binary data.

Here is an example. Try to write Java code that deconstructs an Ethernet packet. Compare the resulting mess of fixed index array references and word types assembled from bytes to the clean and efficient code in C that will superimpose protocol header structs on the buffer data, then traverse the packet using pointer arithmetic!

Since Jancy was conceived as the scripting language for our IO Ninja software, living without pointers was out of the question. Instead, we made data pointers safe. Safe data pointers and safe pointer arithmetic are among the biggest innovations of Jancy.

As with many languages, the Jancy runtime doesn't allow access to data via a pointer that failed the range check. Unfortunately, range checks are not enough for stack data pointers:

```
foo ()
{
    //...

    int* p;

    {
        int a = 10;
        p = &a;
    }

    int b = 20;

    *p = 30; // oh-oh

    //...
}
```

Pointer p obviously passes the range check (it has not been changed!) but accessing this pointer will write to the dead and, possibly, re-allocated location. That happens because stack pointers become invalid even without modification, simply by running out of scope.

To address this issue, Jancy pointer validators also maintain integer thread-local variable holding the target scope level. The Jancy runtime prevents storing an address with a higher scope level at the location with a lower scope level.

Simply put, the approach used by Jancy is this: check the data range at the pointer access, check the scope level at the pointer assignment.

```
int* g_p;

foo ()
{
    int* p = null;

    int x = *p; // error: pointer out of range

    int a [] = { 10, 20, 30 };
    g_p = a;    // error: storing pointer in location with lesser scope level

    int** p2 = &g_p;
    *p2 = &x;   // error: storing pointer in location with lesser scope level

    int i = countof (a);
    x = a [i];  // error: pointer out of range
}
```

Our safe pointers are not thread-safe. It's still possible to corrupt a pointer validator in a multi-threaded

environment. Still, our solution covers a lot of bases and future Jancy versions will likely address the issue by preventing race conditions on pointer validators.

Besides the normal data pointer with validators (*fat* or *safe* data pointers) Jancy also supports *thin* data pointers, which only hold the target address. This might be useful when writing performance-critical code, or for interoperability with the host C/C++ program. *Thin* pointers are not safe.

2.11.2 Class Pointers

Pointer arithmetic is not applicable to class pointers, therefore, class pointer validity can be ensured by performing a null-check on access and a scope level check on assignment. Scope level information could be stored in class header instead of class pointer, so class pointer does not need to be *fat* to be safe.

Jancy provides built-in support for a special kind of class pointers: *weak*. These pointers do not affect the lifetime of an object. Obviously, weak class pointers cannot be used to access an object they point to and can only be cast to strong pointers. If this cast operation returns non-null value, the result can be used to access the object normally; otherwise, the object has already been destroyed.

```
class C1
{
    //...
}

foo ()
{
    C1* c = new C1;
    C1 weak* w = c;

    // if we lose a strong pointer before GC run, the object will be collected

    jnc.runGc ();

    c = w; // try to restore strong pointer

    if (c)
    {
        // the object is still alive
    }
}
```

2.11.3 Const-correctness

As a language with pointers, Jancy fully implements the paradigm of const-correctness. The core idea behind const-correcness is to specifically mark pointers that cannot be used to modify the target object.

Admittedly, const-correctness generally makes it harder to design interfaces and APIs in general as it becomes yet another item for the developer to worry about. At the same time const-correctness greatly improves both the overall type-safety of the language and its ability to self-document.

As in C++, use the *const* modifier to define a const-pointer.

```
struct Point
{
    int m_x;
    int m_y;
}

transpose (
```

```
    Point* dst,
    Point const* src // we can be sure 'src' is not going to change
    )
{
    int x = src.m_x; // so it works even when dst and src point to the same location
    dst.m_x = src.m_y;
    dst.m_y = x;

    // src.m_x = 0; // error: cannot store into const location
}
```

All non-static methods implicitly accept an extra *this* argument, so it is necessary to be able to specify whether *this* is const or not – if yes, then such a method is called a const method.

Certain fields can be modified even from const methods (for example, various kinds of cache fields) – these are mutable fields.

The syntax for declaring const methods and mutable fields has also been borrowed from C++:

```
class C1
{
    int m_field;
    mutable int m_mutableField;

    foo () const
    {
        // ...
    }

    bar (int x)
    {
        // ...
    }
}

baz (C1 const* p)
{
    p.foo ();               // ok, const method
    p.m_mutableField = 100; // ok, mutable field

    p.m_field = 200; // error: cannot store to const location
    p.bar (200);     // error: cannot convert 'C1 const*' to 'C1*'
}
```

2.11.4 Function Pointers

Remember nested C language declarators of death needed to describe a pointer to a function, which returns a pointer to a function, which returns yet another pointer, and so on?

