The Crack Programming Language Guide

Because life is too short to wait for software.

Author: Michael Muller
Contributions by: Shannon Weyrick, Conrad Steenberg

The Crack Programming Language Guide

This is the manual for version 1.5 of the Crack Programming Language. Crack is an imperative, object-oriented language along the lines of Java, Python and C++. It can be compiled in Just In Time and Ahead Of Time modes and has been tested on 32 and 64 bit x86 Linux Debian and Ubuntu systems.

Overview

If you're a seasoned programmer, here is a quick profile to help orient you to Crack:

Major Influences

  C, C++, Java, Python

Syntax

  C-style, curly-brace

Typing

  Static, strong (with some implicit conversion)

Compiler

  native compiled, either JIT (at runtime) or AOT (native binary)

Paradigms

  Object oriented, procedural

Garbage Collection

  Reference counted objects, programmer controlled

OO Features

  Virtual functions, overloaded functions, multiple inheritance, generics.

Crack has been developed on Linux x86 and x86-64. It is highly questionable whether it will build under any other platform. Portability will play a bigger role in future versions of the language.

Installation

See the INSTALL file for the latest installation instructions.

Execution

Crack may be run in one of two ways: as a scripting language (like Python) or as a traditional compiler that creates native binaries (like C++). Both methods will compile to the native instruction set and run similar optimizations. We call these two methods JIT (Just In Time) and AOT (Ahead Of Time) compilation, respectively.

To run a script in the JIT, simply follow the standard "hashbang" practice of putting the crack binary as the program loader in the first line of your script, like so:

```
#!/usr/local/bin/crack
import crack.io cout;
cout `hello world!\n`;
```

If you write this as a script and "chmod u+x" it, when you run it you should see "hello world!" written to the terminal. Changes to the script do not require any explicit compile command - simply rerun the script.

Instead of (or in addition to) the hashbang, you may also run a script directly by calling "crack <scriptfile>".

```
$ crack hello.crk
hello word!
```

To compile your script to a standalone binary (AOT), simply call "crackc" instead of "crack". Thus executing "crackc hello.crk" will produce a binary "hello" which, when run, executes your code and displays output to terminal:

```
$ crackc hello.crk
$ ./hello
hello word!
```

As with all AOT compilers, in this mode of work you must recompile your script and run the resulting binary before seeing any changes to your codebase. However, keep in mind that you are free to develop in the JIT and deploy a final binary with AOT if you'd like. Both methods should run your code in an identical manner.

Note that during native compilation, all imported crack scripts will be "statically" compiled. This means that all imported scripts, including those you import from the crack standard library, will become a static part of your standalone binary and consecutive changes or updates to those scripts will not affect your (previously compiled) standalone binaries. In other words, there are currently no "shared crack libraries."

For more command line options that affect execution and compilation including verbosity, optimization levels, and debug output, see "crack --help".

Hello World

Let's examine the crack "hello world" program again:

```
#!/usr/local/bin/crack
import crack.io cout;
cout `hello world!\n`;
```

If you write this as a script and "chmod u+x" it, when you run it you should see "hello world!" written to the terminal.

As mentioned, the first line is the standard unix "#!" line. It tells the kernel to execute the script by passing its full name as an argument to the "/usr/local/bin/crack" program.

The second line imports the "`cout`" variable from the `crack.io` module. Like C++, Crack uses "`cin`", "`cout`" and "`cerr`" for its standard input, output and error streams. "`cout`" and "`cerr`" are both "formatters," which means that they support the use of the back-tick operator for formatting text.

The third line actually uses the back-tick operator to print some text. We won't go into too much detail on this operator right now (see The Formatter Protocol) - suffice it to say that this line is roughly equivalent to "`cout.format('hello world!\n');`" The "\n" at the end of the string translates into a newline (the ASCII "LF" character, character code 10).

Expressions consisting of a value followed by back-tick quoted text and code are called "interpolation expressions."

Comments

Crack permits the use of C, C++ and shell style comments:

```
    /* C Style comment */
    // C++ style comment
    # shell style comment
```

Crack also supports Doxygen-style doc-comments for classes, functions, global and class-scoped variables:

```
    /** C-style doc-comment */
    /// C++ style doc-comment
    ## shell-ish doc-comment
```

These receive special treatment by the parser: their content is retained and can be emitted (along with a normalized format of the prototypes themselves) using the executor's model builder. The output can then be processed by the "crackdoc" tool to produce documentation.

The `module` keyword can be used to produce documentation for an entire module:

```
    ## This documentation will be associated with the entire module.  Note
    ## that the "module" statement must be the first thing in the module.
    module;

    /// This documentation is associated with the class Foo.
    class Foo {}

    /** And this with the function bar() */
    void bar() {}
```

The `module` keyword is necessary in this case to prevent the doc-comment from being consumed by the next element in the code and to instead be associated with the module. If there are import statements, the module documentation is consumed by the module upon arriving at the first import statement, so the `module` keyword is usually unnecessary.

Variables and Types

Like most languages, Crack allows you to define variables. But unlike most other scripting languages, Crack is statically typed so you must specify (or imply) a type for all variables.

```
    # define the variable i and initialize it to 100.
    int i = 100;
```

You can also define variables using the more terse ":=" operator, which derives the type from that of the value:

```
    i := 100;           # equivalent to "int i = 100;"
    j := uint32(100);   # equivalent to "uint32 j = 100;"
```

If you don't specify an initializer for a variable, the default initializer will be used. For the numeric types, this is zero. For `bool`, it's `false`. For complex types (which we'll discuss later), it is a null pointer.

Constants

The `const` keyword introduces a constant definition. Constants have roughly the same syntax as variable definitions, but they must always have initializers:

```
const int A = 0, B = 1, C = 2;
const X := 100;
```

A constant is a variable whose value can not be modified. In truth, primitive constants (having numeric types) aren't really variables at all. Their values are injected wherever they are used at compile time, saving the space of a variable and the cost of a variable lookup.

Aggregate and primitive pointer constants are normal variables that cannot be modified by assignment. This rule applies to the variables themselves, not the objects they reference. For example, the following code is legal:

```
const array[int] VALS = {3};  # an array of three elements
VALS[0] = 100;
```

While we can modify `VALS[0]`, it would have been an error to do this:

```
VALS = array[int](10);  # ParseError!
```

Built-in Types

The Crack language defines the following set of built-in types - these can be expected to exist in every namespace without requiring an explicit import:

void

> The "void" type - this only exists so you can have a function that doesn't return anything. You cannot declare a variable of type 'void'.

byte

> An 8-bit unsigned integer (like C's `unsigned char`)

bool

> A boolean. Values are **true** and **false**, which are built-in variables.

int16

> A 16-bit signed integer.

uint16

> A 16-bit unsigned integer.

int32

> A 32-bit signed integer.

uint32

> A 32-bit unsigned integer.

int64

> A 64-bit signed integer.

uint64

A 64-bit unsigned integer.

float32

A 32-bit floating point.

float64

A 64-bit floating point.

int

An integer of the C compiler's default int-size for the platform. This must be at least 32 bits and no more than 64 bits in size.

uint

An unsigned integer of the C compiler's default unsigned int-size for the platform. Like int, this must be at least 32 bits and no more than 64 bits in size.

intz

An integer the size of the C compiler's pointer types. `intz` and `uintz` are used for memory sizes, pointer offsets, array indices and cases where we want to represent pointers as integers. `intz` is equivalent to `ssize_t` in C++.

uintz

An unsigned integer the size of the C compiler's pointer. `uintz` is equivalent to `size_t` in C and C++.

float

A floating point of the C compiler's float size for the platform. This must be at least 32 bits and no more than 64 bits in size.

byteptr

A pointer to an array of bytes (roughly like C's `char*`)

voidptr

A pointer to anything (like C's `void*`). All high level classes can implicitly convert to voidptr.

array[*type*]

The low-level array type. You should generally avoid using this in favor of high-level data structures (see `crack.cont.*`). They are not memory-managed, and don't do memory management of their elements.

To use it, you specialize it with another type, for example: `array[int]` is an array of integers.

function[*return-type*, *arg-type* ...]

The type of a function with the corresponding return type and argument types.

VTableBase

The base class used for all classes that can have virtual functions (more on this later).

Object

The implicit base class of all classes that don't define base classes (extends VTableBase)

String

An immutable, memory managed string of bytes. Strings are character set neutral and may contain null bytes.

StaticString

This is a String whose buffer can point to read-only memory.

Class

The type of class objects themselves. Crack classes exist at runtime as well as compile time. See Classes are Variables.

Of these, the `byte`, `bool`, `int`, `uint` and `float` types (including all variations of `int`, `uint` and `float`) are *primitives*. These types are notable in that they are copy-by-value and consume no memory external to the scope in which they are defined.

The `byteptr`, `voidptr` and `array` types are classified as *primitive pointer* types. These are essentially memory addresses: they are copied by value: the memory that they reference is not.

Primitive types, primitive pointer types, and the `void` type are all classified as *low-level* types. They are distinguished from the higher level aggregate types by naming convention: low-level types will always be all lower case (and digits) while high-level types (at least the ones in the standard libraries) will always begin with an upper-case character. You may not currently subclass low-level types (see Inheritance) but this restriction will be lifted in a future version of Crack.

High level or *aggregate* types are first class objects: variables of these types are pointers to allocated regions of memory large enough to accommodate the state data defined for the type. They can be extended to create other high-level types through sub-classing (see Inheritance).

Type names in Crack are very simple. They are either a single word or "*word* `[` *type-list* `]`" The latter form is used both for the built-in types `array` and `function`, and for Generics.

Integer Constants

Integer constants can be defined as an integer value in the code:

```
    int i = 100;
    int j = -1;
```

Integer constants are "adaptive", which means that they will convert to whatever type is required by their usage as long as the value is within the range of values for that type (for example, "uint u = -1" would be illegal).

Integer constants can be defined using hexadecimal, octal or binary notation as well as the default decimal notation. Examples follow:

```
    int x = 0x1AF;  # hex constant
    int o1 = 0117, o2 = 0o117; # both the same octal constant, c-style and
                               # normalized notation.
    int b = 0b10110; # binary constant
```

You can also specify integer constants as the byte values in strings by specifying a "b" (for byte) or "i" (for integer) prefix to a constant string (either single or double-quoted). For example:

```
    b := b'A';  # b is set to 65, the ASCII value of A
    int n = i'\001\0';  # i is set to 256
    int64 magic = i"SpugCrak";  # magic = 6012434588913262955
```

This usage helps to make up for the fact that Crack uses the single quote as an alternate string delimiter, making it otherwise impossible to define single character constants as you would in C. As the example suggests, the more general integer form is useful for specifying magic numbers. The bytes in a string integer constant are ordered from high-byte to low-byte on any architecture, regardless of how integers are stored.

For more information on how numeric constants work with other types, see Numeric Types.

Floating Point Constants

Floating point constants look just like floating point constants in most other languages. They consist of a decimal *mantissa* and an optional *exponent*.

Examples:

```
    float f = 1.0;
    f = 3.1e+6;
    f = 1.2e-14;
    f = .25;
```

For more information on how floating point constants interact with other types, see Numeric Types.

Implicit Conversion

In certain cases, types will automatically convert to other types. Most types will implicitly convert to boolean, allowing pretty much anything to be used as the condition in an `if` or `while` statement.

Aggregate types will implicitly convert to `voidptr`.

Numeric types are slightly more complicated.

First of all, you can always perform *explicit conversion* between any two numeric types using type constructor notation (see Constructors below):

```
    int32 i = {100};  # the '{100}' is equivalent to 'int32(100)'
    i = int32(200);
    byte b = {i};
    b = byte(i + 1000);  # 1200 is bigger than a byte can hold, the result
                         # will be truncated
```

There are two kinds of numeric types, Universal Numeric Types (UNTs) and Platform Dependent Numeric Types. The UNT types are byte, int16, uint16, int32, uint32, int64, uint64, float32 and float64. These are guaranteed to be of the same size regardless of the platform. A numeric type can implicitly convert to a UNT as long as there is no risk of precision loss on any platform. Since we define most PDNTs to be either 32-bit or 64-bit values, this means that we can always implicitly convert from, for example, an unsigned int to a uint64, but we can never implicitly convert from an uint to a uint32.

The PDNTs are int, uint, intz, and float. Any numeric type can implicitly convert to a PDNT.

In general, if you care about precision you should use a UNT. Otherwise, PDNTs can be more convenient.

```
    # implicit conversions
    byte b;
    int32 i32 = b;
    uint32 u32 = b;
    int64 i64 = i32;
    i64 = u32;
    uint64 u64 = u32;
    float32 f32 = b;
    float64 f64 = i32;
    f64 = u32;

    # conversions to PDNTs - note that these may truncate the value and lose
    # information.
    int i = f32;
    uint u = i64;
    intz iz = f64;
    float g = i64;

    # explicit conversions
    i32 = int32(i64);
    b = byte(f32);
    i64 = int64(u64);
```

See the section on Numeric Types for more information on the philosophy and details of how numeric conversions work.

Explicit Conversion

As noted in the section above, we can force a conversion between numeric types with an expression of the form *type-name* **(** *value* **)**. This works for other types, too, and allows us to do some interesting things.

We can convert any pointer type or aggregate type to a `uintz` representing the address of the object:

```
    byteptr b = getBytePtr();
    cout `address of b is $(uintz(b))\n`;

    class A {}  // see "Classes" below.
    A a = {};
    cout `address of a is $(uintz(a))\n`;
```

We can also convert any pointer type to a `byteptr` or `array`:

```
    A a = {};
    byteptr b = byteptr(a);
    byteptr c = {a};  # alternate construction syntax works, too.
    d := byteptr(a);

    ints := array[int](b);
```

These conversions, like `byteptr` and `array` operations in general, are unsafe. They allow you to circumvent the static protections in the language and read and write to unmanaged regions of memory. Nevertheless, they have some valuable uses, like low-level buffer manipulation.

Strings

Most programming languages support strings of characters, which are usually implemented as some kind of array. Crack strings are strings of bytes - you can embed any kind of byte values you want in them (including nulls) and there are no assumptions about encoding (see The ascii Module for information on working with ASCII strings).

String constants are sequences of bytes enclosed in single or double quotes (which are equivalent forms):

```
    String s = "first string";
    t := 'second string';
```

String constants are actually instances of the "StaticString" class - they're just like strings except that since their buffers are constants, they don't try to deallocate on destruction.

As in the other C-like languages, string constants (both single and double quoted) can have escape sequences in them. We've dealt with one of these already ("\n"). The full list is:

\t

ASCII Tab character (9).

\n

ASCII newline character (10).

\a

ASCII alarm character (7).

\r

ASCII carriage return (13).

\b

ASCII backspace (8).

\x *XX*

Two digit hex character value (examples: "`\x1f`", "`\x07`")

\ *OOO*

1 to 3 character octal character value. (examples: "`\0`", "`\141`")

\ *literal-newline*

If you put a backslash in front of the end of the line in a string, the newline is ignored. This allows you to wrap large strings across multiple lines.

In certain cases (most notably, regular expressions) it is desirable that the backslash not be an escape character, but be passed through verbatim. For these cases, we can define a raw string constant using an "r" prefix (similar to python):

```
    val := r'test\nstring';
    cout `$val\n`;  # prints "test\nstring"
```

It is very important to note that strings should be considered immutable. It is inappropriate to modify the contents of a `String` buffer or its size. For that matter, it is inappropriate to do this to any kind of `Buffer` derivative except for those derived from `WriteBuffer`. See Compatibilty for more information.

When defining very long string constants, it is useful to have facilities for wrapping the constant across multiple lines (in general, very long lines are difficult to read). Crack string constants facilitate this by folding adjacent constants (similar to languages like C and python) and also through the definition of "indented strings."

String constant folding is straighforward: adjacent string constants will be treated as a single string constant:

```
a := 'this is a multi-'
     'line string!';
```

In the example above, the value of a is "this is a multi-line string".

All string constants in crack can span multiple lines, so we could have written then same example as:

```
a := 'this is a multi-\
line string';
```

The backslash at the end of the first line escapes the newline so it doesn't show up in the string. If we had omitted it, the value would have been "this is a multi-\nline string".

The problem with this approach to wrapping is that it breaks indentation. For example:

```
String usage() {
    return 'Usage:
foomatic <arg> [options]
This is a lengthy explanation of the purpose of the \
usage function.
Options:
  -h show help
';
}
```

In the example above, we've created a usage message in a single string constant, but we've made a mess out of the indentation of our code. We could have used adjacent string constants to solve this, but that would require a lot of extra quotes and escape characters.

To solve this problem more elegantly, Crack introduces indented string constants. A string prefixed with a capital "I" is treated as an indented string. Indented strings are handled as follows:

- the shortest sequence of whitespace after a newline is stripped from the beginning of all lines.
- An escaped newline also consumes all following whitespace, but the whitespace is still considered for the purpose of establishing the shortest line.

For example, we could have implemented our usage function as follows:

```
String usage() {
    return I'Usage:
             foomatic <arg> [options]
             This is a lengthy explanation of the \
             purpose of the usage function.
             Options:
              -h show help
             ';
}
```

This returns the same string as the earlier example, but most programmers would probably agree that the code is more readable by virtue of the indentation.

Sequence Constants

Most modern languages provide some syntax for defining entire data structures in the code. This is a nice convenience in that the alternative is usually to write a lot of tedious code to compose them.

Crack supports this feature in the form of sequence constants. For example, to define a low-level array of integers we could do this:

```
array[int] a = [1, 2, 3, 4];
```

Sometimes it's useful to be able to do this sort of thing in an expression, for example to pass a sequence argument to a function. Since the compiler doesn't know the type in these cases, we have to specify it explicitly by preceding the list with the type followed by an exclamation point.

For example, we could have used the define operator in conjunction with this notation to write the definition above as follows:

```
a := array[int]![1, 2, 3, 4];
```

Sequence constants can also be used to create instances of aggregate types. Types need only implement The Sequence Constant Protocol as defined below.

As a reminder, we're using `array` here because we've already introduced it. You'll generally want to use the memory-managed `Array` collection instead (see Arrays).

Control Structures

Crack 1.5 supports four control structures: the "if/else" statement, the "while" statement, the "for" statement and the "try/catch" statement. "if" runs code blocks depending on whether a condition is true or false:

```
import crack.io cout;
if (true)
    cout `true is true\n`;
else
    cout `something is wrong\n`;
```

The code above will always print out "true is true".

If we wanted to do something a little more useful, we could have used it to check the command line argument:

```
import crack.sys argv;
import crack.io cout;

if (argv.count() > 1 && argv[1] == 'true')
    cout `arg is true\n`;
else
    cout `arg is false\n`;
```

There's a lot of new stuff going on here: first of all, we're importing the "argv" variable form `crack.sys`. This variable contains the program's command line arguments.

`count()` is a *method* (a function attached to a value called "the receiver") that returns the number of items in argv. `argv[1]` accesses item 1 of the argument list (indexes are zero-based, so item 1 is the second element of the sequence).

The "&&" is a short-circuit logical and: it returns true if both of the expressions are true, but it won't evaluate the second expression unless the first is true. This is important in this case, because if we were to check `argv[1]` in a case where `argv` had less than two elements, a fatal error would result.

There is also a "||" operator which is a short-circuit logical or. It returns true if either expression is true but does not evaluate the second expression if the first is true.

The `if` statement need not be accompanied by an else:

```
    if (argv.count() > 1 && argv[1] == 'true')
        cout `arg is true\n`;
    cout `this gets written no matter what the args are\n`;
```

The code in an `if` or an `else` can either be a single statement, or a sequence of statements enclosed in curly braces:

```
    if (argv.count() > 1 && argv[1] == 'true') {
        cout `arg is true\n`;
        cout `and so are you!\n`;
    }
```

You can also chain if/else statements:

```
    argCount := argv.count();
    if (argCount > 2)
        cout `more than one arg\n`;
    else if (argCount > 1)
        cout `just one arg\n`;
    else
        cout `no args.\n`;
```

Note that blocks of code in curly braces can include the definitions of new variables that are only visible from within that block. Each block is a *namespace* that inherits definitions from the outer namespace. The top-level code in the file is the *module namespace*.

The while statement

The `while` statement repeatedly executes the same code block while the condition is true. For example, we could iterate over the list of arguments with the following code:

```
    import crack.sys argv;
    import crack.io cout;

    uint i;
    while (i < argv.count()) {
        cout `argv $i: $(argv[i])\n`;
        ++i;
    }
```

Note that the code in the while is enclosed in curly braces. In general, the code managed by a control structure can either be a single statement, or a group of statements enclosed in curly braces. The `if` statement works the same way.

This example also introduces the primary feature of the back-tick operator: variable interpolation. A dollar sign followed by a variable name formats the variable. A dollar sign followed by a parenthesized expression formats the value of the expression.

The for statement

Crack supports two different flavors of "for": C-style and "for-each" style (as used in Python, JavaScript, Java and Perl).

The C-style for statement looks like a while statement, but the parenthesized form at the beginning consists of three parts:

- an initializer, which is executed prior to anything else in the loop.
- a condition, used in the same way as the condition in a while loop.
- a post-loop clause, which is called at the end of every iteration before the condition is evaluated.

So for example, the more concise way to express the loop above would be:

```
    for (uint i; i < argv.count(); ++i)
        cout `argv $i: $(argv[i])\n`;
```

Notice that the first section is a variable definition. This is allowed. The semantics are similar to C++: the variable is defined for the scope of the loop.

Any one (or all) of the sections of the parenthesized form after the "for" keyword can be omitted. So this would be the equivalent of a "while" statement:

```
    for (;expr;) stmt
```

Likewise, an infinite loop could be written as:

```
    for (;;) stmt
```

The for statement also supports a for-each style usage:

```
    String argVal;
    for (argVal in argv)
        cout `arg is $argVal\n`;
```

In the example above, `argVal` will be set to each element of argv in succession. There is also a "`:in`" variation on the "in" keyword that defines the variable for the scope of the loop, so we could have omitted the `argVal` definition and done this:

```
    for (argVal :in argv)
        cout `arg is $argVal\n`;
```

The value on the right side of the "in" keyword (in this case `argv`) must conform to The Iteration Protocol as documented below. All high-level container types implement this protocol. Sadly, low-level arrays can not because there is no way to determine their length.

"for-in" creates an underlying iterator variable, but hides it. It is often desirable to have access to the iterator. For example, you may want to modify a sequence as you iterate over it and a the sequence might have a mutator that accepts an iterator.

For this use case, there is also a "for-on" variation of this statement:

```
    for (iter :on argv) {
        argVal := iter.elem();
        cout `arg is $argVal\n`;
    }
```

As the example suggests, the "on" keyword provides a ":on" variation which defines the iterator variable for the scope of the loop.

The try/catch and throw Statements (Exceptions)

Like many modern languages, Crack supports *exceptions*. Exceptions provide a means for consolidating all of your error handling in one place, so it doesn't have to be scattered throughout the code.

Crack exceptions are syntactically similar to C++ and Java exceptions. They are implemented using the `try/catch` and `throw` statements:

```
    try {
        throw Exception('badness occurred');
    } catch (Exception ex) {
        cout `exception caught: $ex\n`;
    }
```

In the trivial example above, the throw statement transfers control to the subsequent catch statement, aborting all processing and cleaning up all variables up to the catch. The "ex" variable would be set to the value of the expression in the throw statement.

Expression handling is most often used across function calls. For example, lets say we have a function that creates file objects:

```
    FileObj open(String filename) {
        if (!exists(filename))
            throw InvalidResourceError(filename);
        else
            return FileObj(lowLeveOpen(filename));
    }
```

Control would be transferred to the first catch in the call stack matching InvalidResourceError (by specifying either that class or a base class).

