Free Pascal Cheatsheet

A quick reference guide for Free Pascal syntax, data types, control structures, procedures, functions, units, and basic OOP concepts.

Basics & Data Types

Program Structure

```
program ProgramName;
uses
  SysUtils, Classes; // Optional units

var
  // Variable declarations

begin
  // Main program code

end.
```

- `program ProgramName;`: Declares the program name.
- `uses Units;`: Imports units providing additional functionality.
- `var`: Section for variable declarations.
- `begin...end.`: Contains the executable statements.

Comments:
- Single line: `// This is a comment`
- Block: `(* This is a multi-line comment *)`
- Alternative block: `{ This is also a multi-line comment }`

Keywords: `begin`, `end`, `program`, `unit`, `interface`, `implementation`, `uses`, `var`, `const`, `type`, `procedure`, `function`, `if`, `then`, `else`, `case`, `of`, `for`, `to`, `downto`, `do`, `while`, `repeat`, `until`, `with`, `try`, `except`, `finally`, `class`, `object`, `property`, `array`, `record`, `set`, `pointer`, `nil`, `exit`, `continue`, `break`, `goto`, `label` (avoid `goto` and `label` if possible).

Identifiers: Must start with a letter or underscore, followed by letters, digits, or underscores. Case-insensitive.

Basic I/O:
- `Read(Variable1, ...);`: Reads values from standard input.
- `Readln(Variable1, ...);`: Reads values and moves to the next line.
- `Write(Expression1, ...);`: Writes values to standard output.
- `Writeln(Expression1, ...);`: Writes values and moves to the next line.

Example:

```
program HelloWorld;
begin
  Writeln('Hello, World!');
end.
```

Tip: Organize code logically within the `begin...end.` block. Use descriptive variable names.

Fundamental Data Types

Integer Types:	`Integer`, `SmallInt`, `LongInt`, `Byte`, `Word`, `Cardinal` (signed/unsigned, varying ranges)
Real Types:	`Real`, `Single`, `Double`, `Extended` (varying precision and range)
Boolean Type:	`Boolean` (`True`, `False`)
Character Type:	`Char` (single character)
String Types:	`string` (dynamic, default), `AnsiString`, `WideString`, `ShortString` (fixed size 1-255)
Type Declaration (`type`):	`type TMyInteger = Integer; TStatus = (Active, Inactive, Pending);`
Variable Declaration (`var`):	`var Count: Integer; Name: string; IsReady: Boolean;`
Constants (`const`):	`const MaxCount = 100; Pi = 3.14159; Greeting = 'Hello';`

Best Practice: Use specific integer types (e.g., `Byte`, `Word`) if you know the range to save memory, especially in embedded systems. Use `Double` for general-purpose floating-point calculations.

Operators

Arithmetic:	`+` (addition), `-` (subtraction), `*` (multiplication), `/` (floating-point division), `div` (integer division), `mod` (modulo - remainder)
Relational:	`=` (equal to), `<>` or `!=` (not equal to), `<` (less than), `>` (greater than), `<=` (less than or equal to), `>=` (greater than or equal to)
Logical:	`and` (logical AND), `or` (logical OR), `not` (logical NOT), `xor` (logical XOR)
Set Operators:	`+` (union), `-` (difference), `*` (intersection), `in` (membership)
String Concatenation:	`+`
Address/Pointer:	`@` (address of), `^` (dereference pointer)
Assignment:	`:=`
Operator Precedence: Similar to mathematics. Parentheses `()` can be used to force evaluation order.	

Example:

```
var
  a, b, c: Integer;
  isTrue: Boolean;
begin
  a := 10;
  b := 3;
  c := a div b; // c is 3
  isTrue := (a > b) and (c
= 3); // isTrue is True
end;
```

Control Flow & Routines

Conditional Statements

`if` **Statement:**

```
if condition then
  Statement;


if condition then
begin
  Statement1;
  Statement2;
end;
```

`if...else` **Statement:**

```
if condition then
  Statement1
else
  Statement2;


if condition then
begin
  Statement1;
end else
begin
  Statement2;
end;
```

`case` **Statement:** For multiple branches based on an ordinal type (integer, char, boolean, enum).

```
case expression of
  Value1: Statement1;
  Value2, Value3: Statement2;
  LowValue..HighValue: Statement3;
  else
    ElseStatement;
end;
```

- `else` is optional.

