**Para-C:** **C-like 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 an extension to C with additional functionality, simplification and helper tools. Including adding more features, like new built-in Macros, simplified or new functions, OOP-structures (Under consideration), list and malloc-handling, expanded data types and additional project-management features.

The compiler will take the Para-C code to compile it down to simple C with the integrated functionality. 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. 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, overloads 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 some ideas of the Zen of Python[[3]](#endnote-3)(Since the Compiler is also written in Cython).

Content

[1. Base structure 4](#_Toc72608112)

[1.1 Implementation 4](#_Toc72608113)

[1.1.1 Overview - Parac Base Library 4](#_Toc72608114)

[1.1.2 The Parac Core Library (PCL) 5](#_Toc72608115)

[1.1.3 Identifiers and Separation of code 5](#_Toc72608116)

[1.2 File structure 6](#_Toc72608117)

[1.2.1 File management in the compiled code 6](#_Toc72608118)

[1.2.2 Importing PARA-files 6](#_Toc72608119)

[1.2.3 Importing C-libraries or headers 7](#_Toc72608120)

[1.3 Style Conventions 8](#_Toc72608121)

[1.3.1 Naming Conventions 9](#_Toc72608122)

[1.4 Name Mangling 10](#_Toc72608123)

[1.5 Built-In Identifiers 10](#_Toc72608124)

[1.5.1 Magic Values in the Para-C source code 10](#_Toc72608125)

[1.5.2 Magic Values in the C source code 10](#_Toc72608126)

[1.6 Running a Program 10](#_Toc72608127)

[1.6.1 Using the CLI 10](#_Toc72608128)

[1.6.2 Project Structure 10](#_Toc72608129)

[1.6.3 Using the parac-config.json file 11](#_Toc72608130)

[1.6.4 Entry-File 12](#_Toc72608131)

[1.6.5 Runtime Entry-Point 13](#_Toc72608132)

[1.7 In-Code Exceptions 13](#_Toc72608133)

[2. Compiler and logical Structure of Para-C 13](#_Toc72608134)

[2.1 Structure 13](#_Toc72608135)

[2.2 CLI 13](#_Toc72608136)

[2.2.1 CLI Reference 13](#_Toc72608137)

[2.3 Compiler Exceptions 13](#_Toc72608138)

[2.3.1 Error-Codes 14](#_Toc72608139)

[3. Para-C Language Reference 14](#_Toc72608140)

[4. Usage-examples of Para-C 14](#_Toc72608141)

[4.1 Using C-Code inside Para-C 14](#_Toc72608142)

[4.1.1 Using Standard C-code 14](#_Toc72608143)

[4.1.2 Using C-libraries 14](#_Toc72608144)

[4.1.3 Restrictions of Para-C 14](#_Toc72608145)

[Endnotes 15](#_Toc72608146)

# Base structure

The structure of Para-C will closely lean to the C-Structure (upwards from C11-Standard) but nonetheless 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 (Bases on C for compatibility)
* Name mangling (Relative to entry-point of the program)
* Exception handling
* Variable System (Required for unique handling and exceptions)
* 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) (Under consideration)

## Implementation

*Para-C intends the GNU C Compiler as the primary compiler for the language, however, there is no limitation to use another one, but unexpected results can likely occur.*

The language serves the main purpose of providing new features for the Base-C language, meaning that any structure used will derive from compiled C-code and library-code that provides that functionality. This language library of Para-C is called the Parac Base Library, which is written in C and for higher-level areas Para-C.

### Overview - Parac Base Library

*From this point on the Parac Base Library will be referred to as PBL*

The PBL is categorised into three parts:

* Core Library (PCL) – The code required for the Para-C 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 (PEL) – Extension Functions and identifiers for specific use cases. (Implementations can be at their core C, but the overhead will always 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 Built-In Library.

### The Parac Core Library (PCL)

The Para-C core library is, as already explained in the previous section, the base for the Para-C programming language. If Para-C functionality is used inside the code that does not exist inside C, the associated core file/library will be imported in the c-code and used.

