NitroPascal Developer Manual

Version: 1.0
Last Updated: 2025-10-07

Table of Contents

---

1. Introduction

What is NitroPascal?

NitroPascal is a modern Pascal dialect designed to transpile cleanly to C++, leveraging Zig for building. It provides Pascal's clarity and readability while generating straightforward C++ code. NitroPascal offers a fresh, streamlined language that maps directly to modern C++ constructs.

Core Mission: Provide a modern Pascal-like language that transpiles to clean, readable C++, enabling:

- Developers who prefer Pascal syntax to target modern platforms
- Clean, maintainable C++ code generation
- Cross-platform compilation via Zig build system
- Direct interop with C/C++ libraries and ecosystems
- Modern development without IDE dependencies

Key Features

Pascal Syntax, C++ Semantics

NitroPascal looks like Pascal but compiles directly to C++:

```
// NitroPascal source
routine add(const x, y: int): int;
begin
  return x + y;
end;
```

Transpiles to:

```cpp
// Generated C++
int32_t add(const int32_t x, const int32_t y) {
  return x + y;
}
```

Modern Language Features

- **Case-sensitive** - Modern convention (unlike traditional Pascal)
- **Explicit types** - No type inference, all types declared
- **Minimal keywords** - Small core language (~41 keywords)
- **Direct C/C++ interop** - Via `extern` declarations
- **Three compilation modes** - program (exe), module (obj), library (dll/so)
- **Conditional compilation** - C-style preprocessor (`#ifdef`, `#define`, etc.)
- **Compiler directives** - Fine-grained build control (`$target`, `$optimize`, etc.)

Zig-Powered Build System

- **Cross-compilation** - Target any platform from any platform
- **Modern toolchain** - No complex IDE dependencies
- **Fast builds** - Leveraging Zig's incremental compilation
- **C++ standard library** - Full access to modern C++20 features

Design Philosophy

1. Simplicity

Small core language - libraries do the work. No built-in I/O, string functions, or math operations. Everything is explicit via external declarations.

2. Transparency

What you write is what you get in C++. No hidden magic, no implicit conversions, no runtime overhead.

3. No Magic

Explicit over implicit. Every operation is visible in the source code.

4. Interoperability

Seamless C/C++ integration. Direct access to system libraries and existing C++ codebases.

5. Practicality

Built for real programs, not academic exercises.

Who Should Use NitroPascal?

Perfect For:

- **C++ developers** who prefer Pascal syntax
- **System programmers** needing direct C interop
- **Cross-platform developers** targeting multiple OSes
- **Projects with algorithms and business logic**

Not Ideal For:

- Projects requiring full Delphi RTL/VCL compatibility
- Rapid GUI development (no visual components)
- Projects needing traditional Pascal case-insensitivity

---

2. Getting Started

Installation

Requirements

- **Zig compiler** (bundled with NitroPascal distribution)
- **Windows, Linux, or macOS**
- **C++ compiler** (provided by Zig)

Building from Source

```bash
# Clone repository
git clone https://github.com/tinyBigGAMES/NitroPascal.git
cd NitroPascal

# Build with Delphi (Windows)
# Open NitroPascal.groupproj in Delphi IDE
# Build the nitro project

# The nitro.exe executable will be in:
# src\nitro\Win64\Release\nitro.exe
```

PATH Configuration

Add the NitroPascal `bin` directory to your system PATH:

Windows:

```cmd
set PATH=%PATH%;C:\Path\To\NitroPascal\bin
```

Linux/macOS:

```bash
export PATH=$PATH:/path/to/nitropascal/bin
```

Your First Project

Create a new project:

```bash
nitro init MyFirstApp
cd MyFirstApp
```

This creates:

```text
MyFirstApp/
  ├── src/
  │   └── MyFirstApp.pas  (entry point)
  ├── generated/          (C++ output)
  ├── zig-out/
  │   └── bin/            (compiled executable)
  └── build.zig           (build configuration)
```

Hello World Example

The generated `MyFirstApp.pas` contains:

```
program MyFirstApp;

extern <stdio.h> routine printf(format: ^char; ...): int;

begin
  printf("Hello from NitroPascal!\n");
  ExitCode := 0;
end.
```

