**Para-C:** **C-based Coding for Advanced Management of Programs and Code**

LUNA KLATZER, HTBLA Leonding

Draft: 21.01.2022 – Preview for v1.0

Introduction

Para-C (From Greek "para": Beside/Alongside C [[1]](#endnote-1)) is a programming language that is designed to integrate other languages and allow for advanced management of programs / code-bases inside a program, where the language will serve as a base for writing overhead and connector programs, which manage instances, can listen for events, stop and start processes and generally manage in- and out-data. This also includes adding more features for the C11 standard, like new built-in functions, decorators, memory-management, additional function-handling, and additional project-management features. To summarise, Para-C will be a simplified and extended C, without the complications of C++ and more straightforward to code.

To achieve the multiple language “support” / integration-functionality, the compiler will take the Para-C code and compile the source code down to simple C and generate the code required to integrate the wanted language, using their required compiler/interpreter for the language. That means that programming in Para-C will be more similar to higher-level languages than to C, due to the new features, keywords and helper functions. Including adding the simple option to integrate and manage code or programs that should be directly embedded into the management program. Using this, you can for example embed async functionality from Python directly into the program, which is not natively supported, and then pass generated data to a C++ program, which then uses that to run something else. This can also include proper management based on web events and data or using the Para-C project configuration to compile code on runtime as well with specified compilers so that in the end the project can be compiled in one go and properly merged with the Para-C program.

Content

[0. Document Style 7](#_Toc89955624)

[1. Theory – Processes and Integration of Executables 8](#_Toc89955625)

[1.1 Simple Tasks 8](#_Toc89955626)

[1.2 Specifying Arguments and Returns 9](#_Toc89955627)

[1.3 Multi-Threading 10](#_Toc89955628)

[2. The Para-C Core Language 11](#_Toc89955629)

[2.1 Overview 11](#_Toc89955630)

[2.2 Implementation 11](#_Toc89955631)

[2.2.1 Overview Parac Base Library 11](#_Toc89955632)

[2.2.2 Identifiers and Separation of code 12](#_Toc89955633)

[2.2.3 Managing C code inside Para-C 12](#_Toc89955634)

[2.2.4 Relative file names 12](#_Toc89955635)

[2.3 Program Structure 13](#_Toc89955636)

[2.3.1 Project Structure 13](#_Toc89955637)

[2.3.2 Project Configuration 14](#_Toc89955638)

[2.3.3 Declarations and Definitions 15](#_Toc89955639)

[2.3.4 Function Definitions 15](#_Toc89955640)

[2.3.5 Blocks 15](#_Toc89955641)

[2.3.6 Entry-File 15](#_Toc89955642)

[2.3.7 Runtime Entry-Point Function 15](#_Toc89955643)

[2.3.8 Specifying an entry-point function 15](#_Toc89955644)

[2.3.9 Lifetime 16](#_Toc89955645)

[2.3.10 Scope and Visibility 16](#_Toc89955646)

[2.3.11 Linkage of Variables 16](#_Toc89955647)

[2.3.12 Para-C Pragmas 16](#_Toc89955648)

[2.3.13 Additional Notes 17](#_Toc89955649)

[2.4 Integration of Processes and Programs 17](#_Toc89955650)

[2.5 Precedence and associativity of operators 17](#_Toc89955651)

[2.6 Lexical Conventions 17](#_Toc89955652)

[2.7 Statements 17](#_Toc89955653)

[2.7.1 Selection Statements 17](#_Toc89955654)

[2.7.2 Iteration Statements 17](#_Toc89955655)

[2.7.3 Compound Statement 17](#_Toc89955656)

[2.7.4 Expression Statement 17](#_Toc89955657)

[2.7.5 Null Statement 17](#_Toc89955658)

[2.8 Operators 17](#_Toc89955659)

[2.9 Namespaces and file management 17](#_Toc89955660)

[2.9.1 Importing PARA-files 17](#_Toc89955661)

[2.9.2 Namespaces 19](#_Toc89955662)

[2.9.3 Including C-libraries or headers 19](#_Toc89955663)

[2.9.4 Compiled code management 19](#_Toc89955664)

[2.10 Declaration and Types 20](#_Toc89955665)

[2.10.1 Properties of Types 20](#_Toc89955666)

[2.10.2 Iterable Type 20](#_Toc89955667)

[2.10.3 Casting and Type-Conversion 21](#_Toc89955668)

[2.10.1 The Any-type 21](#_Toc89955669)

[2.10.2 Typeof 25](#_Toc89955670)

[2.10.3 Typedef 25](#_Toc89955671)

[2.10.4 Sizeof 25](#_Toc89955672)

[2.10.5 Pointers 25](#_Toc89955673)

[2.11 Structures and Simple Classes 25](#_Toc89955674)

[2.12 Enumerators 25](#_Toc89955675)

[2.13 Functions 25](#_Toc89955676)

[2.13.1 Function Overloading 25](#_Toc89955677)

[2.13.2 Simple-Functions 25](#_Toc89955678)

[2.13.3 Decorators 25](#_Toc89955679)

[2.13.4 Named Arguments 25](#_Toc89955680)

[2.13.5 Ellipsis Arguments 26](#_Toc89955681)

[2.13.6 Lambda Functions 26](#_Toc89955682)

[2.14 In-Code Exceptions 27](#_Toc89955683)

[2.14.1 Keywords 27](#_Toc89955684)

[2.14.2 Creating an Exceptions 28](#_Toc89955685)

[2.14.3 Raising the Exception 28](#_Toc89955686)

[2.14.4 Catching Exceptions 29](#_Toc89955687)

[2.14.5 Using a finally and else block 29](#_Toc89955688)

[2.15 Built-In Reserved Identifiers 29](#_Toc89955689)

[2.15.1 Magic Values in the C source code 29](#_Toc89955690)

[2.16 Style Conventions 29](#_Toc89955691)

[2.16.1 Naming Conventions 30](#_Toc89955692)

[2.17 Name Mangling 31](#_Toc89955693)

[3. The Parac Core Library 31](#_Toc89955694)

[3.1 List of Libraries in the PCL 31](#_Toc89955695)

[3.2 PCL Imports inside C 31](#_Toc89955696)

[4. The Para Language Extensions 32](#_Toc89955697)

[4.1 Overview 32](#_Toc89955698)

[4.2 Syntax 32](#_Toc89955699)

[4.2.1 Declaring an extension task 32](#_Toc89955700)

[4.2.2 Calling an extension task 33](#_Toc89955701)

[4.3 Process Management 33](#_Toc89955702)

[4.4 Return Data 33](#_Toc89955703)

[5. The Para-C Pre-Processor 33](#_Toc89955704)