The expressions in a throw clause can not be of arbitrary type. In particular, they must be a type derived from VTableBase. In general, you also want to use types derived from `crack.lang.Exception` in order to get stack traces recorded in the exception.

Certain parts of the exception handling system haven't been implemented yet. If you throw an exception from a constructor or a destructor, instance variables and the object itself will not get cleaned up properly. Also, rethrowing the current exception from a catch clause is not yet implemented.

Having an exception handler doesn't impose any runtime performance penalty. However, throwing an exception is very costly. For this reason, we recommend against using exceptions as a form of general flow control for frequently executed code paths.

Functions

Functions let you encapsulate common functionality. They are defined with a type name, an argument list, and a block of code, just like in C:

```
    int factorial(int val) {
        if (val == 1)
            return 1;
        else
            return val * factorial(val - 1);
    }
```

Also note that Crack supports *recursion*: you can call a function from within the definition of that function.

You can define a function that doesn't return a value by using the special "void" type:

```
    void printInt(int i) {
        cout `$i\n`;
    }
```

Primitive types (such as "int") are always passed "by value." The system makes a copy of them for the function. If they are high-level types, you can modify the objects that they reference from within the function.

Multiple functions can share the same name as long as their arguments differ: this feature is called *overloading*. For example, rather than "`printInt`" above, we could have defined a `print` function for multiple types:

```
    void print(int64 i) {
        cout `int $i\n`;
    }

    void print(uint64 u) {
        cout `uint $u\n`;
    }

    void print(String s) {
        cout `String $s\n`;
    }
```

The compiler chooses a function using a two-pass process: the first pass attempts to find a match based on the argument types without any conversions. The second pass attempts to find a match applying conversions whenever possible.

The general order of resolution in both passes is:

- search for a match in the current namespace by *order of definition*.
- repeat the search in each of the parent namespaces.

So for example, if we called `print()` with a `uint64` parameter, the resolver would check the first `print`, then check the second print, find a match and use `print(uint64 u)`. If we called it with an `int32` parameter, the resolver would try all three functions, and not find a match. It would then repeat the search with conversion enabled and immediately match the first function, because `int32` can implicitly convert to `int64`.

Binary operators ('+', '-', '*' ...) and unary operators ('~', '!', '++' ...) have some additional special steps associated with their resolution, see Operator Overloading for details.

We mentioned searching across namespaces: functions can be defined in most block contexts, including within other functions:

```
    void outer() {
        void inner(int i) {
            cout `in inner\n`;
        }

        inner(100);
    }

    # we can't call "inner() from here...
```

If there were another function, "`inner(uint u)`" defined in the same scope as `outer()`, the resolver would consider `inner(int i)` prior to `inner(uint u)`.

Note that it is currently an error to use variables defined in the outer function in the inner function:

```
    void outer() { int a; int inner() { return a; } } # DOESN'T WORK
```

Attempting to do this will result in a compile-time error.

[it should be noted that a future version of Crack will support this partially, without assignment, and will also support a limited form of closure]

Functions are Values

Functions are first class values in Crack. That is to say, functions are values with well defined types and can be assigned to variables or passed to other functions. The type of a function is a variation of the built-in `function` type parameterized with the return type of the function and the types of all of its arguments.

For example, given the following function:

```
    void f(int a) { cout `got $a\n`; }
```

We can assign it to a variable and then call that variable:

```
    function[void, int] g = f;
    f(100); # prints "got 100"
```

The type `function[void, int]` is the type of a function returning void and accepting a single integer argument.

This feature can be particularly useful when you want to pass a function to another function. For example, we could define a "map" operation to apply an arbitrary function to a low-level array of integers:

```
    void map(array[int] data, uintz size, function[int, int] func) {
        for (uintz i = 0; i < size; ++i)
            data[i] = func(data[i]);
    }

    array[int] arr = [1, 2, 3, 4];
    int addOne(int val) { return val + 1; }
    map(arr, 4, addOne);
```

Some times, passing functions around isn't enough. Some times you need to pass in extra data with a function. Crack also supports "functors" - callable objects. See Functors below for details.

Forward Declarations

Crack is a "single-pass" language. It should be possible for the compiler to generate all code for a module in a single read of a module. That means that in order to use a symbol, the symbol must have already been defined. This causes some problems. For example, you can't just define a pair of recursive functions like this:

```
    int f(int a, int b) {
        if (a + b > 100)
            return g(a, b);  # ERROR: g() is undefined
        else
            return a + b;
    }

    int g(int a, int b) {
        return f(a - b, b);
    }
```

If we reverse the order of the funtions, `f()` will be undefined in the definition of `g()`.

To get around this, Crack allows c-like forward declarations:

```
    int g(int a, int b);  # declare g()

    int f(int a, int b) {
        if (a + b > 100)
            return g(a, b);  # ok because g() has been declared
        else
            return a + b;
    }

    int g(int a, int b) {
        return f(a - b, b);
    }
```

Forward declared functions must be resolved by the close of the scope where they are declared. In the example above, if we had not defined `g()` by the end of the module, we would have gotten an error.

Lambdas

Lambdas are simply anonymous functions that can be used as constants.

For example, if we want to sort an array by some field, we could write:

```
    class Foo {
        String name;
        String data;
    }

    Array[Foo] vals = {};

    ...

    int compareFooByName(Foo a, Foo b) { return cmp(a.name, b.name) }
    vals.sort(compareFooByName);
```

Alternately, we could use a lambda and write:

```
    vals.sort(lambda int(Foo a, Foo b) { return cmp(a.name, b.name) });
```

This saves us the trouble of declaring a function in a separate statement and also avoids poluting the namespace with a single-use definition.

This is a nice feature in and of itself, but it is particularly useful when using macros or any knd of code generation (see The Standard Annotations, below).

The general form of a lambda is:

`lambda` *`return-type`* **`(`** *`arguments`* **`){`** *`function-body`* **`}`**

Note that lambdas are not lexical closures: that is, they do not have access to local variables from enclosing scopes.

Interpolation Expressions

As noted earlier, Crack uses the backtick as an operator to allow formatted output. This kind of expression is called an "interpolation expression," so named because they interpolate the values of expressions nested within them.

The simplest form of interpolation consists of just a variable name:

```
import crack.io cout;
x := 'test';
cout `This is a $x\n`;  # prints "this is a test" to standard output.
```

In order to expand the value of a general expression, a parenthesized expansion must be used:

```
a := 10;
cout `$a * 20 = $(a * 20)\n`;  # prints "10 * 20 = 200"
```

The 'I' prefix can also be used on the interpolation operator to allow you to format it according to the rules of an indented string (see Strings above):

```
cout I`four score
       and seven years ago
       our forefathers ...\n`;
```

Like everything else in Crack, interpolation expressions are implemented through hooks into the language. See The Formatter Protocol below for details.

Aliases

Any definition in Crack can be aliased - that is, we can specify another name to refer to it. This is done using the `alias` statement:

```
void myLongFuncName() {}
class MyLongClassName {}

alias f = myLongFuncName, C = MyLongClassName;

f();  # equivalent to a call to myLongFuncName()
C c = {};  # equivalent to creating an instance of MyLongClassName
```

The `alias` statement is followed by comma-separated a list of alias definitions of the form "*alias-name* = *definition*". A *definition* can be any variable, function, constant or type.

When an alias is defined for an overloaded function, it becomes an alias for all overloads of that function that exist in that context at the point where the alias is defined. For example:

```
import crack.io cout;

void f(int a) { cout `$a\n`; }
void f(bool b) { cout `$b\n`; }

alias g = f;

void f(String s) { cout `$s\n`; }

g('test string');
```

Perhaps unexpectedly, the code above prints "true". "g" is an alias for `f(int)` and `f(bool)`, but not for `f(String)` because that function wasn't defined at the point where the alias was created.

Like all other definitions, aliases are scoped to the block context where they are defined.

Aliases can be useful for providing shorter names, but their main value comes from defining alternate names for Generics.

As noted in Member Access Protection, aliases can also be used to make private variables externally visible.

Operators

Crack supports the complete set of C operators. As in C++, they can be used with non-numeric types through Operator Overloading.

Comparison Operators

Comparison operators compare two values and return a boolean.

==

True if the two values are equal. `1 == 1` is true.

!=

True if the two values are not equal. `1 != 2`

>

True if the left value is greater than the right value. `2 > 1`

<

True if the left value is less than the right value. `1 < 2`

>=

True if the left value is greater than or equal to the right value. `2 >= 2`

<=

True if the right value is greater than or equal to the left value. `2 <= 2`.

is

True if the object on the left is identical (not merely equal) to the object on the right. This isn't defined for numbers, only for aggregates and primitive pointer types. It essentially checks for the equivalence of the pointers.

Basic Arithmetic

All integer and floating point types support the basic arithmetic operators.

+

Add two values. `2 + 2 == 4`

-

Subtract two values. `4 - 2 == 2`

/

Divide one value by another. `6 / 3 == 2`

*

Multiply one value by another

Unary plus and minus are also supported, so we can say:

```
    x := 2;
    y := 10 + -x;  # y is 8
    z := 10 - +x;  # z is also 8
```

Operations on constant integers and floating point values are folded in crack, which means that the operations are performed in the compiler and not in the runtime. This is important because you want a constant to remain constant, even if you perform some set of operations on it so that the result of the operation will preserve the adaptiveness of the original constants. For example:

```
    byte b = 4 + 5;
```

If we didn't do const folding, the result of "4 + 5" would be of type integer, which would require an explicit conversion to be placed into a `byte` value.

Bitwise Operators

All of the integer types support the following bitwise operations:

&

Bitwise and. `5 & 4 == 4`

|

Bitwise or. `5 | 4 == 5`

<<

Shift all bits left by the amount specified on the right. `1 << 2 == 4`

>>

Shift all bits right by the amount specified on the right. `4 >> 2 == 1` If this is done on a signed operator, this is an *arithmetic* shift. Arithmetic shifts preserve the sign of the value shifted.

^

Exclusive or. `5 ^ 6 == 3`

~

Bitwise negate.

Like arithmetic operators, bitwise operators are folded. This has some interesting implications for the bitwise negate because a constant like "~0" needs to be able to fit into whatever size integer we want to map it to, yet it is essentially an unsigned value. In order to handle this, Crack invents the concept of a "negative unsigned." Think of a negative unsigned as an infinitely large value whose high bytes are all 1. All of the other binary operations behave according to this rule. What this ends up providing is the ability to fit the result of an operation on constants into an integer type large enough to accommodate the initial significant bits of the inputs:

```
    byte b = ~0 ^ 5;  # b = 250
```

Augmented Assignment

All of the binary operators but the comparison operators can be used with assignment to do C-like "augmented assignment." In general, the augmented expression `x` *op* `= y` is equivalent to `x = x` *op* `y`.

Specifically, the augmented assignment operators are: `+=`, `-=`, `/=`, `*=`, `&=`, `|=`, `>>=`, `<<=`, `^=`.

The Ternary Operator

Like the C family of languages, crack supports the "ternary operator." The ternary operator evaluates to one expression or another based on the results of a boolean expression:

```
a = b ? c : d;
```

If `b` is true, `a` will be set to `c` and `d` will not be evaluated. If false, `a` will be set to `d` and `c` will not be evaluated..

`c` and `d` may be of different types. If they are, the result will be of the same type as `c` if `d` is of a type derived from `c`'s type (see Inheritance below) and the same type as `d` if `c` is of a type derived from `d`.

If the types of `c` and `d` do not share a common ancestor, the compiler will first attempt to convert `d` to the `c`'s type. That failing, it will attempt to convert `c` to `d`'s type. If none of this works, it will finally give an error.

Precedence

Operator precedence is the same as in C. So the rules are (highest precedence first):

- `() [] . ++ --` (right side operators)
- `++ -- + - ! ~` (left side operators)
- `* / %` (binary operators)
- `+ -` (binary operators)
- `<< >>` (binary shift operators)
- `< <= > >=` (binary comparison operators)
- `== !=` (binary equality check operators)
- `&` (bitwise and)
- `|` (bitwise or)
- `&&` (logical and)
- `||` (logical or)
- `?:` Ternary operator
- `= += -= *= /= %= &= |= <<= >>=` (assignment operators)

The `typeof()` Operator

`typeof()` is a special function built-in to the compiler that produces the type of the expression in its argument. For example:

```
int a;
typeof(a) b = a;
```

Since the variable `a` in the example is of type `int`, `typeof(a) b` declared a variable `b` of type `int`.

It should be noted that `typeof()` determines the type of its argument, but the argument is not evaluated at runtime. So it is safe to pass expressions that would normally have side-effects into `typeof()`. Likewise, it should be noted that `typeof()` produces the *compile-time* type of a value, not its runtime type.

You can use `typeof()` as another way to avoid explicitly specifying types that you don't really care about. For example:

```
typeof(getValue()) val = getValue();
doSomething(val);
doSomethingElse(val);
```

`typeof()` is particularly useful in macros (see The Standard Annotations) and Generics where type information might not be easily accessible. We can use `typeof()` in an `alias` statement to provide names for types in these kinds of situations:

```
alias T = typeof(f());  # make T the return type of the function f().

T getVal();
void setVal(T newVal);
```

Classes

Classes are a feature of object oriented programming languages that combine a set of data variables with a set of special functions called "methods." As a simple example of a class, consider the representation of an x, y graphics coordinate:

```
import crack.io cout, Formatter;

class Coord {
    int x, y;

    oper init(int x0, int y0) : x = x0, y = y0 {}
    oper init() {}

    void formatTo(Formatter out) {
        out `Coord($x, $y)`;
    }
}
```

This class has two "instance variables:" `x` and `y`. These get bundled together in a package whenever we create an instance of the class.

The "oper init" syntax creates a *constructor*, which is a special function that gets called when an instance of the class is created. The constructor performs basic initialization of all of the instance variables. The second "oper init", the one without arguments, is called the "default constructor." As in C++, default constructors get generated automatically if the class has no other defined constructors. If the class does define constructors, and you want a default constructor, you have to specify one explicitly as we've done above.

We can create an instance of `Coord` like so:

```
c := Coord(3, 4);
```

Alternately, we can use a more C-like syntax:

```
Coord c = {3, 4};
```

Both of these are just different syntactic flavors of the same thing: in both cases we're defining a variable "c" that is a *reference* to a `Coord` object that is allocated on the heap. The system initializes this variable by:

- Allocating memory large enough to accommodate a Coord object
- calling the appropriate "oper init" function for the construction arguments ("3, 4" in the examples above).
- Assigning the address of the newly created `Coord` object to `c`.

Note that all variables of class types are references - they behave very much like pointers in C. So if we were to initialize one variable from another, both variables would refer to the same object:

```
c := Coord(3, 4);
d := c;
c.y = 5; # d.y is now also 5
```

This is different from the way that the primitive types behave. Primitive types are always passed "by value." So:

```
c := 100;
d := c;
c = c + 1; # c is now 101, d is still 100
```

We can tell if two variables are references to the same object using the special `is` operator:

```
c := Coord(1, 2);
d := c;
e := Coord(1, 2);
if (c is d)
    cout `this will always be printed\n`;
if (c is e)
    cout `this will never be printed\n`;
```

Note that *identity* (the property tested by the `is` operator) in Crack is a different concept from *equality* (as tested by the == operator). Two objects have the same identity if their underlying references are equal. However, references to two different object may still be equal if they have the same state (as determined by the `cmp()` method). In the example above, it might be reasonable to expect that `c` and `e` are equal, since they both have values (x = 1, y = 2), although in fact they would not be unless `Coord` implemented a `cmp()` method which provided this logic. The `cmp()` method provided by `Object` is simply an identity check.

There is a special constant, "`null`" which allows you to clear these kinds of variables so that they don't reference any object.

```
# initialize c to null, then set it conditionally
Coord c = null;
if (positive)
    c = Coord(1, 1);
else
    c = Coord(-1, -1);
```

You can use the `is` operator on null values:

```
void drawImage(Coord pos, Image img, Coord size) {
    if (size is null)
        copyImage(pos, img);
    else
        stretchImage(pos, img, size);
}
```

For classes derived from `Object`, `null` values are always treated as false:

```
    Coord c = null;
    if (!c)
        cout `this will always be printed\n`;
```

Our `Coord` class also has a `formatTo()` method. This controls how an Object is written using the back-tick operator. For example:

```
    cout `$(Coord(10, 20))\n`; # prints "Coord(10, 10)" to standard output.
```

`formatTo()` uses the instance variables `x` and `y`. One characteristic of methods is that instance variables and other methods can be used without qualification (you don't need a "self" or "this" variable, although this is possible, see below). As another example, we could define a method to give us the square of the distance from the origin as follows:

```
    int distOrgSquared() {
        return x * x + y * y;
    }
```

We could then add this information to our `writeTo()` method:

```
    void writeTo(Writer out) {
        Formatter(out) `Coord($x, $y) [dist squared = $(distOrgSquared())]`;
    }
```

Methods also have a special variable called "`this`". Just as in C++, `this` refers to the object that the method has been called on. In traditional object-oriented parlance, this object is called "the receiver."

We could have rewritten `distOrgSquared()` as follows:

```
    int distOrgSquared() {
        return this.x * this.x + this.y * this.y;
    }
```

The `this` variable is mainly useful for passing the receiver to other functions.

Classes are Variables

In addition to being compile-time entities, Classes are also variables that can be accessed at runtime. They are of type `Class`. So, for example, we can do this:

```
    class Foo {}
    Class foo2 = Foo;
    if (foo.isSubclass(Object))
        cout `Foo is an Object\n`;
```

Methods are Functions

For purposes of typing, methods are functions whose first argument is an instance of the class.

This means that we can assign them to variables:

```
    class A {
        void f(int i) {
            ...
        }
    }

    function[void, A, int] g = A.f;
```

Dereferencing Versus Scoping

The period symbol ('.') can be used for both dereferencing and scoping. Dereferencing is accessing a method or instance variable from an object. Scoping is accessing a name from a nested scope. For example, to dereference a method or instance variable:

```
    class A {
        int i;
        void f() {}
    }

    a := A();
    a.f();
    a.i = 1;
```

But was can also use the period as a scoping operator to indicate "a definition within a namespace:"

```
    function[void, A] g = A.f;

    class B {
        class C {}
    }

    b := B.C();  # Use the nested class B.C.
```

This could produce ambiguities because, as we've said, classes are also instances. For example, let's say we define a class with an "isSubclass()" method:

```
    class A {
        @static bool isSubclass(Class c) { return false; }
    }
```

(See the discussion on Static and Final Methods below)

It isn't immediately obvious whether `A.isSubclass(B)` means:

- Calling the method defined in the class above ('.' is a *scoping operator*)
- Calling the built-in metaclass method `isSubclass()` ('.' is a *dereferencing operator*).

To resolve this, dereferencing always takes precedence over scopoing. If it is possible that the period can be used as a dereferencing operator, it will be used as a dereferencing operator. It will only be used as a scoping operator if there is no conflicting dereferencing usage.

To call the explicitly defined `isSubclass()` method above, you would have to make use of the explicit scoping operator, the double-colon ("::"):

```
    A::isSubclass(B);
```

This operator unambiguously refers to a variable within the scope of a namespace, it can not be used to dereference an instance.

The period operator is preferred for scoping. It's more convenient to type, easier to read and comfortable for users of languages like Python and Java. The explicit scoping operator is seldom necessary.

Constructors

We mentioned the "oper init" functions earlier. These are called constructors. In Java and C++, constructors are defined using a function that looks like the class name. In the interests of providing uniform syntax for all special methods, Crack uses the "oper" keyword to introduce overloaded operators and special methods, including the constructors and destructors.

Constructor definitions have some special syntax. The return type can be omitted, and you can provide an *initializer list* for member variables and base classes.

In the example above, we defined two constructors:

```
    oper init(int x0, int y0) : x = x0, y = y0 {}
    oper init() {}
```

In the first case, the initializer list initializes the x and y member variables from the arguments x0 and y0. Note that the initializers are specified using assignment syntax: "x = x0" instead of the construction syntax that C++ would have used: "x(x0)".

The C++ style construction syntax can be used, too, but it has a different meaning. Construction syntax means "construct the variable with the given arguments." Assignment syntax means "initialize the variable from the given value."

So, for example, "x(x0)" would be equivalent to "x = int(x0)", which is also perfectly legal. The uses for these two types of syntax becomes more obvious when we deal with members that are themselves class instances.

For example, let's say that we want to define a line segment:

```
    class LineSegment {

        # two coordinates
        Coord c0, c1;

        ## Construct from two coordinates.
        oper init(Coord initC0, Coord initC1) :
            c0 = initC0,
            c1 = initC1 {
        }

        ## Construct from raw x and y values
        oper init(int x0, int y0, int x1, int y1) :
            c0(x0, y0),
            c1(x1, y1) {
        }
    }
```

In the first constructor, we're using the assignment syntax because we want to bind the objects passed in (`initC0` and `initC1`) to the c0 and c1 variables. If we had instead used construction syntax:

```
    oper init(Coord initC0, Coord initC1) :
        c0(initC0),
        c1(initC1) {
    }
```

the compiler would have tried to find a `Coord` constructor that accepts another `Coord` object as an argument. Since there is no such constructor, we would have gotten an error. We could have instead done this:

```
    oper init(Coord initC0, Coord initC1) :
        c0(initC0.x, initC0.y),
        c1(initC1.x, initC1.y) {
    }
```

This would have called the two argument constructors and created two new `Coord` objects for c0 and c1. There's an important difference between this and the assignment syntax we started with: with the assignment syntax, c0 and c1 become references to the objects that were passed into them. If we did this:

```
    Coord c0 = {10, 10}, c1 = {20, 20};
    ls := LineSegment(c0, c1);
    c0.x = 20;  # l.c0.x is now also 20.
```

changing `c0.x` in this case also changes the value within `ls` because the `ls`'s `c0` is the same object as the caller's `c0`. If we had instead used the construction syntax, `ls` would have had its own copies of the `Coord` objects, and changing `c0`'s `x` value wouldn't have had any effect on `ls`.

If you don't specify an initializer for one of your instance variables, the constructor will initialize the variable based on whatever initializers you gave it in the instance variable definition. So, for example, if we wanted coordinates to default to "-1, -1" for some reason, we could have done this:

class Coord { int x = -1, y = -1; }

As with ordinary variables, null or zero is used if no initializers are specified (null for aggregate types, zero for primitive numeric types).

Initializers are not necessarily run in the order that you specify them: they are run in the order of member definition. So in our examples above, if we had specified an initializer list of "`: y = y0, x = x0`", `x` still would have been initialized first.

You can define as many constructors as you want as long as their arguments have different types. This is another example of overloading: the compiler can tell the difference between them from their argument types.

The *default constructor* is the constructor without any arguments. If you don't define any constructors in your class, the compiler will attempt to generate a default constructor for you - it will generate a constructor that initializes the members with their variable initializers, using default initializers if there were no explicit initializers. Default initializers are zero for primitive numeric types and null for aggregate types and primitive pointer types.

In future versions of Crack, if a class defines no constructors, it will attempt to inherit all of the constructors of the base classes (see Inheritance).