Nested declarators are evil! Fortunately, there are other ways to achieve the same result. Jancy uses a different approach, which is much easier to read while allowing to declare function pointers of arbitrary complexity.

```
foo ()
{
    // ...
}

int* bar (int x)
{
    // ...
```

```
}

int* function* chooseFunc () (int)
{
    return bar;
}

main ()
{
    function* f () = foo; // boooring!
    int* function* f2 (int) = bar;
    int* function* function* f3 () (int) = chooseFunc; // keep going...
    int* function* function* f4 () (int) = chooseAnotherFunc;
    int* function* function** f5 [2] () (int) = { &f3, &f4 }; // oh yeah!

    (*f5 [0]) () (100); // bar (100)
}
```

Function pointers can be *fat* or *thin*. Thin pointers are just like C/C++ function pointers: they simply hold the address of the code.

```
foo (int a)
{
    // ...
}

bar ()
{
    function thin* p (int) = foo;
    p (10);
}
```

Unlike C/C++, the argument conversion is automated (Jancy compiler generates thunks as needed)

```
foo (int a)
{
    // ...
}

bar ()
{
    typedef FpFunc (double);

    // explicit cast is required to generate a thunk
    FpFunc thin* f = (FpFunc thin*) foo;

    f (3.14);
}
```

The true power comes with *fat* function pointers. Besides the code address, fat pointers also hold the address to the closure object, which stores the context captured at the moment of creating the function pointer.

```
class C1
{
    foo ()
    {
        // ...
    }
}

bar ()
{
    C1 c;

    function* f () = c.foo; // in this case, pointer to 'c' was captured
```

```
    f ();
}
```

Jancy also allows to capture arbitrary arguments in the closure through the use of partial application operator *()*

```
foo (
    int x,
    int y
    )
{
    // ...
}

bar ()
{
    function* f (int) = foo ~(10);
    f (20); // => foo (10, 20);
}
```

You are free to skip arguments during the partial application. For example, you can make it so that the argument 3 comes from the closure, while arguments 1 and 2 come from the call.

```
class C1
{
    foo (
        int x,
        int y,
        int z
        )
    {
        // ...
    }
}

bar ()
{
    C1 c;

    function* f (int, int) = c.foo ~(,, 300);
    f (100, 200); // => c.foo (100, 200, 300);
}
```

Fat function pointers can be *weak*, meaning they do not retain some of the objects in the closure.

```
class C1
{
    foo (
        int a,
        int b,
        int c
        )
    {
        // ...
    }
}

bar ()
{
    C1* c = new C1;

    function weak* w (int, int) = c.foo (,, 3);

    // uncomment the next line and C1 will get collected next gc run
    // c = null;
```

```
    jnc.runGc ();

    function* f (int, int) = w;
    if (f)
    {
        // object survived GC run, call it
        f (1, 2); // c.foo (1, 2, 3);
    }
}
```

2.11.5 Property Pointers

Property pointers are yet another unique feature of Jancy.

Properties are found in many modern languages. What commonly lacks is a developed syntax and semantics of pointer declarations and operators.

Property pointers resemble and are closely related to function pointers. Dealing with property pointers requires a more careful application of address *&* and indirection * operators. This is due to the possibility of implicit invocation of property accessors and the ambiguity induced by such invocation, which can be automatically resolved with function pointers and not with property pointers.

Like the function pointers, property pointers can be *thin* or *fat*. Thin property pointers hold a pointer to a property accessor table.

```
int autoget property g_prop;

g_prop.set (int x)
{
    // ...
}

foo ()
{
    int property thin* p = &g_prop;
    *p = 10;
}
```

Like with the function pointers, the argument conversion is automated (compiler generates thunks if needed).

```
int autoget property g_prop;

g_prop.set (int x)
{
    // ...
}

foo ()
{
    typedef double property FpProp;

    // explicit cast is required to generate a thunk
    FpProp thin* p = (FpProp thin*) &g_prop;

    *p = 2.71;
}
```

Fat property pointers support partial application by capturing arguments in the closure.

```
class C1
{
    int autoget property m_prop;

    m_prop.set (int x)
    {
        // ...
    }
}

foo ()
{
    C1 c;
    int property* p = c.m_prop;
    *p = 100;
}
```