[5.1 Overview 33](#_Toc89955705)

[5.2 Algorithm and Parsing 33](#_Toc89955706)

[5.3 Usage of Pre-Processor Directives 34](#_Toc89955707)

[6. The Para-C Compiler 34](#_Toc89955708)

[6.1 Overview 34](#_Toc89955709)

[6.2 Structure 35](#_Toc89955710)

[6.3 Lexer and Parser 35](#_Toc89955711)

[6.3.1 Overview 35](#_Toc89955712)

[6.3.2 Antlr4 Structure (Parsing Tree Components) 36](#_Toc89955713)

[6.3.3 Pre-Processor Grammar 36](#_Toc89955714)

[6.4 Semantic Analyser 36](#_Toc89955715)

[6.4.1 Overview 36](#_Toc89955716)

[6.4.2 Algorithmic structure 37](#_Toc89955717)

[6.5 File Linker 37](#_Toc89955718)

[6.6 Code Optimiser 37](#_Toc89955719)

[6.7 Code Generator 37](#_Toc89955720)

[6.8 Compiler Warnings 38](#_Toc89955721)

[6.9 Compiler Exceptions 38](#_Toc89955722)

[6.9.1 Error-Codes 38](#_Toc89955723)

[6.10 Tokens 40](#_Toc89955724)

[6.10.1 C Keywords 40](#_Toc89955725)

[6.10.2 Para-C Keywords 43](#_Toc89955726)

[6.10.3 Special Symbols 44](#_Toc89955727)

[6.10.4 Pre-Processor Directives 45](#_Toc89955728)

[Endnotes 46](#_Toc89955729)

# Document Style

**For Proper definition and distinction these styles will be used in the document:**

Standard text:

Normal Font

Name Reference or special name:

Consolas on white background

Code, or Value Reference or Snippet:

Italic Consolas with a grey background

Code Block:

Consolas covering the entire line with a grey background

# Theory – Processes and Integration of Executables

This section is intended as the baseline of the theory of how the so-called processes will be implemented in the language and how they will work and how they can be utilised. It will be the core feature, which will be worked on, besides the core language features to allow for the handling of such tasks with new data types.

## Simple Tasks

The base idea for Para-C are Tasks, which can be spawned with specific arguments, called and then awaited, before receiving the return data from the spawned task.

That means that the data first must be passed, like a function, to the Task and then properly passed onto the process as arguments. This can be likely done over command-line arguments, which will just then include all arguments as parameters to the executable. (or to leave them untouched simple file transfer)

The data, in this case, will be restricted to the base types, which should be implemented in every higher-level/mid-level language:

* Numerics (Integer, Floating-Point Number, Long…)
* Chars (ASCII, UTF-8)
* Iterables (Arrays, Lists, Strings)
* Key-Value Storage (Dictionaries) – Only exception, which can be not supported everywhere.

These data types will then be passed in a standardised format, likely JSON-like to allow for easier readability (and not reinvent the wheel). This means these arguments will then be passed onto the Executable (with a language other than Para-C), which will be compiled/run using the specific Para-C library, which will add the support for such passing and fetching of data.

*Note: In this case, it’s still unclear whether Para-C will add also modules that will add additional source code to the program of the user, or leave it untouched, but force the user to call a “init” function (would require file transfer storage).*

After the data is passed and the user received the data for the process, the process will simply run until the user decides to exit the program. Here a special task-return function will need to be called, which will return the user-defined arguments and pass them into a file, which can then be fetched by the Para-C program.

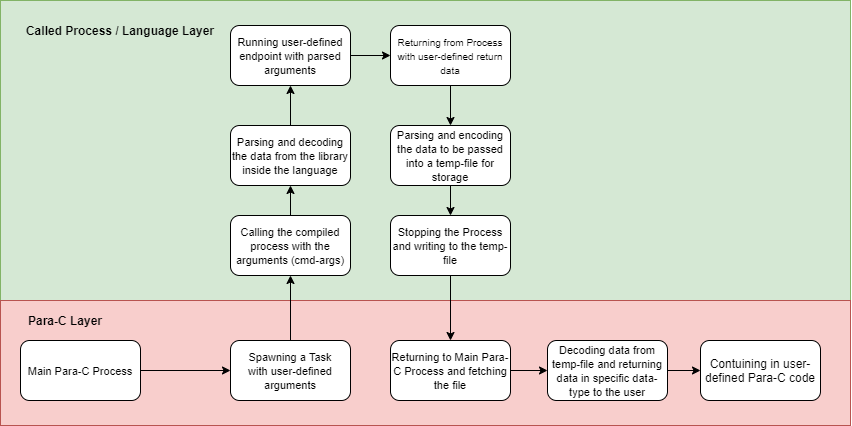
After the process is finished, the C-code under the Para-C code will continue to run and then continue fetching the data from the finished process. This data will then be converted and passed onto the compiler-generated type, which was generated from the configuration of the Processes.

Figure 1 - Process Flow-Chart

## Specifying Arguments and Returns

Specifying Arguments and Returns is a major issue that will need to be addressed with the implementation. Due to the issue of many languages not allowing the integration of dynamic passing and dynamic allocation of memory without restrictions into the language, the user will be forced to specify the arguments that will be passed. (Behind those libraries, this will also require specific allocation of memory for C and C++, special conversion from strings for languages like C# or Java and generally just management of arguments and return data)

One major idea on how to manage this is to manage arguments and parameters is to use language extensions with the compiler. The main idea is to create mini compiler extensions, which will create dynamic source code files for these languages and create functions that will manage how the data should be received and parsed. This can allow that the return types of those functions can be simply set and no dynamic creation is required.

Though this could complicate the process of integrating already compiled executables, which is an intended feature, though it might be required for the integration with Para-C, that the source code must be available. This is undecided though and further decisions will have to be taken when the implementation starts.