Build and run:

```bash
nitro build
nitro run
```

Output:

```text
Hello from NitroPascal!
```

Project Structure

Directory Layout

```text
MyProject/
├── src/                    # Your .pas source files
│   ├── MyProject.pas       # Entry point (program)
│   ├── Utils.pas           # Module (optional)
│   └── MyLib.pas           # Library (optional)
│
├── generated/              # Generated C++ code (auto-created)
│   ├── MyProject.cpp
│   ├── MyProject.h
│   ├── Utils.cpp
│   ├── Utils.h
│   ├── MyLib.cpp
│   └── MyLib.h
│
├── zig-out/               # Build artifacts (auto-created)
│   └── bin/
│       └── MyProject.exe  # Final executable
│
├── .zig-cache/            # Build cache (auto-created)
│
└── build.zig              # Zig build configuration (auto-updated)
```

File Naming Conventions

- **Programs:** `ProjectName.pas` or `main.pas`
- **Modules:** `ModuleName.pas`
- **Libraries:** `LibraryName.pas`

Entry Point Detection

The compiler looks for entry points in this order:

1. `{ProjectName}.pas` (matches project directory name)
2. `main.pas` (fallback)

---

3. CLI Reference

Command Overview

```text
nitro <command> [options]

Commands:
  init              Create a new NitroPascal project
  build             Compile Pascal source to C++ and build executable
  run               Execute the compiled program
  clean             Remove all generated files
  convert-header    Convert C header file to Pascal unit
  version           Display version information
  help              Display help message
```

nitro init

Create a new NitroPascal project.

Syntax

```bash
nitro init <project-name>
```

Parameters

- `<project-name>` - Name of the project to create

Example

```bash
nitro init GameEngine
```

Creates:

```text
GameEngine/
  ├── src/GameEngine.pas
  ├── generated/
  └── build.zig
```

Generated Entry Point

```
program GameEngine;

extern <stdio.h> routine printf(format: ^char; ...): int;

begin
  printf("Hello from NitroPascal!\n");
  ExitCode := 0;
end.
```

nitro build

Compile Pascal source to C++ and build executable.

Syntax

```bash
nitro build
```

Process

1. **Transpilation:** Pascal → C++ (`.pas` → `.cpp/.h`)
2. **Build Configuration:** Updates `build.zig`
3. **Compilation:** Zig compiles C++ to native binary

Output

```text
Entry point: MyProject.pas
Compiling NitroPascal to C++...

✓ Transpilation complete

✓ Updated build.zig

Building with Zig...
  [3/3] MyProject.cpp

✓ Build completed successfully!
```

Error Handling

If compilation fails, detailed error messages are shown:

```text
MyProject.pas(10,5): error: Expected ";"
  x = y + z
    ^
Build failed!
```

nitro run

Execute the compiled program.

Syntax

```bash
nitro run
```

Prerequisites

- Project must be built first (`nitro build`)
- Executable must exist in `zig-out/bin/`

Example

```bash
$ nitro run

Running MyProject...

Hello from NitroPascal!

Program exited with code: 0
```

nitro clean

Remove all generated files and build artifacts.

Syntax

```bash
nitro clean
```

Removes

- `generated/` directory
- `.zig-cache/` directory
- `zig-out/` directory

Output

```text
Cleaning project...

✓ Removed generated/
✓ Removed zig-cache/
✓ Removed zig-out/

✓ Clean completed successfully!
```

nitro convert-header

Convert C header file to Pascal unit (planned feature).

Syntax

```bash
nitro convert-header <input.h> [options]
```

Options

- `--output <file>` - Output Delphi unit filename
- `--library <name>` - Target library name for external declarations
- `--convention <type>` - Calling convention (cdecl, stdcall) [default: cdecl]

Example

```bash
nitro convert-header sqlite3.h --output USQLite3.pas --library sqlite3
```

Status

Not yet implemented. Planned for future release.

nitro version

Display version information.