It is also possible to capture index arguments in the closure, thus reducing dimensions of indexed properties or completely de-indexing them. Skipping indexes is OK, too.

```
property g_prop
{
    int get (
        unsigned i,
        unsigned j
        )
    {
        // ...
    }

    set (
        unsigned i,
        unsigned j,
        int x
        )
    {
        // ...
    }
}

foo ()
{
    int indexed property* p (unsigned) = g_prop [] [20];
    *p [10] = 100; // => g_prop [10] [20] = 100;
}
```

Like function pointers, property pointers can be *weak*, meaning that they do not retain selected objects in the closure from being collected by the garbage collector.

```
class C1
{
    int autoget property m_prop;

    // ...
}

C1.m_foo.set (int x)
{
    // ...
}

foo ()
{
    C1* c = new C1;
```

```
    int property weak* w = &c.m_prop;

    // uncomment the next line and C1 will get collected next gc run
    // c = null;

    jnc.runGc ();

    int property* p = w;
    if (p)
    {
        // object survived GC run, access it
        *p = 100;
    }

    return 0;
}
```

2.12 Control Flow

2.12.1 if-else

Bla-bla-bla

2.12.2 for

Bla-bla-bla

2.12.3 while/do

Bla-bla-bla

2.12.4 break/continue

Jancy features multi-level loop jumps. These are achieved with *break-n* and *continue-n*:

```
int a [3] [4] =
{
    { 1,  2,  3,  4 },
    { 5,  6, -7,  8 },
    { 9, 10, 11, 12 },
};

for (size_t i = 0; i < countof (a); i++)
    for (size_t j = 0; j < countof (a [0]); j++)
        if (a [i] [j] < 0)
        {
            // negative element is found, process it...

            break2; // exit 2 loops at once
        }
```

2.12.5 switch

Jancy encloses all the case blocks in switch statements into implicitly created scopes. This means you are free to create and use local variables in switch statements:

```
foo (int x)
{
    switch (x)
    {
    case 0:
        int i = 10;
        break;

    case 1:
        int i = 20; // no problem: we are in different scope

    case 2:
        int i = 30; // no problem even when we fall-through from previous case label
        break;

    default:
        int i = 40; // still ok. you've got the idea
    }
}
```

Multi-level breaks can be applied to switch statement as well. In example below *break2* is used to break out of the switch statment and then out of the outer loop:

```
for (;;)
{
    Request request = getNextRequest ();

    switch (request)
    {
    case Request.Terminate:
        break2; // out of the loop

    case Request.Open:
        // ...
        break;

    case Request.Connect:
        // ...
        break;

    // ...
    }
}
```

2.12.6 once

Jancy provides an elegant syntax for lazy initialization. Prefix the necessary piece of code with *once* and the compiler will generate a thread-safe wrapper. The latter will ensure that this code executes once per each program run.

```
foo ()
{
    once initialize ();

    // ...
}
```

If your lazy initialization requires more than a single statement, enclose the entire block of your initialization code in a compound statement:

```
foo ()
{
    once
    {
        initializeTables ();
        initializeMaps ();
        initializeIo ();

        // ...
    }

    // ...
}
```

Jancy also provides a way to run the lazy initialization once per thread. Use *thread once* to achieve this:

```
foo ()
{
    thread once initializeThread ();

    // ...
}
```

2.13 Multicasts and Events

Multicasts are compiler-generated classes capable of accumulating function pointers and then calling them all at once. The main (but not the only) application of multicasts is the implementation of a publisher-subscriber pattern.

A multicast stores function pointers, so a multicast declaration looks similar to a function pointer declaration (and just like a function pointer it can be thin, fat, or weak).

```
multicast m (int);
```

A multicast class provides the following methods (this is for the multicast from the above example):

```
void clear ();
intptr setup (function* (int)); // returns cookie
intptr add (function* (int));   // returns cookie
function* remove (intptr cookie);
function* getSnapshot ();
void call (int);
```

The set() and add() methods return a cookie which can later be used to efficiently remove the function pointer from the multicast.