## Multi-Threading

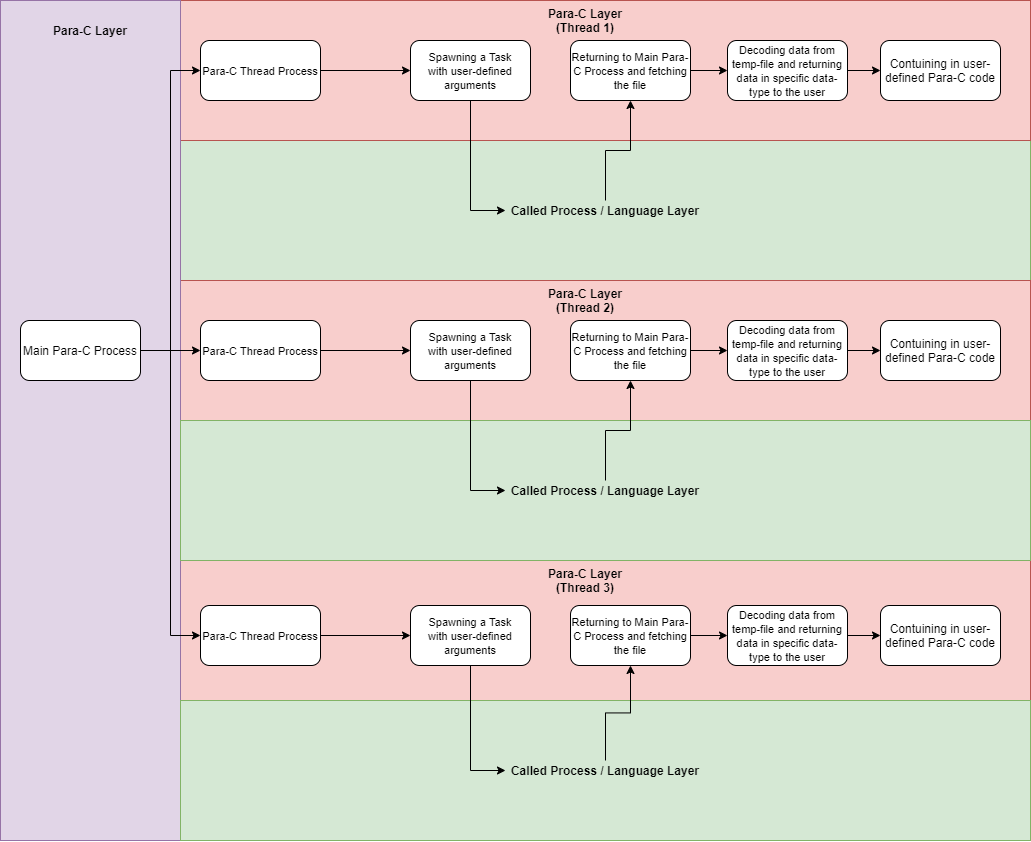


Figure 2 - Multi-Threading in a flow-chart

# The Para-C Core Language

## Overview

The Para-C Core Language is the fundamental language and base from where language extensions will derive from. It is an optimised and updated version of the C11-Standard, which implements improved concepts and handling for better management and usability. It is usable using the Para-C Compiler, which will transpile the Para-C code into C-code or directly binary executable. (*See* [*The Para-C Compiler*](#_The_Para-C_Compiler))

## Implementation

*Para-C intends the GCC C Compiler as the primary compiler for the language and will build upon it to implement functionality, though perhaps the Visual Studio Compiler may be used for Windows, especially if advanced threading needs to be implemented.*

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 Para-C 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 functionality Para-C (will be compiled on runtime as well).

### 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 (PBIL) – Built-in functions that are automatically available in the Para-C code (Imported in the C-code).
* Extension Library (PEL) – Mainly, but not necessarily, language extensions, functions for specific use cases and specific identifiers. (Non-Langauge extensions 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. In the case of language extensions, they will have two libraries, one Para-C and one in the target language that is integrated)

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.

### Identifiers and Separation of code

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

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 add a comment with additional information and separate them clearly from the user code.

These should though usually be hidden behind PBL macros or functions.

### Managing C code inside Para-C

Since due to an [issue](https://github.com/Para-C/Para-C/issues/32) where it was uncertain on how to manage the array syntax, either using the C syntax, implementing the C# syntax to make an array a type or implement both for compatibility, the decision was reached to entirely move onto C# syntax and abandon most of the C-intended support.

This does not though mean you can not use C code inside Para-C using [Extension Tasks](#_The_Para_Language_1), though, for the most part, Para-C will intend to not redundantly support C, as Para-C is already more intended for writing new overhead and management code, to simply certain applications, rather than allow for a higher-level implementation of older code. This means that you *can* use C, or if needed migrate to Para-C, as the syntax is very similar, with a few exceptions as explained in the issue with the array syntax.

### Relative file names

In Para-C the implementation of name mangling means that file names are also relative to their name and place. The relative name of a file is represented by their parents and its name without the file extensions. The root path or project path will be the starting point, meaning that a file inside that directory will have the relative name <filename> and any file in a directory will have the name of the directory (parent). Spaces are not allowed in any file or directory name.

*Examples for Relative Names:*

* name.para -> name
* name/second\_name.para -> name.second\_name

## Program Structure

### 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/library-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 with code or additionally with a readme, .git folder, init-configuration etc.

A 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 were not, then the compiler would ignore these as they are not visibly needed in the program).

### Project Configuration

The parac-config.json file is the project configuration implementation, which will use CMake in the background. All specified options will be passed to the compiler, which will change behaviour based on the configuration and create the output.

**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 e.g. MIT or GPL (Accessible using \_\_*license*\_\_)
* *entry-point*– Absolute or Relative path to the entry-point file (Must be at the highest level of the project e.g. inside a source folder or next to the parac-config.json file)
* *include*– A list of all files and directories that should be included in the program e.g. data files, configuration files etc. that are not automatically included with the entry-point.
* *parac-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 or equal to 0.1
  + *>~0.1* – Version must be greater than 0.1
  + *<=0.1* – Version must be lower or equal to 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)

### Declarations and Definitions

Like in C, Para-C uses declarations and definitions to manage its variables:

* Declarations may be used as prototypes and definitions of *the scope of a variable*, but not assigning a value and allowing for the option of it being re-declared again as many times as wanted. These declarations look exactly like in C and are simply made up of the type and the variable\_identifier.

type identifier;

* Definitions are the default and required way of assigning a value to a variable, like in C, a variable may be declared and defined at the same moment, creating the following syntax:

type identifier = value;

Though unlike in C in Para-C, there is usually no *real* declared variable, except partially in headers, due to it being intended as a prototype file. This should be easily visible in the PBL, as the Macros intended for declaring variables are already allocated and assigned a default value. This is to prevent issues with accessing declared variables in C / unassigned and unset memory, which is in C undefined behaviour.

This also means that all types are defined with a *meta* property (Of type *PblVarMetaData\_T*), which has itself the property *defined*, which if *false* means the value of the type is not initialised / commonly NULL.

### Function Definitions

### Blocks

### Specifying an entry-point function

In Para-C declaring the entry-point with the C-standard *int main()* is not allowed. This is because the Compiler generates the main() function itself based on the program and requirements. Therefore, it uses instead a new syntax where the entry-point function is declared using *entry status <function-name>*.