Example `if`:

```
var
  x: Integer;
begin
  x := 15;
  if x > 10 then
    Writeln('x is greater than 10')
  else
    Writeln('x is 10 or less');
end;
```

Example `case`:

```
var
  Grade: Char;
begin
  Grade := 'B';
  case Grade of
    'A': Writeln('Excellent');
    'B', 'C': Writeln('Good');
    'D'..'F': Writeln('Poor');
    else
      Writeln('Invalid Grade');
  end;
end;
```

Tip: Use `begin...end;` blocks even for single statements in `if` / `else` for better readability and easier future expansion.

Best Practice: Use `case` for multiple checks against a single variable; it's often more readable and efficient than nested `if` statements.

Loop Statements

`for` **Loop:** Iterates a fixed number of times. Loop variable must be an ordinal type.

```
for LoopVar := StartValue to EndValue do
  Statement;


for LoopVar := StartValue downto EndValue do
begin
  Statement1;
  Statement2;
end;
```

- `LoopVar` is automatically declared local to the loop in newer Pascal standards.

`while` **Loop:** Executes as long as the condition is true (condition checked *before* execution).

```
while condition do
  Statement;


while condition do
begin
  Statement1;
  Statement2;
end;
```

`repeat...until` **Loop:** Executes at least once, repeats until the condition is true (condition checked *after* execution).

```
repeat
  Statement1;
  Statement2;
until condition;
```

Loop Control:

- `break;`: Exits the innermost loop immediately.
- `continue;`: Skips the rest of the current loop iteration and proceeds to the next.

Example `for`:

```
var
  i: Integer;
begin
  for i := 1 to 5 do
    Writeln('Count: ', i);
end;
```

Example `while`:

```
var
  i: Integer;
begin
  i := 0;
  while i < 5 do
  begin
    Writeln('Count: ', i);
    Inc(i); // i := i + 1
  end;
end;
```

Example `repeat`:

```
var
  i: Integer;
begin
  i := 0;
  repeat
    Writeln('Count: ', i);
    Inc(i);
  until i >= 5;
end;
```

Tip: `while` is best when the loop might not need to execute at all. `repeat...until` is good when it must execute at least once. `for` is ideal for known iteration counts.

Procedures & Functions

Procedure: A routine that performs an action but does not return a value.

Declaration:

```
procedure ProcedureName(Parameters);
  // Local declarations (var, const, type)
begin
  // Procedure body
end;
```

Function: A routine that computes and returns a value.

Declaration:

```
function FunctionName(Parameters): ReturnType;
  // Local declarations (var, const, type)
begin
  // Function body
  Result := CalculatedValue; // or FunctionName := CalculatedValue;
end;
```

Parameters:

- **By Value (default):** A copy of the argument is passed. Changes inside the procedure/function do not affect the original variable. `ProcedureName(Param1: ParamType; Param2: ParamType);`

Parameters (cont.):

- **By Reference (`var`):** The memory address of the argument is passed. Changes inside the routine *do* affect the original variable. `ProcedureName(var Param1: ParamType);`
- **Output (`out`):** Similar to `var`, but indicates the parameter is only used for output (value on entry is undefined). `ProcedureName(out Param1: ParamType);`

Example Procedure:

```
procedure Greet(Name: string);
begin
  Writeln('Hello, ', Name, '!');
end;

// Calling:
Greet('World');
```

Example Function:

```
function Max(a, b: Integer): Integer;
begin
  if a > b then
    Result := a
  else
    Result := b;
end;

// Calling:
var
  m: Integer;
begin
  m := Max(10, 20);
  Writeln('Max is: ', m);
end;
```

Scope: Identifiers declared inside a routine are local to that routine and cease to exist when it finishes. Variables declared in the `var` section of the main program or a unit's implementation are global.

Forward Declaration: Needed if routine A calls routine B, but routine B is declared later.

```
procedure B; forward;

procedure A;
begin
  B;
end;

procedure B;
begin
  Writeln('In B');
end;
```

Tip: Use procedures for actions, functions for calculations. Use `var` only when you need to modify the argument passed to the routine.