This means the user does not have to import any headers themselves, since the compiler automatically will import all core library headers that are needed. (This only goes for core functionality and keywords, everything else is treated as regular libraries that can be imported)

#### PCL Imports inside C

The PCL imports won’t be inserted into every file, but inserted into the project-wide header file \_\_parac\_\_.h. This header file will be imported into every resulting C file, meaning if a package is used it will be available in every other file.

For clarification reasons, the compiler will still log errors for imports for unknown identifiers inside Para-C even if they are imported in another file. That means if a library is imported in one file and another file wants to access it without importing it, it will fail due to the compiler not finding the import in the associated file.

### Identifiers and Separation of code

Any PBL identifiers in the C-code will have a clear prefix “\_pbl\_” (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))*

If the Para-C compiler declares new variables, calls new functions or updates values that are not part of the user code, but required for other functionality, such as new keywords, special function calls etc. the compiler will attempt to comment with additional information on these lines and separate them clearly from the user code to avoid that code gets mixed up or confused.

## File structure

The file structure in Para-C is similar to C and works around the compiler directory, the PBL, the compiler libraries and the project module. Nonetheless due to the structure of Para-C, importing in a project will be different from a regular C/C++ project and depends on the C-version on the compiler-generated \_\_parac\_\_.h header file (This file will have the identifier ph\_ to signalise in the code the declarations are from the base parac header).

### File management in the compiled code

The compiler-generated header file \_\_parac\_\_.h will define important identifiers inside Para-C which are used throughout the compiled code. That means the compiler will fetch all required imports, paths and additional data and insert them into the file. The file will be placed at the highest level of the project hierarchy, meaning the defined project\_path (*For more detailed info see* [*Using the parac-config.json file*](#_Using_the_parac-config.json)) will be used to insert it. This header file can be modified if the user intends to do so.

However, it is discouraged to change the \_\_parac\_\_.h file, and instead use the universal non-os-dependent parac-config.json file.

The compiled header file should only be changed if it is necessary for specific changes that need to be done that are not available in the normal parac-config.json file. This is because the \_\_parac\_\_.h is configured for the specific system and compiler where the compilation was run. Import paths will likely or almost always not work on other systems and the program will fail to compile or unwanted results are going to occur.

### Importing PARA-files

*Note: For the sake of clearness, importing and including is used here in the same context and way*

The standard in-code importing system will not be different from C and will base on a standard header file. In this header, all public available identifiers can be specified, which are then either written in the header itself or the source file (.para). This header can be either included inside your file or another header, which can then be included as well.

For Libraries in Para-C, the simple #include <library.h> can be used since for C library imports additional commenting is required.

Example:

* Standard Header: #include “header.ph”
* PBL Library Header: #include <library.ph>

#### Renaming included identifiers

The only difference Para-C is introducing is the way you handle the name mangling and possible duplicate identifier names. That means Para-C will introduce new syntax to handle specific cases and “rename” the identifiers to avoid duplicate naming.

The word rename is here in quotation marks since due to the name mangling there can never be the case that a variable has the same name as another variable from another file. Still, in the user code, the mangling is not yet applied meaning that by importing any header containing a variable declaration with an already existing name, the compiler will be unable to identify which variable is the “correct” one.

In this case, you can reorder your code and use a #define macro to point to your variable, but for the sake of easier readability it will use the following syntax for renaming imported values:

#from “<header-file>” include <variable-name> as <new-name>

This will “rename” the variable or signalise the compiler that the new variable name will point to that mangled name. The mangled variable will still be imported in the C-version of the code, but in the Para-C version, it will no longer interfere with existing variables.

### Importing C-libraries or headers

Since Para-C is based on C and backwards compatibility, C-code can be easily included into Para-C, by just either importing the library/header which should be available in the standard C-library or including the header file specified.