Example:

entry status Main()

{

// code

}

*entry*, in this case, is a new keyword like *static*, which hints to the compiler that this function is the entry-point function. The keyword *status* is here a new built-in type, which should be returned from the function. This data type is a struct, which can contain status-code, additional stdout strings or an entire exception that should be raised.

*Note: All functions are automatically wrapped using a status return to integrate exception returns and calling stacks. If the code is hinted to be native C, automatically the return is the actual type specified.*

The reason for this seemingly odd decision is partly based on the Para-C language structure, which automatically builds in exception-catching and additional return functionality. Therefore, the language itself provides the entry-point function, which runs the specified entry-point function with additional wrapping and checking that is hidden from the user code. Because of that defining the entry-point does not require the method name to be explicitly Main, meaning a name like *entry status MyProgramMain()* is also valid.

However, it is recommended to use the standard identifier *Main*.

### Scope and Visibility

The Scope of a variable defines which parts of the program may access the variable that was defined. There are a few scopes that a variable may be defined:

* “Project/Global Scope” - The scope that inclusion is handled, which may be accessed by all files that are connected. This is though in reality accessing the scope of another file and importing it into the local file one using the pre-processor.
* Local File Scope (Local-Global Scope) - Local File scope, which defines the local variables, which may be accessed by all function scopes and compound scopes in this file.
* Function Scope / Function Definition Scope.
* Compound Statement ({ }).

Important to note with scopes is that it also defines the lifetime of a variable, and as such, it should be watched out for that a pointer does not point to a variable that was already de-allocated / reached the end of its lifetime. Para-C like C does not handle dangling pointers, though will attempt to raise a warning when a possible dangling pointer is encountered.

### Lifetime

The lifetime of a variable is the period of program execution time, where the variable may be accessed by the user code. This is an important thing for Para-C, as it utilises a Garbage Collector for collecting un-used variables, and to remove/free memory as soon as possible to not waste memory while executing the program. Especially for long-running programs, the lifetime should be kept short and variables only used in the scope it is needed.

Para-C, in this case, will with large allocations especially inside struct types (type-definitions using structs) give the freeing task to the Garbage Collector, which will run every few execution cycles and clean up what it can find.

### Linkage of Variables

### Para-C Pragmas

At the moment, there are no specific pragmas available yet, though with later updates there may be ones.

### Additional Notes

1. Line-Endings are strictly limited to \n,\r\n and\r
2. Comments are strictly ignored and before starting compilation removed. This means that the errors reported in the program will *NOT* contain any comments.

## Integration of Processes and Programs

## Precedence and associativity of operators

## Lexical Conventions

## Statements

### Selection Statements

### Iteration Statements

### Compound Statement

### Expression Statement

### Null Statement

## Operators

## Namespaces and file management

*This entire section needs to be reworked! See* [*https://github.com/Para-C/Para-C/issues/74*](https://github.com/Para-C/Para-C/issues/74)

## Declaration and Types

### Type Allocation and Management

### Properties of Types

### Iterable Type

Inside Para-C, the iterable type is a combination of a standard C array and a dynamic list. This means, using type[] identifier will result automatically in a dynamic list/array, but if it is never resized it will practically be a standard array.

This also means that there is no need for conversion between lists and arrays since in both cases functions for the list can be applied without any need for conversion.

When specifying the length of an iterable, the length will be used to predefine defaults, meaning that if length 7 is specified and only 3 items are passed, the left 4 will be the default value. This also occurs when passing no items, but passing a length.

*Examples:*

int[3] x;// Uninitialised, meaning that the length does not matter

int[0] x = { }; // Empty iterable with length 0

int[5] x = { 1, 2, 3, 4, 5 }; // Creating item with length 5 (would result in the same without the length specifier)

int[] y = { 1, 2, 3, 4, 5}; // Specifying no length, but passing 5 items, creating an iterable with 5 items and length 5

### Casting and Type-Conversion

#### Cast Expression

#### “as“ expression

*Runtime keyword with Compiler optimisations*

##### Overview

Converting types inside Para-C is made a lot simpler with the “as” keyword, which acts as a universal convert. It takes the variable before it and converts it to the type specified after it:

<variable> as <type>

Under the hood this is will be a function that will take in Para-C types and appropriately convert the type to the wanted type. This will be done on runtime with the any-type and a void\* pointer to the actual value.

##### Usage

The “as” keyword can be applied to any expression that evaluates to a single value, like a cast this means that the type will be used and passed like specified.

*Example with simple variable declaration:*

float variable = 3.45;

int variable\_2 = variable as int;

*Example with a calculation:*

float variable = 3.4;

float variable2 = 3.3333333;

float sum = variable2 as int \* variable; // int \* float = float

### The Any-type

#### Overview

The any-type is a special type, which allows every type to be passed onto it. Meaning it can store a type, and have itself be overwritten by another type. There is a distinction between the any-type and an any-type array though, meaning it needs to be explicitly specified whether the type should be an array/list or a single value.

#### Theory

In C the void\* pointer can be used to *point* to any value. This restricts the usage though, due to the usage of a pointer and a value, meaning that the user needs to watch out for the lifetime of both variables. This often can result in cases, where the value gets destroyed before the pointer, meaning the pointer becomes a dangling pointer, which will raise an error when trying to be used to access the already destroyed value.

In Para-C this is solved through implementing a Compiler-managed type, which is dynamically allocated to the needs of the user. This means when creating an any-type the compiler will allocate memory and the any-type will fetch using a void\* pointer the stored value, but with the difference that the allocated memory exists as long as the pointer exists. Resulting in that when the value gets out of scope e.g. should be destroyed, that the memory used is freed at the same time as the pointer is destroyed, meaning the user will not have to manage the lifetime of the value and correct passing to avoid a dangling void\* pointer.

If the value gets overwritten, the program will simply attempt to reallocate the memory and increase its size, so that pointers to that value are continuing to work as wanted, preventing dangling pointers as well (This can not be guaranteed though, as the operating system manages the memory and may change the location. This is though nothing the user will have to worry about, as Para-C uses wrapper types that are allocated once and as such pointers will always be valid, even if the underlying types are altered).

#### C-Implementation

In the C-implementation, the function void\* malloc() will be used to allocate the needed memory for the variable. This means that a new space of memory is allocated, which will be pointed to by the returned void\* pointer. The min. size of this allocated block will equal the size of a NULL pointer.

This also means that the any-type can represent an array or list, meaning:

int[] array = { 2, 3 };

any[] variable = array as any[2];

int variable2 = variable[0]; // => 2

is valid. In this case, the array is converted to an array with two elements. This means the variable will be an array containing a list of any-type variables, which point to allocated memory, where the values were copied to.

