**Para-C:** **C-based Coding designed to be simple and fast**

LUNA KLATZER, HTBLA Leonding

LORENZ HOLZBAUER, HTBLA Leonding

Introduction

Para-C (From Greek Origin: Beside C [[1]](#endnote-1)) is a programming language designed to integrate C, compile to C and serve as a helper and simplifier to write better code in C, but with additional functionality. Including adding more features, like new built-in Macros and functions, OOP-structures, more straightforward array and malloc-handling, expanded data types and simplified functions.

The compiler will take the code with C-base syntax and additional Para-C syntax and keywords and compile it down to simple C. That means that programming in Para-C will be similar but simpler and well looking due to the simplifications, new structures, keywords and helper functions. The Compiler will additionally provide simple checking, through the fact of using the classic compiler structure:

#### The lexical analyser (Tokenizer)

#### Syntax analyser

#### The semantic analyser (Logical analyser)

#### Code Generator and optimiser

which means syntax issues, logic errors and general errors will be detected and logged. The C-code compiler will do the rest of the job, which will be preferably GCC.

Syntax-wise Para-C will still lay onto C to avoid causing issues with more compiler code that would be required for a new syntax that can’t be easily integrated into the C-syntax. So newer structures won’t look so new, and possibly similar to C# or C++, like data-types, one-liners and getters etc.

Furthermore, formatting and non-fetal syntax warnings will also be reported, as a help/motivator to avoid causing inconsistent writing and style. Including possibly harder conventions, that will try to improve on the loose C-conventions, which are more open to writing code. That means Para-C will introduce more conventions regarding naming, type declarations, formatting, commenting and will likely also adopt a few Python conventions[[2]](#endnote-2) and integrate the ideas in the Zen of Python[[3]](#endnote-3)(Since the Compiler is also written in Cython).

Content

[1. Base structure 4](#_Toc72002104)

[1.1 Implementation of Para-C inside C 4](#_Toc72002105)

[1.2 The Parac base library 5](#_Toc72002106)

[1.2.1 Magic Values in the Para-C source code 5](#_Toc72002107)

[1.2.2 Magic Values in the C source code 5](#_Toc72002108)

[1.3 File structure 5](#_Toc72002109)

[1.3.1 Para-C modules and files 6](#_Toc72002110)

[1.3.2 Importing PARA-modules 6](#_Toc72002111)

[1.3.3 Importing C-modules 6](#_Toc72002112)

[1.4 Para-C identifiers and structures 6](#_Toc72002113)

[1.5 In-Code Exceptions 6](#_Toc72002114)

[2. Compiler and logical Structure of Para-C 6](#_Toc72002115)

[2.1 Structure 6](#_Toc72002116)

[2.2 Compiler Exceptions 6](#_Toc72002117)

[2.2.1 Error-Codes 6](#_Toc72002118)

[3. Usage-examples of Para-C 7](#_Toc72002119)

[3.1 Project Configuration 7](#_Toc72002120)

[3.2 Using C-Code inside Para-C 7](#_Toc72002121)

[3.2.1 Using Standard C-code 7](#_Toc72002122)

[3.2.2 Using C-libraries 7](#_Toc72002123)

[3.2.3 Restrictions of Para-C 7](#_Toc72002124)

[Endnotes 8](#_Toc72002125)

# Base structure

The structure of Para-C will closely lean to the C-Structure (upwards from C11-Standard) but still have its independent system apart from it. That means it will include its own:

* Structure for its Parac-modules and C-modules (*See* [*File Structure*](#_File_structure))
* Import-structure
* Built-in macros[[4]](#endnote-4)
* Built-in functions[[5]](#endnote-5) (which will partly replace the C-functions for easier handling)
* OOP-Structure using GObject[[6]](#endnote-6) and system for integrating C-code and Macro definitions.

## Implementation of Para-C inside C

The language serves the main purpose of helping and providing features for C-code, meaning that any structure used will derive from the Parac Base Library which is written in C. The Parac Base Library will be categorised into three parts

* Core Library – The code required for new keywords, functions, identifiers and additional core functionality.
* Built-In Library – Built-in functions that are automatically available in the Para-C code (Imported in the C-code).
* Extension Library – Extension Methods and identifiers for specific use cases. (Implementations are going to be at their core C, but the overhead will be written in Para-C, meaning it will be compiled as well at runtime, but only if it was imported)

The base modules and any additionally used structure will be imported into the project at the top of the file. These imports will be separated from the C-imports of the user which are not associated with the Parac Base Library.