To signalise the compiler the header is in native C, a comment needs to be added next to the import, either as:

* One-Liner-Comment: // -\*- lang: C -\*-
* Or Multi-Line-Comment – Before and after the imports: // -\*- lang: C -\*-

Due to compatibility and proper workflow reasons, the compiler will also go through the included c-header and source code and possibly change minor details to fit the Para-C code. Still, the compiler will not do any major changes and attempt to do as little as possible to keep the functionality alive as wanted.

*Note: Since Para-C is based on C and not too different C++, C++ code may be included, but due to restrictions of C++, it might not work and break.*

Example:

* C-header file:

#include “<path\_to\_header>.h” // -\*- lang: C -\*-

#include <c\_library.h> // -\*- lang: C -\*-

* Multi-line Import:

// -\*- lang: C -\*-

#include <stdio.h>

#include <stdbool.h>

// -\*- lang: C -\*-

## Style Conventions

Since Para-C is written in C, style conventions won’t be different in the compiled code or PBL, except the user-specified ones, still inside Para-C naming conventions are a bit different from C for better differentiation of certain types:

* Line-Length Limit is 79 characters for one line
* 4 Spaces per Indentation
* Functions should be declared with the return type definition at the front and the name following in one line. The arguments can be split if it exceeds the line-length limit. Declaring the return type over the name is not allowed, even if it is commonly used around some C-developers.

Example:

void MyFunction(

int arg1, int arg2, int arg3, ...

); // Tab before the arguments

or

void MyFunction(

int arg1,

int arg2,

int arg3,

...

); // Tab before the arguments

* Arrays/Lists should be strechted evenly over multiple lines if the content exceeds the line-limit.

Example:

// One Liner (if it doesn’t exceed the line limit again)

char[] char\_array = {

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ...

}; // Additional one tab

or

// Matrix-Style Array Distribution

char[] char\_array = {

1, 2, 3,

4, 5, 6,

7, 8, 9,

10, 11 ...

}; // Additional one tab in every row

* Indentation level should be around 4/5 to allow readability (Still there is no limitation other than the compiler limitations)

### Naming Conventions

|  |  |  |
| --- | --- | --- |
| **Type** | **Public** | **Internal\*** |
| Header-file | snake\_case |  |
| Source-file | snake\_case |  |
| Structures (Classes, Structs) | PascalCase | \_PascalCase |
| Exceptions | PascalCase (with Error at the end) |  |
| Functions/Methods | PascalCase | \_PascalCase |
| Variables | snake\_case | \_snake\_case |
| Instances | snake\_case | \_snake\_case |
| Constants | \_SNAKE\_CASE | \_SNAKE\_CASE |
| Types | snake\_case (with possibly \_t at the end) |  |

*\* Internal in this context means variables inside a file, function or structure that should be seen as “private”. Not everything can be internal such as files and exceptions, so in those cases, the field will be empty.*

## Name Mangling

## Built-In Identifiers

Inside Para-C macros and reserved identifiers are used to store program-vital data, serve as functions and store configuration data. The identifiers are categorised into library functions, magic identifiers and magic constants. (Similar to the Python magic methods and double underscore definition)

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

### Magic Values in the C source code

## Running a Program

### Using the CLI

The Para-C compiler CLI provides multiple different commands on how to run the program (For more detailed info see [CLI Reference](#_CLI_Reference)):

* parac compile – Compiles the program down to C and creates a build folder with the raw C code and a dist folder containing the executable with the specified environment. (Starts on default with no arguments an interactive prompt)
* parac run – Compiles and runs the program in the command line, where stdout, stderr and stdin are redirected into the program.
* parac new – Creates a new empty project based on the default project structure. It will contain a parac-config.json file for the project configuration where the values are going to be pre-set, but changeable.
* parac run --compiled – If parac run is used with the --compiled option, then the passed directory is a compiled directory with C-code produced by parac compile, which should be run.

### Project Structure

*Note: To compile or run a program in Para-C, a project setup or configuration is not required since the compiler will compile either way based on an entry file. Still, for organised libraries or programs, it is recommended to use the provided tools to properly manage it.*

In Para-C the Project structure bases on a module-like structure, where a configuration file, called parac-config.json,is used to declare project settings and set general project info. A Project in this case is a simple folder with a configuration file outside of it with a possible readme, .git folder etc.