#### Usage and Converting

The any-type can be initialised and passed to a function like any other type and will simply be overwritten if a different type is passed onto it.

*Note: Due to the difference between any and any-type array, specifying any as the type of a parameter does* ***not*** *mean it will allow any variable! It needs to be specified whether an array or simple value since a normal any-type can only hold one single value at one moment.*

*Simple Example:*

any variable = 2; // the value is set to 2

variable = “3”; // the value is replaced by 3

print(variable); // 3 is printed

1. The any-type can also be simply converted to another type and then passed to another variable:

any variable = “3”;

int variable\_2 = variable as int;

print(variable\_2);

1. If an array should be converted to an array the following syntax can be used:

any[] <name> = <array> as any[<size>];

Note that the any-array now contains allocated memory, meaning that the types are not strict anymore and so the following would be allowed:

{ “name”, 3, 4, ‘c’, ‘\n’ };

1. If the user wants to pass a strict-type variable as an any-type parameter to a function, the following syntax needs to be used:

function(param as any);

or for an array:

function(param as any[<size>]);

Since the any-type counts as a type like any other type, it will not be automatically converted and the compiler will raise a mismatching type error.

#### Getting the currently used type

To fetch the current type, typeof *(See for more info* [*Typeof*](#_Typeof)*)* can be used, which can be used to compare against other types or initialise an entirely new variable.

*Example with declaring a new variable with typeof:*

any variable = 1;

typeof(variable) variable\_2 = 2; // Will evaluate to int (the actual type)

*Example with converting a variable to the currently used type:*

any variable = 1;

int variable\_2 = “3” as typeof(variable); // => 3

### Typeof

*Compile-time and Runtime keyword*

### Typedef

*Runtime keyword*

### Sizeof

*Compile-time keyword*

### Pointers

## Structures and Simple Classes

## Enumerators

## Functions

### Function Overloading

### Simple-Functions

### Decorators

### Named Arguments

### Ellipsis Arguments

### Lambda Functions

Para-C supports like other higher-level programming languages Lambda-Functions, which, as already explained in [Functions](#_Functions), can be passed onto a variable and called like any other regular function. A lambda function in this case can also be used with a decorator (*See* [*Decorators*](#_Decorators)) which wraps the function and applies additional functionality.

#### Syntax

A lambda function is started using a parameter declaration and then an arrow to signalise the call behaviour. This is similar to the Simple-Functions, which use a standard function definition and an arrow to start the wanted block.

After the arrow, there are two possible behaviours the user can choose from:

* Statement Lambda, which is equivalent to a regular function:

() => { <list-of-statements> }

* Expression Lambda, which is equivalent to a simple function:

() => <expression>

The return of a lambda function will always be Any, meaning that any form of value can be returned. If passed to a lambda<T> variable, which acts like a function-pointer, the argument will need to be specified, like this:

lambda<types-of-args> name = (args) => lambda-body;

*Example with statement lambda:*

lambda<int> multiply\_by\_two = (int x) => { return x \* 2; };

*Example with expression lambda:*

lambda<int> multiply\_by\_two = (int x) => x \* 2;

#### The lambda<T> type

The lambda<T> type is the type that is returned when creating a lambda function. This type represents a basic struct that contains the information for the parameter types. The return-type is though always any meaning that it depends on the actual lambda which value is returned since it will allow any value. This is for ease and better functionality when passing so that different return types can be allowed.

*Example with a basic declaration:*

lambda<int, float, string> name = …

1. If wanted, the argument names can also be explicitly declared. This does not alter the functionality though, but only provides the option to name the arguments when calling the function (*See for more info* [*Named Arguments*](#_Named_Arguments)).

*Example:*

lambda<int x> multiply\_by\_two = (int x) => { return x \* 2; };

*Note: The variable names need to match the specified names in the parenthesis of the lambda declaration.*

## In-Code Exceptions

Para-C provides Exceptions similar to C++, but with a bit of pythonic-like syntax-sugar added.

These are implemented using compiler-generated compile-types, which automatically implement the user-specified return types. These compile types contain the actual return type and the Para-C return type struct, which defines the exception return.

If an exception was raised, the struct will contain the exception and the call stack. The call-stack will be in this case a compiler-generated variable that keeps track of the current stack, by passing itself to every function and being updated there.

### Keywords

*raise* – Keyword used to raise an Exception. The following value represents the exception and must be an instance of the exception struct.

*try* – Keyword used to start a try code-block, where exceptions will be passed to the following except statements if the type is included.

*except* – Keyword used to start a handler code-block, which will be called if the expression inside the parenthesis matches the exception type.

*finally* – Keyword used to start a finally code-block, which will be called after all the previous code is executed (including try and except statements). The only exception is if the exception was not included in the except cases or another exception was raised inside an except code-block

*else* – Keyword used to start an else-block, which will be called if no exception is raised in the try-block.

### Creating an Exceptions

Raising an exception is relatively simple, though unlike implementations like Python, not type-secured, meaning that there is only a single exception type that may be used to initialise an exception.

The following properties may be set for the exception:

* msg
* name
* filename
* line
* line\_content

### Raising the Exception

To raise an exception, the error struct needs to be initialised and then with the *raise* keyword raised/thrown.

Example with variable creation:

CustomError e = {

.error\_msg=”Something bad”,

.display\_name=”Error”

};

raise e;

Example with direct initialisation:

raise (CustomError) {

.error\_msg=”Something bad”,

.display\_name=”Error”

};

### Catching Exceptions

### Using a finally and else block

## Built-In Reserved Identifiers

Inside Para-C special macros and reserved identifiers are used to store program-vital data, serve as functions an

### Magic Values in the C source code

## 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 120 characters for one line
* 2 Spaces per Indentation
* Functions should be declared with the return type definition at the front and the name following in a single 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 my\_function(

int arg1,

int arg2,

int arg3,

...

); // Tab before the arguments

* Arrays/Lists should be stretched 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 | Any |  |
| Source-file | Any |  |
| Structures (Classes, Structs) | PascalCase | \_PascalCase |
| Exceptions | PascalCase (with Error at the end) |  |
| Functions/Methods | PascalCase | \_ snake\_case |
| Property field (struct or simple class) | snake\_case |  |
| Variables | snake\_case | \_snake\_case |
| Instances | snake\_case | \_snake\_case |
| Constants | PascalCase with a leading *k* | \_SNAKE\_CASE |
| Built-In Types | alllowercase | (Forbidden) |
| Typedef Types | PascalCase with a trailing *\_T* | (Forbidden) |

*\* 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

Para-C applies, like C++, name mangling to its variables and identifiers, the only exception being pre-processor directives or defines, meaning that the user can generate two functions with the same name, but different parameter types (also called function overloading). To that, names that are included from other files will be wrapped with the specific file name and place to differentiate it from variables with the same name. This allows that imported names are renamed, or referenced using their parent header name (*See for more info* [*Importing PARA-files*](#_Importing_PARA-files))

# The Parac Core Library

The Para-C core library is, as already explained in the implementation section ([*Implementation of Para-C inside C*](#_Implementation_of_Para-C)), 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 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.