Best Practice: Keep routines short and focused on a single task. Use descriptive names.

`Exit` Statement: Exits the current routine immediately.

```
procedure Example;
begin
  Writeln('Before exit');
  Exit; // Leaves the procedure
  Writeln('After exit'); // Never reached
end;
```

Note: In a function, you can set `Result` (or the function name) before calling `Exit` to specify the return value.

Data Structures & Units

Arrays

Static Arrays: Fixed size determined at compile time.

Declaration:

```
var
  Numbers: array[1..10] of Integer;
  Letters: array[0..255] of Char;
  Matrix: array[1..3, 1..4] of Real;
```

- Index ranges can be any ordinal type.

Accessing Elements: Use square brackets `[]`.

```
begin
  Numbers[1] := 10;
  Matrix[2, 3] := 5.5;
  Writeln(Letters[Ord('A')]); // Access by char ordinal value
end;
```

Dynamic Arrays: Size can change at runtime. Managed automatically (reference counted).

Declaration:

```
var
  DynArray: array of Integer;
  DynMatrix: array of array of string;
```

Managing Dynamic Arrays:

- `SetLength(ArrayVar, Size);`: Resizes the array. Existing elements are preserved up to the minimum of the old and new sizes.
- `Length(ArrayVar);`: Returns the current size (number of elements).
- `High(ArrayVar);`: Returns the upper bound index (`Length - 1`).
- `Low(ArrayVar);`: Returns the lower bound index (always 0).

Example Dynamic Array:

```
var
  Names: array of string;
  i: Integer;
begin
  SetLength(Names, 3);
  Names[0] := 'Alice';
  Names[1] := 'Bob';
  Names[2] := 'Charlie';

  for i := 0 to High(Names) do
    Writeln(Names[i]);

  SetLength(Names, 5); // Names[0..2] kept, Names[3..4] added (empty strings)

  // Array is automatically deallocated when it goes out of scope
end;
```

Tip: Use static arrays when the size is known and fixed at compile time. Use dynamic arrays when the size needs to change or is unknown until runtime.

Best Practice: Avoid very large static arrays on the stack; declare them globally or dynamically allocated if possible.

Array Literals (modern FPC):

```
const
  MyArray: array[1..3] of Integer = (10, 20, 30);

var
  AnotherArray: array of string;
begin
  AnotherArray := ['Apple', 'Banana', 'Cherry'];
end;
```

Records

Definition: A composite data type that groups related fields of potentially different types.

Declaration:

```
type
  TPerson = record
    Name: string;
    Age: Integer;
    IsStudent: Boolean;
  end;

var
  Person1: TPerson;
```

Accessing Fields: Use the dot `.` operator.

```
begin
  Person1.Name := 'Alice';
  Person1.Age := 30;
  Person1.IsStudent := False;

  Writeln(Person1.Name, ' is ', Person1.Age, ' years old.');
end;
```

`with` **Statement:** Simplifies access to record fields within a block.

```
begin
  with Person1 do
  begin
    Name := 'Bob';
    Age := 25;
    IsStudent := True;
  end;
end;
```

- **Caution:** Use `with` carefully, especially with nested records or objects, as it can make code less clear if not obvious which record/object is being referenced.

Records with Methods (Object Pascal extension): Records can contain procedures and functions.

```
type
  TPoint = record
    X, Y: Real;
    procedure Init(AX, AY: Real);
    function Distance(Other: TPoint): Real;
  end;

// Implementation of methods goes in the implementation section
of a unit.
```

Example Record with Method:

```
// In Interface section of Unit
type
  TPoint = record
    X, Y: Real;
    procedure Init(AX, AY: Real);
  end;

// In Implementation section of Unit
procedure TPoint.Init(AX, AY: Real);
begin
  X := AX;
  Y := AY;
end;

// Usage:
var
  Pt: TPoint;
begin
  Pt.Init(10.0, 20.0);
  Writeln('Point: (', Pt.X:0:1, ', ', Pt.Y:0:1, ')');
end;
```

Tip: Records are value types (like basic types). When you assign one record variable to another, the entire content is copied. Objects are reference types.