Syntax

```bash
nitro version
```

Output

```text
 _  _ _ _           ___                  _ ™
| \| (_) |_ _ _ ___| _ \__ _ ___ __ __ _| |
| .` | |  _| '_/ _ \  _/ _` (_-</ _/ _` | |
|_|\_|_|\__|_| \___/_| \__,_/__/\__\__,_|_|
      Modern Pascal * C Performance

Version 1.0.0.0

Copyright © 2025-present tinyBigGAMES™ LLC
All Rights Reserved.

Licensed under BSD 3-Clause License
```

nitro help

Display help message.

Syntax

```bash
nitro help
```

Aliases

- `nitro -h`
- `nitro --help`

---

4. Language Specification

Lexical Structure

Character Set

NitroPascal uses **UTF-8** encoding. Source files must be valid UTF-8.

Case Sensitivity

NitroPascal is **case-sensitive** (unlike traditional Pascal):

```
var count: int;      // Different from
var Count: int;      // Different from
var COUNT: int;      // All three are distinct

routine test();      // ✓ Correct
Routine test();      // ✗ Error - keyword must be lowercase
```

Naming Conventions

Keywords: lowercase (`module`, `routine`, `var`, `type`, `begin`, `end`)

Built-in types: lowercase (`int`, `uint`, `double`, `bool`, `string`)

User types: PascalCase (`Point`, `Vector`, `MyRecord`)

Variables/routines: camelCase (`userName`, `itemCount`, `calculateTotal`)

Comments

```
// Single-line comment (C++ style)

(*
   Multi-line comment
   (Pascal style)
*)

{
  Alternative multi-line comment
  (Curly brace style)
}
```

Note: Comments do not nest.

Identifiers

```
Identifier = Letter { Letter | Digit | "_" } .
Letter     = "a".."z" | "A".."Z" .
Digit      = "0".."9" .
```

Rules:

- Must start with a letter
- Can contain letters, digits, underscores
- Case-sensitive
- Cannot be a reserved keyword

Valid:

```
count
userName
my_variable
Item2
_internal  // Leading underscore allowed
```

Invalid:

```
2count     // Starts with digit
my-var     // Contains hyphen
routine    // Reserved keyword
```

Keywords (Reserved)

NitroPascal has **41 reserved keywords** (must be lowercase):

```text
and         array       begin       break       case
const       continue    div         do          downto
else        end         extern      false       finalize
for         halt        if          import      library
mod         module      nil         not         of
or          program     public      repeat      return
routine     shl         shr         then        to
true        type        until       var         while
xor
```

Operators and Delimiters

Arithmetic: `+` `-` `*` `/` `div` `mod` `shl` `shr`

Comparison: `=` `<>` `<` `>` `<=` `>=`

Logical: `and` `or` `not` `xor`

Assignment: `:=`

Pointer: `^` `@`

Delimiters: `(` `)` `[` `]` `.` `,` `:` `;` `..`

Special: `$` for directives, `#` for preprocessor

Literals

Integer Literals

```
42          // Decimal
0xFF        // Hexadecimal (prefix 0x)
0b1010      // Binary (prefix 0b)
```

Floating-Point Literals

```
3.14
2.5e10
1.23E-5
.5          // 0.5
```

Character Literals

```
'A'
'9'
'\n'        // Newline
'\t'        // Tab
'\\'        // Backslash
'\''        // Single quote
```

String Literals

```
"Hello, World!"
"Line 1\nLine 2"
"Tab\there"
"Quote: \"Hello\""
""           // Empty string
```

Escape sequences:

- `\n` - newline
- `\t` - tab
- `\r` - carriage return
- `\\` - backslash
- `\"` - double quote
- `\'` - single quote
- `\0` - null character

Boolean Literals

```
true
false
```

Nil Literal

```
nil          // Null pointer
```

Type System

Built-in Types

Type	C++ Type	Size	Range
`int`	`int32_t`	4 bytes	-2,147,483,648 to 2,147,483,647
`uint`	`uint32_t`	4 bytes	0 to 4,294,967,295
`int64`	`int64_t`	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`uint64`	`uint64_t`	8 bytes	0 to 18,446,744,073,709,551,615
`int16`	`int16_t`	2 bytes	-32,768 to 32,767
`uint16`	`uint16_t`	2 bytes	0 to 65,535
`byte`	`uint8_t`	1 byte	0 to 255