Some of these methods have operator aliases:

```
multicast m ();
m.setup (foo);     // same as: m = foo;
m.add (bar);       // same as: m += bar;
m.remove (cookie); // same as: m -= cookie;
clear ();          // same as: m = null;
```

The following example demonstrates some of the basic operations on multicasts:

```
foo (int x)
{
    // ...
}

bar (
    int x,
    int y
    )
{
    // ...
}

baz ()
{
    multicast m (int);
    intptr fooCookie = m.add (foo); // same as: m += foo;

    m += bar ~(, 200); // add a pointer with partial application
    m (100); // => foo (100); bar (100, 200);

    m -= fooCookie;
    m (100); // => bar (100, 200);
    m.clear (); // same as: m = null;

    // ...
}
```

Events are special pointers to multicasts. They restrict access to multicast methods *call*, *setup*, and *clear*.

```
foo (int x)
{
    // ...
}

bar ()
{
    multicast m (int);

    event* p (int) = m;
    p += foo; // ok
    p (100);  // error: 'call' is not accessible
    p.clear ();  // error: 'clear' is not accessible
}
```

Declaring a variable or a field with the event type yields a dual access policy. Friends of the namespace have multicast access to it, aliens have event access only. Read more about the dual access control model here.

```
class C1
{
    bool work ()
    {
        // ...

        m_onComplete (); // ok, friends have multicast access to m_onComplete
        return true;
    }

    event m_onComplete ();
}

foo ()
{
    // ...
}
```

```
bar ()
{
    C1 c;
    c.m_onComplete += foo; // ok, aliens have event access to m_onComplete
    c.work ();

    c.m_onComplete (); // error: 'call' is not accessible
}
```

Converting from a multicast to a function pointer is inherently ambiguous: should the resulting pointer be *live* or *snapshot*? In other words, if after creating a function pointer we modify the multicast, should this function pointer see the changes made to the multicast or not?

To deal with this ambiguity, Jancy multicast classes provide the *getSnapshot* method. Casting a multicast to a function pointer implicitly yields a *live* pointer, while the getSnapshot() method returns a snapshot.

```
foo ()
{
    // ...
}

bar ()
{
    // ...
}

baz ()
{
    multicast m () = foo;

    function* f1 (int) = m;                 // live
    function* f2 (int) = m.getSnapshot (); // obviously, a snapshot

    // modify multicast

    m += bar;

    f1 (); // => foo (); bar ();
    f2 (); // => foo ();

    return 0;
}
```

2.14 Reactive Programming

Jancy is one of the few imperative languages with the support for reactive programming.

Reactive programming is something that anyone who ever used Excel is intuitively familiar with. Writing a formula in cell A that references cell B creates a *dependency*. Change the value in B, and the cell A will get updated, too. There is no need to write an event handler to be invoked on every update in cell B – all changes are tracked automatically.

Things are not so easy in common programming languages. UI widgets provide events that fire when certain properties change, and if you need to track these changes and do something in response then you write an event handler, subscribe to an event, and update dependent controls/value from within the handler.

Jancy brings the Excel-like automatic execution of a formula when values referred to by that formula change. Write a relation between two or more UI properties, and the updates will happen automatically.

```
// ...
m_isTransmitEnabled = m_state == State.Connected;
m_actionTable [ActionId.Disconnect].m_isEnabled = m_state != State.Closed;
// ...
```

How does Jancy know where to use Excel-like execution and where to use the traditional imperative approach?

Reactors.

You declare dedicated sections of reactive code, or so-called reactors. Expressions within reactors behave like formulas in Excel and get automatically re-evaluated when bindable properties referred by the given expression change. All the dependency building, subscribing, and unsubscribing happens automatically behind the scenes.

```
reactor TcpConnectionSession.m_uiReactor ()
{
    m_title = $"TCP $(m_addressCombo.m_editText)";
    m_isTransmitEnabled = m_state == State.Connected;
    m_actionTable [ActionId.Disconnect].m_isEnabled = m_state != State.Closed;
    m_adapterProp.m_isEnabled = m_useLocalAddressProp.m_value;
    m_localPortProp.m_isEnabled = m_useLocalAddressProp.m_value;
}
```

Reactors specify the boundaries of where to use the reactive approach. In addition you are also in full control of when to use it because reactors can be started and stopped as needed.

```
TcpConnectionSession.construct ()
{
    // ...
    m_uiReactor.start ();
}
```

Sometimes, expressions don't quite cut it when it comes to describing what has to be done in response to a property change: e.g. running a cycle, or executing a sequence of statements. Using expressions in *reactor* blocks might not provide enough control over which actions must be taken in response to what property change.

