

Cosmium Programming Language Specification

Retcinder

Version 0.0.0-rc0, 2026-10-12: conceptual designing phase

Table of Contents

Why a new language?	1
Why not (insert language here)?	1
Goals	1
Specification information	2
The language	3
Changes from C	3
Types	3
Classes and Interfaces	4
Enums	5
Namespaces	6
Generics	7
Operator Overloading	9
The standard library	13
string.cosh	13
vector.cosh	14

Why a new language?

C++, as a language, is poorly constructed.

As a few examples...

- Every new release comes with a huge swath of features that don't mesh with each other, resulting in a bloated non-cohesive language.
- Several aspects of the language feel pointless or a detriment. As a few examples:
 - Exceptions are a poor way to handle errors
 - Modules serve little benefit and worsen subsequent build times.
- Several aspects of C++ are undesirable
 - Poor compile-time error messages, especially when you get into classes and generics.

As a result, I felt the need to make my own language, serving as a rethinking of C++.

Why not (insert language here)?

Other languages deviate from C/C++ in ways I do not like. Usually with:

- Alternative memory management handling (e.g. borrow checker, garbage collection)
- Ship a package manager with the language
 - I prefer languages without them, so that using a system package manager or a high quality community one is preferred.
- Do away with C-syntax altogether.
- Utilise module imports instead of headers.
- Weird arbitrary changes (e.g. `main` returning `void` instead of `int`)

Some languages are also too willing to permit LLM contributions to their critical components, which I believe should be avoided completely due to ethical concerns regarding actions by major LLM companies.

These are not things I want in the programming languages I use.

Goals

Make a language that:

- Doesn't overcomplicate itself, keeping the stdlib to a good amount of features that fit.
- Keep features feeling natural (i.e. not introduce weird features that don't fit)
- Improves on C++ whenever possible.
- Improves on C when possible while avoiding breakage in `extern "C"` blocks.

Specification information

This specification will define syntax using EBNF (Extended Backus Normal Form (also known as Extended Backus-Naur Form)) as defined by ISO/IEC 14977.

We define the following basic rules that will be shared across all EBNF syntax segments:

```
paren_begin = "(" ;
paren_end   = ")" ;
square_begin = "[" ;
square_end   = "]" ;
curly_begin  = "{" ;
curly_end    = "}" ;

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
        | "H" | "I" | "J" | "K" | "L" | "M" | "N"
        | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
        | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
        | "c" | "d" | "e" | "f" | "g" | "h" | "i"
        | "j" | "k" | "l" | "m" | "n" | "o" | "p"
        | "q" | "r" | "s" | "t" | "u" | "v" | "w"
        | "x" | "y" | "z" ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;

identifier = letter, (letter | digit | "_")* ;

whitespace = " " ;

req_whitespace = ( whitespace )* ;

opt_whitespace = ( whitespace )* | ε ;
```

The language

Changes from C

Additions

- There is now the `and` & `or` keywords, acting identically to `&&` and `||`.
 - This was added to make it less confusing when you're dealing with operations using `&` or `|`, along with being easier to read.
 - This is preferred in Cosmium source code and should be used in learning materials.
- `null` keyword (disabled within `extern "C"` blocks).
 - Exactly the same as `NULL` and is preferred in Cosmium source code.
 - Use `nullptr` for pointers, not `null`.

Deprecations

- `wchar_t`
 - Use `char/char8_t` for UTF-8, `char16_t` for UTF-16 and `char32_t` for UTF-32.
 - Deprecated due to its implementation defined in such a way where it differs in size between platforms rather than architecture.
 - See Windows `sizeof(wchar_t)` vs Linux `sizeof(wchar_t)` for why.

Types

Numerous basic types are provided. These are available regardless of whether you have access to the standard library or not.

Types

- `iN` (signed N bits where N can be between 2 and 65535 (inclusive))
- `uN` (ditto of the above but unsigned and can be as small as 1 bit)
- `int` (signed 32 bits on 32-bit and 64-bit devices, signed 16 bits on 16-bit devices)
- `bool` (u1, auto-convertible into C's `_Bool`)
- `char` (signed byte, regardless of platform)
- `char8` (ditto)
- `char16` (UTF-16 char)
- `char32` (UTF-32 char)
- `uchar/uchar8/uchar16/uchar32` (ditto of the 4 above but unsigned)
- `fN` (N-bit floating point number, where N is any number between 16 and 65535)
- `float` (f32 equivalent)

- double (f64 equivalent)
- bfloat16 ([16-bit alternative float format](#))

Classes and Interfaces

Classes and interfaces are a way to create objects with functionality attached.

Interfaces may be implemented fully but is generally advised to only be either partially or fully unimplemented.

Classes must be fully implemented.

You can use the **implements** keyword to make a class implement an interface and you can use the **extends** keyword to extend an existing class.

Classes may extend one class and implement as many interfaces as is desired.

Interfaces can extend other interfaces to require implementing those other interfaces.

IMPORTANT

There is currently no way to have a constructor return an error. You will need to handle this manually.

This is something desired for the language and should be available in a future revision.