Type	C++ Type	Size	Range
`double`	`double`	8 bytes	IEEE 754 double
`float`	`float`	4 bytes	IEEE 754 float
`bool`	`bool`	1 byte	`true` or `false`
`char`	`char`	1 byte	ASCII character
`string`	`std::string`	Variable	Managed string type
`pointer`	`void*`	4/8 bytes	Generic pointer

Type Aliases

```
type
  Counter = int;
  Name = string;
```

Transpiles to:

```cpp
using Counter = int32_t;
using Name = std::string;
```

Enumerations

```
type
  Color = (red, green, blue);
  Status = (idle, running, stopped);
```

Transpiles to:

```cpp
enum Color {
  red,
  green,
  blue
};

enum Status {
  idle,
  running,
  stopped
};
```

Enum values are integers starting from 0.

Records (Structs)

Records provide structured data types similar to C structs.

Syntax:

```
type
  Point = record
    x, y: double;
  end;

  Person = record
    name: string;
    age: int;
    active: bool;
  end;
```

Transpiles to:

```cpp
struct Point {
  double x, y;
};

struct Person {
  std::string name;
  int32_t age;
  bool active;
};
```

Arrays (Fixed Size)

Arrays provide fixed-size collections of elements.

Syntax:

```
type
  Buffer = array[0..1023] of byte;
  Matrix = array[0..9, 0..9] of double;
```

Transpiles to:

```cpp
using Buffer = std::array<uint8_t, 1024>;
using Matrix = std::array<std::array<double, 10>, 10>;
```

Pointers

```
type
  PInt = ^int;
  PPoint = ^Point;
```

Transpiles to:

```cpp
using PInt = int32_t*;
using PPoint = Point*;
```

String Type

The `string` type is a **managed type** that maps to `std::string`.

```
var
  name: string := "Alice";
  greeting: string;
begin
  greeting := "Hello, " + name;  // Concatenation
end;
```

Transpiles to:

```cpp
std::string name = "Alice";
std::string greeting;
greeting = "Hello, " + name;
```

Operations:

- Assignment: `s1 := s2`
- Concatenation: `s1 + s2`
- Comparison: `s1 = s2`, `s1 <> s2`, `s1 < s2`, etc.

Method Calls:

Since `string` maps to `std::string`, you can call any C++ string method:

```
var
  s: string := "Hello, World!";
  len: int;
  sub: string;
begin
  len := s.length();              // Call std::string::length()
  sub := s.substr(0, 5);          // Returns "Hello"
  s.clear();                      // Empty the string
end;
```

Subrange Types

Subrange types allow you to define a restricted range of values for a type.

Syntax:

```
type
  Index = 0..9;
  Letter = 'A'..'Z';
  Percentage = 0..100;
```

Transpiles to:

```cpp
using Index = int32_t;  // Range: 0-9
using Letter = char;     // Range: 'A'-'Z'
using Percentage = int32_t;  // Range: 0-100
```

Note: Range checking is not enforced at runtime - subranges are type aliases with documentation value.

Function Pointer Types

Function pointer types allow you to declare types for routines.

Syntax:

```
type
  MathFunc = routine(const x: double): double;
  CompareFunc = routine(const a, b: int): int;
```

Transpiles to:

```cpp
using MathFunc = double (*)(const double);
using CompareFunc = int32_t (*)(const int32_t, const int32_t);
```

Usage:

```
var
  func: MathFunc;
  ptr: ^routine(const x: int): int;  // Pointer to function
begin
  func := @sqrt;  // Address of routine
end;
```

Compilation Units

Program

A **program** compiles to an executable with a `main()` function.

```
program MyApp;

extern <stdio.h> routine printf(format: ^char; ...): int;

var
  count: int;

begin
  count := 42;
  printf("Count: %d\n", count);
  ExitCode := 0;
end.
```

Rules:

- Must have `begin...end.` block (becomes `main()`)
- Can declare types, constants, variables, routines
- Can import other modules
- Implicit `ExitCode` variable (type `int`, default `0`)
- Implicit `halt(code: int)` routine for early exit