## List of Libraries in the PCL

* langext.parah– Language Extension Library containing structs, types and functions for integrating and handling extension tasks.
* sysio.parah –System IO Library implementing the C library (stdio.h)

## 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.

This means the user will have to properly manage inclusion so that the linker can properly merge the files in the end to the resulting c-file.

# The Para Language Extensions

## Overview

The Para Language Extensions are extensions for integrating other programming languages into Para-C. Using Extension tasks, a program can be embedded into the Para-C Program and called with parameters. The Program will wait until the Process has finished and return its returned data if they are given.

These Processes are compiled with the main program, meaning the binaries are created and able to be shipped with the entire Project. The only requirements are that the compiler or interpreter of the language is added and that the compiler knows how to embed the process into the Para-C program. This is done using the installable extensions and libraries inside the language.

## Syntax

The Language Extension Syntax is relatively simple and consists of the extension task declaration that defines code from another language that should be embedded and the call for that task.

### Declaring an extension task

To declare an extension task this syntax is used:

*exttask <lang> <identifier> (<args>) {config}*

The identifier in this case is like a regular function that can be called inside the Para-C program. After that, the arguments, which will be passed onto the called file or code, and the configuration, which defines how the compiler should treat the extension task, are declared. This also includes its return arguments, though how they will be specified is not decided *yet*.

*exttask* <lang> <identifier> *(<args>)* {config}

*Example:*

exttask python TestFunc2(int x, int y, int z) {

    file: "file.py",

    file\_entry: "main"

}

### Calling an extension task

To call a task, the following syntax is used:

spawn <func\_name>(<args>);

To catch the return simply assign the value to a variable. The type of return ExtTaskReturn(struct instance) can be imported from exttask.parah

ExtTaskReturn <variable\_name> = spawn <func\_name>(<args>);

*Note that this is only syntax sugar and behind the actual type is a compiler-generated type, which will specifically put the return types into the type. This is to avoid adding complicated memory management, which allows for multi-dynamic types. (It is also considered to add the configuration to the project config, though it is uncertain at the moment)*

## Process Management

## Return Data

# The Para-C Pre-Processor

## Overview

The Para-C Pre-Processor is, like in the GCC C Compiler, a component besides the core language that will be called to interpret and process pre-processor directives that alter the source code or interact with the compiler. These directives use a Hashtag(#) as the prefix for the directives (*For a list of directives see* [*Pre-Processor Directives*](#_Pre-Processor_Directives)).

## Algorithm and Parsing

The Pre-Processor utilises, like the core language, Antlr4 as the Parser and Lexer, but in this case, the Para-C code will be “ignored” and only the pre-processor directives are going to be parsed and then used to process the file. Meaning the Pre-Processor will use the context and the stored lines to create the resulting file/files.

To that, a major difference to the core language is that while processing; all files are going to be directly included in the process and checked to correctly apply pre-processor effects. (This will be done over the *#include* directive)**.** These files will then also be passed to the Compiler, which will use them in the Linker (*See* [File Linker](#_File_Linker_1)) and Multi-File Semantic Analyser (*See* [*Multi-File Semantic Analyser*](#_Semantic_Analyser_(Entire)) to correctly analyse included files as well.

After finishing the parsing and modification process, the pragmas and the generated altered code will be passed to the Compiler. Meaning that in the Compiler, pre-processor directives will not be visible anymore and error logs will display the altered code, meaning it will be visible how the pre-processor interpreted the specific directives and altered the code.

## Usage of Pre-Processor Directives

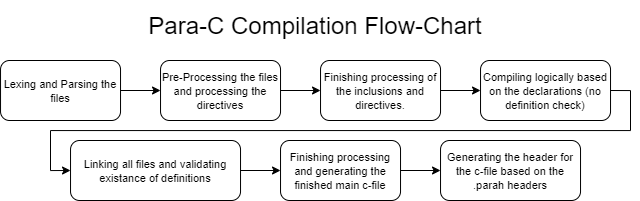
# The Para-C Compiler

## Overview

The Para-C Compiler is the Compiler responsible for the Para-C Core Language (*See* [*The Para-C Core Language*](#_The_Para-C_Core)) and the Para-C Language Extensions (*See* [*The Para Language Extensions*](#_The_Para_Language)). It parses, analyses and links the files together to generate a C-source code output or using the GCC C Compiler (GNU Compiler Collection C Compiler) an executable.

*Note: If language extensions are used, the compiler generates executables for each language extension (Even if multiple files and components are used, the compiler will merge them into a single program, which will, depending on the program, call different modules and code), meaning that the output will contain multiple executables. There are tools though that merge these into a single executable, to make shipping the code easier.*

## Structure



The Structure of the Compiler is made up of multiple components and modules, which have specified jobs assigned e.g. components will not interact with each other and the compiler will use the components to put all pieces together, which in the end make up the finished program.

*The different modules are:*

The steps of a compilation process (Flowchart)

* The Pre-Processor, which will alter the code based on the pre-processor directives (*See* [*The Para-C Pre-Processor*](#_The_Para-C_Pre-Processor_1))
* The Lexer and Parser, which generate the tokens and the logic trees of the program. (Includes the Conversion to Logical Tokens, which simplifies the tokens to make the job easier for the Semantic Analysis. *See* [*Lexer and Parser using Antlr4*](#_Lexer_and_Parser))
* The Single-File Semantic Analyser (*See* [*Single-File Semantic Analyser*](#_Semantic_Analyser_(Single-File)))
* The File Linker (*See* [*File Linker*](#_File_Linker_1))
* The Code Optimiser (*See* [*Code Optimiser*](#_Code_Optimiser))
* The Code Generator (*See* [*Code Generator*](#_Code_Generator))

## Lexer and Parser

### Overview

The Para-C Core Language uses for easier parsing Antlr4, which automatically generates python source files containing the parser, listeners and lexer based on the *ParaC.g4* file defining the grammar of the language. The generated code will be wrapped inside a module, which will call and use the generated code. Using that, it will convert on runtime the inserted file into a list of logic trees, which contain all needed information for statements. These logic trees are then returned and used to compile the program.

The Lexer and Parser will start by first parsing the main file and then afterwards all included files (Data of the included files will be passed by the Pre-Processor). These will then be wrapped and passed to the Semantic Analyser.