The *onevent* declaration in *reactor* blocks gives you fine-grained control over dependencies and at the same time frees you from manually binding/unbinding to/from events:

```
reactor g_myReactor ()
{
    onevent bindingof (g_state) ()
    {
        // handle state change
    }

    // onevent statement allows binding to any events, not just to 'onChanged'

    onevent g_onApplyIpSettings ()
    {
        // apply IP settings...
    }
}
```

All in all, reactors simplify UI programming by an order of magnitude.

2.15 Automaton Functions

Jancy features automaton functions to provide a built-in support for creating protocol analyzers, text scanners, lexers and other recognizers.

If you ever used tools like Lex, Flex, Ragel etc then you are already familiar with the idea. If not, then it is pretty simple, actually. First, you define a list of recognized lexemes in the form of regular expressions. Then you specify which actions to execute when these lexemes are found in the input stream. Jancy compiler will then automatically build a DFA to recognize your language.

```
jnc.AutomatonResult automaton fooBar (jnc.Recognizer* recognizer)
{
    %% "foo"
        // lexeme found: foo;

    %% "bar"
        // lexeme found: bar;

    %% [0-9]+
        // lexeme found: decimal-number

        char const* numberString = recognizer.m_lexeme;

    %% [ \r\n\t]+
        // ignore whitespace
}
```

Automaton functions cannot be directly called – you need a recognizer object of type jnc.Recognizer to store the state of DFA and manage accumulation and matching of the input stream.

```
jnc.Recognizer recognizer (fooBar);
```

Class jnc.Recognizer features a method *recognize* to do recognition in one go:

```
bool result = try recognizer.recognize ("foo bar 100 baz");
if (!result)
{
    // handle recognition error
}
```

Even more importantly, it's also OK to perform recognition incrementally – chunk by chunk. This is cructial when analyzing protocols operating over stream transports like TCP or Serial, where it is not guaranteed that a message will be delivered as a whole and not as multiple segments.

```
try
{
    recognizer.write (" ba");
    recognizer.write ("r f");
    recognizer.write ("oo ");
    recognizer.write ("100");
    recognizer.write ("000");

    // notify recognizer about eof (this can trigger actions or errors)

    recognizer.eof ();

catch:
    // handle recognition error
}
```

Like Ragel, Jancy-generated recognizer support mixed-language documents. Developer can switch languages at will, by adjusting the value of field *m_automatonFunc* at appropriate locations.

In the sample below the first automaton recognizes lexeme *foo* and switches to the second automaton upon discovering an opening apostrophe:

```
jnc.AutomatonResult automaton foo (jnc.Recognizer* recognizer)
{
    %% "foo"
        // lexeme found: foo

    %% '\''
        recognizer.m_automatonFunc = bar; // switch language

    %% [ \r\n\t]+
        // ignore whitespace
}
```

The second automaton recognizes lexeme *bar* and switches back to the first automaton when a closing apostrophe is found if and only if it's not escape-protected by a backslash prefix.

```
jnc.AutomatonResult automaton bar (jnc.Recognizer* recognizer)
{
    %% "bar"
        // lexeme found: bar

    %% "\\'"
        // ignore escape-protected apostrophe

    %% '\''
        recognizer.m_automatonFunc = foo; // switch language back

    %% [ \r\n\t]+
        // ignore whitespace
}
```

Of course it's possible to maintain a call stack of previous automaton function pointers and thus implement a recognizer for nested language documents of arbitrary complexity.

2.16 Schedulers

Jancy implements the concept of function pointer scheduling. When passing a function pointers as a callback of some sort (completion routine, event handler etc) you are free to assign it a user-defined scheduler. The purpose of this scheduler is to ensure the execution of your callback in the correct environment (i.e. a specific worker thread, from within a Windows Message handler, under lock/mutex, and so on).

The scheduler is a built-in interface of the Jancy compiler:

```
namespace jnc {

class Scheduler
{
    abstract schedule (function* f ());
}

} // namespace jnc {
```

Note that even the *schedule* method accepts a pointer to a function with no arguments, and you can schedule functions with arbitrary argument list, as arguments will be captured in the closure object.