Module

A **module** compiles to object code (`.obj` or `.o`) for static linking.

```
module MathUtils;

extern <math.h> routine sqrt(x: double): double;

public type Point = record
  x, y: double;
end;

public routine distance(const p1, p2: Point): double;
var
  dx, dy: double;
begin
  dx := p2.x - p1.x;
  dy := p2.y - p1.y;
  return sqrt(dx * dx + dy * dy);
end;

routine square(const x: double): double;
begin
  return x * x;
end;

end.
```

Rules:

- No `begin...end.` block
- `public` items exported (in header)
- Non-public items are internal (static)
- Cannot be executed directly
- Must be imported by programs or other modules

Library

A **library** compiles to a shared library (`.dll`, `.so`, `.dylib`).

```
library Calculator;

extern <stdio.h> routine printf(format: ^char; ...): int;

var
  operationCount: int;

public routine add(const a, b: int): int;
begin
  operationCount := operationCount + 1;
  return a + b;
end;

public routine getCount(): int;
begin
  return operationCount;
end;

begin
  // DLL initialization (DLL_PROCESS_ATTACH)
  operationCount := 0;
  printf("Calculator library loaded\n");
end;

finalize
  // DLL cleanup (DLL_PROCESS_DETACH)
  printf("Calculator library unloaded (%d operations)\n", operationCount);
end.
```

Rules:

- `public` items are DLL-exported
- Optional `begin` block for initialization
- Optional `finalize` block for cleanup
- Cannot be executed directly

Declarations

Import Declaration

```
import ModuleName;
```

Imports a NitroPascal module.

Extern Declaration

```
extern <stdio.h> routine printf(format: ^char; ...): int;
extern "mylib.h" routine myFunction(const x: int): int;
extern dll "kernel32.dll" stdcall routine GetCurrentProcessId(): uint;
```

Declares external C/C++ functions.

Type Declaration

```
type
  Counter = int;
  Point = record
    x, y: double;
  end;
```

Constant Declaration

```
const
  maxSize: int = 100;
  pi: double = 3.14159265359;
  appName: string = "MyApp";
```

Variable Declaration

```
var
  x: int;
  y, z: double;
  name: string := "Default";
```

Routine Declaration

```
routine add(const x, y: int): int;
begin
  return x + y;
end;

public routine multiply(const a, b: int): int;
begin
  return a * b;
end;
```

Statements

Compound Statement

```
begin
  statement1;
  statement2;
  statement3;
end;
```

Assignment Statement

```
x := 42;
name := "Alice";
ptr^.value := 100;
```

If Statement

```
if condition then
  statement;

if condition then
  statement1
else
  statement2;

if condition then
begin
  statement1;
  statement2;
end;
```

While Statement

```
while condition do
  statement;

while x < 10 do
begin
  x := x + 1;
  printf("x = %d\n", x);
end;
```

Repeat Statement

```
repeat
  statement;
  statement;
until condition;
```

For Statement

```
for i := 1 to 10 do
  statement;

for i := 10 downto 1 do
  statement;
```

Loop Variable:

- Implicitly declared (scoped to loop)
- Cannot be modified inside loop
- Not accessible after loop

Case Statement

```
case expression of
  value1: statement1;
  value2: statement2;
  value3..value4: statement3;
else
  defaultStatement;
end;
```

No fall-through: Each case implicitly breaks.

Break Statement

```
break;
```

Exits the innermost loop.

Continue Statement

```
continue;
```

Continues to next iteration of loop.

Return Statement

```
return;              // Procedures
return expression;   // Functions
```

Halt Statement

```
halt(exitCode);
```

Terminates program immediately with exit code.

Expressions

Primary Expressions

- Integer literals: `42`, `0xFF`, `0b1010`
- Float literals: `3.14`, `2.5e10`
- String literals: `"Hello"`
- Char literals: `'A'`
- Boolean literals: `true`, `false`
- Nil literal: `nil`
- Identifiers: `variableName`
- Parenthesized: `(expression)`

Postfix Expressions

- Function call: `func(arg1, arg2)`
- Method call: `obj.method(args)`
- Array indexing: `arr[index]`
- Field access: `record.field`
- Pointer dereference: `ptr^`

Unary Expressions

- Unary plus: `+x`
- Unary minus: `-x`
- Logical not: `not flag`
- Address-of: `@variable`

Binary Expressions

Multiplicative:

- Multiply: `a * b`
- Divide: `a / b` (float result)
- Integer divide: `a div b`
- Modulo: `a mod b`
- Shift left: `a shl b`
- Shift right: `a shr b`
- Logical and: `a and b`

Additive:

- Add: `a + b`
- Subtract: `a - b`
- Logical or: `a or b`
- Logical xor: `a xor b`

Relational:

- Equal: `a = b`
- Not equal: `a <> b`
- Less than: `a < b`
- Less or equal: `a <= b`
- Greater than: `a > b`
- Greater or equal: `a >= b`

Type Cast

```
var
  x: int := 42;
  y: double;
begin
  y := double(x);
end;
```

Operator Precedence (Highest to Lowest)

1. `@` (address-of), `^` (dereference), `not`
2. `*` `/` `div` `mod` `shl` `shr` `and`
3. `+` `-` `or` `xor`
4. `=` `<>` `<` `<=` `>` `>=`

Parameter Passing

By Value (Default)

```
routine test(x: int);
begin
  x := x + 1;  // Modifies local copy only
end;
```

By Const Reference

```
routine test(const x: int);
begin
  // x := x + 1;  // ✗ Error - cannot modify const
  writeLn(x);
end;
```

For large types (records, arrays), `const` uses C++ const reference.

By Reference (var)

```
routine increment(var x: int);
begin
  x := x + 1;  // Modifies caller's variable
end;
```

By Reference (out)

```
routine getValue(out result: int);
begin
  result := 42;
end;
```

Note: `out` is semantically the same as `var` in v1.0.

Variadic Parameters

```
extern <stdio.h> routine printf(format: ^char; ...): int;
```

The `...` indicates variadic parameters (C-style).

5. Compiler Directives

Directives Overview

Compiler directives control how NitroPascal transpiles and builds your code. They are specified at the top of your source file using the `$` prefix followed by the directive name and a quoted value.

Syntax:

```
$directive "value"
```

Example:

```
$target "x86_64-windows"
$optimize "release_fast"

program MyApp;
begin
  // Your code here
end.
```

Rules:

- Must appear **before** `program`, `module`, or `library` keyword
- Directive names are case-insensitive
- Values must be enclosed in double quotes
- Multiple directives can appear in any order

$target Directive

Specifies the target platform for cross-compilation.

Syntax

```
$target "<arch>-<os>[-<abi>]"
$target "native"
```

Values

Special:

- `"native"` - Build for current platform (default)

Format: `<arch>-<os>[-<abi>]`

Supported Architectures:

```text
x86_64, x86, aarch64, arm, armeb, aarch64_be, aarch64_32,
arc, avr, bpfel, bpfeb, csky, dxil, hexagon, loongarch32,
loongarch64, m68k, mips, mipsel, mips64, mips64el, msp430,
nvptx, nvptx64
```

Supported Operating Systems:

```text
windows, linux, macos, freebsd, netbsd, openbsd, dragonfly,
wasi, emscripten, cuda, opencl, glsl, vulkan, metal,
amdhsa, ps4, ps5, elfiamcu, tvos, watchos, driverkit,
mesa3d, contiki, aix
```

Supported ABIs:

```text
gnu, musl, msvc, android, eabi, eabihf, ilp32, simulator
```

Examples

```
$target "native"                // Current platform
$target "x86_64-windows"        // 64-bit Windows
$target "x86_64-linux"          // 64-bit Linux
$target "aarch64-macos"         // Apple Silicon macOS
$target "aarch64-linux-gnu"     // ARM64 Linux with GNU libc
$target "x86_64-linux-musl"     // x86_64 Linux with musl libc
$target "wasm32-wasi"           // WebAssembly with WASI
```

Validation

Invalid targets produce compile errors:

```
$target "invalid-platform"  // ✗ Error: Invalid Zig target
$target "x86_64"            // ✗ Error: Missing OS component
```

$optimize Directive

Controls compiler optimization and safety checks.