Inheritance

One important property of object-oriented programming languages is *inheritance*: the ability to create a new class by extending an existing class. Crack supports inheritance with a syntax similar to that of C++. Let's say that we wanted a coordinate like in our last example, only we also wanted it to have a name. We could create a new class for this:

```
    class NamedCoord {
        int x, y;
        String name;
    }
```

but then we'd have to write everything that we wanted to reuse over again in the new class. And every time we fixed a bug in `Coord`, we'd have to fix the same bug in `NamedCoord`. Inheritance provides a better way to reuse code:

```
class NamedCoord : Coord {

    String name;

    oper init(int x, int y, String name0) : Coord(x, y), name = name0 {}

    void formatTo(Formatter out) {
        out `NamedCoord($x, $y)`;
    }
}
```

In the example above, we're creating a new class called `NamedCoord` that is derived from `Coord`. It will inherit all of `Coord`'s instance variables and methods. We call `Coord` `NamedCoord`'s *base class*. `NamedCoord` is a *subclass* or *derived class* of `Coord`.

In addition to allowing reuse of code, inheritance also has the advantage that instances of the derived class can be used in situations that call for an instance of the base class. So if we had a function that accepted a `Coord`, we could pass it a `NamedCoord`:

```
void drawLine(Coord c0, Coord c1) { ... }

NamedCoord c1 = {1, 2, 'c1'}, c2 = {3, 4, 'c2'};
drawLine(c1, c2);
```

Note that this is not conversion: instances of `NamedCoord` are already instances of `Coord`. As such, function calls passing classes derived from argument types will match in the first resolution pass.

One of the first things we have to deal with in creating `NamedCoord` is `Coord`'s constructor. Note that in the new initializer list, we have an entry for the base class as well as for the `name` variable. If we didn't specify a constructor, the compiler would have used the default constructor if there was one.

Like member initializers, base classes are initialized in the order in which they are defined. All base class initializers are run before any of the instance variable initializers for the class. Consider the following example:

```
import crack.io cout;

class A {
    oper init(String name) { cout `initializing $name\n`; }
    oper init() {}
}

class B : A {
    A a1, a2;

    # the order of initializers is ignored.
    oper init() : a2('a2'), a1('a1'), A('base class') {}
}

# create a temporary instance of B, prints
B();
```

This will print the following:

```
initializing base class
initializing a1
initializing a2
```

Going back to our `NamedCoord` example, we also defined another `formatTo()` method:

```
    void formatTo(Formatter out) {
        out `NamedCoord($x, $y)`;
    }
```

We did this because `Coord`'s `formatTo()` method writes out "Coord($x, $y)". We want to write "NamedCoord($x, $y)".

Sometimes you want to call the base class version of a function that is overridden in the derived class. Most often this is used to extend the base class functionality. Crack lets you do this by qualifying the method with the class name. For example, we could have instead overridden `formatTo()` like this:

```
    void formatTo(Writer out) {
        out.format('Named');
        Coord.formatTo(out);
    }
```

Note that this makes use of scoping behavior, so `Coord::formatTo(out)` would have been equivalent (see Dereferencing Versus Scoping).

Multiple Inheritance

Crack supports *multiple inheritance*: you can have any number of base classes. With the exception of the special `VTableBase` class, it is an error to inherit from the same base class more than once, even indirectly. Eventually, Crack will support this using virtual base classes like in C++.

Member Access Protection

Like other object oriented languages, Crack allows you to hide methods and instance variables that are used internally by a class but not part of its public interface. Like Python, Crack provides this access protection through the use of naming conventions. Unlike Python, Crack's naming conventions cannot be easily circumvented.

Crack gives special access protection to names that begin with an underscore:

- A name beginning with a single underscore is *module protected*. At module scope, such a symbol is inaccessible from other modules. When used in a class, such a symbol is accessible within the module where it is defined and in methods of all derived classes, regardless of whether they are declared in the same module. (See Access Protection for more information on module protected symbols in the module's scope)
- A name beginning with a double underscore is private - it is only accessible in the class where it is defined.

As an example, let's consider a `Polygon` class that includes a list of coordinates:

```
    class Polygon {
        class Vertex {
            Coord coord;
            Vertex next;
            oper init(Coord coord, Vertex next) :
                coord = coord,
                next = next {
            }
        }

        # first vertex in the list
        Vertex __first;
        # public method to add coordinates.
        void add(Coord c) {
            Vertex v = {c, __first};
            __first = v;
        }

        # protected method to allow derived classes to get a specific
        # coordinate
        Coord _getCoord(int index) {
            cur := __first;
            while (index-- && cur)
                cur = cur.next;

            return cur ? cur.coord : null;
        }
    }
```

In the `Polygon` class, the `__first` instance variable is private - the fact that the sequence of coordinates is implemented as a linked list is an implementation detail of `Polygon`, and we want to be able to change it without changing the interface and affecting users of the class. It can not be accessed outside of the scope of the class.

The `_getCoord()` method is protected: it can only be used from within the module or from classes derived from `Polygon`. This allows derived classes to have special knowledge of the sequence of coordinates without access to the private `__first` variable.

Access protection is based on the name used to access a definition, which is not necessarily the name that it is defined as. This is lets you create public aliases for private defintions (see Aliases below). For example, in the following code:

```
    int _private;  # is private to the module
    alias public = _private;  # But 'public' is public!
```

The `public` alias essentially leaks access to `_private`, which may be desirable in some circumstances.

Destructors

In addition to "`oper init`" constructors, Crack classes can have destructors. These are called by `Object.oper release()` when an object's reference count drops to zero. They can also be called explicitly by objects implementing their own memory management strategies.

You can implement the destructor for a class by defining an "`oper del`" method:

```
    class Noisy {
        oper del() { cout `Noisy object deleted\n`; }
    }

    Noisy x = {};  # Prints a message when x goes out of scope.
```

After calling the user defined code, `oper del` automatically calls `oper release` on all of the instance variables that have an `oper release` method (see Reference Counting). It then automatically calls the `oper del` method of each of its base classes. In both cases, these calls are in reverse order of initialization: first the instance variables in the reverse order that they are defined, then the base classes in the reverse order that they are listed.

Because of all of this automatic destruction, most `oper del` method don't need to have any user code at all - everything takes care of its own cleanup. If you don't define an `oper del` method, the compiler will generate one by default.

The only cases where you really need to define an `oper del` method are in the case of certain external consequences: for example, a File object might want to make sure that its file descriptor is closed upon destruction.

It should be noted that an object must do nothing to change its own reference count during processing of `oper del`, such as assigning it to an external variable, or inserting it into an external collection. For classes derived from `Object`, doing this will result in a runtime error. Future versions of Crack may have some degree of protection against this, but for now - don't do it.

Abstract Methods

There are times when you want to require a derived class to provide an implementation for a method. For example, let's say we're implementing a drawing system and we want to have a `Shape` class with a draw method:

```
class Shape {
    void draw() {
        # "shape" doesn't know how to draw itself
    }
}

class Line : Shape {
    Coord a, b;
    oper init(Coord a, Coord b) : a = a, b = b {}
    void draw() {
        graphics.drawLine(a, b);
    }
}
```

This works, but the problem is that if we create a new shape and forget to implement the `draw()` method, we don't know about it. We could throw an exception from `Shape.draw()`, but then we still don't know about it until runtime. We want to be able to say "you can not create an instance of a class derived from `Shape` without implementing `draw()`."

This is where *abstract methods* come in. Abstract methods let you declare a method without a body (sort of like a forward declaration) and require you to implement this method for every class that you wish to instantiate.

Abstract methods are defined using the `@abstract` annotation. They can only be used in a class that has also been defined as abstract. We would write our `Shape` class like this:

```
@abstract class Shape {
    @abstract void draw();
}
```

Now, in order to create a new class derived from shape we must either implement `draw()` or also declare the derived class as abstract.

Abstract methods are especially useful in Interfaces, described below.

Static and Final Methods

By default, all methods in a class derived from `VTableBase` are virtual. Since `Object` is derived from `VTableBase`, that means that in practice most of the methods that you write are virtual.

This isn't always what you want. For one thing, virtual methods are relatively expensive compared to normal functions. Calling them involves looking up their addresses at runtime from the vtable, which may be overkill if you have no intention of ever overriding them. Furthermore, because the vtable is associated with an instance, a virtual method can not be called on a null value. Finally, there are situations where you may want a function to be scoped to a class, but it isn't really a method: it doesn't need to access `this`.

To deal with these cases, Crack defines two special built-in annotations: `@static` and `@final`. (See Annotations for a more general description of this feature)

`@static` and `@final` can be used in front of a method definition to give it special status. `@final` indicates that the method should not be virtual. This makes the method more like a normal function: no vtable lookup is required, and the receiver can be null. Using `@final` prevents the method from ever being overridden. If you try to override it in a derived class, you will get an error.

`@static` effectively makes the method a normal function scoped to the class. As a result, the function can be called without a receiver.

For examples, let's return to our `Coord` class:

```
class Coord {

     # ... methods defined above omitted ...

     ## Returns true if the coordinate is non-null and not the origin.
     @final bool isNonOrigin() {
        return !(this is null) && (x || y);
     }

     ## Returns the length of an x, y position, whether or not it's
     ## wrapped in a Coord object.
     @static int length(int x, int y) {
        # let's pretend we have an integer square root function
        return squareRoot(x * x + y * y);
    }
}
```

Final Classes

The @final annotation can also be applied to a class. A final class may not be used as a base class for other classes:

```
@final class A {}
class B : A {}  # ERROR: deriving from a final class.
```

Final classes have no performance benefits over non-final classes. They are mainly useful for ensuring that new members can be added to the class without breaking compatibility.

For non-final public classes, you can not introduce a public or protected member without breaking compatibility. A user may have derived from the class and introduced another member with a conflicting name. Using `@final` guards against this.

The Special Base Classes

There are three "special" base classes in Crack:

- `Object`
- `VTableBase`
- `FreeBase`

The first two are available from any Crack code, `FreeBase` must be explicitly imported from `crack.lang`.

`Object` is the default base class for all other classes. If you don't specify any base classes, your class will implicitly be derived from `Object`. (that's not entirely true: there is a bootstrapping mode in which classes have no default base class, but that's another story).

`Object` supports a general set of functionality that is applicable to most types, including:

- Reference counting.
- Boolean conversion.
- Formatting.
- Comparison operators.

`VTableBase` is the base class for all classes with a vtable, which is the implementation mechanism of virtual functions. It is a special class that is defined by the compiler, and it has no special contents other than a hidden vtable pointer instance variable.

`Object` is derived from `VTableBase`, so by default most methods in `Object` and all of its derived classes are virtual.

`FreeBase` is a base class that can be used in cases where you don't want to be derived from `Object` . `FreeBase` does not support virtual functions, memory management, or anything you don't put into your derived class. If you're going to use it, you should at minimum figure out how to deal with memory management.

There are situations where you get a base class but you suspect or know that it is a derived class. Like C++, Crack lets you *typecast* a base class to a derived class using `cast()` and `unsafeCast()`.

Typecasting is generally deprecated in object-oriented paradigms. However, there are certain situations where it is necessary, and others where it is just the easiest way to get something done.

The `cast()` function accepts an object as an argument and returns the object cast to its type:

```
class A {}
class B : A {}
A a = B();
B b = B.cast(a);
```

If the object is not actually an instance of the type we are casting to, the `cast()` function throws a `BadCastError`.

Sometimes you want to be able to perform some action on a value only if it is a certain type. This can be done by checking the type prior to casting, or by performing the cast followed by the action in a `try` block, but these approaches both have performance problems. To address this, there is an overload of `cast()` which accepts a default value. If the value cannot be cast to the type, the default value is returned.

Typically, this is used with a default value of `null`. For example, a comparison function would typically be implemented as:

```
class A {
    String foo;

    int cmp(Object other) {
        if (o := A.cast(other, null))
            return cmp(foo, o.foo);
        else
            return Object.cmp(other);
    }
}
```

The `cast()` function is defined for all classes that derive from `VTableBase` (including all classes derived from `Object`).

For classes not derived from `VTableBase`, you can use `unsafeCast()`:

```
import crack.lang FreeBase;
class Rogue : FreeBase {}

FreeBase f = Rogue();
Rogue r = Rogue.unsafeCast(f);
```

Unlike `cast()`, `unsafeCast()` does no checking whatsoever - the programmer is responsible for insuring that the object is of the type that he is casting it to. If it's not, `unsafeCast()` will happily deliver a reference to an invalid object. It also doesn't work well with multiple inheritance. For anything other than the linage of first base classes, `unsafeCast()` won't do the pointer offsetting necessary for conversion.

For classes derived from `VTableBase`, you can verify that an object is an instance of a class by looking at the associated "class" attribute:

```
Foo obj;
Coord c = null
if (obj.class.isSubclass(Coord))
    c = Coord.unsafeCast(c);
```

Every object derived from `VTableBase` has a special `class` attribute - it's like an instance variable, only you can't assign it. It is implemented using a virtual function. The `class` attribute returns the object's class (recall that classes are also values that exist at runtime). So we could also do something like this:

```
Coord c;
if (c.class is Coord)
    cout `this will always get printed\n`;
```

Note that you usually don't want to use the `is` operator to check the class because it's usually acceptable for the class to be either the same as the class you are checking for or *derived from* the class you are checking for. This is particularly bad for checking the types of exceptions thrown from the standard library, as these may be replaced with derived classes in future compatible versions of the language. Use `isSubclass()` instead.

Interfaces

Crack doesn't have virtual base classes, but it does have interfaces. In practice, you can do almost anything that you can do with virtual base classes using Crack's underlying mechanism for this, but we'll mainly focus on the interface use case.

The `crack.ann` module provides two special annotations: `@interface` and `@implements` (for more general information see Annotations).

To use them, define an interface class:

```
@import crack.ann interface, implements;

@interface Drawable {
    # we don't have to define Drawable as abstract, interfaaces are always
    # abstract.
    @abstract void draw(Canvas canvas);
}
```

In the example above, we're defining a Drawable interface that provides a draw() method that generates a runtime error if called.

Now lets create a flavor of our LineSegment class that implements it:

```
    class DrawableLineSegment : LineSegment @implements Drawable {
        void draw(Canvas canvas) {
            canvas.drawLine(c0, c1);
        }

        oper init(Coord initC0, Coord initC1) : LineSegment(initC0, initC1) {}
    }
```

We can now pass DrawableLineSegment to any function that accepts Drawable objects without having to worry about doing anything special to accommodate reference counting. That is to say, we can do something like this:

```
    void myFunction(Drawable x) {
        x.draw();
    }
```

Of course, we could do this with normal inheritance, too, but that doesn't solve the problem for classes that we don't control (like those provided by an external module), or for cases where we want an object to implement more than one interface.

This approach has some limitations. The `@interface` and `@implementation` annotations are really just syntactic sugar for class definitions. `@interface` generates a class definition from the syntax that follows it and injects three special methods: `oper bind()`, `oper release()` and `oper from` *IFaceName* `()` (where *IFaceName* is the name of the interface, `Drawable` in the example above).

`oper from` is a special operator created to facilitate interfaces (as well as general translation across inheritance hierarchies). For interfaces, these methods are defined with a return type of `Object` and they exist to provide an instance of `Object` for the `oper bind()` and `oper release()` methods to delegate to. These are defined as `@abstract` in the interface class.

`@implements` does something similar with the implementation class: for every interface class name that follows it (you can have a comma separated list of them) it generates an `oper from` *IFaceName* `()` method that simply returns "this". Implementation classes must therefore derive from Object, either indirectly or directly and they must do so explicitly. So, "`class A @implements IFace`" would be illegal, you'd have to use "`class A : Object @implements IFace`" instead. You can avoid the additional verbosity by using the `@impl` annotation, which also expands the ": Object" part, letting you write:

```
    class A @impl IFace {
```

These annotations are a very weak form of syntactic sugar that mostly works, however failure to use them correctly may produce non-obvious errors. For example, if you try to inherit from an interface without using `@implements` (or explicitly defining the `oper from` methods yourself) you'll get an error like:

```
    ParseError: myfile.crk:100: Non-abstract class Bar has abstract methods:
      crack.lang.Object myfile.crk.Foo.oper from myfile.crk.Foo()
```

Likewise, redefining the name of an interface class from within an interface or its implementation is bad:

```
    @interface A {
        void A() {}  # DON'T DO THIS.
    }

    class B @impl A {
        void A() {}  # OR THIS.
    }
```

Special Methods

Certain methods have special meaning within the language or the standard libraries.

In the table below, **final** is used to designate methods that are declared with the `@final` annotation in `crack.lang.Object`. As explained in Static and Final Methods, this indicates that the method is non-virtual - even if the class derives from VTableBase, the method will not be turned into a virtual method. As such, the method can not be overloaded. It may also be invoked using a `null` value as the receiver.

`oper init`

A constructor.

`oper to` *type*

If this method is defined for a type, instances of the class can be implicitly converted to that type. Specifying the return type is optional. If it is specified it must be the same as the type in the name.

`oper from` *type*

This is a special function used to implement interfaces. The interface annotations defines it as an abstract method and the @implements macro generates an implementation for it.

The only thing special about "oper from" is that its name is unambiguously derived from the specified type. This allows it to be used by multiple interfaces without resulting in a name collision if those interfaces are mixed together by a derived class.

`oper del`

The destructor.

`oper bind`

(**final**) Called when an instance of the type is created or assigned to a variable.

`oper release`

(**final**) Called when a variable of the type goes out of scope.

`oper call`

Called when you call the value as if it were a function (see Functors below).

`bool isTrue()`

Returns true if the object is "true" when converted to a boolean. This is a virtual function defined in `Object` that is called for non-null values by `oper to bool()`. It allows derived classes to easily override conversion to `bool`.

`int cmp(Object other)`

Compare the object with another object. Return a value that is greater than zero if the receiver is greater than `other`, returns a value less than zero if it is less than `other`, and returns zero if the two objects are equal.

If you implement this, all of the normal comparison operators ("==", "!=", "<", ">", "<=" and ">=") will work for you.

`void formatTo(Formatter out)`

Format the receiver to `out`. This is used to allow the object to write itself in its most natural representation - whatever that means for the object type.

`void format(` *type* `object)`

This method is used by the back-tick operator to format objects of specific types in specific ways. See The Formatter Protocol.

`void append(` *type* `object)`

This method is used to construct a sequence constant if it is available. See Sequence Constants and The Sequence Constant Protocol.

Operator Overloading

The `oper` keyword originated as a short form of the "operator" keyword in C++ which is designed to allow you to define your own implementation of the operators (e.g. "+", "-", ">" ...).

The following operators can be overloaded:

`oper +(` *type* `other)`

Binary plus.

`oper -()`

Unary negate.

`oper -(` *type* `other)`

Binary minus.

`oper *(` *type* `other)`

Binary multiply.

`oper /(` *type* `other)`

Binary divide.

`oper %(` *type* `other)`

Binary remainder.

`oper [](` *type* `index)`

Array element access.

`oper []=(` *type* `index,` *type* `value)`

Array element assignment.

`oper --()`

Unary pre-decrement (post-decrement, pre-increment and post-increment don't exist yet, not sure why this one does).

`oper !()`

Unary boolean negate.

`oper ~()`

Unary bitwise negate.

`oper ==(` *type* `other)`

(**final**) Binary "equals." `Object` implements this as "`cmp(other) == 0`".

`oper !=(` *type* `other)`

(**final**) Binary "not equals." `Object` implements this as "`cmp(other) != 0`".

`oper <(` *type* `other)`

(**final**) Binary "less than." `Object` implements this as "`cmp(other) < 0`".

`oper <=(` *type* `other)`

(**final**) Binary "less than or equal to." `Object` implements this as "`cmp(other) <= 0`".

`oper >(` *type* `other)`

(**final**) Binary "greater than." `Object` implements this as "`cmp(other) > 0`".

`oper >=(` *type* `other)`

(**final**) Binary "greater than or equal to." `Object` implements this as "`cmp(other) >= 0`".

`oper |(` *type* `other)`

Bitwise or.

`oper &(` *type* `other)`

Bitwise and.

`oper <<(` *type* `other)`

Left shift.

`oper >>(` *type* `other)`

Right shift.

`oper ^(` *type* `other)`

Exclusive or.

`oper` *op* `=(` *type* `other)`

(all Augmented Assignment operators). If these exist, they will be called when the augmented assignment operators are used. If they are not defined, they can still be used if the type defines the simple operator that it is based on. So for example, if a class defines "`oper +`", you can use the += operator on an instance without defining it.

oper call

The function call operator: implementing this makes your objects callable using the argument list of the operator definition. You can declare "oper call" multiple times with different argument types to make your object behave like an overloaded function.

As in the last section, **final** means the method is non-virtual for Object derivatives and cannot be overridden.

The primitive types mostly have intrinsic implementations of the operators.

Certain operators can not be overloaded:

- The ternary operator.
- The short-circuit and and or operations ("&&" and "`||`")
- The assignment operator ("`=`").
- The define operator ("`:=`").
- The "is" operator.

All of the binary operators have corresponding "reverse operators." For example, `oper r+()` is the reverse operator of `oper +()`. Reverse operators are used to check for an overload on the right operand of a binary expression.

Binary operations have special resolution rules based on the normal overload resolution order. When trying to find a match for a binary operation, the compiler:

- Checks for a method named for the operator on the left operand with an argument matching the type of the right operand.
- Checks for a method named for the reverse operator on the right operand with an argument matching the type of the left operand.
- Checks for a function named for the operator with two arguments where the first matching the left operand and the second matching the right operand.

Each of these steps uses the existing method resolution rules as defined above in Functions and below in Method Resolution in Classes.

Unary operators also have a special resolution rule: the compiler checks for a unary method named for the operator with no arguments prior to checking for a function with a single argument in the enclosing namespaces.

Functors

If a class defines the special `oper call` method, it is possible to call an instance of the class with the same syntax that you would use to call a function. For example:

```
class Action {
    void oper call(int arg) { ... }
}
```

```
    Action a = {};
    a(100);
```

This ends up calling the `oper call` method.

This sort of pattern is commonly known as a "function object" or simply a "functor." This feature will be expanded in future versions of the language to provide implicit conversion from normal functions to functors.

As it stands, there are a set of generic interfaces (see Generics below) that make it easier to work with functors with up to five arguments. These are defined in the `crack.functor` module.

For example, let's say we have a `Scheduler` class that simply performs actions at a specified time. We could use a functor returning void with no arguments to represent our actions:

```
    import crack.functor Functor0;

    @import crack.ann implements;

    class Scheduler {
        void schedule(Time time, Functor0[void] func) { ... }

        void run() { ... }
    }

    class Alarm @impl Functor0[void] {
        void oper call() {
            cout `Time to wake up!\n`;
        }
    }

    Scheduler s = {};
    s.schedule(Time('10 AM'), Alarm());
```

`Functor0` is a "functor with zero arguments." If it were a two argument functor we would use `Functor2`:

```
    class CompareX @impl Functor2[int, MyObj, MyObj] {
        int oper call(MyObj a, MyObj b) {
            return cmp(a.x, b.x);
        }
    }
```

Having to derive a class every time you want to do this can be tedious, so the `crack.functor.FunctorX` classes each include a `Wrap` nested class that wraps a plain function with the same signature. So we could have implemented our alarm like this:

```
    import crack.functor Functor0;

    ...

    void alarm() {
        cout `Time to wake up!\n`;
    }

    Scheduler s = {};
    s.schedule(Time('10 AM'), Functor0[void].Wrap(alarm));
```

Alternately, we could have added a convenience overload to Scheduler to do the conversion for us:

```
    class Scheduler {
        void schedule(Time time, Functor0[void] func) { ... }
        void schedule(Time time, function[void] func) {
            schedule(time, Functor0[void].Wrap(func));
        }

        void run() { ... }
    }


    ...

    Scheduler s = {};
    s.schedule(Time('10 AM'), alarm);
```

The `FunctorX.Wrap` classes also have corresponding `FunctionX` generics defined in `crack.functor`. These primarily exist for backwards compatibility, but are also safe to use as alternate names.

