jc Programming Language

Language Design Document

Contents

Motivation

Games programming has always been about pushing hardware to its limits. Low-level games engineers aspire to write fast, performant code that keeps framerates high and crashes low. Traditionally this was achieved using first assembly language, then C and later C++. It should be noted that the higher-level languages still expose low-level memory management and utilities while still being high-level enough for humans to use. It should also be noted that these higher-level languages are both general-purpose programming languages; they are not specialised towards any one discipline of software development. The games industry's last major programming language switchover was from C to C++ in the late 1990s and early 2000s. This has largely stayed the same for the last couple of decades.

The *jc* programming language aims to be the next logical step in low-level games engineering. It is a domain-specific language designed to cater to the needs of games engine programmers. It will retain the high-performance and low-level functionality of C and C++ while adding new features from other programming languages developed since the advent of C++.

Language

General Syntax

Source code (also known as modules) will be made of combinations of statements, functions and scopes. All statements will be delimited by semicolons.

Comments

Comments will use traditional C/C++ style comment syntax with // denoting a single line comment and /* …. */ denoting multi-line comments.

Data Types

Data types will have their sizes specified in their type name.

Data Type Name	Type	Size (bits)
i8 / int8	integer	8
i16 / int16	integer	16
i32 / int32	integer	32
i64 / int64	integer	64
u8 / uint8 / byte	unsigned integer	8
u16 / uint16	unsigned integer	16
u32 / uint32	unsigned integer	32
u64 / uint64	unsigned integer	64
f32 / float32	floating point	32
f64 / float64	floating point/double	64
char	unsigned integer	16
bool	boolean	8
Pointer (declared as * after another data type name e.g. i8*)	pointer	32

Scopes

As in other C-like languages scope blocks will be denoted using curly braces ({}). The syntax examples will show how this works. When a variable is created inside a scope block it is stack-allocated and will be freed once the scope block has ended.

Defer

The defer keyword runs the statement directly after the scope containing it. For example:

```
bool read_file(char *filepath)

{

        FILE *f = fopen(filepath); //opens the file
```

```
    defer fclose(f); //runs fclose at the end of the function
    //DO SOME FILE STUFF BETWEEN HERE AND THE END OF SCOPE
} //defer gets called just before the scope exits here
```

Functions

Function Declaration

Functions are declared using the convention set out in C and C-derived languages:

<return type> <function name>(<list of argument with type and name, comma separated>)

Example: i32 fibonacci(i32 n);

Function Definition

Functions are also defined in a manner similar to C and C-derived languages, with function bodies contained in their own scope block:

<return type> <function name> (<list of argument with type and name, comma separated>)

{

<function body>

}

Example:

```
i32 fibonacci (i32 n)
{
    if (n <= 1)
        return n;
    return fibonacci (n – 1) + fibonacci(n – 2);
}
```

Modules/Namespaces

Modules are the language's mechanism for separating programs into smaller parts. Unlike C/C++ which specifies header and implementation files, the language will take after Python, Java and C# among others and use a modular system. Each module can contain definitions and declarations, with each acting similarly to a namespace in C++.

The language will also have namespaces. This will theoretically make it easier to produce libraries with multiple modules.

Modules will be imported using the **import** keyword. For example importing a module called foobar would look like:

import foobar;

Functions are exported from modules using the **export** keyword. Functions that are not exported will not be visible when the module is imported. Only function declarations need to be exported.

This example would export the function fibonnaci:

export i8 fibonacci(i8 n);

It is also legal syntax to export during function definition. Both the definition and the declaration can be exported. If only one is exported the first declaration is favoured by the compiler.

Type Inference

The compiler will also have optional type inference. This will use the **auto** keyword.

All of these would be considered legal syntax:

i8 foo = 10; //type is i8

auto bar = foo; //type will be deduced as i8

auto x = 1.0f; //type will be deduced as f32

auto y = 10; //type will be deduced as i32 by default

Operator and Function Overloading

Overloading operators and functions will be supported. Quite what form this will take is currently dependent on decisions regarding whether the language will support object-oriented programming and if so, how fully it will be supported.

Function overloading will be supported for functions with different arguments or return values but with the same name.

Nested Functions

Nested functions will be allowed to aid development. These will be functions that are only accessible inside the function they are defined in.

Struct Namespace Capture

Struct function arguments can be explicitly brought into the current scope namespace with the keyword capture in the function definition. (SUBJECT TO CHANGE: Unknown how to do this intelligently with language syntax. Keyword seems a bit sloppy/C# style)