Syntax

```
$optimize "mode"
```

Values

Mode	Speed	Size	Safety	Debug Info	Use Case
`"debug"`	Slowest	Largest	Full	Yes	Development (default)
`"release_safe"`	Fast	Medium	Full	No	Production
`"release_fast"`	Fastest	Medium	Minimal	No	Performance-critical
`"release_small"`	Fast	Smallest	Minimal	No	Embedded/Size-constrained

Examples

```
$optimize "debug"          // Development build
$optimize "release_safe"   // Production with safety
$optimize "release_fast"   // Maximum performance
$optimize "release_small"  // Minimal binary size
```

Default

If not specified, defaults to `"debug"`.

$exceptions Directive

Enables or disables C++ exception handling.

Syntax

```
$exceptions "on|off"
```

Values

- `"off"` - Disable exceptions (default, smaller/faster code)
- `"on"` - Enable exceptions (required for try/catch)

Examples

```
$exceptions "off"  // Default: -fno-exceptions
$exceptions "on"   // Enables C++ exceptions
```

Note

Disabling exceptions reduces code size and improves performance, but you cannot use try/catch/throw.

$strip_symbols Directive

Controls whether debug symbols are stripped from the final binary.

Syntax

```
$strip_symbols "on|off"
```

Values

- `"off"` - Keep debug symbols (default)
- `"on"` - Strip debug symbols (smaller binaries)

Examples

```
$strip_symbols "off"  // Keep symbols for debugging
$strip_symbols "on"   // Strip for release builds
```

Effect

Can significantly reduce binary size (30-50% reduction typical).

$module_path Directive

Adds directories to search for imported NitroPascal modules.

Syntax

```
$module_path "path1;path2;path3"
```

Format

Semicolon-separated list of paths (relative or absolute).

Examples

```
$module_path "C:\MyModules"
$module_path "lib;shared;../common"
$module_path "/usr/local/np/modules"
```

Behavior

- Multiple `$module_path` directives are cumulative
- Paths can be relative or absolute
- Searched in order when resolving `import` statements

Multiple Directives

```
$module_path "lib"
$module_path "shared"
$module_path "../common"
```

Equivalent to:

```
$module_path "lib;shared;../common"
```

$include_path Directive

Adds directories to search for C/C++ header files referenced in `extern` declarations.

Syntax

```
$include_path "path1;path2;path3"
```

Examples

```
$include_path "C:\Libraries\include"
$include_path "vendor/SDL2/include;vendor/opengl"
```

Use Case

When using C libraries via `extern <header.h>`.

$library_path Directive

Adds directories to search for system libraries during linking.

Syntax

```
$library_path "path1;path2;path3"
```

Examples

```
$library_path "C:\Libraries\lib"
$library_path "/usr/local/lib;/opt/libs"
```

$link_library Directive

Specifies system libraries to link against.

Syntax

```
$link_library "lib1;lib2;lib3"
```

Format

Library name (without `lib` prefix or extension).

Examples

```
$link_library "SDL2"
$link_library "opengl32;gdi32;user32"
$link_library "m;pthread"
```

Platform Notes

- **Windows:** Links `library.lib` or `library.dll`
- **Linux/macOS:** Links `liblibrary.so` or `liblibrary.a`

Complete Example

```
// Cross-compile for Linux with full optimization
$target "x86_64-linux-gnu"
$optimize "release_fast"
$strip_symbols "on"

// Link against system libraries
$link_library "m;pthread"

// Add custom module paths
$module_path "lib;shared"

program MyLinuxApp;

extern <math.h> routine sqrt(x: double): double;

begin
  printf("Square root of 16 is: %f\n", sqrt(16.0));
  ExitCode := 0;
end.
```

Directive Validation

All directive values are validated at parse time:

- **`$target`**: Must be a valid Zig target triple or `"native"`
- **`$optimize`**: Must be one of: `debug`, `release_safe`, `release_fast`, `release_small`
- **`$exceptions`/`$strip_symbols`**: Must be `on` or `off`
- **Path directives**: Paths are not validated until build time

Invalid directives will cause compilation to fail with a helpful error message.

---

6. Conditional Compilation

Conditional Overview

NitroPascal supports C-style preprocessor directives for conditional compilation. This allows you to include or exclude code blocks based on defined symbols, enabling platform-specific code, feature flags, and debug/release configurations.

Preprocessor Directives

#define

Defines a symbol. Once defined, the symbol can be tested with `#ifdef`.