Method Resolution in Classes

When resolving an overloaded method, Crack uses the same rules as for normal function resolution: check each method in each namespace in the order defined, then do the same in the parent namespaces. If no result is found, repeat with conversions. The only exception is that overrides or implementations of virtual functions are not considered in the method resolution order.

For classes, "parent namespaces" are the base classes. So if we have:

```
    class Base {
        void func(B b) {}
        void func(A a) {}
    }

    class Derived : Base {
        void func(C c) {}
        void func(A a) {}
    }
```

when we try to resolve `func(val)`, the compiler will check:

- `Derived.func(C c)`
- `Base.func(B b)`
- `Base.func(A a)`

Note that `Derived.func(A a)` is not checked: this is because it is just an override of `Base.func(A a)`. Crack ignores overrides when considering method resolution. If it didn't, it would be impossible to override a function without also matching more specific overloads defined in the base class. For example, imagine that if, in the example above, B were derived from A. Matching `Derived.func(A a)` would hide the more specific `Base.func(B b)` which is probably not what you want.

Forward Declarations of Classes

As with functions, some times we want to reference a class before it is defined. We can forward declare a class in the same way as a function:

```

    class A;

    @abstract class B {
        @abstract void useA(A a);
    }
    class A {}
```

There are some heavy restrictions on what you can do with forward declared classes. You can't call any methods on them, you can't instantiate them and you can't even define variables to be of that type. About the only things that you can do with them is use them as argument types and pass those arguments to other functions. However, these things are sufficient for defining interfaces (as in the example above) which is usually all you need to bootstrap these kinds of collaborations.

Method Elision

Under certain cicrumstances, it is useful to remove inherited methods from the compiled representation of a class. Crack has special function declaration syntax to support this. By declaring a function followed by the "delete" identifier, you effectively hide the function within that context:

```
    class A {
        void f() {}
    }

    class B : A {
        void f() delete;
    }

    A().f(); # OK
    B().f(); # ERROR: f() is undefined in B.
```

This feature was introduced to support elision of `oper bind` and `oper release`, specifically to allow the implementation of raw (non-reference-counted) pointers. It may also be useful for providing limited interfaces to an existing class.

Note that this only affects the visibility of a method at compile-time: for example, deleting a destructor of a class derived from `Object` will not prevent the base class destructors from being called when the object is destroyed.

Generics

In strongly typed languages like Crack, it is highly desirable to be able to express a type in terms of other types. For example, if we have an array container, we want to be able to define an "array of integers" or an "array of strings." Without this feature, you need to do a lot of typecasting when extracting an object from the array.

Crack provides this feature in the form of *generics*. A generic is a class that can accept other types as parameters, similar to the way that the primitive built-ins `array` and `function` do.

You define a generic like any other class definition only with a parameter list:

```
    class Pair[FirstType, SecondType] {
        FirstType first;
        SecondType second;

        oper init(FirstType first, SecondType second) :
            first = first,
            second = second {
        }
    }
```

In this example, we have defined a `Pair` generic that is simply a holder for two variables, `first` and `second`, each of which can have their own type.

To construct a `Pair` of an integer and a string:

```
p := Pair[int, String](100, 'string value');
```

You can use a `Pair[` *t1* `,` *t2* `]` specification anywhere a type is allowed. When such a specification is encountered, the Crack compiler instantiates the generic for its parameter types by actually compiling the code of the generic with its parameters aliased to the types provided. It only does this the first time it encounters the generic - after that it just imports the original instantiation.

Generics in Crack are essentially macros. When they are defined, the contents of the generic class is simply recorded. When instantiated, that recording is played back in an "ephemeral module" containing:

- Aliases for all of the names contained in the original module where the generic was defined.
- The generic parameter types for the instantiation, aliased to the the generic parameter names.

Because of this approach, compile-time errors in generics do not manifest until the generic is instantiated.

Generic classes also support inheritance. Using the above example we can derive a class `PrimitivePair` from `Pair` as follows:

```
class PrimitivePair[FirstType, SecondType] : Pair[FirstType, SecondType] {

    oper init(FirstType f, SecondType s) : Pair(f, s) {}

    oper +=(FirstType f){
        first += f;
    }
}
```

Appendages

Appendages allow you to add functionality to a class hierarchy without using inheritance. As an example of why you would want to do this, let's go back to a stripped-down form of our `Coord` example:

```
class Coord {
    int x, y;
    oper init(int x, int y) : x = x, y = y {}
}

class NamedCoord : Coord {
    String name;
    oper init(int x, int y, String name) : Coord(x, y), name = name {}
}
```

Let's say that you want to add a few methods to `Coord` but `Coord` and `NamedCoord` are part of an external module. Without appendages, you wouldn't be able to add them as methods. You could extend `Coord`, but then you would lose the ability to use those methods on `NamedCoord`.

To deal with this, you'd have to instead create functions:

```
float distanceFromOrigin(Coord c) {
    return sqrt(float(c.x) + float(c.y));
}

int area(Coord c) {
    return c.x * c.y;
}
```

This works, but it's suboptimal:

- You lose the *receiver* **.** *method* **()** syntax and the implicit use of instance variables (you can't just use `x` and `y`, you have to use `c.x` and `c.y`).
- You have to import the methods individually into any other module that uses them.

Appendages let you bundle functionality that is associated with a class external to the class and its hierarchy. An appendage based solution would look like this:

```
    class SpatialCoord = Coord {
        float distanceFromOrigin() {
            return sqrt(float(x) + float(y));
        }

        int area() {
            return x * y;
        }
    }
```

Note the use of the equal sign ("=") in the class definition where a colon would normally be. This identifies `SpatialCoord` as an appendage. In this relationship `Coord` is the *anchor class*. It is the base class of the hierarchy that the appendage's methods manipulate.

To use `SpatialCoord`, you just convert any kind of `Coord` object:

```
    a := SpatialCoord(Coord(10, 20));
    b := SpatialCoord(NamedCoord(10, 20, 'my location'));

    cerr `area is $(b.area())\n`;
```

For an appendage, calling the class (`SpatialCoord(...)`) doesn't create a new instance of SpatialCoord. Rather, it converts the argument (which must be an instance of `Coord` or a derived class) to a SpatialCoord. Doing this is equivalent to using `unsafeCast`:

```
    a := SpatialCoord.unsafeCast(Coord(10, 20));
```

The major difference being that it isn't unsafe: `unsafeCast()` accepts any kind of argument, appendage initialization only accepts an instance of something derived from the anchor class, and does the correct pointer offseting to ensure a reference to the body of the anchor class.

An appendage may also be derived from one or more other appendages, as long as they all have the same anchor class. For example, we might use this feature to bundle together a set of appendages:

```
    class DumpingCoord = Coord {
        void dump() {
            cerr `$(coord)\n`;
        }
    }

    class MyCoord = SpatialCoord, DumpingCoord {}
```

Appendages are limited in that they may add no additional state to an object. You can not define instance variables or virtual functions in an appendage.

Modules

We've been making casual use of the `import` statement throughout this document. The `import` statement is used to import symbols from modules, for example we've use it to import the global variable `cout` from the `crack.io` module:

```
    import crack.io cout;
```

The general format of the import statement is:

```
    import module-name name-list;
```

module-name is a dot-delimited module name. *name-list* is a comma separated list of functions, variables and classes defined in the module that you wish to import into the current namespace. A name can also have the form "*local-name* = *source-name*", allowing you to import a definition under a different alias. This feature is typically used to avoid name collisions.

For example, let's say that we had two modules that both define classes names "Document." We could use both types in the same module by importing them with aliases:

```
    import acme.xml.dom DOMDocument = Document;
    import libre.word WPDocument = Document;
```

Module names correspond directly to directory and file names in the Crack "library path." When resolving a module name, the system:

- checks to see if the module has already been loaded, if so it just uses the existing module information.
- splits the name up by periods, concatenates all but the last part of the name into a relative directory path. The last part of the name becomes the filename. Example: "foo.bar.baz" -> path = "foo/bar", filename = "baz"
- for every directory in the crack library path, search for a subdirectory matching the path and the filename with a ".crk" extension
- when we find it, compile it. Prior to executing any code in the importing module, the module top-level code will be executed (everything that's not in a function or class).

So for example, to load the `crack.lang` module for the first time we:

- Search the library path for "crack/lang.crk"
- Compile the file.

The crack library path is specified with the "-l" option values on the command line. By default, the executor inserts the $PREFIX/lib/crack$VERSION path and the current directory into the beginning of the search path.

Variables defined in the module top-level are not released until program termination. Cleanups are called in the reverse order of definition.

Access Protection

Not all symbols can be imported. Formally, you can only import public symbols that are defined in the module that you are importing them from.

A "public symbol" is an identifier that doesn't begin with an underscore. Identifiers beginning with a single underscore are classified as "module protected." These symbols can only be used within the module where they are defined and (if they are class-scoped) from within derived classes (see Inheritance below). You can't import module protected symbols.

You are also limited to importing symbols that are actually defined in the module that you are importing them from. Crack doesn't allow "second order imports." For example, if module `x` imports function `f()` from module `y`, module `z` may not import `f()` from module `x`. This is explicitly disallowed to avoid the problem of "leaky imports," where a programmer (often accidentally) imports low-level symbols from a higher level module, subjecting their code to breakage if the implementation of the higher level module changes and it no longer imports the low-level symbol.

Second order imports are also disallowed in the case of overloaded functions where some of the overloads are implemented locally. For example, if your module does this:

```
    import mod func;  # 'func' is an overloaded function.

    void func() {}
```

An error will be generated if you import `func` from this module because the symbol `func` includes overloads from `mod`. This is even true in the pathological case where all overloads in the second order module are overriden in the imported module. If not, the owner of that module could introduce another overload (an action which should be backwards compatible) and break compatibility.

There are times when you want to allow other modules to import a second-order symbol. For example, sometimes you want to set up a "convenience module" that consolidates a group of entities from a disparate collection of other modules. You can do this by explicitly marking the symbols as safe for import using the `@export_symbols` annotation:

```
    import x a;
    import y b;

    @export_symbols a, b;  # a and b can now be imported from this module.
```

Note that this mechanism can not be used to make module protected symbols importable. To import a symbol, it must still be public. To offer a non-public symbol for import, you must define an alias.

Note that aliases by themselves to not circumvent the second-order import rule. To export symbols from another module you must use @export_symbols, and that goes for aliases for generic instantiatons, too:

```
    import crack.cont.hashmap HashMap;

    class MyObj {}
    alias MyMap = HashMap[String, MyObj];

    # Even though X is defined in this module, X[int] is not because generics
    # are instantiated in their own modules.
    class X[T] {}
    alias XInt = X[int];

    @export_symbols MyMap, XInt;
```

We can now import `MyMap` and `XInt` from other modules, providing them with a more abbreviated name for the type and insulating them from implementation changes.

Relative Imports

Crack allows you to elide the package name (all but the last segment of the module name) to import sibling modules or their children. For example, in module foo.bar:

```
# This is module foo.bar in "foo/bar.crk"

import .baz A;     # Imports from foo.baz
import .bot.bog B; # Imports from foo.bot.bog.
```

The leading period triggers these semantics.

Doing this consistently thorughout a module tree effectively makes the tree "portable." You can move it to anywhere in the import tree of a program that is using it and still be able to import it.

There are several advantages to this over absolute names:

- Easier refactoring. A subsystem organized in this way can be easily moved in its entirety to another location in the source tree.
- Namespace hygiene. For example, if you depend on two packages both rooted at "parser" you could simply relocate one (or both) of them to more specificly named directories. More generally, you can organize an import tree any way you see fit.
- Dealing with multiple versions of the same package. For example, if you have two components that depend on a different version of the same dependency (the "diamond dependency" problem) you might be able to install both versions in your tree under separate, versioned directories.

Because of relative imports, it's possible to deploy a package consisting as a set of interdependent modules as a set of modules in the top-level directory: the user can simply unpack this directory into any location in their module path.

Auto-imports

It's generally preferable for a programmer to not have to deal with imports at all: as long as names are unique (or can be given unique aliases) within the scope of a project, it is much easier to simply define all of the imports in a common module and be done with them. The auto-import annotation allows us to do this. For example, we can define a "my_imports" module as such:

```
import crack.compiler CrackContext;
@import crack.ann auto_imports;
@auto_imports {
    crack.lang AppendBuffer, Buffer, Exception;
    crack.io cin, cout, cerr, StringFormatter;
}
```

And then use it from another module as follows:

```
@import my_imports @auto_import; @auto_import;

cout `We can use cout without importing it!\n`;
```

`auto_imports` registers the modules and symbols as *lazy imports*. These modules are only imported if the identifiers associated with them are referenced but not defined in the module.

The contents of an `@auto_imports` block is a list of import statements without the import keyword (since these are the only things allowed in the block, there is no need to distinguish them).

You can also use relative import notation to define auto-imports:

```
@auto_imports {
    .mod func, CONSTANT;
}
```

When using relative auto--imports, the module name is presumed to be relative to enclosing name of the module defining the auto-imports. So if the module above is "`foo.imports`", "`.mod`" would load from the module "`foo.mod`". In this way, packages can provide their own "imports" files and remain relocatable to any location within the module tree.

Auto imports currently do not work for compile-time imports (imported with `@import`).

When defining auto-import modules, it is important to consider backwards compatibility. Users may make use of multiple auto-import modules in any order. Namespace collisions are not a problem in this context, the last lazy import definition wins. However, if the owner of an auto-import module introduces a new definition in another version of their code, they may end up overriding an earlier auto-import that the user was relying on:

```
# "foo" provides an auto-import for getFuncie.
@import foo foo_auto_import = auto_import; @foo_auto_import;
# "bar" also provides an auto-import for getFuncie, but didn't used to!
@import bar bar_auto_import = auto_import; @bar_auto_import;

# User was expecting foo.getFuncie(), but gets bar.getFuncie().
getFuncie();
```

For this reason, it is recommended that import modules be versioned. By appending a version number to the name of the imports file (e.g. "imports1_5") you can ensure that the set of imports referenced by a particular external user remains constant, while still providing a way to extend your set of auto-imports.

Crack provides an auto-import module for most of its standard library. For version 1.5 of crack, this module is `crack.imports1_5`. Note that the first use of this module does add noticeable overhead to the compile time of a program that makes use of it. Tracing the actual imports of the modules (with the "-t LazyImports" option) can be used to determine the actual imports used by all modules and convert the auto-import into traditional import statements.

Protocols

There are certain syntactic conventions in crack that map to specific compile time interfaces - that is, the compiler will translate these kinds of statements or expressions into a set of method calls, partially depending on whether these methods are defined. These special methods are similar to Interfaces but they are strictly relevant at compile time and as such do not rely on virtual methods for their implementation. We call these method sets *protocols*.

The language elements that currently make use of protocols are the `for/in` statement, the string interpolation expression and sequence construction.

At this time, there is no way to formally indicate that a type implements a protocol: if a type provides the necessary methods, it is considered to implement the protocol and can be used in the relevant context.

The Iteration Protocol

The following `for/in` statement:

```
for (x in container)
    cout `$x\n`;
```

translates to roughly the following code:

```
for (tempIter := container.iter(); tempIter; tempIter.next()) {
    x = tempIter.elem();
```

```
        cout `$x\n`;
    }
```

Note that the tempIter variable is purely illustrative. There is an iterator variable generated, but it is hidden so as not to clash with any user defined variables.

For a class to be iterable using `for/in`:

- It must provide an `iter()` method that returns an object conforming to the iterator protocol (an "iterator type").
- The iterator type must have an `elem()` method that returns the element referenced by the iterator.
- The iterator type must have a `next()` method that forwards the iterator to the next element.
- The iterator type must convert to `bool`, converting to `true` if the iterator is valid, `false` if not (if it has run out of elements).

The Formatter Protocol

As we've shown, the back-tick operator allows us to do formatted output of static data, variables and expression values:

```
    int a;
    cout `a = $a, a + 1 = $(a + 1)\n`;
```

See Interpolation Expressions above for details on what this means. The interpolation expression above is equivalent to the following code:

```
    if (cout) {
        cout.enter();
        cout.format('a = ');
        cout.format(a);
        cout.format(', a + 1 = ');
        cout.format(a + 1);
        cout.format('\n');
        cout.leave()
    }
```

The `enter()` and `leave()` methods are optional - they are called if they are present. If the `leave()` method exists, its return value is the result value of the interpolation expression. If not, the original target (cout, in this case) is the result.

The `enter()` method is typically defined with a return type of `void`. If the return type is not void, the return type is used as the formatter for the remainder of the interpolation expression: `format()` and `leave()` methods are called on this value instead of the original expression (although the original expression is still used to determine whether the interpolation sequence is invoked at all). If `out` is an object whose `enter()` method returns a non-void value, then `` out `this is $x` `` would be equivalent to the following code:

```
    if (out) {
        temp := out.enter();
        temp.format('this is ');
        temp.format(x);
        temp.leave();
    }
```

The `cout` variable used above is defined in `crack.io` as an instance of `Formatter`. Interpolation expressions are not limited to use with `Formatter`, they can be used on any object that supports conversion to boolean and `format()` methods for all of the values in the expression. For example, we could create our own formatter that could be used in the expression above:

```
class SumOfInts {
    int total;

    ## ignore static strings.
    void format(StaticString s) {}

    ## make integer formatting add the value to the sum.
    void format(int val) { total = total + val; }

    ## initialize to zero
    void enter() { total = 0; }

    ## return the total
    int leave() { return total; }
}

SumOfInts sum;
result = sum `a = $a, a + 1 = $(a + 1)\n`;

# result is 2a + 1
```

More often when doing this, you'll want to derive from `StandardFormatter` and extend its functionality:

```
import crack.io StandardFormatter;

## Formatter that encloses strings in quotes.
class StrQuoter : StandardFormatter {

    oper init(Writer w) : StandardFormatter(w) {}

    ## Write strings wrapped in quotes.
    void format(String s) {
        if (s := String.cast(obj, null)) {
            rep.write('"');
            rep.write(s);
            rep.write('"');
        } else {
            StandardFormatter.format(obj);
        }
    }
}

String s = 'string value';

# wrap standard output's underlying writer with our formatter and use it
# to format the value.
StrQuoter(cout.rep) `value is $s\n`;
```

Note that we have to check for a string dynamically using casting. StandardFormatter doesn't have a `format(String)` method. If we had defined one at this level, it would have matched all `String` types, including `StaticString`, which is the type of the static portion of the interpolation string. Adding a `format(StaticString)` method wouldn't have helped us, because this is a virtual function defined in `StandardFormatter`, and by method resolution order, the local `format(String)` method would have been checked first.

You can create your own `StandardFormatter` objects given a `Writer` object. There are already a few specializations of this class in the `crack.io` module.

`StringFormatter` allows you to construct a string using a formatter:

```
    import crack.io StringFormatter;

    f := StringFormatter();
    f `some text`;
    s := f.string();  # s == "some text"
```

It's often useful to be able to format a string in an expression. For example, throwing an exception like this is unwieldy:

```
    if (foo > MAX_FOO) {
        StringFormatter out = {};
        out `Value of foo out of bounds: $foo`;
        throw AssertionError(out.createString());
    }
```

It would be much cleaner to format and generate the string within the argument list of `AssertionError`. `FStr` allows us to do this. `FStr` creates a new string formatter for every interpolation expression and returns the created string from its "leave()" method. So we could replace the code above with:

```
    if (foo > MAX_FOO)
        throw AssertionError(FStr() `Value of foo out of bounds: $foo`);
```

Using the Formatter FieldSet Interface

The `Formatter` interface derives from `FieldSet`, which allows users to inject and obtain state information from anything implementing it (see Field Sets). This can be used to implement special formatting functionality and to vary the formatting behavior of specific classes.

One utility that exploits this feature is `crack.io.fmtutil.Indenter`. As its name suggests, the `Indenter` class provides indented output that can be used on any `Formatter` object. As an example:

```
    class Foo {
        String name;
        Array[Foo] children;

        ...

        void formatTo(Formatter out) {
            indent := Indenter.wrap(out);
            indent `this is $name:\n`;

            ++indent;  # Indent one level.
            for (child :in children)
                indent `$child`;
            --indent;  # dedent one level.
        }
    }
```

`Indenter` works by intercepting newlines and injecting indentation after them. If used as the formatter for nested elements, as in the example, it will apply indendation regardless of whether the nested elements use `Indenter.wrap()`. If passed a formatter that is already an Indenter, `Indenter.wrap()` will cast the formatter to an indenter rather than creating one or trying to retrieve one from the formatter's `FieldSet`.

The `Formatter` `FieldSet` can also be used to vary the behavior of object format methods. The following example illustrates the use of a stored attribute to switch formatting between a debug representation and the normal output form:

```
    import crack.io cout, Formatter;
    import crack.io.fmtutil Indenter;
    import crack.fieldset FieldSet;
```

```
@import crack.fieldset_ann fieldset_accessors;

class A;

# AFmt is a class that can be stored in a FieldSet.  It's purpose is to
# dispatch formatting to different methods depending on which
# specialization is active in the Formatter.
@abstract class AFmt {
    @abstract void format(Formatter out, A inst);
    @fieldset_accessors(AFmt);
}

AFmt _makeAPrettyFmt();

class A {

    void formatTo(Formatter out) {

        # Get the AFmt object for the Formatter, and invoke it's format
        # method to do the formatting.
        AFmt fmt;
        if (!(fmt = AFmt.get(out))) {
            fmt = _makeAPrettyFmt();
            fmt.putIn(out);
        }

        fmt.format(out, this);
    }

    # Different veriations of the formatTo() method called by AFmt
    # specializations.

    void formatReprTo(Formatter out) {
        out `repr A`;
    }

    void formatPrettyTo(Formatter out) {
        out `pretty A`;
    }
}

# When this is associated with a Formatter, we format "reprs" of A.
class AReprFmt : AFmt {
    void format(Formatter out, A inst) {
        inst.formatReprTo(out);
    }
}

# When this is associated with a Formatter, we "pretty format" A.
class APrettyFmt : AFmt {
    void format(Formatter out, A inst) {
        inst.formatPrettyTo(out);
    }
}

AFmt _makeAPrettyFmt() {
    return APrettyFmt();
}

APrettyFmt().putIn(cout);
cout `$(A())\n`;  # Prints 'pretty A'

AReprFmt().putIn(cout);
cout `$(A())\n`;  # Prints 'repr A'
```

The Sequence Constant Protocol

Defining a sequence constant translates to the construction of an object and the insertion of a set of elements into that object. For example, the definition:

```
    array[int] a = [1, 2, 3];
```

Is equivalent to the following code:

```
    array[int] a = array[int](3);
    a[0] = 1;
    a[1] = 2;
    a[2] = 3;
```

Likewise, this code generates a linked list and iterates over it:

```
    for (elem :in List!['foo', 'bar'])
        cout `got $elem\n`;
```

is roughly equivalent to this:

```
    List temp = {};
    temp.append('foo');
    temp.append('bar');
    for (elem :in temp)
        cout `got $elem\n`;
```

The "temp" variable is here for purely illustrative purposes, no named variable is actually defined.

When a sequence constant is defined, the compiler does the following:

- Searches for a constructor for the type that can accept an unsigned integer or uintz value This value is assumed to be the number of elements in the sequence. - That failing, it searches for a default constructor for the type.

For each element between the brackets, the compiler:

- Searches for an `append()` method that will accept the element type as an argument.
- That failing, it searches for an array assignment method (`oper []=()`) that assigns that element (using an integer index).

This approach allows all sequence types to easily implement sequence constants.