### Antlr4 Structure (Parsing Tree Components)

The basic structure of the Parsing Tree is based on the compilationUnit, which describes the entire file and the EOF (End of File Character). The main component, one layer under that, is the translationUnit, which describes the actual code that will be processed in the compiler. The translationUnit contains a list of externalItems, which can be either stray Semicolons(;) or a declaration/definition of a variable or function.

### Pre-Processor Grammar

Since the Pre-Processor (See [The Para-C Pre-Processor](#_The_Para-C_Pre-Processor)) is syntax-wise different than the standard language, mainly noticeable due to the line endings (pre-processor statements with hard-line endings (\n or \r\n, while the normal lines use declared line breaks in form of a Semicolons(;)), it uses its independent grammar file, and as such lexer and parser. Those will parse everything and pass it onto the Pre-Processor itself to generate the altered files.

## Semantic Analyser

### Overview

The Semantic Analyser is the first step after the parsing process, which will introduce logical checking on the file and validate whether statements are valid or not.

This step will mainly work on converting the code into the specific internal tokens that then can be used to properly process and link all files together. This task as a whole is then split into these two separate items:

* Processing the core file, where all inclusions were placed and so all declarations must be available for the logical checking step aka. type-checking and logical cohesion.
* Processing all files that were mentioned by the header aka. all source files where the definitions should be placed. Those files will be all independently processed and their inclusions will also be placed inside the files. Here it’s important to note though, that due to this logic, there may **never** be definitions in headers, as such they would cause errors later in the linker, as it can not predict which value is to be used.

### Algorithmic structure

The Semantic Analysis will go through each externalDeclaration (Can be either a declaration or functionDefinition) and go through each block individually and check for logical correctness. This means that it will treat each function block as a single token to handle and validate each in its context.

## File Linker

File linking in Para-C is similar to C, with the key difference being though, that the result of the generation is a single C file, not byte code. This means that the linker has the main task to fetch all definitions and link them together creating, in the end, the resulting C-file (This will be though managed by the code generator).

All definitions will be placed in the C-file, as well as the declarations in the C-Header. This is to preserve the declaration logic and avoid reference errors in the C code.

## Code Optimiser

The Code Optimiser will be the last step when processing everything, and attempt to check for duplicate declarations, unnecessary variable calls and in general things that just are not that necessary to be in source-code. Here it will still utilise the Para-C logic tokens, and pass them to the Code generator, which will compile the Para-C logic tokens into C logic tokens.

## Code Generator

As the name states, the code generator will convert the compiled C logic tokens into stable code, by adding all required references and creating the required structure for the functionality to work.

## Compiler Warnings

The Compiler while running will check for basic information and will report on possibly problematic issues such as logical issues, possible loss of data or problematic usages of certain types. While running these will be counted and at the end of the run logged as a summary of the process.

To that, syntax warnings for non-fatal formatting and inconsistency issues can be reported, as a help to avoid causing inconsistent writing and style. This also includes the partly stricter conventions, that try to improve on the loose C-conventions, which are more open to writing code.

## Compiler Exceptions

Exceptions inside Para-C are categorised into two categories:

* Non-Fatal Exceptions, which do not interfere with continuing to check the file and
* Fatal Exceptions, which can not be ignored and cause the compiler to interrupt the process and exit.

If only Non-Fatal Exceptions get noticed by the compiler, the compilation will finish with a summary containing a counter for all errors and warnings while running the compilation.

If a Fatal-Issue is received and causes a hard interrupt while running, the return code will be an error code that is specified here. This helps for better categorisation for certain errors. An error message will also appear with the error code at the end of the file, including a trace-back if the issue is a bug inside the compiler. (*Note that the actual return code used with exit() is 1 for errors. This is due to the structure of many os-systems that require that return codes should not exceed the 256 (8-Bits) range*

### Error-Codes

*All Exceptions inherit from the base code (99) and their respective parent code e.g. for 204 -> 200.*

* 99 Base Error:

99 – BaseError/ParacCompilerError: Base Error every other exception inherits of.

* 1\*\* Internal Errors:

100 – InternalError: An Exception in the Internal parts of the compiler that are not related to the compilation.

101 – InterruptError: The compiler received an interrupt while running. (Derives from the Python Base Exception KeyboardInterrupt)

102 – FailedToProcessError: A specific error that is raised inside a compilation process or pre-processor process, which represents a failure in processing the wanted input. This class replaces the actual error that would be logged and all error logs will be printed onto the console.

* 2\*\* User Input Errors:

200 – UserInputError: General Exception due to faulty input of the user

201 – FileAccessError: General Exception due to failed interaction with a file

202 – FilePermissionError: Failed to access (read, write) to existing file due to missing permissions

203 – FileNotFoundError: The File was not found and does not exist! If the file can’t be seen it will be treated as well as FileNotFound.

204 – IsDirectoryError: File is a directory

205 – InvalidArgumentsError: The passed flags or arguments are invalid and can’t be processed.

206 – ConfigNotFoundError: The configuration file for the project was not found.

207 – CCompilerNotFoundError: Failed to locate the configured C Compiler. Path does not exist. (If the file can’t be executed, FilePermissionError will be raised)

* 3\*\* Lexical Errors:

300 – LexerError: An issue occurred in the Tokenizer / Lexical Analyser step of compiling. (Derive from the Antlr4 lexer errors)

* 4\*\* Parser Errors:

400 – ParserError: An issue occurred in the Parser (Logic Tree generator), which tries to convert the generated Antlr4 tokens into proper Logical Para-C tokens

401 – SyntaxError: A syntax issue occurred while processing that is a direct result of the user failing to input valid code.

* 5\*\* Logical Errors:

500 – LogicalError: An issue occurred while walking through the program, which was caused due to logical irregularity and incompatible statements.

* 6\*\* Linker Errors:

600 – LinkerError: An issue occurred while linking the files together and checking dependencies and mergeability. (Logical issues like double declarations or importing a name that was already defined will be treated as linker error since they directly result from the linking process)

* 9\*\* Other Errors:

900 – UnassociatedError: Exception of type other that is assignable to any other type of exception

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

## Tokens

### C Keywords

*Note: Contains only basic definitions/explanations. For more info see C Keywords [[2]](#endnote-2)*

*auto* – Storage Class Keyword declaring the usage of the default Storage Class in C (Para-C).

*const* – Keyword used to define a Constant Variable that can only be defined once.

*double* – Built-In C Floating Point Datatype (64-Bit).

*float* – Built-In C Floating Point Datatype (32-Bit).