To assign a scheduler you use @ operator (at):

```
class WorkerThread: jnc.Scheduler
{
    override schedule (function* f ())
    {
        // enqueue f and signal worker thread event
    }

    workerThread ()
    {
        for (;;)
        {
            // wait for worker thread event

            function* f () = getNextRequest ();
            f ();
        }
    }
}

foo (int x);

bar ()
{
    WorkerThread workerThread;

    function* f (int) = foo @ workerThread; // create a scheduled pointer

    (foo @ workerThread) (100); // or schedule now

    // ...

    f (200); // call through a scheduled pointer
}
```

Below is a real-life example (from our IO Ninja software) of assigning a socket event handler (which gets fired from within the socket IO thread) and scheduling it to be called from the main UI thread:

```
TcpListenerSession.construct (doc.PluginHost* pluginHost)
{
    // ...

    m_listenerSocket = new io.Socket ();
    m_listenerSocket.m_onSocketEvent +=
        onListenerSocketEvent @ pluginHost.m_mainThreadScheduler;

    // ...
}
```

2.17 Dual Modifiers

The namespace member access control model of Jancy differs from that of most object-oriented languages.

- There are only two access specifiers:
  - public
  - protected
- Member access can be specified in two styles:
  - C++-style (i.e. a label)

  - Java-style (i.e. a declaration specifier)

- The default access specifier is public – even for classes
- Global namespace members can also have access specifiers just like named type members (and why not?)

The main difference, of course, is the first one. For each namespace the rest of the namespaces falls into one of the two categories:

- Friends
  - This namespace
  - Nested namespaces
  - Namespaces of derived types
  - Extension namespaces
  - Namespaces declared as *friend*'s
- Aliens
  - Everybody else

Friends have access to all the members including protected ones. Aliens can only access public members.

Admittedly, this approach definitely affords the developer a much lesser degree of flexibility in fine-tuning of who-can-access-what (other object oriented languages typically provide from three to five access specifiers).

On the positive side, this simplified binary model of friends vs aliens and opens up the possibility of dual modifiers, that is, the modifiers having one meaning for friends and another for aliens.

2.17.1 The dual modifier *readonly*

One of the most common elements used in virtually every program is a read-only field. A class sets and modifies a field; all users of the class can only read this field.

Conventionally this is implemented by declaring a private field and a public getter.

The implementation relying on dual modifiers looks a lot more natural, as it is compact – Jancy's dual modifier *readonly* is ignored by friends and means *const* for aliens:

```
class C1
{
    int readonly mreadOnly;

    foo ()
    {
        mreadOnly = 10; // for insiders it's a regular field
    }
}

bar ()
{
    C1 c;
    int x = c.mreadOnly; // no problem
    c.mreadOnly = 20;    // error: cannot assign to const-location
}
```

2.17.2 The dual modifier *event*

Events represent yet another extremely common programming element that requires dual access control.

The owner of an event must have the full control over this event, including the possibility of actually firing it. Subscribers are only able to add and remove event handlers.

The dual modifier *event* provides full multicast-access to friends and event-only access to aliens:

```
class C1
{
    event m_onCompleted ();

    work ()
    {
        // ...

        m_onCompleted (); // insiders have multicast-access to m_onCompleted
    }
}

onCompleted ()
{
    // ...
}

foo ()
{
    C1 c;
    c.m_onCompleted += onCompleted; // aliens have event-access to m_onCompleted
    c.m_completeEvent (); // error: aliens have no multicast-access to m_onCompleted
}
```

2.18 Literals

Jancy has three kinds of literals:

- C-literal;
- Hex literal;
- Bin literal;
- Multi-line literal;
- Formatting literal.

The first one is the good old C-style literal. It defines a statically allocated const char array.

```
char a [] = "hello world";
```

The second kind is the hex literal. This kind of literals allows for a nice and clean way of defining in-program const binary data blocks (i.e. icons, public keys etc) Just like C-literals, hex literals define a statically allocated const char array.

```
char b [] = 0x"61 62 63 20 64 65 66 00";
// same as: char b [] = { 0x61, 0x62, 0x63, 0x20, 0x64, 0x65, 0x66, 0x00  }
```

It's OK to use upper-case or lower-case and group hex codes with spaces to your liking (or not use spaces for grouping at all):

```
char c [] = 0x"696a 6b6c 6D6E 6F70 0000";
```

Concatenating hex and C-literals can be used for removing trailing zeroes from C-literals thus producing non-zero-terminated literals:

```
char d [] = "non-zero-terminated" 0x"";
```

The third and last kind of Jancy literals is the formatting literal. Literals of this kind bring Perl-style formatting into our C-family language. A formatting literal produces a dynamically allocated char array on the GC heap:

```
int i = 100;

char const* c = $"i = $i";
```

Jancy allows the use of expressions and printf-style formatting:

```
int i = 100;
char a [] = "hello world";
uint_t h = 0xbeef;

char const* c = $"i = $i; a [6] = $(a [6]; c); h = 0x$(h; 08x)";
```