Reference Counting

Reference counting is a simple form of memory management. Every object is assigned a *reference count*, which is essentially the number of other objects or variables referencing the object. When a new reference is added, the reference count is increased. When a reference is removed, the reference count is decreased. When the reference count drops to zero, the destructor is called and the object's memory is released.

Crack's reference counting mechanism is actually implemented in the language as part of the implementation of `Object` in the `crack.lang` module. The compiler uses two special hooks - the "`oper bind`" and "`oper release`" methods - to notify an object when a reference is being added (by calling "`oper bind`") and released (by calling "`oper release`"). These methods are implicitly non-virtual: they cannot be overridden by a derived class, do not make use of the vtable and therefore they can be safely applied to null objects.

It is possible to implement the bind and release methods in classes derived from `FreeBase` or `VTableBase` to implement your own memory management. For example, the `Wrapper` class in the (now deprecated) `crack.exp.bindings` module uses it `oper release` to always free the `Wrapper` instance when it is released, allowing it to essentially exist in the scope in which it is defined. Note that if you were to pass such an object out of that scope, the results would be undefined.

For efficiency, Crack does not bind and release every time you might expect: for one thing, objects passed as function arguments are often not bound and released for the function call - we know that the external caller has a reference to these objects. The called function can simply borrow them.

Crack also has the notion of "productive" and "non-productive" expressions. A productive expression is one that produces a reference. A non-productive expression simply borrows an existing reference. Variable references are always non-productive. Functions returning values are (almost) always productive.

The compiler will call `oper bind` when assigning a non-productive value to a variable, or when returning a non-productive value. It will call "oper release" when a variable goes out of scope or when productive temporary value is cleaned up. In general, temporaries get cleaned up at the end of the outermost expression of a statement. For the "&&" and "`||`" operators, temporaries get cleaned up for the secondary expression prior to cleanup of outer expressions.

Related to productivity is the concept of volatility: a variable is "volatile" if it can be modified in a function call. Globals and instance variables are volatile, function local variables are not (because a called function has no way to access them). An expression, in turn, is volatile if it is a simple reference to a volatile variable. When a volatile expression is passed as a function argument or used as the receiver in a method call, "oper bind" is called on the value. Otherwise, the function or method could change the value of the volatile variable and delete the object while it's still being referenced by a parameter.

There's one other thing you need to be aware of about reference counting: the mechanism is susceptible to the problem of reference cycles - this is when an object directly or indirectly references itself. When this happens, the entire cycle of objects can become unfreeable, potentially resulting in a memory leak. This is because each object retains a reference from the last object in the sequence, so even when all external references are removed, none of the objects will drop to a reference count of zero.

The best way to work around this is with the `RawPtr` class. `crack.rawptr.RawPtr` is a generic appendage that causes the compiler to omit calls to bind and release for its specializations. See `crack.rawptr` for a usage example. You still need to be aware of classes that have cyclic references. When they are present, use RawPtr. Cautiously.

Primitive Bindings

Crack allows you to directly import and call functions from shared libraries. A special variation of the `import` statement allows you to import symbols from a shared library:

```
    # import malloc() and free()
    import "libc.so.6" free, malloc;
```

After doing this, it is necessary to provide declarations of the functions you've imported:

```
    byteptr malloc(uint size);
    void free(byteptr val);
```

You can then use them like any other function:

```
    mem := malloc(100);
    free(mem);
```

Many C functions require special arguments like pointers to integers. These can be passed using primitive arrays:

```
# import "free()"
import "libc.so.6" free;
void free(voidptr mem);

# import C function "void doSomething(int *inOutVal)"
import "somelib.so" doSomething;
void doSomething(array[int] inOutVal);

# call it
v := array[int](1);
v[0] = 100;
doSomething(v);

# clean up our array
free(v);
```

The `crack.exp.bindings` defines an `Opaque` class. This can be used for structures returned from C functions that contain no user-serviceable parts. For example:

```
import crack.exp.bindings Opaque;
import "libFoo.so" Foo_Create, Foo_Destroy;
class Foo : Opaque {}

# create a Foo instance, then destroy it.
foo := Foo_Create();
Foo_Destroy(foo);
```

`Opaque` doesn't attempt to free the object, so it is important that you manage the object correctly yourself.

Sometimes C functions want to accept a function pointer to use as a callback. You can get this effect by defining a function and using a parameter type of `voidptr` for the callback parameter:

```
import "libFoo.so" Foo_SetCallback;
void Foo_SetCallback(Foo obj, voidptr callback);

void myCallback(Foo obj) { cout `callback called\n`; }
Foo_SetCallback(Foo_Create(), myCallback);
```

This won't work for overloaded functions: the compiler won't be able to tell which overload to use.

Crack's current approach to bindings is not without its problems:

- This whole business is extremely platform dependent. There's no guarantee that the shared libraries that you're importing will have the same names on other platforms, or that the functions you try to import will not be implemented as macros.
- You can't currently import global variables.
- `crack.exp.bindings` is experimental and may be discontinued. Its use is considered deprecated.

The primitive import mechanism exists mainly to facilitate quick hacks. For anything more significant than that, create an extension instead (see Extensions).

Annotations

Annotations are a *meta-programming* mechanism: they essentially allow you to extend the Crack compiler in Crack. The annotation system is very much a work in progress: much remains to be done with it. That said, you can still use it to do some very impressive things.

As a simple example, lets say that we want to define a macro that expands to write a marker to standard output. In module `myann` we could define:

```
import crack.compiler CrackContext;

void writeMarker(CrackContext ctx) {
    # the built-in annotations @FILE and @LINE refer to the current
    # filename and line number (something like "myann.crk" and 7, in this
    # case).
    ctx.inject(@FILE.buffer, @LINE, 'cout `marker\\n`;'.buffer);
}
```

From another module, we can use the macro by doing a *compile-time import* of it and invoking the annotation:

```
@import myann writeMarker;

# equivalent to 'cout `marker\n`;'
@writeMarker
```

In `myann`, we define the `writeMarker()` function, which accepts a `CrackContext` object as its sole argument. `CrackContext` is an interface to the compiler. We can use it to inject text, read and inject tokens, set callbacks from the compiler, emit errors and warnings and create other annotations.

`CrackContext` and all of the other objects defined in `crack.compiler` are fairly primitive: they are defined before the basic types (`String` and `Object`) even exist. As such, all of these interfaces must be built up from the primitive types. That's why we pass `@FILE.buffer` and `'cout marker\n;'.buffer` to inject instead of just `@FILE` and `'cout marker\n;'`. `CrackContext.inject()` accepts a `byteptr` argument. The `buffer` instance variable is a `byteptr` that points to the raw character data of the string.

The first and third arguments to `inject()` must also be *null terminated*. That means that the data must end with a zero byte value (this is the native way that strings are represented in C). All static strings in Crack happen to be null terminated, so we can easily do this with a static string. If we wanted to do the same with a constructed string we'd have to do something to guarantee that the string were null terminated (converting it to a `CString` would work, but there are often cheaper ways to accomplish this. Also note that the entire practice of injecting strings should be avoided in favor of the use of Token Sequence Literals.)

Using an annotation requires some special syntax. First, we import it using the `@import` statement. `@import` is just like `import` except that it imports the module at compile time into the annotation namespace. Symbols in the annotation namespace can be invoked with the syntax "@" *symbol*. *symbol* must refer to a function accepting a `CrackContext` object as its argument.

An annotation is invoked at compile time immediately after tokenization of the symbol associated with it.

Here's a more complicated example involving reading tokens. In this example, we process a simple variable definition and generate a getter and a setter for it:

```
import crack.compiler CrackContext;

void attr(CrackContext ctx) {
    # get the next token, assume it is a type
    type := ctx.getToken();
    if (!type.isIdent()) ctx.error(tok, 'Identifier expected');

    # get the attribute
    var := ctx.getToken();
```

```
        if (!var.isIdent()) ctx.error(tok, 'Identifier expected');

        # get a closing semicolon
        tok := ctx.getToken();
        if (!tok.isSemi())
            ctx.error(tok, 'semicolon expected after attribute definition');

        # get the token text - note that like all other strings in this
        # interface, these are of type byteptr, not String
        typeText := type.getText();
        varText := var.getText();

        # format and inject code (note that we add a null character at the end)
        StringFormatter fmt = {};
        fn := @FILE; ln := @LINE; fmt `
            $typeText __$varText;

            $typeText get_$varText() { return __$varText; }
            void set_$varText($typeText val) { __$varText = val; }
\0`;
        ctx.inject(fn, ln, fmt.createString().buffer);
```

In addition to injecting full strings, you can also store single tokens: see the documentation in the header files in the `compiler` directory of the source tree for details.

When doing injection, one thing to keep in mind is that single tokens and blocks of injected text will be parsed in reverse order of how they were inserted: so if we inject block A and then B, the parser will read the contents of B and then of A. The tokens *within* an injected block are not in reverse order (the inject function takes care of this) but the blocks themselves are.

You generally don't want to import annotation modules using the normal import. The `crack.compiler` module, which is needed for pretty much all annotations, does not exist at runtime for AOT compiled code, only for JIT, so you will not be able to compile your program to a stand-alone executable if you ignore this rule.

Inspecting Parser State

`CrackContext` provides two functions for inspecting parser state: these are `getScope()` and `getParseState()`. `getScope()` returns one of `SCOPE_MODULE`, `SCOPE_FUNCTION` or `SCOPE_CLASS` depending on whether the innermost enclosing context is within a module, function or class.

`getState()` currently only returns an indicator as to whether the parser is in the "base state." A return value of `STATE_BASE` indicates that we are not currently in the middle of parsing a definition or statement. For example, in the following code:

```
    int x;
    for (x = 0; x < 100; ++x)
        cout `$x\n`;
```

Annotations found immediately before the `int`, `for` and `cout` words and after the final semicolon would find the parser in the base state. Annotations anywhere else would not be in the base state.

A return value of `STATE_OPT_ELSE` indicates that we've completed parsing an if statement and the next thing might be an else clause.

These functions can be used in conjunction with parser callbacks to ensure that an annotation used to modify a definition or statement precedes the right kind of syntactic construct.

Here's an example of an annotation that verifies that it has been used in the base state of a class:

```
    void checkBaseState(CrackContext ctx) {
        if (ctx.getScope() != SCOPE_CLASS ||
            ctx.getParseState() != STATE_BASE &&
            ctx.getParseState() != STATE_OPT_ELSE
            )
            ctx.error('checkBaseState can only be used in a class definition '
                       'outside of a nested method or instance variable '
                       'definition.'.buffer
                      );
    }
```

Registering Parser Callbacks

The `addCallback()` and `removeCallback()` methods allow you to register and deregister parser callbacks. The parser calls these callbacks at certain points in the subsequent code.

Like annotations, callbacks are called with a `CrackContext` object as their only argument.

The following callbacks are available:

`PCB_FUNC_DEF`

Called immediately after the opening parenthesis of the parameter list of a function definition.

`PCB_FUNC_ENTER`

Called immediately after the opening curly brace of a function definition.

`PCB_FUNC_LEAVE`

Called immediately before the closing curly brace of a function definition.

`PCB_CLASS_DEF`

Called immediately after the "class" keyword in a class definition or forward declaration.

`PCB_CLASS_ENTER`

Called immediately after the opening brace of a class definition.

`PCB_CLASS_LEAVE`

Called immediately prior to the closing brace of a class definition.

`PCB_VAR_DEF`

Called immediately after the assignment operator, semicolon, or comma following a variable definition.

`PCB_EXPR_BEGIN`

Called immediately before parsing the outermost expression in a statement or the initialization clause of a for loop.

`PCB_CONTROL_STMT`

Called immediately after the keyword signifying a control statement (`while`, `for` or `if`).

As stated earlier, we can combine parser state with a callback to insure that an annotation is placed in a certain location in the code. The following code creates a `funcMod` annotation that can only be called prior to a function definition:

```
    Array[Callback] callbacks;

    void deregisterCallbacks() {
        for (cb :in callbacks)
            removeCallback(cb);
    }

    void good(CrackContext ctx) {
        deregisterCallbacks(ctx);
    }

    void bad(CrackContext ctx) {
        deregisterCallbacks(ctx);
        ctx.error('@funcMod can only be used before a function '
                   'definition'.buffer
                  );
    }

    void funcMod(CrackContext ctx) {

        if (ctx.getScope() != SCOPE_CLASS ||
            ctx.getParseState() != STATE_BASE
            )
        ctx.error(" annotation can not be used here (it must precede a "
                   "function definition in a class body)".buffer
                   );

        callbacks.append(ctx.addCallback(PCB_FUNC_DEF, good));
        callbacks.append(ctx.addCallback(PCB_CLASS_DEF, bad));
        callbacks.append(ctx.addCallback(PCB_EXPR_BEGIN, bad));
        callbacks.append(ctx.addCallback(PCB_CONTROL_STMT, bad));
    }
```

Function Flags

The parser contains a function flag register that can be used by an annotation to set flags on function definitions that follow it. For example, we could implement the `@static` built-in annotation as follows:

```
    import crack.compiler CrackContext, FUNCFLAG_STATIC;

    void static(CrackContext ctx) {
        ctx.setNextFuncFlags(FUNCFLAG_STATIC);
    }
```

The only other function flags supported at this time are `FUNCFLAG_FINAL` and `FUNCFLAG_ABSTRACT`.

Error Contexts

Sometimes it is useful to cause the parser to print additional information with an error message. For example, if you get a syntax error in a macro, you would ideally like to see the location within the macro that caused the error as well as the location where the macro was invoked.

You can do this with error contexts. Error contexts are string messages that are printed on the line following the original error message. There can be as many lines of error context as you want, they are stored as a stack and the last context is printed first.

Error contexts are pushed using `CrackContext.pushErrorContext()`. They can be removed from the stack either by calling `CrackContext.popErrorContext()` or by putting back a special token of type `TOK_POPERRCTX`. The latter causes the error context to be popped the next time the token is read, which is useful if your annotation injects a sequence of tokens and then returns - the error context will remain active until after your sequence of tokens is parsed.

As an example, let's say that we have registered `beginClass()` and `endClass()` as the `PCB_CLASS_ENTER` and `PCB_CLASS_LEAVE` callbacks. We could add the message "in a class!" to the error message with the following code:

```
    void beginClass(CrackContext ctx) {
        ctx.pushErrorContext('in a class!'.buffer);
    }

    void endClass(CrackContext ctx) {
        ctx.popError();
    }
```

An annotation can register other annotations. For example, if we wanted to create an annotation to define our `beginClass()` and `endClass()` annotations, we could do this:

```
    void registerAnnotations(CrackContext ctx) {
        ctx.storeAnnotation('beginClass'.buffer, beginClass);
        ctx.storeAnnotation('beginClass'.buffer, endClass);
    }
```

The new annotations are scoped to the lexical context from which they are called - for example, if registerAnnotation() is called from the top-level block of a function, the annotations will only exist within the scope of the function.

When registering an annotation, it is often useful to associate the annotation with user data. For example, the following example is a very weak macro facility. `@record` binds the next token to an identifier, when the identifier is used as an annotation it expands to its value:

```
    import crack.compiler CrackContext, Token;

    void _playback(CrackContext ctx) {
        tok := Token.unsafeCast(ctx.getUserData());
        ctx.putBack(tok);
    }

    void record(CrackContext ctx) {
        # get the identifier (error checking omitted)
        ident := ctx.getToken();

        # get the token to attach to it
        val := ctx.getToken();

        # register the identifier as an annotation, store 'val' as user data.
        ctx.storeAnnotation(ident.getText(), _playback, val);

        # VERY IMPORTANT: need to bind a reference to the user data so it
        # doesn't get deallocated
        val.oper bind();
    }
```

We could use it like this:

```
    @record var 'some string';
    String x = @var;
```

Note that we perform `oper bind()` on the value token after storing it with the annotation. This prevents the token from being garbage-collected when `record()` terminates. User data is a `voidptr`, not an object, and therefore it can not be managed. We also have to use an `unsafeCast()` to restore it to a `Token`.

We can access existing annotations with `CrackContext.getAnnotation()`. Annotation objects contain the following access methods:

```
voidptr getUserData();
byteptr getName();
voidptr getFunc();
```

We've demonstrated obtaining and injecting tokens with the `getToken()` and `putBack()` methods. We can also create our own tokens:

```
import crack.compiler CrackContext, Token, TOK_STRING;

# annotation to convert the next token to a string
void stringify(CrackContext ctx) {
    tok := ctx.getToken();
    ctx.putBack(Token(TOK_STRING, tok.getText(), tok.getLocation()));
}
```

Usage would be as follows:

```
String x = @stringify test;
```

Details of the annotation interface can be found in the header files in the `crack/compiler` directory installed in your include tree.

Built-in Annotations

Crack defines the following special built-in annotations:

@final

Marks a class method as final.

@static

Marks a class method as static. (See Static and Final Methods above)

@abstract

Marks a class or method as abstract. (See Abstract Methods above)

@encoding

Identifies the source file's encoding. (See Encoding below)

@FILE

Expands to a string containing the current filename.

@LINE

Expands to an integer containing the current line number.

The Standard Annotations

The `crack.ann` module defines a set of standard annotations. Three of them are `@interface`, `@implements` and `@impl` where are documented above in Interfaces. The others allow you to define macros. Crack macros are similar to C preprocessor macros with some improvements.

To define a macro for use within a module, simply use the `@define` annotation:

```
@import crack.ann define;

@define max(Type) {
    Type max(Type a, Type b) {
        return a > b ? a : b;
    }
}

int a = 3, b = 5;
x := @max(a, b);   # x becomes 5
```

When defining a macro, the identifier after the `@define` annotation is the macro name. The argument list follows, then the body of the macro enclosed in curly braces.

Defining a macro creates a new annotation with the macro name. Macro annotations parse an argument list that follows them and then expand the macro with the arguments. Macros are scoped to the context in which they are defined - so if you define a macro in the body of a class, it can only be used from within that class.

When a macro is expanded, all of the tokens within the curly braces are expanded into the location of the macro. All macro arguments are expanded to the values passed in for those arguments.

Macros also set error contexts for the span of their expansion - if you get a syntax error in the body of the macro, the error message will report the location of the error within the definition of the macro, followed by the location of the annotation that expanded the macro.

As in C macros, you can do argument concatenation and stringification. A single dollar sign in a macro stringifies the following argument, a double dollar sign concatenates the argument with the adjacent tokens.

For example, if we wanted to define a macro to produce command-line flags, we might do something like this:

```
@define flag(Type, name) {
    Type flag_$$name = Type$$Flag($name);
    flags.addFlag(flag_$$name);
}

@flag(int, port);
```

This would generate the following code:

```
Type flag_port = intFlag('port');
flags.addFlag(flag_port);
```

By default, macros are limited to the modules that they are defined in. To make a macro available to other modules, you have to export it:

```
@import crack.ann define, export, exporter;

# Indicates that the module will be exporting macros.
# What this actually does is to import various utility functions from
# crack.ann needed to do an export.
@exporter
```

```
@define myMacro(a, b) { ... }
@export myMacro
```

We can then import the macro as an annotation to another module:

```
@import mymodule myMacro;

@myMacro(1, 2)
```

Assertions

`crack.ann` also defines an `@assert` annotation that can be used to verify that a given expression is true:

```
@import crack.ann assert;

@assert(2 + 2 == 4);
```

`@assert` raises an `AssertionError` if the condition is false with a string containing the location of line and a representation of the failing expression.

Structs

`crack.ann` also defines a `@struct` annotation that can be used to generate a very limited class with a constructor that allows you to initialize all of its member variables. For example:

```
@import crack.ann struct;

@struct Coord { int x; int y; }
Coord c = {10, 20};
```

Structs are very limited as of Crack 1.5. In particular:

- They don't support inheritence.
- You can't add any other methods.
- Instance variables definitions must be strictly of the form "*type name*;" (we couldn't have written "int x, y;" above).

`crack.ann` also includes a more limited macro called `cvars` ("constructed variables"). `cvars` is followed by a block of variable definitions which are expanded to the variable definition list and the constructor:

```
@import crack.ann cvars;

class Foo {
    @cvars {
        String a;
        int b;
    }
}

# Equivalent to:

class Foo {
    String a;
    int b;

    oper init(String a, int b) : a = a, b = b {}
}
```

This form allows inheritence and the definition of other methods.

Other standard annotations

The standard annotation module contains other annotations as well. See Auto-imports for information on automating lazy-imports. See Interfaces for information on the `@interface` annotation.

Annotations in Generics

There are limitations on the ways that you can use annotations with generics. Generics are implemented as a simple stream of language tokens that are replayed by the compiler when the generic is instantiated. Unfortunately, annotations can affect parser state in ways that can not always be passed through to the instantiation. For this reason, there are limitations on the kinds of mutations that can be passed in from the context of a generic to the code within the generic. The only thing that is preserved are the imports.

So this code would not work:

```
    @import crack.ann define;
    @define func() { void f() {} }

    class A[T] {
        @func()   # func does not exist!
    }
```

Likewise, registering parser callbacks, error contexts or new annotations will not affect the instantiated generic code. The only way to do so is to perform all mutations within the scope of the generic. So, for example, we could have successfully written our last example like this:

```
    @import crack.ann define;  # Imports are preserved.

    class A[T] {
        @define func() { void f() {} }
        @func()
    }
```

In this case, `define` is passed through (because it's an import) and `func` will be defined within the generic at the time of its instantiation.

Token Sequence Literals

When using annotations for code generation, it's often necessary to emit boilerplate code and inject it into the tokenizer. We've demonstrated how this can be done using the `inject()` and `putBack()` methods of `CrackContext`, a better approach is through the token sequence literal annotations - `@tokens` and `@xtokens`.

`@tokens` is an annotation defined in `crack.ann` that consumes a sequence of tokens following it and expands to a `NodeList` object, which can then be expanded into a `CrackContext`, creating a sort of macro system for annotations. For example, if we wanted to define a macro that wrote the current line number to standard error, we could do something like this:

```
    import crack.ann deserializeNodeList;;
    @import crack.ann tokens;

    void linenum(CrackContext ctx) {
        @tokens {
            cerr `$(@FILE):$(@LINE)\n`;
        }.expand(ctx, null);
    }
```

The `@tokens` annotation consumes the curly-brace delimited block following it and emits the code to construct a `NodeList` from it. We then call the `expand()` method to expand the node list into the usage context (the second argument to `expand()` is an argument vector, which can be null here because the `NodeList` contains no parameters).

It is worth noting exactly what is going on here, as this involves annotations at multiple levels:

- `@tokens` consumes the delimited list of tokens following it and constructs its own internal NodeList.
- It then calls `serializeNodeList()` to convert the node list into a string constant token.
- Finally, it emits a call to deserializeNodeList() with the string constant as its argument.

In this way, a serialized `NodeList` that exists at compile-time is passed through to the code that subsequently reuses it at runtime.

The list of tokens following the `@tokens` annotation can be delimited by curly braces, square brackets, or parentheses. The only restriction is that if the symbol is nested within the sequence, it must include a matching terminator within the sequence. So something like this is perfectly legal:

```
    # The curly is unmatched, but the sequence is delimited by the square
    # brackets.
    @tokens [ while (true) { ].expand(ctx, null);
```

The `@tokens` annotation is great for constant sequences, but typically in code generation scenarios we want to parameterize our code blocks. The `@xmac` token lets us do this:

```
    import crack.ann deserializeXMac;
    @import crack.ann xmac;

    void exception(CrackContext ctx) {
        tok := ctx.getToken();

        @xmac {
            class $className {
                int i;
            }
        }.set('className', tok).expand(ctx);
    }
```

In this case, the `className` variable is a parameter. We substitute it for a token (which will be converted to a node list) and then expand it into the context. The `@xmac` annotation expands to an expression returning a `crack.ann.XMacro`. `XMacro`'s `extend()` method doesn't accept an argument list, but it maintains one internally. All variables must be defined at the time of expansion or an `InvalidStateError` exception will be thrown.