*int* – Built-In C Numeric Datatype (16-Bit or 32-Bit).

*short* – Built-In C Numeric Datatype (16-Bit).

*struct* – Datatype used to declare a Structure Variable or Structure Datatype (if used with typedef).

*unsigned* – Keyword used with Numeric and Floating-Point Datatypes to declare them as unsigned (they can only hold positive values).

*break* – Keyword used to break out of the current loop (Restricted to loops).

*continue* – Keyword used to continue and jump to the next iteration of the loop (Restricted to loops).

*else* – Keyword used to declare an else branch / code-block, which will be called if **all** previous if or else-if statements have failed.

*for* – Keyword used to start a for loop (must be followed by parenthesis containing an expression that either evaluates to *true* (1) or *false* (0)).

*long* – Built-In C Numeric Datatype (64-Bit or 32-Bit, if the used OS is in 32-Bit).

*signed* – Keyword used with Numeric and Floating-Point Datatypes to declare them as signed (they can hold both positive and negative values, used by default).

*switch* – Keywords used to start a switch statement (must be followed by parenthesis containing an expression).

*void* – Built-In C Datatype signalising “nothing”. Used for no-return function or in the context of a pointer (Declaring a variable using void is forbidden).

*case* – Keyword used inside a switch statement to start a case branch. Must be followed by a constant value.

*default* – Keyword used inside a switch statement to start a default branch. Called when all previous cases were *false*.

*enum* – Keyword used to define an Enum Variable containing integer constants

*goto* – Keyword used to jump to a declared label (Not recommended inside Para-C, since it provides unnecessary assembler-like functionality that can be easily achieved using other simpler systems)

*register* – Storage Class Keyword used to hint that a variable *should* be stored inside a register instead of ram. (Very low capacity, but a far better speed on read and write)

*sizeof* – Built-In Function which can be called to get the size of a specified variable. (Returns the size in bytes)

*typedef* – Keyword used to create a new custom type

*volatile* – Keyword used to signalise that the keyword can be changed in an unspecified way by the hardware. This also means that the compiler will not do any optimisations based on the logic of the program, since the value might change during runtime even if the compiler does not see that.

*char* – Built-In C Datatype representing a char / numeric value for a char (4-Bit)

*do* – Keyword used to start a do-while loop code-block.

*extern* – Storage Class Keyword used to define a global variable that is visible to all object modules. It will point to the address of the prior declared variable with the same name and can not be initialised.

*if* – Keyword used to declare an if-branch / code-block, which will be called if the statement inside the parenthesis evaluates to *true*.

*return* – Keyword used to return from the current function to the caller of the function. Can contain a value if the function-type is not *void*

*static* – Storage Class Keyword used to define a static variable, which will retain its value until the end of the program. This means if a static variable is declared inside a function and the value is increased inside that function, calling the function multiple times will use the old value and contain from there, meaning the old value will **not** bereinitialised. (Similar to constant, but with the difference, the value can be changed)

*union* – Union Keyword used to create a union. The union can contain multiple types and will reuse the same storage for all types, meaning that the biggest type inside the union will define the size of it. If initialised only one value can be used at the same time.

*while* – Keyword used to start a while loop (must be followed by parenthesis containing an expression that either evaluates to *true* (1) or *false* (0)).

### Para-C Keywords

*entry* – Keyword used to hint the compiler at the entry point function of the program. This can be only used once and multiple usages will raise an error during compilation.

*status* – Built-In datatype which represents an exit status for a program.

*raise* – Keyword used to raise an Exception. The following value represents the exception and must be an instance of the exception struct.

*try* – Keyword used to start a try code-block, where exceptions will be passed to the following except statements if the type is included.

*except* – Keyword used to start a handler code-block, which will be called if the expression inside the parenthesis matches the exception type.

*finally* – Keyword used to start a finally code-block, which will be called after all the previous code is executed (including try and except statements). The only exception is if the exception was not included in the except cases or another exception was raised inside an except code-block

*else* – Keyword used to start an else-block, which will be called if no exception is raised in the try-block.

*as* – Convert keyword which will convert the value before the keyword to the type specified after the keyword

*typeof* – Keyword used to get the type of a variable and compare it against others. (Exists in native C but only as an extension and for declaring a variable with the type of another variable)

*spawn* – Keyword used to spawn/create an Extension Task and call it

### Special Symbols

|  |  |
| --- | --- |
| Arithmetic Symbols | Addition(+), Subtraction(-), Modulo(%), Multiplication(\*), Division(/) |
| End of line | ; |
| Sequencing | , |
| Code-Block | { } |
| Subexpression Grouping | ( ) |
| Assignment | = |
| Special Assignment | +=, /=, \*=, -=, %=, &=, |=, ^=, <<=, >>= |
| Increasement or Decrement | ++, -- |
| Condition Evaluation | Questionmark(?), Colon(:) |
| Comparison | ==, !=, <, <=, >, >= |
| De- or Reference | \*, &, [ ] |
| Member Selection | ->, Dot(.) |
| Pre-processor Directive | # |
| Bit-Operator | &, ~, |, ^ |
| Decorator Specifier | @ |
| Logical | &&, ||, ! |
| Shift Operator | >>, >>>, <<, <<< |

### Pre-Processor Directives

|  |  |
| --- | --- |
| #define | Substitutes a pre-processor macro. (Can either be a single *value* or *(expression)*) |
| #include | Inserts a particular header from another file. |
| #undef | Undefines a pre-processor macro. |
| #ifdef | If the identifier following the directive is defined, the block afterwards will be utilised. |
| #ifndef | If the identifier following the directive is not defined, the block afterwards will be utilised. |
| #elifdef | Else-If the identifier following the directive is defined, the block afterwards will be utilised. |
| #elifndef | Else-If the identifier following the directive is not defined, the block afterwards will be utilised. |
| #if | Tests if a compile-time condition is true. |
| #else | The alternative for #if. Following Code is inserted if the previous if was evaluated as *false* |
| #elif | #else and #if in one statement. (adds another possible branch) |
| #endif | Ends pre-processor conditional. |
| #error | Prints error message on stderr. |
| #pragma | Issues special commands to the compiler, using a standardized method. |
| #line | Tells the compiler to set the line number and filename (optional) for the next line. |
| defined(…) | Can be used inside #if or #elif to check whether an item is defined. Logical Operators are allowed between items inside define() |

Endnotes

1. Para meaning and origin: [[link]](https://en.wiktionary.org/wiki/%CF%80%CE%B1%CF%81%CE%AC#Preposition) [↑](#endnote-ref-1)
2. C Keywords: [[link]](https://en.cppreference.com/w/c/keyword) [↑](#endnote-ref-2)