Possible Look of such a structure:

src/

main.para

main.ph

other.para

other.ph

parac-config.json

LICENSE

README.md

Here the src folder will contain all source files and data required for the program. Inside the parac-config.json the entry-file was set to ./src/main.para, meaning the compiler will start compiling and pre-processing from there and check all included headers and libraries, which are in this case the other.ph and main.ph headers (The compiler includes these two headers because in the main.para file they were included. If they weren’t, then the compiler would ignore these as they are not visibly needed in the program).

### Using the parac-config.json file

Using the parac-config.json file is relatively simple and the possible options simply need to be changed or added. The compiler will take the file when compiling and based on that configure and create the project.

**Possible Configuration:**

* name –Name of the Project/Program (Accessible using \_\_name\_\_)
* description– Description of the Project/Program (Accessible using \_\_description\_\_)
* version– Version of the Project/Program (Accessible using \_\_version\_\_)
* author– Author of the Project/Program (Accessible using \_\_author\_\_)
* license– Distribution License Type for the Project/Program eg. MIT or GPL (Accessible using \_\_license\_\_)
* entry-point– Absolute or Relative path to the entry-point file
* include– A list of all files and directories that should be included in the program eg. data files, configuration files etc. that are not automatically included with the entry-point.
* compiler-version– Wanted version of the compiler that should be used. If the version of the compiler used does not match this an exception will be thrown during compilation.

Syntax:

* + >=0.1 – Version must be greater than 0.1
  + >~0.1 – Version must be greater than 0.1
  + <=0.1 – Version must be lower or equal than 0.1
  + <~0.1 – Version must be lower than 0.1
  + ==0.1 – Version must be equal to 0.1
* compiler-options– List of all options that should be used for the Para-C compiler
* c-compiler-options– List of all options that should be used for the compiler (In this case it is intended for the GNU C Compiler)

### Entry-File

#### Compilation Specification

Unlike in C, the compiler in Para-C is not designed to be able to compile files without an entry-point or reference on how the program should be run. This is because the compiler goes out from the entry-point file and from there handles all name mangling and imports. This means that to compile a project or file, the entry-file must be explicitly marked as an entry-point. All files and headers used will be compiled as well, but unused files will be ignored even if they are in the same folder, due to uncertainty about how to handle them.

This entry-point file can be either set in the parac-config.json file or using the parac compile command where the prompt will ask for the wanted configuration and entry-point. The recommended name for the main entry-point is main.para or index.para.

### Runtime Entry-Point

An entry-point in the program is the function that should be called on runtime, to start the entire program. It is not necessarily needed, for example in library code, where the code is imported into another program.

Para-C will also allow pre-compilation of library code, where the user wants to use them in a C-environment and avoid Para-C mangling and runtime handling.

#### Specifying an entry-point function

In Para-C declaring the entry-point with the C-standard int main() is not allowed. It uses instead a unique “datatype” called EntryPoint for hinting at the type of the return, which is in this case, the entry-point of the program. The function will allow returns/return-codes like in C, which can signalise the state of the program.

Example:

EntryPoint Main() { return 0; } // Program that returns immediately

The reason for this seemingly odd decision is based on the Para-C language structure, which automatically builds in exception-catching and mangling. The PBL, therefore, provides the entry-point function, which runs the specified entry-point function with additional wrapping and checking that is hidden from the user code.

Due to that functionality, defining the entry-point does not require the method name to be explicitly Main, meaning a name like EntryPoint MyProgramMain() is also valid. However, it is recommended to use the standard Main() name for visibility.

## In-Code Exceptions

# Compiler and logical Structure of Para-C

## Structure

## CLI

### CLI Reference

## 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 and compiler.

* **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.

# Para-C Language Reference

*(This part of the document serves as the reference for all keywords, identifiers, functions etc. that are added in Para-C and will provide information on how to properly use them)*

# Usage-examples of Para-C

## 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)