Any Para-C identifiers in the C-code will have a clear prefix “\_\_pbl\_” and the suffix “\_\_” (“pbl” = “parac base library”), and are going to be signalised if needed, using comments to separate user and compiler code.

*Note: To not be confused with the reserved identifiers in C, the Para-C identifiers have the suffix of two underscores instead of just the regular prefix of one or two which are reserved by the C-conventions. (See C11 – 7.1.3 Reserved Identifiers[[7]](#endnote-7))*

The identifiers are categorised into magic methods, magic declarations and magic constants. (Similar to the Python magic methods and double underscore definition.). These will also not interfere with the standard identifiers in C or overwrite GCC compiler identifiers to avoid unexpected results.

If return values are used, they will be passed to locally declared variables. If new declarations are made by the Para-C compiler that did not exist prior in the Para-C code these will be declared either at the top of the function or file and surrounded by comments signalising these were automatically generated. To that comments signalising their use case and position might also be added to help find the use case positions if needed. That means the goal of the compiler will be to make as little logical changes to the code to preserve its written nature and intent.

## The Parac base library

### Magic Values in the Para-C source code

### Magic Values in the C source code

## File structure

The file structure in Para-C is similar to C and works around the compiler and the system-specific libraries and modules that can be imported. Para-C will work mainly around the Parac Base Library, the executable Compiler itself, the C-modules which are used in Para-C and Para-C Extensions.

The importing and module system of a project will lay on the `\_\_parac\_\_.h` language file, which will define important identifiers inside Para-C which are used throughout the compiled code. That means the compiler will fetch all required data and insert them into the file which can then be modified as well if the user intends to do so. This will also include important paths, such as import paths, module paths, etc. which are vital for the proper workflow of Para-C.

Still, it is discouraged to change the `\_\_parac\_\_.h` in the compiled code, since the parac-config.json file can be used to define project details, versioning, and identifiers used in the \_\_parac\_\_.h file (*See* [*Project Configuration*](#_Project_Configuration)). The file should only be changed if required or specific changes need to be done that are not available in the normal parac-config.json file. This is because the \_\_parac\_\_.h is **only** intended to be used in the specific environment where the compilation was run since import paths will likely not work on other systems and unexpected results can occur.

### Para-C modules and files

### Importing PARA-modules

### Importing C-modules

## Para-C identifiers and structures

## In-Code Exceptions

# Compiler and logical Structure of Para-C

## Structure

## Compiler Exceptions

Exceptions in the Para-C compiler

### Error-Codes

* 1\*\* Internal Errors:

101 – Interrupt Error: The compiler received an interrupt while running. (derives from the Python BaseException KeyboardInterrupt)

102 – Config Not Found: The parac-config.json file for the project was not found, which is responsible for configuring the Project.

* 2\*\* File Exceptions

200 – File Error: General File Exception in the Compiler

201 – File Permission Error: Failed to access (read, write) to existing file due to missing permissions

202 – File not Found: The File was not found and does not exist! If the file can’t be seen it will be treated as well as a File not Found Error.

203 – Is Directory: File is a directory

* 9\*\* Other Exceptions

901 – Unknown Error: Received an unknown exception while running.

# Usage-examples of Para-C

## Project Configuration

## Using C-Code inside Para-C

### Using Standard C-code

### Using C-libraries

### Restrictions of Para-C

Endnotes

1. Para meaning and origin: [[here]](https://en.wiktionary.org/wiki/%CF%80%CE%B1%CF%81%CE%AC#Preposition) [↑](#endnote-ref-1)
2. PEP8 – Conventions of Python: [[here]](https://www.python.org/dev/peps/pep-0008/) [↑](#endnote-ref-2)
3. PEP20 – Zen of Python: [[here]](https://www.python.org/dev/peps/pep-0020/) [↑](#endnote-ref-3)
4. List of Pre-defined macros in C: [[here]](https://gcc.gnu.org/onlinedocs/cpp/Predefined-Macros.html) [↑](#endnote-ref-4)
5. List of Built-in C-functions: [[here]](https://www.tutorialspoint.com/ansi_c/c_function_references.htm) [↑](#endnote-ref-5)
6. Introduction to GObject: [[here]](https://www.freedesktop.org/software/gstreamer-sdk/data/docs/latest/gobject/howto-gobject.html) [↑](#endnote-ref-6)
7. C11– Draft April 12, 2011: [[here]](http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf) [↑](#endnote-ref-7)