It's also OK to specify some of injected values (or all of them) in the argument list of the formatting literal and reference these values by index. You can even reference the same value multiple times to display it using different format specifiers:

```
char const* c = $"rgb dec = (%1, %2, %3); rgb hex = (%(1;x), %(2;x), %(3;x))" (
    (colorTable [i].m_value & 0xff0000) >> 16,
    (colorTable [i].m_value & 0x00ff00) >> 8,
    (colorTable [i].m_value & 0x0000ff)
    );
```

Last but not least, all literal kinds can be concatenated and combined. If the combination does not include formatting literals, then the result is a statically allocated const char array. If the combination includes formatting literals then it will produce a dynamically allocated char array on the GC heap:

```
int i = 100;
char a [] = "hello world";

char const* c =
    0x"61 62 63"
    " ...concatenated to... "
    $"i = $i; a = $a; "
    0x"64 65 66"
    " ...end."
    );
```

2.19 Exception Handling

Jancy exceptions handling model applies a layer of syntactic sugar over good old C-style error code checking. As a result, it is extremely transparent and easy to support from the host C/C++ application.

A function marked by the *throws* modifier will have its return value interpreted as an error code. Intuitive defaults are assumed: *false* for bools, negative for signed integers, *-1* for unsigned integers and *null* for pointers.

```
bool foo (int a) throws
{
    printf ("foo (%d)\n", a);
    return a > 0;
}
```

If return values match, the error code is automatically propagated:

```
int foo (int a) throws;

int bar (int a) throws
{
    // ...

    baz (a);

    // ...
}
```

The *try* operator shields an expression from *throwing*:

```
int result = try baz (-5);
```

The *try* block shields a parent scope from *throwing* even if this parent scope has no *catch*:

```
foo ()
{
    // ...

    try
    {
        baz (20);
        baz (-1);
        baz (21); // never get here
    }

    // ...
}
```

catch and *finally* can be within any scope:

```
int bar (int a) throws
{
    // ...

catch:
    printf ("bar.catch\n");
    return -5;

finally:
    printf ("bar.finally\n");
}
```

When calling a function, the developer can use either an error code check or exception semantics depending on what's more appropriate or convenient in each particular case.

```
int main ()
{
    // ...
```

```
    int result = try bar ();
    if (result < 0)
    {
        // handle error
    }
}
```

Chapter 3

Jancy Standard Library

3.1 std.Error

Bla-bla-bla

3.2 std.String

Bla-bla-bla

3.3 std.StringRef

Bla-bla-bla

3.4 std.StringBuilder

Bla-bla-bla

3.5 std.ConstBuffer

Bla-bla-bla

3.6 std.ConstBufferRef

Bla-bla-bla

3.7 std.BufferRef

Bla-bla-bla

3.8 std.Buffer

Bla-bla-bla

3.9 std.List

Bla-bla-bla

3.10 std.StringHashTable

Bla-bla-bla

3.11 std.VariantHashTable

Bla-bla-bla

3.12 io.FileStream

Bla-bla-bla

3.13 io.MappedFile

Bla-bla-bla

3.14 io.NamedPipe

Bla-bla-bla

3.15 io.Socket

Bla-bla-bla

3.16 io.SocketAddressResolver

Bla-bla-bla

3.17 io.Serial

Bla-bla-bla

3.18 io.PCap

Bla-bla-bla

3.19 io.SshChannel

Bla-bla-bla

Chapter 4

Jancy Extensions

Bla-bla-bla

4.1 ABI-compatibility with C/C++

After the proper declaration of a data type in the Jancy scripts and in the host C/C++ application it becomes possible to directly pass data through arguments and return values without the need to explicitly push and pop the stack of the virtual machine or pack data into variant-like containers.

The following types are supported:

- All primitive C/C++ types (also integer types with inverted byte order, a.k.a. bigendians)
- Structs (with arbitrary pack factor)
- Unions
- Bit fields (in structs and unions)
- Arrays
- C/C++ data and function pointers

The following calling conventions are supported:

- cdecl (Microsoft/gcc)
- stdcall (Microsoft/gcc)
- Microsoft x64
- System V

The above brings simplicity and effectiveness to the application-script interaction.