Sets

Definition: An unordered collection of unique elements of the same ordinal type.

Declaration:

```
type
  TDigitSet = set of 0..9;
  TCharSet = set of Char;

var
  Digits: TDigitSet;
  Vowels: set of Char;
```

- Base type must be ordinal with no more than 256 possible values.

Set Literals: Use square brackets `[]`.

```
begin
  Digits := [0, 2, 4, 6, 8];
  Vowels := ['A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u'];
  Digits := []; // Empty set
end;
```

Set Operations:

- **Union:** `+` (combines elements)
- **Difference:** `-` (elements in the first set but not the second)
- **Intersection:** `*` (elements common to both sets)
- **Membership:** `in` (checks if an element is in a set)

Set Operations (cont.):

- **Assignment:** `:=`
- **Comparison:** `=`, `<>` (equality), `<=` (subset), `>=` (superset)
- **Adding/Removing:** `Include(SetVar, Element);`, `Exclude(SetVar, Element);`

Example Set Operations:

```
var
  Set1, Set2, Set3: set of 1..10;
  i: Integer;
begin
  Set1 := [1, 2, 3, 4, 5];
  Set2 := [4, 5, 6, 7];

  Set3 := Set1 + Set2; // Set3 is [1, 2, 3, 4, 5, 6, 7]
  Set3 := Set1 - Set2; // Set3 is [1, 2, 3]
  Set3 := Set1 * Set2; // Set3 is [4, 5]

  if 3 in Set1 then
    Writeln('3 is in Set1');

  Include(Set1, 6); // Set1 is now [1, 2, 3, 4, 5, 6]
  Exclude(Set1, 1); // Set1 is now [2, 3, 4, 5, 6]

  if Set2 <= Set1 then // Is Set2 a subset of Set1?
    Writeln('Set2 is a subset of Set1'); // No, 7 is not in Set1
end;
```

Tip: Sets are very efficient for checking membership of small ordinal ranges. They are often used instead of boolean arrays or multiple `or` conditions.

Units

Definition: A compilation unit that allows code organization, modularity, and separate compilation.

Structure:

```
unit UnitName;

interface
  // Public declarations (visible to other units/programs that 'use' this unit)
  // Types, constants, variables, procedure/function headers, class/object definitions

implementation
  // Private declarations (only visible within this unit)
  // Full procedure/function bodies declared in interface, private routines
  // Unit initialization/finalization (optional)

initialization
  // Code executed when the unit is loaded

finalization
  // Code executed when the program/library using the unit is unloaded

end.
```

Using Units: Add the unit name to the `uses` clause.

```
program MyProgram;
uses
  MyUnit, SysUtils;

begin
  // Use types, procedures, functions declared in MyUnit and SysUtils
end.
```

Example Unit (`MyUnit.pas`):

```
unit MyUnit;

interface

type
  TMessageType = (Info, Warning, Error);

procedure LogMessage(Msg: string; MsgType: TMessageType);
function GetVersion: string;

implementation

// Private helper procedure
procedure InternalLog(Msg: string);
begin
  Writeln('INTERNAL: ', Msg);
end;

procedure LogMessage(Msg: string; MsgType: TMessageType);
begin
  case MsgType of
    Info:    Writeln('INFO: ', Msg);
    Warning: Writeln('WARNING: ', Msg);
    Error:   Writeln('ERROR: ', Msg);
  end;
  InternalLog('Logged: ' + Msg);
end;

function GetVersion: string;
begin
  Result := '1.0';
end;

initialization
  Writeln('MyUnit Initialized');

finalization
  Writeln('MyUnit Finalized');

end.
```

Example Program Using the Unit:

```
program TestMyUnit;
uses
  MyUnit;

begin
  LogMessage('Starting program', Info);
  Writeln('Unit Version: ', GetVersion);
  LogMessage('Something went wrong', Warning);
end.
```

- When compiled and run, you'll see output from `initialization`, procedure calls, and `finalization`.

Dependencies: Units listed in the `uses` clause of the `interface` section affect the public interface of the unit. Units listed only in the `implementation` section do not affect the public interface but are needed for the unit's internal code.