Syntax

todo

Sample usage (using [SDL3](#))

```
struct Vector2i {
    int32_t x;
    int32_t y;
};

interface IWindow {
    bool maximise() unimplemented;

    bool setVSyncState() unimplemented;

    bool setWindowTitle(std.string) unimplemented;

    std.string getWindowTitle() unimplemented;
};

class SDLWindow implements IWindow {
    constructor(std.string title, Vector2i size, uint64_t flags) {
        win = SDL_CreateWindow(title.c_str(), size.x, size.y, flags);
    }

    destructor() {
        SDL_DestroyWindow(win);
    }
}
```

```

    }

    bool maximise() {
        return SDL_MaximiseWindow(win);
    }

    bool setWindowTitle(std.string newTitle) {
        return SDL_SetWindowTitle(newTitle);
    }

    std.string getWindowTitle() {
        return std.string { SDL_GetWindowTitle(win) };
    }

protected:
    SDL_Window* win;
};

class SDLWindowWithRenderer extends SDLWindow {
    // Ommitting contents for brevity.
protected:
    SDL_Renderer* renderer;
};

```

CAUTION

This example code does not do error checking at all. You may experience unhelpful arbitrary crashes.

NOTE

determine whether **implements INTERFACE for CLASS** (implementing custom functionality in external classes) is a good idea or not.

Enums

Enums are a named set of key-value pairs. You can optionally specify a type for its maximum size (must be an integer type).

Note that unlike C, enums are not namespace-polluting by default. If you need that behaviour, prefix the enum with **extern "C"** or contain it within an **extern "C"** block.

Syntax

```
todo
```

Sample usage

```

enum Foo {
    Bar,
    Baz
};

```

```
extern "C" enum EggColor {
    EGG_STANDARD,
    EGG_GREEN
};

enum Foobert : uint8_t {
    Barbert,
    Bazbert
};
```

Enum Unions

Enum Unions are an Algebraic Data Type (ADT) where the key in the enum is used to change which struct is being referenced.

Enum Union Sample

```
enum union Foo {
    Bar { uint16_t x; uint32_t y; },
    Baz { uint64_t z; std::string err; }
};

void printFoo(const Foo inFoo) {
    switch (inFoo) {
        case Bar: {
            printf("inFoo is Bar: %d %d\n", inFoo.x, inFoo.y);
        } break;
        case Baz: {
            printf("inFoo is baz: %d \"%s\"\n", inFoo.z, inFoo.err);
        } break;
    }
}

int main(void) {
    Foo bfoo = Foo::Bar { 12, 30 };
    Foo zfoo = Foo::Baz { UINT64_MAX, "No err"s };

    printFoo(bfoo);
    printFoo(zfoo);
}
```

Namespaces

Namespaces are a way to separate functions, classes and features into separate "modules". You access namespaces using the `.` symbol (e.g. `foo.bar()`).

You can use the contents of another namespace in your current namespace by "using" it with `using namespace NAMESPACE`.

NOTE `using namespace NAMESPACE;` is forbidden in header files.

NOTE You should generally avoid using `using namespace std;` in specific as it is bad practice.

Syntax

```
namespace_define_begin = namespace_keyword, req_whitespace, identifier, ( ".",  
identifier )*, opt_whitespace, curly_begin ;  
namespace_define_end = curly_end, ";" ;  
  
namespace_using = using_keyword, req_whitespace, namespace_keyword, req_whitespace,  
identifier, ( ".", identifier )*, opt_whitespace, ";" ;  
  
namespace_keyword = "namespace" ;  
using_keyword = "using" ;
```

Sample usage

```
namespace foo {  
    void bar() { printf("Baz!\n"); }  
};  
  
int main(void) {  
    foo.bar();  
  
    using namespace foo;  
  
    bar();  
  
    return 0;  
}
```

Generics

Generics are a way to allow a class or function to take in multiple types.

You can specify a required interface or constraint before the generic typename for extra checks.

If you pass in an object that cannot be used (either due to invalid function calls or being incompatible with the generic's definition), compiling will fail.

Syntax

todo

Sample usage

```
void print<T>(T input) {
```

```

    puts(std.toString(input));
}

void renderObject<IRenderable T>(T obj) {
    if (!obj.visible)
        return;

    obj.draw();

    if (gDebug)
        obj.drawDebug();
}

class ResourceManager<T> {
    constructor();

    bool createResource(std.string filepath);

    T getResource(std.string id);

private:
    std.vector<T> arr {};
};

// Use multiple generics
class Result<Value, Error> {
    constructor();
    constructor(Value v);
    constructor(Error e);

    bool hasValue();
    bool hasError();

    Value getValue();
    Error getError();

private:
    Value value;
    Error error;
}

```

Constraints

Constraints are a way to limit the usage of generics to only be valid types. These are compile time checks that cover implementations and whether certain statements can compile.