If `@xmac` is followed immediately by an asterisk, it will expand its variables from your local context. So we could have written the code above like this:

```
    void exception(CrackContext ctx) {
        className := ctx.getToken();

        @xmac* {
            class $className {
                int i;
            }
        }.expand(ctx);
    }
```

A more complete example of this concept can be found in `example/struct_ann.crk`, which provides a toy implementation of the `@struct` annotation using `@xmac`.

Extensions

In addition to the approach for calling C functions identified in Primitive Bindings, Crack supports a formal extension concept allowing you to load specially crafted shared libraries as if they were Crack modules.

When importing a module, if the executor is unable to find a native Crack module in the search path, and if there is a shared library with that name on the search path, and if the shared library defines a pair of initialization functions corresponding to the module name, the shared library is loaded as an extension module, and the initialization function is called at load time.

Initialization functions must have a name corresponding to the canonical name of the module. There are two of them, the "compile init" function and the "runtime init" function. The convention for forming their names is to join all of the elements of the canonical name with underscores and then append either "_cinit" (for the compile-time function) or "_rinit" (for the runtime function) to the result. So for the extension module `foo.bar.baz`, the initialization functions would be named `foo_bar_baz_cinit` and `foo_bar_baz_rinit`.

Initialization functions should be C++ functions declared as 'extern "C"'. The compile-time init function accepts a parameter of type `crack::ext::Module *`. The role of the compile-time initialization function is to populate the module object with the module's interface. To describe it another way, this initialization function configures everything that is necessary to import the module at compile time. The compile-time initialization function is not called for extensions loaded from an AOT compiled program, use the runtime-initialization function for this purpose.

The runtime initialization function is called at runtime during the initial load of the module (only once). Note that when running in JIT mode, both the compile-initialization and runtime-initialization functions are called.

For example, let's say that we want to provide a binding for a C function called `myfunc()`:

```
void myfunc(char *data) { printf("my data: %s", data); }
```

Our init function (again, assuming the module is `foo.bar.baz`) would be as follows:

```
#include <crack/ext/Module.h>
#include <crack/ext/Type.h>
#include <crack/ext/Func.h>

extern "C" void foo_bar_baz_cinit(crack::ext::Module *mod) {
    // define a function returning void
    Func *func = mod->addFunc(mod->getVoidType(), "myfunc", (void *)myfunc);
    // add the "data" argument, parameters are the type and the parameter
    // name
    func->addArg(mod->getByteptrType(), "data");
}

extern "C" void foo_bar_baz_rinit() {
    // do module runtime initialization.
}
```

The details of the extension interface are documented in the header files in the `crack/ext` directory installed with your distribution. To compile your extension, you'll want to:

- build it into a shared library, linked with whatever support code it needs (usually the code it's wrapping)

- Add the include directory for crack headers to the include path (usually something like `/usr/local/include/crack-1.5`)

The Extension Generator

Creating extensions by hand is way too much work. In the interests of simplifying it we have the `crack.extgen` module. This defines a `@generateExtension` annotation that hijacks the compiler to generate a source file defining the extension. Its usage is best illustrated by example:

```
@import crack.extgen generateExtension;

# generate an extension that will be imported as 'foo.bar'
@generateExtension foo.bar {

    # name of the source file to generate.
    @filename "foobar.cc"

    # code to inject into the extension source file
    @inject '#include "FooBar.h"\n';

    # function accepting a char *
    void puts(byteptr string);

    # function accepting a float and an integer pointer
    int hairy(float val, array[int] result);

    # opaque type and a function that uses it.
    class MyType;
    void myFunc(MyType val);

    # constants (must be integer or float types)
    const int
        # with no value it will be set to the value of the C constant
        SAME_NAME_AS_C_CONSTANT,

        # when defined from an identifier, the identifier is the C
        # constant
        SIMPLE_NAME = LESS_AESTHETIC_NAME_OR_VALUE,

        # when assigned from an integer, the value is the integer
        ONE_HUNDRED = 100,

        # a string value can be used to indicate a complex expression
        SIMPLE_NAME = 'ONE_HUNDRED >> 1';

    # @ifdef lets you define bindings based on preprocessor values (there
    # is also an @ifndef)
    @ifdef FOO_FEATURE_X {
        int useFeatureX();
    }
}
```

Numeric Types

The rules with which Crack deals with numeric types are complicated but predictable.

The general philosophy is that the Platform Dependent Numeric Types (PDNTs, which are `int`, `uint`, `intz`, `uintz` and `float`) should be easily usable in most cases where the programmer cares more for performance and compatibility with C libraries than for precision. For this reason, implicit conversions to PDNTs are allowed from all other numeric types and PDNTs are said to "dominate" in binary operations, meaning that the type of the result of a binary operation involving a PDNT will be the type of the first PDNT (from left to right) in the operation.

The Universal Numeric Types (UNTs, which are `byte`, `int32`, `uint32`, `int64`, `uint64`, `float32`, and `float64`) are intended for use in situations where the programmer cares very much about precision. You can implicitly convert to a PDNT only in cases where no loss of precision is possible.

In binary operations (for example '+', '-', '*' ...) involving multiple numeric types, PDNTs are said to "dominate." That is, if one operand is a PDNT and the other is a UNT, the the PDNT flavor of the operation is always matched. So `float32(5) + int(5)` is of type `int`.

If both values in a binary expression are PDNTs, the type of the left-most operand dominates, so `float(5) + int(5)` is of type `float`.

Note that integer and floating-point constants, although they default to the corresponding PDNT types in situations where type cannot otherwise be inferred, do not dominate - so an expression like: `int32(val) + 1` is of type `int32`.

All of this behavior is implemented using the normal Crack method resolution and conversion rules. The trick is that PDNT binary operations are implemented as methods and reverse methods and UNT binary operations are implemented as functions in the default namespace. UNT operations are also defined in order of smallest to largest types, so binary operation involving two UNTs will match the first operation with a type large enough to accommodate both operands.

As an example, when resolving `int64(a) + int(b)`:

- check `int64` for an `oper +()` method accepting first an `int` argument and then an argument that can be converted to `int` (no match)
- check `int` for an `oper r+()` method (the *reverse operator* of `oper +`) that accepts:
    - an `int64` argument (no match)
    - an argument that `int64` can be converted to (matches `oper r+(int)` because all numeric types can convert to the `int` PDNT)

On the other hand, let's consider `int64(a) + byte(b)`:

- Check `int64` for an `oper +()` method accepting a `byte` argument or an argument that can be converted to `byte` (no match)
- Check `byte` for an `oper r+()` method accepting an `int64` argument or an argument that can be converted to `int64` (no match)
- Check the global scope for `oper +(int64, byte)` (no match)
- Check each of the following to see if `int64` and `byte` can be converted to each of the arguments
    - `oper +(byte, byte)` (no match)
    - `oper +(int32, int32)` (no match)
    - `oper +(uint32, uint32)` (no match)
    - `oper +(int64, int64)` (matched!)

As a special exception, we don't check for methods or reverse methods when performing binary operator resolution on constants: doing so would cause constants (whose default types are PDNTs) to dominate. This would lead to silly results like `float64(a) + 1` producing a result of type `int`.

Threading

Crack provides rudimentary support for multi-threading. Writing thread-safe programs is difficult in any sufficiently primitive language, and crack is no exception.

Crack's reference counting mechanism is thread-safe, however it's bind and release semantics are not: there is no guarantee of order of operations when it comes to assigning reference counted variables, and without proper synchronization it may still be possible to get into a situation where you are referencing illegal memory.

When sharing objects between threads, always be sure to do one of the following:

- Fully transfer ownership of the object to the other thread.
- Fully protect the state of the object with a `Mutex`.
- Treat the object as immutable.

Follow these rules diligently and you won't have any problems. Keep in mind that module-scoped globals are very easy to share accidentally.

See Threads for information on the `crack.threads` module used to support threading.

Debugging

Crack has only minimal support for debugging. If your program seg-faults or aborts, you can at least get a sparse stack-trace by running it in JIT mode under a fairly recent version of GDB (7.0 or later).

The recommended approach to debugging Crack programs is to instrument the code with console writes, assertions, and other forms of intrusive inspection. This can be done for the standard library or any intermediate package by cloning all or a portion of the library tree and adding instrumentation to the cloned code.

Encoding

Crack scripts are essentially ASCII-compatible binary data. Symbols, keywords numerics and whitespace are ASCII characters, other elements can utilize byte values from 0x80 to 0xFF. Specifically, comments, and all types of strings can contain any bytes whatsoever (of course, the delimiter and escape character have special meanings, as does the dollar sign for interpolated strings). Identifiers treat the high byte values 0x80-0xFF in the same way as alphabetic characters - an identifier may include these at any position.

These features allow you to use any ASCII compatible encoding for Crack scripts. To identify the encoding for tools like editors and web browsers that care about encodings, the language features a built-in "encoding" annotation. This annotation has no meaning to the compiler, it is simply a formal declaration of the encoding. It must be followed by a string identifying the name of the encoding. For example, to indicate that a script uses UTF-8 encoding, one might begin the script like this:

```
#!/usr/bin/env crack
@encoding 'utf8'

// script contents...
```

The `encoding` annotation should be either the first or second line of the script. If a script or module doesn't specify an encoding, tools are free to attempt to infer it from the contents of the script. In the absence of high-byte values, the encoding is implicitly ASCII.

Viewing Module Documentation with `crackdoc`

The `crackdoc` program lets you view the contents of modules along with the contents of the "doc-comments" for each definition. `crackdoc` is compiled and installed along with the crack runtime code, so it should always be available. To use it, just specify the name of the module that you want to see the documentation for:

```
$ crackdoc crack.lang
```

This will display the colorized documentation for `crack.lang` in the `less` paginator.

By default, `crackdoc` searches for modules in the default library search locations, obtained from the crack executor itself. The "-l" option can be used to specify an alternate search path.

You can also use the "-r" option to view the colorized raw source code of the module. This can be very useful in cases where `crackdoc` isn't good enough, the module hasn't been well documented or if you just want to see the full implementation of the module that will actually be used by the executor.

`crackdoc` actually invokes the compiler on the target module, so the first time it is run it will take a few seconds. After that, it caches the formatted documentation in the `~/.crack/doc` directory.

`crackdoc` is still very much a work in progress. It doesn't work for generics yet and doesn't do a very good job with inheritance. Nevertheless, it's still quite useful.

Appendix 1 - Libraries

Crack comes with its own (sparse but growing) set of standard libraries. These are loosely organized as the modules under `crack` and `crack.exp`. `crack.exp` includes modules that are designated as "experimental." Modules in the experimental tree do not have the same guarantees as the other modules. Their interfaces may change going forwards. In some cases, these are older, deprecated libraries that may be removed entirely.

Special Language Types

Strings and Buffer Types

Strings are derived from `Buffer`. A buffer is just a `byteptr` (stored in the instance variable `buffer`) and a size (stored in the instance variable `size`).

The memory that a buffer references is presumed to be read-only, but nothing in the language enforces this. For cases where a writable buffer is desired, use something derived from a `WriteBuffer` subclass. `WriteBuffer` inherits from `Buffer`, but the read-only requirement is relaxed: it is legal to modify the buffer of a `WriteBuffer`.

Most output functions accept a `Buffer`, most input functions accept a `WriteBuffer`. For cases where you want a `WriteBuffer` that frees its own buffer memory, use a ManagedBuffer.

Constant strings are actually of type `StaticString`. This is a class for strings with buffers that are not to be deleted (because static strings can reference data in read-only memory segments). Constant strings are always guaranteed to be null terminated, making them suitable for use with low-level C functions defined in extensions and primitive bindings. This is not true of strings or buffers in general, see CString for a more general solution.

Like any object, a buffer can be converted to a `bool`. A buffer is **true** if it has a non-zero size.

Any two buffers can be compared - a byte-for-byte comparison is performed. If one buffer is identical to the beginning of the other, but is shorter, the shorter buffer is "less than" the larger buffer.

The `String` class is one of the few types that don't need to be imported. `Buffer` and its other descendants can be imported from `crack.lang`.

CString

When dealing directly with C functions expecting null terminated strings, we often need to convert a Crack string to a null-terminated value. `CString` is one way to do so. Given an arbitrary buffer type, `CString` will create a copy of the buffer and insure that there is a null terminator at `buffer[size]` (note that the size of a `CString` excludes the byte for the null terminator).

This allows us to safely do something like this:

```
import runtime strcpy;

String userInput = getUserInput();
data := byteptr(userInput.size + 1);

strcpy(data.buffer, CString(userInput).buffer);
```

ManagedBuffer

A managed buffer is a `WriteBuffer` that manages its underlying buffer memory. These are designed for use with IO operations. The typical use case is to read into it and then convert it to a string, either by copying the underlying buffer or by passing ownership of it to the new string:

```
# create a 1K managed buffer.
ManagedBuffer buf = {1024};

# read it, store the size of what we read
buf.size = cin.read(buf);

# convert it to a string and take ownership of the buffer (the second
# argument of the constructor is "takeOwnership")
String s = {buf, true};
```

Note that `Reader` includes a utility method that does this more intelligently (it only transfers ownership if the amount read is at least 75% of the buffer size):

```
# 's' is a string.
s := cin.read(1024);
```

There is also an `AppendBuffer` which is a managed buffer that will grow as you add data to it.

SubString

The SubString class provides a lightweight string that references the buffer of an existing string. You can create one with the `slice()` and `substr()` methods of a normal string (or of another `SubString`).

`slice()` and `substr()` are very similar, but they have slightly different semantics. `slice(a, b)` returns the substring between positions 'a' and 'b' in the string (inclusive for 'a', non-inclusive for 'b'). `substr(a, b)` returns the substring starting at 'a' of length 'b'. Both methods can omit the second argument, in which case the remainder of the string starting at 'a' is returned, and both allow you to provide negative offsets from the end of the string.

Examples:

```
data := 'sample string';

# all of these assign the SubString 'str' to s
s := data.slice(7, 10);
s = data.substr(7, 3);
s = data.slice(-6, -3);
s = data.substr(-6, 3);

# both assign the SubString 'string' to s
s = data.slice(7);
s = data.substr(7);
```

You can also create a substring explicitly:

```
import crack.lang SubString;

s := SubString('this is a test', 10, 4);
```

`SubString` is derived from `String`, so you can use it in the same way that you would a string without the cost of managing and copying a portion of the buffer.

Standard Exceptions

The following standard exception types are defined in `crack.lang`:

`Exception`

The exception base class. Generally all exception types should derive from this. It provides stack trace functionality.

`BadCastError`

Thrown by the `cast()` function when used on an object that is not of the specified type.

`AssertionError`

Thrown when an internal invariant is broken. This should generally indicate a bug in the module that throws it.

`IndexError`

Thrown when an array operation is passed an invalid index (greater than or less than the number of elements).

`InvalidResourceError`

Thrown when an operation is performed on a resource that does not exist or is not appropriate for this kind of action (such as trying to open a directory for read).

`InvalidStateError`

Thrown when an operation is performed on an object that is not in an appropriate state for the operation (such as trying to write to a closed file).

`InvalidArgumentError`

Thrown when a function is passed an invalid argument.

Containers

Containers provide support for the basic sequential and mapping data structures expected from a modern programming language. Crack's container library is defined in a set of modules in `crack.cont`. Each module is named for the container generic that it provides, for example, high-level arrays are in `crack.cont.array` and the array class itself is called `Array`.

All containers have a `count()` method that returns the number of elements in the container. They also convert to boolean based on whether or not they are empty:

```
c := createContainer();
if (c)
    cout `container has $(c.count()) elements\n`;
else
    cout `container is empty\n`;
```

Iterators

Iterators are like iterators in Java. All container types have an `iter()` method which returns an `Iterator` object for the container.

Iterators allow you to iterate over the set of objects in a container one at a time. As an example, consider an array:

```
Array[ElemType] arr = createArray();

for (i := arr.iter(); i; i.next())
    cout `got $(i.elem())\n`;
```

`arr.iter()` creates an iterator over the array initialized at the first item. An iterator is "true" for as long as there are elements remaining in the collection. `next()` forwards the iterator to the next element, and `elem()` returns the element associated with the iterator.

Iterators conform to The Iteration Protocol, and as such can be used more succinctly in a `for` loop:

```
for (elem :in arr)
    cout `got $elem\n`;
```

Arrays

Arrays are a collection that preserves element order with fast random-access assignment and retrieval. Unlike their low-level counterparts, the are safe for general use. They manage both the size of the underlying array and the reference counts of the elements.

To construct an array with the expected number of elements:

```
import crack.cont.array Array;
Array[String] arr = {10};  # 10 element array of Strings
```

The number of elements specified is the array *capacity* - this is the number of elements that can be added before it is necessary for the array to reallocate its underlying, low-level array. It is not the size of the array. If you attempt to access `arr[0]` in the example above, you would get an `IndexError`.

Like other containers, arrays have a `count()` method that returns the actual number of elements in the array. At the time of construction, the count is zero.

To add an element to an array, use the `append()` method:

```
arr.append('first element');
arr.append('second element');
cout `count = $(arr.count())\n`;  # writes "2"
```

Note that `append()` will occasionally be an O(n) operation: if we don't have capacity for a new element, `append()` will reallocate the array to one twice as big.

To get an element in an array, use the bracket operator:

```
cout `elems: $(arr[0]), $(arr[1])\n`;
```

You can also replace an existing element in an array using element assignment:

```
arr[1] = 'new second element';
```

Note that this only works to replace an element: `arr[2] = 'something'` would result in a runtime error.

Finally, like all containers, you can iterate over the elements of an array:

```
for (i :on arr)
    cout `elem: $(i.elem())\n`;
```

Array iterators expose an "index" field allowing you to obtain the index of the current element. This is useful in cases where you want to mutate the array during iteration:

```
# Zero out all elements divisible by three.
for (i :on arr) {
    if (!(i.elem() % 3))
        arr[i.index] = 0;
}
```

It should be noted that there is a specialization of `Array[String]` called `StringArray` defined in the `crack.strutil` module. In addition to the standard `Array` functionality, it's contains methods for obtaining a primitive array and for joining the array into a single delimited string.

Linked Lists

The `List` class implements a linked list. Linked lists aren't very good for random access, but they do support constant time insert and append, making them preferable to arrays in certain cases.

The `List` interface looks very much like the `Array` interface:

```
import crack.cont.list List;

List[String] l;

# add some elements
l.append('first');
l.append('second');

# iterate over them
for (elem :in l)
    cout `got element $elem\n`;
```

Random access lookup, delete and insert are all available, although they are all O(n) operations:

```
cout `$(l[1])\n`;       # prints "second"
l.delete(0);            # deletes 'first'
l.insert(0, 'first');   # reinserts 'first'
```

You can also push a new element onto the front of the list (equivalent to `insert(0, ...)` but terser and slightly faster):

```
l.pushHead('zero');
```

`pushTail()` is also available as a synonym for `append()`.

Likewise, you can pop items off the head of the list:

```
elem := l.popHead();  # after the last operation, elem == 'zero'
```

`popTail()` can not be implemented efficiently: if you need this, use a `DList`.

Doubly-Linked Lists

The `DList` class implements a doubly-linked list. `DList` provides some additional functionality over `List` and can facilitate faster operations, but at the cost of higher memory usage (each `DList` element requires an additional pointer). `DList` is part of the `crack.list` module.

`DList` supports inserting and deleting the element referenced by an iterator:

```
# all sequence types can be initialized with a sequence constant.
DList[String] dl = ['first', 'delete me', 'temp', 'before me'];

for (i :on dl) {

    # delete the 'delete me' string from the list
    if (i.elem() == 'delete me') {
        dl.delete(i);

    # insert 'inserted' before 'before me'
    } else if (i.elem() == 'before me') {
        dl.insert(i, 'inserted');
        i.next();  # skip the item we just inserted
    }
}
```

After insert, the iterator points to the new element. After delete, it points to the element after the deleted element.

`DList` also supports a bi-directional iterator:

```
# get a bi-directional iterator - an argument of "true" causes
# the iterator to be initialized to the end of the list.  "false" would
# start it at the beginning.
i := dl.bidiIter(true);
while (i) {
    cout `$(i.elem())\n`;
    i.prev();
}
```

You can use a bi-directional iterator everywhere you can use a normal iterator.

In addition to the `popHead()` function, `DList` supports a `popTail()` function:

```
    DList dl;
    dl.append('first');
    dl.append('second');
    elem := dl.popTail();  # elem == 'second'
```

TreeMap

A "map" or an "associative array" is a container that stores a set of values each indexed by a key. Both keys and values can be arbitrary types. The Crack library provides both hashtable and red-black tree map implementations, `TreeMap` is a red-black tree. Red-black trees provide a natural sort order, and provide O(log n) insertion and lookup with constant time rebalancing.

To create red-black tree map:

```
    import crack.cont.treemap TreeMap;

    # maps are defined with the types of the key and the value.
    map := TreeMap[String, String]();
```

To add elements:

```
    map['first'] = '1';
    map['second'] = '2';
    map['third'] = '3';
```

To access them:

```
    cout `$(map['first'])\n`;  # writes "1"
```

Iteration is the same as for arrays, except that the elements are `KeyVal` objects. As their name suggests, `KeyVal` objects have a `key` and `val` attribute:

```
    for (kv :in map)
        cout `map[$(kv.key)] = $(kv.val)\n`;
```

HashMap

A `HashMap` provides the same interface as `TreeMap`, but is backed by a hashtable instead of a red-black tree. Hashtables should provide significantly faster inserts and lookups (generally O(1), although insertions will occasionally be O(N) when table grows beyond its capacity and both insertions and lookups can be a multiple of constant time in cases of hash collisions). The downside is that they do not preserve the order of their keys. If you iterate over a hashtable the order of elements returned is undefined.

Since the interface is the same as that of `TreeMap`, we won't go over it in detail except to provide an example of how it is constructed:

```
    import crack.cont.hashmap HashMap;

    HashMap[String, int] map = {};
    map['first'] = 100;
    map['second'] = 200;
```

Algorithms (Sort Functions)

The `crack.algorithm` module includes two sorting algorithms for arrays. For example:

```
    import crack.cont.array Array;
    import crack.algorithm QuickSort, InsertionSort;
    Array[int] arr = getAnArray();

    # try out both kinds of sort
    QuickSort[Array[int]].sort(arr);
    insertionSort[Array[int]].sort(arr);
```

For aggregate types, sorting relies on the comparison functions of `Object`, which in turn rely on the overridable `cmp()` method. For primitive types, the type must define the comparison operators.

The strutil Module

`crack.strutil` contains utility functions for working with strings that have external dependencies making them unsuitable for inclusion in the `String` or `Buffer` classes themselves.

```
    import crack.strutil split, StringArray;

    # create a StringArray, which is a class derived from Array[String] only
    # containing a little extra functionality.
    StringArray arr = ['first', 'second', 'third'];

    # concatenate all of the elements together, separated by a space
    text := arr.join(' ');  # text = 'first second third';

    # split them backk apart (see also crack.ascii.wsplit())
    arr2 := split(text, ' ');  # arr2 = ['first', 'second', 'third']
```

The ascii Module

The Crack language strives to be encoding-neutral as much as possible. That's why the `String` class and the `crack.strutil` modules generally avoid providing functionality for dealing with whitespace or capitalization. Nevertheless, there are times when you care about encodings, and there's no encoding more ubiquitous than ascii. Therefore, we include this kind of functionality in the `crack.ascii` module:

```
    import crack.ascii radix, strip, toUpper, wsplit;

    # strip trailing and leading whitespace
    a := strip('  this is a string   ');  # a = 'this is a string'

    # uppercase it
    b := toUpper(a);  # b = 'THIS IS A STRING'

    # split it into words
    words := wsplit(b);  # b = ['THIS', 'IS', 'A', 'STRING']

    # obtain the ascii representation of a number in a given base (base 16, in
    # this case)
    c := radix(255, 16);  # c = 'ff';
```

Program Control (sys)

`crack.sys` contains symbols for interacting with the executable context. It mainly defines the `exit()` function and `argv` variable.