Tip: Group related procedures, functions, types, and constants into units. This improves code organization and reusability.

Best Practice: Keep the `interface` section clean and expose only what is necessary for users of the unit. Implement details in the `implementation` section.

OOP & Advanced Topics

Objects and Classes

Object Types (Legacy): Based on Object Pascal, closer to C++ structures with methods.

Declaration:

```
type
  TMyObject = object
    Field1: Integer;
    procedure Method1;
  end;
```

- Allocated on stack or heap. Not reference-counted by default.

Class Types (Modern): Based on Delphi/Object Pascal, closer to Java/C# classes. Reference-counted by default (automatic memory management via `TObject` hierarchy).

Declaration:

```
type
  TMyClass = class
    Field1: Integer; // Published, Public, Protected, Private, Strict Private
    procedure Method1; // Public, Protected, Private, Strict Private
    constructor Create; // Special method for object creation
    destructor Destroy; override; // Special method for object destruction
  end;
```

Visibility Specifiers (Classes):

- `published`: Visible at runtime (for properties in RTTI/streaming).
- `public`: Accessible from anywhere.
- `protected`: Accessible within the class and its descendants.
- `private`: Accessible only within the unit where the class is declared.
- `strict private`: Accessible only within the class itself.

Instantiation & Usage (Classes):

```
var
  Obj: TMyClass;
begin
  Obj := TMyClass.Create; // Call constructor
  Obj.Field1 := 10;
  Obj.Method1;
  Obj.Free; // Call destructor (important!)
end;
```

- Use `Obj.Free` instead of `Dispose(Obj)` for classes.

Example Class:

```
// Interface section
type
  TPerson = class
  private
    FName: string;
    FAge: Integer;
  public
    constructor Create(AName: string; AAge: Integer);
    destructor Destroy; override;
    procedure DisplayInfo;
    property Name: string read FName write FName; // Property
    property Age: Integer read FAge; // Read-only property
  end;

// Implementation section
constructor TPerson.Create(AName: string; AAge: Integer);
begin
  inherited Create; // Call parent constructor
  FName := AName;
  FAge := AAge;
end;

destructor TPerson.Destroy; // No need to call inherited Destroy unless overridden
begin
  Writeln('Destroying Person: ', FName);
  // Clean up resources if any
  inherited Destroy; // Call parent destructor (good practice)
end;

procedure TPerson.DisplayInfo;
begin
  Writeln('Name: ', FName, ', Age: ', FAge);
end;

// Usage:
var
  P: TPerson;
begin
  P := TPerson.Create('Eva', 28);
  P.DisplayInfo;
  Writeln('Accessing property: ', P.Name);
  // P.Age := 30; // Error: property is read-only
  P.Free;
end;
```

Inheritance: Define new classes based on existing ones using `:`. Methods can be `override`n or `virtual` / `abstract`.

```
type
  TStudent = class(TPerson) // TStudent inherits from TPerson
    StudentNo: string;
    procedure DisplayInfo; override; // Overriding parent method
  end;
```

Polymorphism: Using virtual methods. A variable of a parent class type can hold an object of a descendant class type, and calling a virtual method executes the descendant's version.

```
procedure TPerson.DisplayInfo; virtual; // Declare as virtual

// In implementation of TStudent:
procedure TStudent.DisplayInfo; override;
begin
  inherited DisplayInfo; // Call parent version
  Writeln('Student No: ', StudentNo);
end;

// Usage:
var
  P: TPerson;
begin
  P := TStudent.Create('David', 20); // Assign descendant to parent variable
  TStudent(P).StudentNo := 'S123'; // Cast to access descendant fields
  P.DisplayInfo; // Calls TStudent.DisplayInfo due to virtual/override
  P.Free;
end;
```

Abstract Classes/Methods: Declared with `abstract`. Cannot be instantiated directly. Must be implemented by non-abstract descendants.

```
type
  TShape = class(TObject)
    procedure Draw; abstract;
  end;
```

Tip: Prefer classes over objects for modern applications, especially when automatic memory management is beneficial. Always pair `Create` with `Free`.