Consider the following example of mapping Jancy declarations to C++ implementation:

```
opaque class Socket
{
    // ...

    SocketAddress const property m_address;
    SocketAddress const property m_peerAddress;

    bool readonly m_isOpen;
    uint_t m_syncId;

    Socket* operator new ();

    bool open (
        Protocol protocol,
        SocketAddress const* address = null
        ) throws;

    void close ();

    // ...
}
```

The implementation in C/C++ would look something like:

```
class Socket: public jnc::IfaceHdr
{
public:
    // conventient macros for name-to-address mapping

    JNC_BEGIN_CLASS ("io.Socket", ApiSlot_Socket)
        JNC_OPERATOR_NEW (&Socket::OperatorNew)
        JNC_CONST_PROPERTY ("m_address",     &Socket::getAddress)
        JNC_CONST_PROPERTY ("m_peerAddress", &Socket::getPeerAddress)
        JNC_FUNCTION ("open",     &Socket::open)
        JNC_FUNCTION ("close",    &Socket::close)

        // ...
    JNC_END_CLASS ()

    // these fields are directly accessed from Jancy

    bool m_isOpen;
    uint_t m_syncId;

    // ...

    // these methods are directly called from Jancy

    static
    Socket*
    operatorNew ();

    sockaddr
    AXL_CDECL
    getAddress ();

    sockaddr
    AXL_CDECL
    getPeerAddress ();

    bool
    AXL_CDECL
    open (
        int protocol,
        jnc::DataPtr addressPtr
        );
```

```
    void
    AXL_CDECL
    close ();

    // ...
};
```

The described compatibility also means you can copy-paste C definitions of communication protocol headers (such as TCP, UDP, etc.). C is the de-facto standard of system programming and its possible to find C definition for virtually any protocol in existence. Need to use this protocol from Jancy for analysis, implementation, or testing? Copy-paste the definition of protocol headers into Jancy!

```
enum IpProtocol: uint8_t
{
    Icmp = 1,
    Tcp  = 6,
    Udp  = 17,
}

struct IpHdr
{
    uint8_t m_headerLength : 4;
    uint8_t m_version      : 4;
    uint8_t m_typeOfService;
    bigendian uint16_t m_totalLength;
    uint16_t m_identification;
    uint16_t m_flags;
    uint8_t m_timeToLive;
    IpProtocol m_protocol;
    bigendian uint16_t m_headerChecksum;
    uint32_t m_srcAddress;
    uint32_t m_dstAddress;
}
```

4.2 Opaque Classes

When implementing the interaction between your Jancy script and the host C/C++ application you will often need to hide the details of C++ implementation of classes exported to the Jancy namespace. Jancy simplifies the job by providing opaque classes.

```
opaque class Serial
{
    uint_t autoget property m_baudRate;
    SerialFlowControl autoget property m_flowControl;
    uint_t autoget property m_dataBits; // typically 5..8
    SerialStopBits autoget property m_stopBits;
    SerialParity autoget property m_parity;

    // ...

    Serial* operator new ();
}
```

The corresponding C++ implementation class would look somewhat like this:

```
class Serial: public jnc::IfaceHdr
{
public:
    JNC_BEGIN_CLASS ("io.Serial", ApiSlot_Serial)
        JNC_AUTOGET_PROPERTY ("m_baudRate",    &Serial::setBaudRate)
```

```
        JNC_AUTOGET_PROPERTY ("m_flowControl", &Serial::setFlowControl)
        JNC_AUTOGET_PROPERTY ("m_dataBits",    &Serial::setDataBits)
        JNC_AUTOGET_PROPERTY ("m_stopBits",    &Serial::setStopBits)
        JNC_AUTOGET_PROPERTY ("m_parity",      &Serial::setParity)

        // ...

        JNC_OPERATOR_NEW (&Serial::operatorNew)
    JNC_API_END_CLASS ()

    uint_t m_baudRate;
    axl::io::SerialFlowControl m_flowControl;
    uint_t m_dataBits;
    axl::io::SerialStopBits m_stopBits;
    axl::io::SerialParity m_parity;

    // ...

protected:
    // hidden implementation

    axl::io::Serial m_serial;
    mt::Lock m_ioLock;
    uint_t m_ioFlags;
    IoThread m_ioThread;

};
```

Opaque classes can be neither derived from nor allocated statically, on stack, or as a class field member. This is because the Jancy compiler has no information about their full layout – they are opaque after all.

Opaque classes can only be allocated on the heap and only if their declaration includes *operator new*. The developer can choose which opaque classes should be creatable and which ones should be exposed as non-creatable host interfaces.

4.3 Static Extensions

Bla-bla-bla

4.4 Dynamic Extensions

Bla-bla-bla