The `exit()` function can be used to terminate the program at any point with the given exit code:

```
    import crack.sys exit;
    # terminate with the normal code.
    exit(0);
```

The `argv` variable is a `StringArray` instance that allows you to access the command line arguments:

```
import crack.sys argv;

# write out all of the args except the program name (argv[0])
uint i = 1;
while (i < argv.count()) {
    cout `arg $i = $(argv[i++])\n`;
}
```

The `env` variable is a HashMap containing the environment variables. It can also be used to change environment variables:

```
import crack.sys env;
path := env['PATH'] + ':/opt/my/programs';
env['PATH'] = path;
```

Field Sets

Field sets allow you to associate arbitrary subsystem specific data with a context object. They are provided by the `crack.fieldset` module.

An example of when you would use a `FieldSet` is in an HTTP request handler chain. Let's say you have a set of request handlers that are called sequentially by your server. The first does authentication, the second does dispatch based on URL path, and the third finally handles the request. Both of the latter modules may need authentication information from the first, so a logical choice would be to add authentication information as an attribute of the request object. In this way, authenticators can fill it in and consumers can read it from a standardized place.

But there may be many authenticator implementations that all provide different forms of state information. As an HTTP framework implementor, you wouldn't want to define authentication state that is tightly coupled with your request object. One solution would be to add a request factory method to the HTTP server, allowing frameworks to customize their request objects, but that in turn makes it difficult to combine elements of different frameworks.

Field sets provide an elegant solution to this problem. `crack.http.HTTPRequest` implements the `FieldSet` interface, allowing arbitrary objects to be attached to it. So for example, for our authentication data we could define an `AuthData` class and use it as follows:

```
import crack.fieldset FieldSet;
@import crack.fieldset_ann fieldset_accessors;

class AuthData {
    String userId;
    oper init(String userId) : userId = userId {}

    # Macro defining the methods allowing AuthData to be added and removed
    # from a FieldSet.
    @fieldset_accessors(AuthData);
}

void authHandler(Request req) {
    ... do authentication ...

    # Store the auth data.
```

```
        AuthData(userId).putIn(req);
    }
```

Our application handler could then access it using the `get()` method:

```
    void appHandler(Request req) {
        auth := AuthData.get(req);

        if (!auth) {
            req.sendRedirect(303, LOGIN_URL);
        } else {
            ... serve page ...
        }
    }
```

The `Formatter` interface is also derived from `FieldSet`. For more information, see Using the Formatter FieldSet Interface.

Input/Output

The `crack.io` module contains crack's basic input/output interfaces. The `Reader` and `Writer` classes are the core interfaces for reading and writing data. These make use of the `Buffer` hierarchy and are implemented as interfaces so that they can be combined with each other and other classes.

The `crack.io` module defines three global variables: `cin`, `cout`, and `cerr`. These are wrappers around the system standard input, output and error streams. `cin` is a kind of `Reader` (an `FDReader`, to be precise) and `cout` and `cerr` are both `Formatters`. `Formatter`, in turn, implements `Writer`, so we can use these streams to illustrate the `Reader` and `Writer` interfaces.

`Writer` has one salient method called `write()`. This method writes a buffer to the underlying output object:

```
    cout.write('some data');
```

Since `String` is derived from `Buffer`, we can use the `write()` method to write a constant string. Classes implementing `Writer` must implement the `write()` method.

`Reader` provides two methods: an efficient method that reads into a `WriteBuffer` and a heavier form that reads a buffer of specified size and returns a string.

```
    ManagedBuffer buf = {1024};  # ManagedBuffer is derived from WriteBuffer
    bytesRead = cin.read(buf);  # read up to 1024 bytes into 'buf'
```

`read(WriteBuffer)` modifies the size of its input buffer and also returns the number of bytes read.

We could have done something similar with the heavy-weight form:

```
    String data = cin.read(1024);
```

This is more expensive than `read(WriteBuffer)` because it actually allocates a buffer and a `String` object.

We could implement the most basic functionality of the `cat` command like this:

```
    ManagedBuffer data = {4096};
    while (data.size = cin.read(data)) {
        cout.write(data);
        data.size = 4096;
    }
```

`cin`, `cout` and `cerr` are implemented using `FDReader` and `FDWriter`. These classes respectively implement `Reader` and `Writer` to read from and write to a file descriptor. You can obtain a file descriptor from any of the low-level IO functions on your system ("`open()`", for example).

The `crack.io` module also provides the `Formatter` class. See The Formatter Protocol for details on how to use this.

Threads

The `crack.threads` module provides high-level thread functionality. It includes the `Thread`, `Mutex`, `MutexLock`, `Condition` and `Queue` classes.

To create a thread, derive from the `Thread` base class and implement the `run()` method:

```
import crack.threads Thread;

class MyThread : Thread {
    void run() {
        # Note that cout, cin and cerr are threadsafe: their only state is
        # a file-descriptor.
        cout `hello from the background!\n`;
    }
}
```

We can then start use `start()` and `join()` to start the thread running in the background and wait for it to complete:

```
MyThread thread = {};
thread.start();

# block until completion
thread.join();
```

The simplest synchronization primitive is the `atomic_int` type. `atomic_int` is an integer (an `intz`, actually, because it is the size of a pointer) whose only operations are performed atomically. Atomic operations are visible to all threads and are not subject to reordering by the compiler. They are used to implement the reference counting mechanism in `Object`, and can also be used to implement other forms of thread-safe counters and control mechanisms. For example, here is a simple active thread counter:

```
atomic_int count;

class MyThread : Thread {
    void run() {
        count += 1;
        ... do thread actions ...
        count -= 1;
    }
}
```

The only operations supported for `atomic_int` are:

- Augmented assignment for addition and subtraction: `count += 1`
- Implicit conversion to `int`, `uint`, `intz` and `uintz`, `int64` and `uint64`.

Mutexes

Mutexes synchronize access to shared resources. Only one thread can have a mutex locked at any given time. If a thread tries to lock a mutex that is currently locked by another thread, it will block until the mutex becomes available.

The `Mutex` class provides the `lock()` and `unlock()` methods:

```
int value;
Mutex valueMutex = {};

valueMutex.lock();
value = 100;
valueMutex.unlock();
```

There is also a `MutexLock` class that manages locking and unlocking a mutex for the context it is defined:

```
void f() {
    # The mutex will be unlocked when this goes out of scope and is
    # deleted at the end of the function call.
    lock := MutexLock(valueMutex);
    value = 200;
}
```

Note that `MutexLock` is rather expensive, as it involves a memory allocation and free. However, it also has the virtue of being exception safe.

Conditions

Conditions allow you to send events between threads. They must be used in conjuntion with a `Mutex`. The mutex protects a shared resource, the condition notifies observers of changes in that resource:

```
# Create a condition associated with our value.
Condition cond = {valueMutex};

# Change the value.
if (true) {
    lock := MutexLock(valueMutex);
    value += 1;
    cond.signal();
}

# Thread that waits on changes.
class Waiter : Thread {
    void run() {
        lock := MutexLock(valueMutex);
        while (true) {
            # The condition atomically releases the mutex while it's
            # waiting for an event.
            cond.wait();

            cout `value changed, new value = $value\n`;
        }
    }
}
```

Queues

`Queue` is the most sophisticated inter-thread communication mechanism provided by `crack.threads`. As its name implies, it is a thread-safe message queue. It is also a generic.

Here's how we could use Queue to implement a worker pool:

```
    Queue[Functor0[void]] actions = {};
    class WorkerThread : Thread {
        void run() {
            func := actions.get();
            func();
        }
    }
    Array[WorkerThread] pool = {};
    for (int i = 0; i < 10; ++i) {
        thread := WorkerThread();
        pool.append(thread);
        thread.start();
    }

    void doSomething() {
        cout `blah`;
    }
    actions.add(Functor0[void](doSomething));
```

Line Readers

The `crack.io.readers` module provides a `LineReader` class that lets you read from a writer a line at a time:

```
    import crack.io.readers LineReader;

    reader := LineReader(cin);
    for (line :in reader)
        cout `got line: $line\n`;
```

Dealing with the Filesystem

The `crack.fs` module provides tools for working with the filesystem. It allows you to both manipulate filesystem meta-data and to do file IO.

Eventually, `crack.fs` will support a virtual filesystem allowing each program to have its own configurable view.

The primary interface of `crack.fs` is the `Path` class. Instances of `Path` correspond to entities in the filesystem, whether or not they currently exist.

By way of an example, to check for the existance of a file and write it if it doesn't exist, we could do something like this:

```
    import crack.fs makePath, Path;
    Path p = makePath('myfile.txt');
    if (!p.exists())
        p.writer().write('Sir, I exist!');
```

`makePath()` creates a new `Path` object from a path string in a way that is familiar to most programmers: the path string can be absolute or relative to the current working directory.

The more idiomatic way to do this would have been to use the "cwd" object:

```
    import crack.fs cwd, Path;

    Path p = cwd/'myfile.txt';
```

`Path` objects overload "oper /" to do path concatenation. We don't generally recommend overloading an operator in a way that provides different semantics from its normal meaning, but the forward slash is such an intuitive way to join paths together that we make an exception for it. It should be used with a `Path` instance on the left side and a string on the right.

The `cwd` object is a `Path` object representing the current directory. It can also be used to set the current directory with a special `set()` method:

```
    cwd.set('/home/weezer');
```

The string passed to `set()` can be either an absolute or relative directory. There is also an overload that accepts another `Path` object. The specified directory must exist - attempting to change to a non-existent directory will result in an exception.

In addition to file operations, `Path` objects can be used to create directories, remove files and directories and (as seen above with `exists()`) obtain status information.

Process

Crack provides a method for executing and controlling sub-processes and retrieving their output via the `Process` class, which is located in `crack.process`

You run a new process by constructing a Process class with the full command line to execute. If successful, you can retrieve the process ID with getPid().

Once running, you can poll() the process to find its status, or wait() for the process to finish execution. Once finished, you can call getReturnCode() for the final return code.

To retrieve stdout and stderr output from the process, use getStdOut() and getStdErr().

Alternately, you can use the `run()` method and pass the process a handler to handle standard input and standard error, or add the process and a handler to a `Poller` to multiplex it with other IO (see Networking below).

Regular Expressions

Crack provides a regular expression library in the form of `crack.regex`. You use it by creating a `Regex` object:

```
    import crack.io cin, cout;
    import crack.io.readers LineReader;
    import crack.regex Regex, Match;

    rx := Regex('(\\w+)=(.*)');
    String line = null;
    LineReader src = {cin};
    while (line = src.next()) {
        Match m = rx.search(line);
        if (m)
            cout `got $(m.group(1)) = $(m.group(2))\n`;
    }
```

The `Regex.search()` method returns a `Match` object if there was a matching substring and `null` if there wasn't. The `Match` object allows you to get the entire text of the matched substring, the start and end positions of it, and also allows you to get this information for parenthesized subgroups (employed in the example above).

We uses values of 1 and 2 to access the subgroups: the zeroth subgroup is the text of the entire expression.

To get the start and end position of the matched substring:

```
    uint startPos = m.begin();
    uint endPos = m.end();
```

Likewise, we can do this with subgroups:

```
    uint start1 = m.begin(1);
    uint end1 = m.end(1);
```

Note again that the zeroth subgroup (`m.begin(0)`) is equivalent to the entire group, so we start with group 1.

We can also use named subgroups. For example, we could have written our expression like this:

```
    rx := Regex('(?P<var>\\w+)=(?P<val>.*)');
```

And then the output statement would look like this:

```
    cout `got $(m.group('var')) = $(m.group('val'))\n`;
```

Likewise, the `begin()` and `end()` functions can be used with group names.

Named subgroups can greatly enhance the readability of regular expression code.

`crack.regex` is a wrapper around the PCRE library (Perl-compatible regular expressions). As the name suggests, the regular expression syntax supported by this library is very close to the syntax supported by Perl 5.

Math

The `crack.math` library provides access to functions and constants from the C standard math library. A description can be found on WikiPedia or a more complete reference at Dinkumware.

Each function has three versions: one overloaded for different floating point types, and one each for `float32` and `float64` values respectively with the precision postfix that can be passed to methods expecting an unambiguous function type. I.e. there are functions `sin`, `sin32` and `sin64`.

The library provides:

trigonometric functions

`sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`.

logarithmic/exponential functions

`exp`, `exp2`, `expm1`, `log`, `log10`, `log1p`, `log2`, `logb`, `modf`, `hypot`.

power and absolute value functions

`pow`, `sqrt`, `cbrt`, `fabs`, `hypot`.

gamma and error functions

`erf`, `erfc`, `lgamma`, `tgamma`,

integer functions

`ceil`, `floor`, `nearbyint`, `rint`, `round`, `trunc`, `nextafter`.

library error handling functions

> `errno`, `setErrno`, `testexcept`, `clearexcept`.

other functions

> `fdim`, `fmod`, `remainder`, `copysign`, `fpclassify`, `isfinite`, `isinf`, `isnan`, `isnormal`, `sign`, `atoi`, `atof`, `usecs`.

floating point classification and error condition constants

> `HUGE_VAL`, `INFINITY`, `NAN`, `FP_INFINITE`, `FP_NAN`, `FP_NORMAL`, `FP_SUBNORMAL`, `FP_ZERO`, `FP_ILOGB0`, `FP_ILOGBNAN`, `ALL_EXCEPT`, `INVALID`, `DIVBYZERO`, `OVERFLOW`, `UNDERFLOW`.

physical constants

> `E`, `LOG2E`, `LOG10E`, `LN2`, `LN10`, `LNPI`, `PI`, `PI_2`, `PI_4`, `PI1`, `PI2`, `SQRTPI2`, `SQRT2`, `SQRT3`, `SQRT1_2`, `SQRTPI`, `GAMMA`.

Below is a simple example:

```
import crack.math sqrt, pow, PI;

float length(float x, float y) {
    # x*x + y*y would perform better but would be less illustrative
    return sqrt(pow(x, 2) + pow(y, 2));
}

len := length(10, 5);
x = cos(30.0 / 180 * PI) * len;
y = sin(30.0 / 180 * PI) * len;
```

Networking

The `crack.net` module includes support for TCP/IP socket level programming.

A socket is a communication endpoint for TCP/IP communications. Sockets can either be connection channels, allowing you to send data to a matching socket elsewhere on the network, or listeners which wait for new connections.

We create a socket with the `makeSocket()` function:

```
import crack.net makeSocket, AF_INET, SOCK_STREAM;

s := makeSocket(AF_INET, SOCK_STREAM, 0);
```

This creates a "stream" (TCP) socket for a reliable connection over the INET protocol.

With a little more code, we can create a very basic server program:

```
import crack.net makeSocket, InetAddress, AF_INET, INADDR_ANY, SOCK_STREAM;
import crack.exp.error err;

srv := makeSocket(AF_INET, SOCK_STREAM, 0);

# allow the socket's port to be immediately reused after the program
# terminates (useful for debugging and server restarts)
srv.setReuseAddr(true);
```

```
    # bind to port 1900 on all interfaces
    if (!srv.bind(InetAddress(INADDR_ANY, 1900))) {
        cerr `error binding socket`;
        exit(1);
    }

    # queue up to 5 connections
    srv.listen(5);

    # accept loop
    while (true) {

        # accept a new client connnection
        accepted := srv.accept();
        if (!accepted) {
            cerr `Error accepting socket.\n`;
            continue;
        }

        # read up to 1K from the new connection
        data := accepted.sock.read(1024);

        # write it back to the new connection (we could have also used the
        # send() and recv() functions)
        accepted.sock.write(data);
    }
```

A client script might look like this:

```
    import crack.sys exit, strerror;
    import crack.net InetAddress, makeSocket, AF_INET, SOCK_STREAM;
    import crack.lang ManagedBuffer;
    import crack.io cout, cerr;

    s := makeSocket(AF_INET, SOCK_STREAM, 0);

    # connect to localhost (127.0.0.1)
    if (!s.connect(InetAddress(127, 0, 0, 1, 1900))) {
        cerr `error connecting: $(strerror())`;
        exit(1);
    }

    # this time, we'll use send and recv
    if (s.send('some data', 0) < 0) {
        cerr `error sending: $(strerror())`;
        exit(1);
    }

    # receive from the socket
    ManagedBuffer buf = {1024};
    int rc;
    if ((rc = s.recv(buf, 0)) < 0)
        cerr `error receiving: $(strerror())`;
    buf.size = uint(rc);

    cout `got $(String(buf))\n`;
```

When writing a server, you usually want to manage multiple connections. You can use the `Poller` class for this. `Poller` wraps the POSIX poll interface, which allows you to wait for an event on a set of `Pollable` objects. (`Socket` is derived from `Pollable`, so you can use this to manage sockets. We'll use `SocketApd` below, an appendage on `Socket` which adds some extra functionality.)

`Poller` allows you to add any number of `Pollables` and the events you care about on them. You can then wait for an event (or a timeout) using the `wait()` method. Here is an example of a simple server program (it just echos back whatever it reads from a client) written using `Poller`:

```
import crack.net makeSocket, InetAddress, Poller, PollEvent,
    PollEventCallback, SocketApd, AF_INET, INADDR_ANY, POLLIN, POLLOUT,
    POLLERR, SOCK_STREAM;
import crack.logger error;
import crack.sys exit;

@import crack.ann impl;

class ClientHandler @impl PollEventCallback {
    SocketApd sock;

    oper init(SocketApd sock) : sock = sock {}

    int oper call(Poller poller, PollEvent event) {
        if (event.revents & POLLIN) {
            data := sock.read(1024);
            if (!data) {
                poller.remove(sock);
                return 0;
            }

            # This won't make for a very robust server.  Sockets should
            # really be non-blocking and should keep a buffer and send as
            # much data as they can when ready to write (when the POLLOUT
            # flag is set).  But it's good enough for an example.
            sock.write(data);
        } else if (event.revents & POLLHUP) {
            poller.remove(sock);
        }

        return POLLIN | POLLERR;
    }

    oper del() {
        error `socket destroyed`;
    }
}

class ServerHandler @impl PollEventCallback {
    SocketApd sock;
    oper init(SocketApd sock) : sock = sock {}
    int oper call(Poller poller, PollEvent event) {
        if (event.revents & POLLIN) {
            accepted := sock.accept();
            poller.add(accepted.sock, ClientHandler(accepted.sock));
        }
        if (event.revents & POLLERR)
            error `Error on server socket.`;
        return POLLIN | POLLOUT
    }
}

srv := makeSocket(AF_INET, SOCK_STREAM, 0);

# allow the socket's port to be immediately reused after the program
# terminates (useful for debugging and server restarts)
srv.setReuseAddr(true);

# bind to port 1900 on all interfaces
if (!srv.bind(InetAddress(INADDR_ANY, 1900))) {
    error `Error binding socket `;
    exit(1);
}

# queue up to 5 connections
srv.listen(5);

Poller p = {};
p.add(srv, ServerHandler(srv));

# accept loop
while (p)
    p.waitAndProcess(null);
```

Serving HTTP

As of 0.13, Crack includes the `crack.http` modules containing functionality commonly used in an HTTP server. Functionality is broken out as follows:

`crack.http.core`

Core functionality, mainly the `Request` class.

`crack.http.server`

Contains the `Server` class, which implements an HTTP server.

`crack.http.util`

Utilities. Includes different kinds of request dispatchers, tools for parsing URLs and for dealing with query parameters,

The HTTP serving system is designed around handler chains. A handler chain is a sequence of handlers of type `Functor1[bool, Request]` that are called in sequence, and provided with the current `Request` object, until one of them returns true (or until the end of the chain). This approach is useful for defining default behavior and composing functionality. `Requesta` is a `FieldSet`, allowing handlers to attach arbitrary data to it. These concepts can be used to build sites and web applications out of loosely-coupled subsystems.

`http.auth` is one such system. It defines a handler (`AuthHandler`) to be chained in front of other handlers that require authentication. When invoked in the context of an HTTP request that includes a valid session cookie, it injects a `User` object into the `Request` instance. In the absence of a valid session cookie, it redirects the client to a login page which should be served by `LoginHandler`.

As a complete example, let's consider a simple web server with user authentication. (Note that this code can be found in the source tree as " `example/httpsrv.crk`".)

```
    import crack.http.auth AuthHandler, AuthInfoImpl, LoginHandler, StupidRNG,
        User;
    import crack.http.core HandlerFunc, Request;
    import crack.http.server Server;
    import crack.http.util Chain, MethodHandler, PathDispatcher, NOT_FOUND;
    import crack.io FStr;

    server := Server(8080);

    # Create a random number generator that the authInfo object will use for
    # generating session cookies.  We need to give it a 64 bit salt value and an
    # unsigned integer seed value.  You should use a better (more random) salt
    # value and a different seed value.
    rng := StupidRNG(uint64(i'123abczx') << 16 | i'y475oKjW', 123763);

    # Create our authentication database and give it a user.  (The password isn't
    # stored in plain text.  Instead we follow the standard practice of storing
    # the hash of the password and salt.)
    authInfo := AuthInfoImpl(rng);
    authInfo.addUser('user', 'password', 'salt');

    # Our "hello user!" request handler.
    bool hello(Request req) {

        # Get the User record for the request.  This gets stored by the AuthHandler.
        user := User.get(req);

        req.sendReply(200, 'text/plain', FStr() `Hello $(user.id)!`);
        return true;
    }

    # Our root page handler.  Let's just have it redirect to the "/hello"
    # entrypoint.
    bool root(Request req) {
        req.sendRedirect(303, '/hello');
        return true;
    }

    # Create a path dispatcher to route requests to the correct entrypoints.
    disp := PathDispatcher();

    # Create a "/hello" entrypoint, stick an AuthHandler in front of it.
    disp['hello'] = Chain![AuthHandler(authInfo), MethodHandler('GET', hello)];

    # Add the login handler.  The name "login" is special because "/login?url=" is
    # wired into AuthInfoImpl.  To change this entrypoint, you must change it
    # here where it is registered and you must also override getLoginURL() in
    # AuthInfoImpl.
    disp['login'] = LoginHandler(authInfo);

    # Add a handler for the root page.
    disp[''] = HandlerFunc(root);

    # Add the path dispatcher and the NOT_FOUND handler, which serves a 404 if
    # nothing before it terminates the request processing.
    server.addHandler(disp);
    server.addHandler(NOT_FOUND);

    # Run the server.
    server.run();
```

This illustrates a simple HTTP server with basic authentication and provides a rough layout as to how the library is distributed across the `core`, `server`, `util` and `auth` modules.

Please note that that HTTP interfaces in Crack are very immature. While the memory management features in Crack's lower level code ensure some degree of safety, we can make no claim as to the security of these systems. Use them at your discretion, and if you choose to use them in a public-facing service, be sure to put them behind some form of trustworthy reverse proxy.

Logging

The `crack.logger` module provides classes for logging messages with configurable formatting, severity level and output destination.