Best Practice: Use properties to control access to internal fields, even if it's just a simple read/write. This allows adding validation or logic later without changing the class interface.

File Handling

File Types:

- `TextFile`: For plain text files.
- `File of Type`: For binary files storing records or specific types.
- `File`: Untyped binary files (for low-level operations like copy).

Basic Text File I/O (`TextFile`):

1. Declare file variable: `var MyFile: TextFile;`
2. Assign file name: `AssignFile(MyFile, 'data.txt');`
3. Open file: `Reset(MyFile);` (for reading) or `Rewrite(MyFile);` (for writing, creates/overwrites) or `Append(MyFile);` (for appending).
4. Read/Write: `Read(MyFile, ...);`, `Readln(MyFile, ...);`, `Write(MyFile, ...);`, `Writeln(MyFile, ...);`
5. Close file: `CloseFile(MyFile);`

Error Checking: Use `{$I-}` directive before file operations and check `IOResult` afterwards, or use `SysUtils.IOResult`. Better yet, use `try...except` blocks around file operations.

Example Text File Write:

```
var
  F: TextFile;
  FileName: string = 'output.txt';
begin
  AssignFile(F, FileName);
  {$I-}
  Rewrite(F);
  {$I+}
  if IOResult = 0 then
  begin
    Writeln(F, 'This is the first line.');
    Writeln(F, 'This is the second line.');
    CloseFile(F);
    Writeln('File ''', FileName, ''' written successfully.');
  end
  else
    Writeln('Error writing file ''', FileName, '''');
end;
```

Example Text File Read:

```
var
  F: TextFile;
  FileName: string = 'output.txt';
  Line: string;
begin
  AssignFile(F, FileName);
  {$I-}
  Reset(F);
  {$I+}
  if IOResult = 0 then
  begin
    Writeln('Reading file ''', FileName, ''':');
    while not Eof(F) do // Check for End of File
    begin
      Readln(F, Line);
      Writeln(Line);
    end;
    CloseFile(F);
  end
  else
    Writeln('Error reading file ''', FileName, ''' (does it
exist?)');
end;
```

Binary File I/O (`File of Type`):

1. Declare file variable: `var MyBinFile: file of TPerson;` (using the TPerson record from before)
2. Assign file name: `AssignFile(MyBinFile, 'people.dat');`
3. Open file: `Reset(MyBinFile);` or `Rewrite(MyBinFile);`
4. Read/Write: `Read(MyBinFile, RecordVar);`, `Write(MyBinFile, RecordVar);`
5. Positioning: `Seek(MyBinFile, RecordIndex);` (0-based index)
6. Get position: `FilePos(MyBinFile);`
7. Get size: `FileSize(MyBinFile);` (number of records)
8. Close file: `CloseFile(MyBinFile);`

Example Binary File Write:

```
var
  F: file of TPerson;
  P: TPerson;
begin
  AssignFile(F, 'people.dat');
  Rewrite(F);

  P.Name := 'Alice'; P.Age := 30; Write(F, P);
  P.Name := 'Bob';   P.Age := 25; Write(F, P);
  P.Name := 'Charlie'; P.Age := 35; Write(F, P);

  CloseFile(F);
  Writeln('Binary file written.');
end;
```

Example Binary File Read & Seek:

```
var
  F: file of TPerson;
  P: TPerson;
begin
  AssignFile(F, 'people.dat');
  Reset(F); // Open existing binary file

  if FileSize(F) > 1 then
  begin
    Seek(F, 1); // Move to the second record (index 1)
    Read(F, P);
    Writeln('Read record at index 1: Name=', P.Name, ', Age=',
P.Age);
  end;

  CloseFile(F);
end;
```

Tip: Always close files when you are finished with them using `CloseFile`. Use `Eof` for text files and `Eof(FileVar)` or checking `FilePos < FileSize` for binary files to detect the end.

Best Practice: Use `try...finally` blocks to ensure `CloseFile` is always called, even if errors occur during file processing.

```
var F: TextFile;
begin
  AssignFile(F, 'test.txt');
  try
    Rewrite(F);
    Writeln(F, 'Test');
    // Potential error might occur here
  finally
    CloseFile(F);
  end;
end;
```