You can also alias functions within the constraint, in the event of name collision.

todo*Sample usage*

```

constraint CHashable {
    require implements IStringable;

    rule std.hash<T>(input) -> std.convertTo<size_t>;

    alias str = IStringable::toString;
};

// Use as you would for an interface generic
void printHash<CHashable T>(T in) {
    printf("%s", in.str.cStr());
}

```

Operator Overloading

COMMENT

I am not proud of this, I might remove some or all of this in the future.

Operator Overloading is the means by which you customise what operators (+-*/&|<=> and == << >> [] () →) do, either for a class or struct).

You can also have multiple overloads for the same operator, provided they have different signatures.

It is highly discouraged to call functions within an overloaded operator unless it is absolutely necessary.

Arguments (lhs is used to determine what type of object this operator is for)

Symbo l	FunctionName	Out of Class	In Class	Return Type
+	Operator	const T* lhs, const T* rhs	const T* rhs	(Any except void)
-	Operator -	const T* lhs, const T* rhs	const T* rhs	(Any except void)
*	Operator *	const T* lhs, const T* rhs	const T* rhs	(Any except void)
/	Operator /	const T* lhs, const T* rhs	const T* rhs	(Any except void)

%	Operator %	const T* lhs, const T* rhs	const T* rhs	(Any except void)
=	Operator =	(Forbidden)	const T* rhs	(Any except void)
+=	Operator +=	const T* lhs, const T* rhs	const T* rhs	(Any except void)
-=	Operator -=	const T* lhs, const T* rhs	const T* rhs	(Any except void)
*=	Operator *=	const T* lhs, const T* rhs	const T* rhs	(Any except void)
/=	Operator /=	const T* lhs, const T* rhs	const T* rhs	(Any except void)
&=	Operator &=	const T* lhs, const T* rhs	const T* rhs	(Any except void)
=	Operator =	const T* lhs, const T* rhs	const T* rhs	(Any except void)
^=	Operator ^=	const T* lhs, const T* rhs	const T* rhs	(Any except void)
!	Operator !	const T* lhs, const T* rhs	const T* rhs	(Any except void)
~	Operator ~	const T* lhs, const T* rhs	const T* rhs	(Any except void)
&	Operator &	const T* lhs, const T* rhs	const T* rhs	(Any except void)
	Operator	const T* lhs, const T* rhs	const T* rhs	(Any except void)
^	Operator ^	const T* lhs, const T* rhs	const T* rhs	(Any except void)
* (Ptr)	Operator *	const T* lhs	(None)	(Any except void)
<	Operator <	const T* lhs, const T* rhs	const T* rhs	bool
==	Operator ==	const T* lhs, const T* rhs	const T* rhs	bool
>	Operator >	const T* lhs, const T* rhs	const T* rhs	bool
<=>	Operator <=>	const T* lhs, const T* rhs	const T* rhs	i2 (-1, 0 or 1)
""(any)	Operator ""(any)	(See below)	(Forbidden)	(Any except void)
[]	Operator []	(Forbidden)	(Any)	(Any except void)

()	Operator ()	(Forbidden)	(Any)	(Any)
→	Operator →	const T* lhs	(None)	(Any except void)

Literals: permitted args

- const char*
- uint64_t
- f32
- f64
- f128
- char
- char8
- char16
- char32
- const char*, size_t
- const char8*, size_t
- const char16*, size_t
- const char32*, size_t

Syntax

todo

Sample usage

```
struct Vector2 {
    float x;
    float y;
}

Vector2 Operator+(const Vector2* lhs, const Vector2* rhs) {
    return { lhs->x + rhs->x, lhs->y + rhs->y };
}

class Array<Value> {
    std::result<Value, std::string> Operator[](const int which) {
        if (which > length || which < 0)
            return "Exceeds array bounds";

        return contents[which];
    }

    size_t size() { return length; }

private:
```

```
Value* contents;  
size_t length;  
}
```

The standard library

IMPORTANT

Very unfinished! Everything needs to be defined and elaborated on.

string.cosh

Strings are a class object for storing sequences of text.

CAUTION

String objects are not null terminated. They will **gain** a null char on conversion into a c_str.

Definition

```
namespace std {
    class basicString<CharType> {
        constructor();
        constructor(basicString<CharType> copy);
        constructor(CharType* initialText);

        bool isEmpty();

        CharType* cString();

        size_t size();

    protected:
        CharType* text;

        size_t length;
    };

    // Cosmium-styled typedef
    typedef string = basicString<char>;
}
```

Sample usage

```
int main(void) {
    std.string hi = "Hello World!"s;

    printf("%s\n", hi.cStr());

    return 0;
}
```

vector.cosh

Vectors are a way of storing multiple of a given type in a dynamically sizing array.

Definition

```
namespace std {  
    class vector<T> {  
        constructor();  
  
        bool isEmpty();  
  
        size_t size();  
  
    protected:  
        T* data;  
  
        size_t length;  
    };  
}
```

Sample usage

```
int main(void) {  
    auto foo = std.vector<Bar> {};  
  
    foo.pushBack(Bar { 1, 2 });  
    foo.pushBack({ 3, 4 });  
  
    for (Bar bar : foo) {  
        printf("%d %ld", bar.baz, bar.egg);  
    }  
  
    return 0;  
}
```