Basic Use

In many cases, it is adequate to simply log to standard error. The easiest way to do this is to simply import log-level formatters from the logger module and format to the default logger:

```
import crack.logger debug, error, fatal, info, setLogLevel, warn, DEBUG;

setLogLevel(DEBUG);  # Log debug messages and above.
info `This is an info message.`;
error `Something bad happened!`;
```

In this example, we set the "log level" to DEBUG so that debug messages will be written. The default log level is ERROR, so without this statement we would have only seen the error message, not the info message.

Be aware that the default logger is not thread-safe.

Core functionality

The core of the module is the `Logger` class which implements 5 logging severity levels: `FATAL`, `ERROR`, `WARN`, `INFO`, `DEBUG`. Only messages with a severity level higher than the minimum for the `Logger` object will output.

An example looks like this:

```
import crack.logger Logger, ERROR, DEBUG;
import crack.io cerr;

logger := Logger(cerr, ERROR);
logger.log(DEBUG, "This will not be logged");
logger.log(ERROR, "This will be logged");
```

The output will look something like this:

```
2012-03-24T15:40:35 [ERROR] This will be logged
```

The first argument of the constructor can be any `Formatter` object. Other convenient constructors are:

```
Logger(String filename, uint level);
Logger(Writer w, uint level);
```

The `Logger` class defines convenience methods `fatal`, `error`, `warn`, `info`, `debug` to log messages with the eponymous severity levels:

```
logger.debug("This will not be logged");
logger.error("This will be logged");
```

Formatter protocol

The `logger` module also provides a `LogFormatter` class that implements the `Formatter` protocol. Extending the example above to use a `LogFormatter` instead, we get:

```
import crack.logger LogFormatter;

debug := LogFormatter(l, DEBUG);
error := LogFormatter(l, ERROR);

debug `This message will not be logged`;
error `This message will be logged`;
```

That is, a `LogFormatter` instance always logs messages with the given severity level. The `logger` module defines the formatters `fatal`, `error`, `warn`, `info`, `debug` that logs messages to `cerr` using the default log format.

The `LogFormatter` class also implements the convenience constructors `LogFormatter(String filename, uint level)`, `LogFormatter(File file, uint level)`, and `LogFormatter(Writer w, uint level)`. Using these, the severity level of the `Logger` and the `LogFormatter` is the same. To change the log level of the undelying `Logger` instance, the method `setLoggerLevel(uint level)` can be used. The severity level of a `LogFormatter` instance can be changed using the `setLevel(uint level)` method.

The `crack.logger` module provides a set of `LogFormatter` global variables for the default logger to simplify logging at different severities. As you might expect, they are "debug", "info", "warn", "error" and "fatal".

Message formatting

The `Logger` class implements log message fields, named `msg`, `datetime`, `progname`, `escapemsg`, and `severity` that allow you to specify the format of a message written to the logfile. Every logger has a sequence of message fields defining the format used for the logger. The default fields are `datetime`, `severity`, `msg`, leading to the output shown in the example above.

The fields can be specified using the constructors `Logger(Formatter fmt, uint level, Array[String] fieldNames)` or using a space delimited field specification `Logger(Formatter fmt, uint level, String fieldSpec)`, as follows:

```
l := Logger(cerr, FATAL, "datetime severity progname escapemsg");
```

The `progname` field is simply the value of `argv[0]`, while `escapemsg` is the log message with special characters represented using the escape sequences produced by the `ascii.escape` module.

The format of the `datetime` field is determined by a string in `strftime`, format, with the default being `"%Y-%m-%dT%H:%M:%S"`, as defined by ISO-8601. This format is only able to produce timestamps with second accuracy, as it relies on the `crack.time.Date` class.

Message fields

Field representations are constructed by `MessageField` instances. To add a new type of message field, define a class as follows:

```
import crack.runtime usecs;

MilliSecField : MessageField {
    oper init(LoggerOptions options): MessageField(options) {}

    void format(uint level, String msg) {
        options.fmt.format(usecs() / 1000);
    }
}
```

The field can be registered with the `Logger`, and the fields specified with:

```
logger.setNamedField("msecs", MilliSecField(logger.options));
logger.setNamedFields("datetime msecs severity progname escapemsg");
```

Logger options

The `Logger` class contains an `options` member of type `LoggerOptions`, with members, defined as:

```
    class LoggerOptions {
        uint seq;            // Message sequence number
        String timeFormat;   // strftime date format
        String sep, eol;     // Field separator and end of line
        Formatter fmt;       // Output formatter
        uint level;          // Logging level

        oper init() {}
    }
```

These are typically passed to a MessageField object during construction and can be used to vary the formatting of the message field.

GTK

`crack.gtk` is a very minimal GTK module. It currently provides classes for the following GTK objects:

`Toplevel`

A top-level (window manager level) window.

`Tooltips`

Used to tie fly-over help text to widgets.

`Button`

A pushbutton widget.

`Entry`

A single line text entryfield.

`HBox`

A container that arranges child widgets horizontally in a row.

`VBox`

A container that arranges child widgets vertically in a column.

`App`

Models an application with an init function and a main loop.

Here's a sample program:

```
    import crack.exp.gtk App, Button, Entry, Handlers, HBox, Label, Toplevel,
        VBox;
    import crack.sys argv;
    import crack.io cout;

    App app;

    # this is going to be our window.
    class MyWindow : Toplevel {

        # create a bunch of child widgets.
        Label lbl = {'Your Name:'};
        Entry dataEnt;
        Button doneBtn = {'Close'};
        VBox v1 = {false, 10};
        HBox h1 = {false, 10};

        # print out all of the data collected by the window.
        void getData() {
            cout `Entry text: $(dataEnt.getText())\n`;
        }

        # handler for the button class - print out the data and quit.
        class DoneBtnHandler : Handlers {
            # note that this creates a reference cycle - we need to do
            # something to break the cycle if we care about memory leaks.
            MyWindow win = null;
            oper init(MyWindow win0) : win = win0 {}
            bool onClicked() {
                win.getData();
                app.quit();
                return false;
            }
        }

        oper init() {
            # arrange all of the widgets
            add(v1);
            v1.add(h1);
            h1.add(lbl);
            h1.add(dataEnt);
            v1.add(doneBtn);

            # set the done handler.
            doneBtn.setHandlers(DoneBtnHandler(this));
            doneBtn.handleClicked();

            # show all of the widgets
            lbl.show();
            dataEnt.show();
            doneBtn.show();
            v1.show();
            h1.show();
            show()
        }
    }

    app.init(argv);
    MyWindow win = {};
    app.main();
```

Crack's GTK libraries are really a toy implementation or proof of concept. You can probably implement minimal programs with them, you can implement larger programs if you care to augment the library.

Command Processing

Crack provides a command line option library similar to Java's jcommander or python's argparse. It is still experimental, and therefore the interface may be subject to change.

A command is defined as a function with a special annotation:

```
import crack.cont.array Array;
import crack.cont.treemap TreeMap;
import crack.exp.cmd ArgInfo, ArgInfoBase, ArgParser, ArgVal,
    CommandArgParser, CommandInterface, ConverterBase, ValueMap;
import crack.io Formatter;
import crack.strutil StringArray;

@import crack.ann impl;
@import crack.exp.cmd.ann command;

@command int main(String arg, @flag('-f') bool foo) {
    cout `first argument is $arg\n`;
    cout `flag value is $foo\n`;
}

exit(main(args.subarray(1))());
```

Positional arguments are simply defined as normal arguments. Flags are defined using the @flag annotation. (This isn't a real annotation, it receives special treatment by the `@command` annotation. It should not be imported).

The `@command` annotation parses what looks like a function, but it generates a class. When we call the function, we generate an instance of the class which is populated by the array of arguments passed into it,

The function that we defined ends up being the "`oper call()`" method in this class, so we actually do the processing by calling the instance. The argument list that we pass in gets translated to attributes of the class.

Arguments are converted to values based on their types. The command processing code defines built-in converters for Strings, integers and booleans, as well as lists of these (for use across Multiple Arguments).

Flags

Flags are defined by adding the `@flag` pseudo-annotation before an argument. `@flag` has an argument list consisting of the flag names that are associated with that argument. These must be literal strings (variables and non-literal constants are not allowed).

A flag name should begin with either a single or a double hyphen: the single hyphen is used for single character flag names ("short names"), the double hyphen is used for "long names." For example, to have a flag associated with a logfile, we might do this:

```
@command int main(@flag('-l', '--logfile') String logfile) {
    ...
}
```

All flags have an associated value except boolean flags. Short flag names can have the value passed either immediately following the flag name or as the next command line argument:

```
$ command -llogfile.txt     # this works
$ command -l logfile.txt    # so does this.
```

Likewise, long flag arguments can also be specified as part of the flag (separated by an equal sign) or as the next argument:

```
$ command --logfile=logfile.txt
# command --logfile logfile.txt
```

Boolean flags do not require an argument. Short boolean flags *may not* have an argument - their presence indicates "true", their absence indicates "false". Long flags may be negated by either prefixing the flag name with "--no-" or by adding "=false" to the end of the flag ("=true" is also supported). Long binary flags do not support the use of the value as the next argument.

```
$ command --happy        # Sets "happy" to true.
$ command --no-happy     # Sets "happy" to false.
$ command --happy=false  # So does this.
$ command --happy false  # Sets happy to true, "false" is the next
                         # positional argument!
```

Subcommands

You can create "subcommands" by defining a command with a parent. For example, to define the "get" and "put" subcommands, we can do this:

```
import crack.exp.cmd SubcommandArgInfo;

@command int main(@flag('--global-flag') String globalFlag,
                  @subcmd
                  ) {
}

@command(parent = main) int get(String arg) {
    cout `get called with $arg.  Global flag = $(parent.globalFlag)`;
}

@command(parent = main) int put(String arg) {
    cout `put called with $arg.  Global flag = $(parent.globalFlag)`;
}
```

The main command has a special pseudo-argument, "`@subcmd`", which indicates the position of the subcommand.

The subcommands must each contain a parameter ("`parent = main`") indicating the parent command.

We can now call the commands as follows:

```
$ command --global-flag=blech get foo
get called with foo.  Global flag = blech
$ command put bar --global-flag=barch
put called with bar.  Global flag = barch
```

Subcommaneds can have flags and arguments of their own. From within the subcommand's function, the parent command's attributes are accessible through the "parent" object (see the use of `globalFlag` in the example above. To use a name other than "parent" for this, you can specify "`parentAttr = `*`name`*" in the `@command` parameters.

Optional Arguments

Optional arguments are specified using a default value syntax. For example:

```
@command int foo(String optional = 'value') {
    ...
}
```

Optonal arguments must be single token literals. Most flags should generally be defined using a default value (the alternative is that they be required).

Positional optional arguments must be specified at the end of the argument list. You can not mix optional positional arguments with multiple arguments.

Multiple Arguments

The "`@multi`" pseudo-annotation is used to define arguments that may be specified multiple times. These should be used with an Array of the basic type (in truth, any container with default or element count constructor and an `append()` method which adds an element can be used, but it is necessary to define a converter tor the type, see Defining Converters below):

```
@command int main(@multi Array[String] words) {
    ...
}
```

The command definition above consumes all (non-flag) arguments and puts them in 'words'.

You can do the same thing with flags:

```
@command int main(@multi @flag('-f') Array[bool] vals) {
    ...
}
```

Defining Converters

There are a canned set of types that can be used for command arguments, currently these are `String`, `bool`, `int` and arrays of these (`crack.cont.array.Array`, not the built-in `array` type).

However, it is often desirable to pass more elaborate types, such as dates, or to enforce constraints on string types (such as that they be file names, or members of a set of canned values).

The canned argument types are defined as *converters*, and user-defined converters may be provided to extend (or override) this set.

Defining a converter is usually a simple matter of defining a function to convert from a string to the required type. For example, let's say we want an argument that passes colors by name:

```
@struct Color {
    int red, green, blue;
}

Color convertColor(String name) {
    if (toLower(name) == 'red')
        return Color(0xff, 0, 0);
    else if (toLower(name) == 'green')
        return Color(0, 0xff, 0);
    else if (tolower(name) == 'blue')
        return Color(0, 0, 0xff);
    else
        # Default to black.
        return Color(0, 0, 0);
}
```

Now we create an array of converters to give to our command:

```
import crack.exp.cmd Converter;

converters := Array[ConverterBase]![Converter[Color].Wrap(convertColor)];
```

And finally, register it in the command annotation:

```
@command(converters = converters) int main(Color color) {
    ...
}
```

Note that converters are not inherited from parent commands: the converter argument must be passed to every command definition where the extended types are desired.

Also note that converters for the existing types override the existing definitions.

Multiple argument types must have their own converters. For this, use the `MultiArgConverterImpl` generic paramterized by the collection type and the element type. To define converters for the Color type above:

```
import crack.exp.cmd Converter, MultiArgConverterImpl;

converters := Array[ConverterBase]![
    Converter[Color].Wrap(convertColor),
    MultiArgConverterImpl[Array[Color], Color]
];
```

The MultiArgConverterImpl requires that the container implement the `append()` method for the element type. For containers that do not provide an append, you can provide your own implementation derived from `MultiArgConverter[T]` (where `T` is the element type).

Documentation Generation

The `@command` annotation does special handling of doc-comments, and you can use the `addHelpCommand()` function to add a special "help" subcommand:

```
import crack.exp.cmd.doc addHelpCommand;

## This is a command.
@command void main(
    ## The "foo" flag.
    @flag('-f') int foo = -1,
    @subcmd
) {
}

addHelpCommand(main_subcmds, main);
```

After this, running "`command help`" will produce a help message with the command documentation, including documentation on flags and subcommands. Running "`command help` *`subcommand`*" will provide help on *subcommand*.

The Legacy `crack.cmdline` Module

The `crack.cmdline` module also allows you to do options processing. It is more work to use than `crack.exp.cmd`, but has the advantage of not currently being in experimental.

```
import crack.sys argv;
import crack.exp.cmdline CmdOptions, CMD_BOOL;

CmdOptions opts = {};

options.add('help', 'h', 'Show help text', 'f', CMD_BOOL);
options.add('port', 'p', 'Port number', '', CMD_INT);

options.parse(argv);

if (options.getBool('help'))
    showUsage();
# get the port number
port := options.getInt('port');
```

There are numerous other modes of interaction with this module, see the module itself and the test script for details.

Appendix 2 - Migration

Crack follows Semantic Versioning (see semver.org). This means that all versions of Crack with a major version number of "1" will be backwards compatible until the major version number changes. See Compatibility for caveats and details.

The remainder of this appendix describes the things you need to know when upgrading between versions.

From 1.1 through to 1.5

No known changes affecting compatibility.

From 1.0 to 1.1

- Though intended to be backwards compatible, we did introduce a fairly minor incompatibility: the `crack.compiler.Location` class adds the `getStartCol()` and `getEndCol()` methods. This was judged to be acceptable because, even though user code could conceivably derive from `Location` and implement conflicting methods, there is no purpose in doing so because instances of the class can not be constructed by the user.

From 0.13 to 1.0

- Reference counts are now incremented for function arguments and method receivers borrowed from global or instance variables.

From 0.12 to 0.13

- `crack.net.httpsrv` has been removed and replaced with the `crack.http` modules (see Serving HTTP).
- The fuse module callbacks now require 64-bit file offsets (change these to `uint64`).
- Buffer sizes are now of type `uintz` instead of `uint`.

From 0.11 to 0.12

- Many methods that were unnecssarily virtual are now @final.
- The deprecated "createString()" and "createCString()" methods in StringWriter and StringFormatter have been removed.
- The "do" keyword has been reserved and can no longer be used for variables.
- the crack.exp.error module has been removed.

- The `split()` functions in `crack.strutil` now treat adjacent delimiters as independent separators (e.g. "split('foo..bar', b'.') -> ['foo', '', 'bar']" instead of ['foo', 'bar']). The `tsplit()` functions preserve the original behavior.
- In The Formatter Protocol, The return value of the `enter()` method was previously ignored, but is now used as the formatter object for format() methods and leave().
- Buffer size is now defined as uintz. All other indexes and sizes to strings and buffers are uintz or intz.
- `crack.net.httpsrv` has been broken out into a set of modules under `crack.http`.

From 0.10 to 0.11

- The low-level alsa midi functions have been regenerated to feel more object-oriented (the high-level crack wrappers remain backwards compatible).
- The name of the "EACCES" constant has been fixed (it was "EACCESS", which is not what it is in POSIX).

From 0.9 to 0.10

- crack.fs.Path.makeDir() has been converted to a final function, makeDir(bool errorOnExists) is now the abstract that must be overriden.

From 0.8 to 0.9

- The `URLEncode` and `URLDecode` functions in `crack.net.curl` have been renamed to `urlEncode` and `urlDecode`.
- Changed the protected variable interface to MersenneTwister in `crack.random`.
- Cache file format has changed, if experimenting with the cache you'll need to purge your existing cache files from ~/.crack/cache.

From 0.7 to 0.8

- Renamed `crack.cont.numericarray.NumericArray` to `Matrix`.
- Moved the single parameter overload of `strutil.split()` to `crack.ascii` and renamed it `wsplit()`.
- Changed the semantics of annotations in generics, see Annotations in Generics for details.

From 0.6 to 0.7

- Object writing is now done using a Formatter instead of a Writer. All objects now have a formatTo() method.
- WriteBuffers now have a 'cap' field indicating capacity. AppendBuffer's 'pos' field has been eliminated and the 'size' field now serves this function. Reader.read() is now expected to set the 'size' field of the WriteBuffer it is passed.

- `StaticString` is now derived from `CString`.
- `crack.net.Pollable` has been removed and replaced with `crack.io.FileHandle`.
- Removed `SOCK_NONBLOCK` and `SOCK_CLOEXEC` for compatibility reasons. Use `fcntl(int fd, int flag)` with `O_NONBLOCK` instead. `FD_CLOEXEC` could be used as a flag where it is supported. It is non-portable, however, and is not defined in the `runtime` module.
- Added precision postfix (32 and 64) math functions. Overloaded non-postfix versions are still there.
- Renamed `TreeMap`, `HashMap`, and `OrderedHashMap` `contains()` method to `hasKey()`.
- Renamed `OrderedHashMap.getIndex()` method to `getItemAt()`.
- The `now()` function in the time module is now a static method of the `Time` class.

From 0.5 to 0.6

- Removed `crack.container` module (use `crack.cont.*` instead).
- Removed `crack.lang.MixIn` (use Interfaces instead).
- The extgen, readers, gtk, fs, ann, cmdline, serial, regex
- The implementation of instance variables in extensions was changed to make them more portable. Instead of trying to create a mapping of a native class, the interface now requires an offset to the instance variable.
- Access protection now works: symbols beginning with a single underscore can only be accessed from the module they are defined in, symbols beginning with a double underscore are private to their class.
- Second-order imports are now disallowed - to import a symbol from a module, the symbol must either be defined in the module or explicitly exposed with an @export_symbols annotation.

From 0.4 to 0.5

- the `toBool()` converter has been replaced with `oper to bool()`
- Macro style generic containers are deprecated: use real generics instead. (This currently just triggers a warning, but macro-generics will be removed by 1.0).
- crack.exp.dir `Directory.dirIter()` has been replaced with `Directory.dirs()`, and `Directory.fileIter()` with `Directory.files()` which now return `List[Directory].Iter` and `List[FileInfo].Iter` objects, respectively.
- `DList.last()` was renamed `DList.prev()`

From 0.3 to 0.4

- The `enter()` and `leave()` methods are now called from a string interpolation expression if they exist. This could present a problem if a formatter had them defined for some other purpose.

From 0.2 to 0.3

- The following words are now keywords, and no longer available for use as identifiers: `in` `on` `for`
- The semantics of aggregate variable definitions without an explicit initializer have changed. Uninitialized variables use to initialize to an object created with the default constructor, they now initialize to null. So for example in this code:

  ```
  class A {}
  A x;
  ```

  `x` would have been assigned to an instance of `A` in version 0.2, in version 0.3 it would be assigned to null.

  The -m flag can be used to help you identify instances where your code is likely to be confused by this. It generates a warning for places where you are using default initialization (presumably expecting default initialization) and also where you are assigning the variable to null (since it is now the default and does not need to be specified explicitly).

  We recommend that you run all code that you are upgrading with this option and audit all sites where one of these warnings are triggered.

Compatibilty

Crack promises to be backwards compatible across major versions starting at 1.0. So all 1.x versions of the language and its core libraries should be compatible with earlier 1.x versions. We define compatibility as follows:

Code that functions without error in one version will function without error in a compatible version.

There are some caveats to this rule:

- The crack ABI is still a work-in-progress. We make no guarantees about the compatilbility of object files or IR code produced using one version of crack with those produced by a compatible version of the executor.
- We may modify error or logging output. Programs should not rely on the text of error or logging messages.
- We may subclass exceptions. Programs should not rely on constructs like `ex.class is Exception` for establishing the type of an exception, use something like `ex.isa(Exception)` instead.
- Constructs in the language or libraries that produce an error may not produce an error in a compatible version. Programs should not rely on language syntax or runtime error conditions being "locked down."
- The `buffer` and `size` variables of the `Buffer` class and all derived classes (including `String`) may be treated as read-only in future, compatible versions of the language. Classes derived from `WriteBuffer` are exempt from this.

Appendix 3 - Library Conventions

The standard library uses the following naming conventions:

- Module names are all lower-case and may contain underscores: `my_module`
- Class names are capitalized camel-case: `ClassName`

- Method and function names are uncapitalized camel-case: `doSomething()`
- Instance variables and module scoped variables are uncapitalized camel-case: `myInstVar`
- Constants are all upper-case and may contain underscores: `MY_CONST`
- Annotations are all lower-case and may contain underscores: `an_annotation`.

As the language dictates, module-protected members begin with underscores.

The following rules apply to method and function names (we use the word "function" in the following context to refer to both function and methods):

- They are usually verbs.
- Methods that convert an object to a new representation of a different type are of the form "to*ClassName*". Example: `toString()`
- Functions that always create a new object should be of the form "make*Noun*". In situations where the noun is unambigously a noun, never a verb, the method may have an alias that is simply the noun. Examples:

  ```
  class Path {
      Writer makeWriter() ...
      Writer writer() ...
  }
  ```

- the "get" prefix is used for methods that return a contained object (although it may be lazily created at the time of the first call to the "get" method). The "set" prefix is used to indicate a method that sets the corresponding attribute. The most common case of these is attribute access, for example:

  ```
  class Person {
      String getName() ...
      void setName(String name) ...
  }
  ```

  These may also be used for access of parameterized values, as in "`getRelative(COUSIN)`" or simply `get(Key key)` as in the case of Map containers.

Appendix 4 - Notes on Caching

The Crack JIT stores meta-data and compiled LLVM bitcode in a cache in the user's home directory. Keeping this information cached significantly improves the load time of Crack programs. While this feature has been in use for quite a while and has no known bugs, a bug in the caching mechanism can produce some very confusing results. Different behavior can manifest depending on whether code is actually being compiled (for the first time or due to a change in the source code) or simply loaded from the cache.

Cached files are stored in the user's $HOME/.crack/cache directory and caching is controlled by the -C, -K, and --allow-sourceless options, the "cachePath" builder option and the CRACK_CACHING environment variable.

Caching can be disabled entirely by providing the -K command line option or by setting the "CRACK_CACHING" environment variable to false. An alternate cache path can be specified by adding a builder option "-b cachePath=*path*" in JIT mode (caching is not used in AOT mode).

If you suspect that you have discovered a bug in caching, try moving $HOME/.crack/cache to $HOME/.crack/cache.org and run your program again. If it succeeds the second time, then you have most likely discovered a caching bug and we'd greatly appreciate it if you'd report this, ideally including the cache files and source files that manifested it.