Syntax:

```
#define SYMBOL_NAME
```

Example:

```
#define DEBUG
#define WINDOWS_BUILD
```

Characteristics:

- Symbols are case-sensitive (`DEBUG` - `debug`)
- Symbols are boolean flags (defined or not defined)
- No value assignment (use constants for values)

#undef

Undefines a previously defined symbol.

Syntax:

```
#undef SYMBOL_NAME
```

Example:

```
#define DEBUG
// ... code ...
#undef DEBUG  // DEBUG is no longer defined
```

#ifdef

Includes the following code block only if the symbol is defined.

Syntax:

```
#ifdef SYMBOL_NAME
  // code included if SYMBOL_NAME is defined
#endif
```

Example:

```
#ifdef DEBUG
  extern <stdio.h> routine printf(format: ^char; ...): int;
  routine Log(const msg: ^char);
  begin
    printf("[DEBUG] %s\n", msg);
  end;
#endif
```

#ifndef

Includes the following code block only if the symbol is NOT defined.

Syntax:

```
#ifndef SYMBOL_NAME
  // code included if SYMBOL_NAME is not defined
#endif
```

Example:

```
#ifndef PRODUCTION
  extern <stdio.h> routine printf(format: ^char; ...): int;
  routine DevLog(const msg: ^char);
  begin
    printf("[DEV] %s\n", msg);
  end;
#endif
```

#else

Provides an alternative code block when the condition is false.

Syntax:

```
#ifdef SYMBOL_NAME
  // code if defined
#else
  // code if not defined
#endif
```

Example:

```
#ifdef WINDOWS
  extern dll "kernel32.dll" stdcall routine Sleep(ms: uint);
#else
  extern <unistd.h> routine usleep(us: uint);
#endif
```

#endif

Ends a conditional block started by `#ifdef` or `#ifndef`.

Syntax:

```
#ifdef SYMBOL_NAME
  // conditional code
#endif
```

Conditional Examples

Platform-Specific Code

```
program CrossPlatform;

#define WINDOWS

extern <stdio.h> routine printf(format: ^char; ...): int;

routine ClearScreen();
begin
  #ifdef WINDOWS
    extern dll "kernel32.dll" routine system(const cmd: ^char): int;
    system("cls");
  #else
    extern <stdlib.h> routine system(const cmd: ^char): int;
    system("clear");
  #endif
end;

begin
  ClearScreen();
  printf("Platform: ");
  #ifdef WINDOWS
    printf("Windows\n");
  #else
    printf("Unix/Linux\n");
  #endif
end.
```

Debug vs Release Builds

```
module Logger;

#define DEBUG

extern <stdio.h> routine printf(format: ^char; ...): int;
extern <stdlib.h> routine exit(code: int);

public routine Log(const msg: ^char);
begin
  #ifdef DEBUG
    printf("[DEBUG] %s\n", msg);
  #endif
  // Release build: routine does nothing
end;

public routine Assert(const condition: int; const msg: ^char);
begin
  #ifdef DEBUG
    if condition = 0 then
    begin
      printf("[ASSERT FAILED] %s\n", msg);
      exit(1);
    end;
  #endif
  // Release build: assertions removed
end;

end.
```

Feature Flags

```
module GameEngine;

#define FEATURE_MULTIPLAYER
#define FEATURE_ACHIEVEMENTS
// #define FEATURE_VR  // Commented out - not ready yet

extern <stdio.h> routine printf(format: ^char; ...): int;

public routine InitializeEngine();
begin
  printf("Initializing Game Engine...\n");

  #ifdef FEATURE_MULTIPLAYER
    printf("Multiplayer: Enabled\n");
  #endif

  #ifdef FEATURE_ACHIEVEMENTS
    printf("Achievements: Enabled\n");
  #endif

  #ifdef FEATURE_VR
    printf("VR: Enabled\n");
  #endif
end;

end.
```

Nested Conditionals

```
program ConfigExample;

#define PLATFORM_WINDOWS
#define BUILD_TYPE_DEBUG

extern <stdio.h> routine printf(format: ^char; ...): int;

const
  #ifdef PLATFORM_WINDOWS
    #ifdef BUILD_TYPE_DEBUG
      LOG_LEVEL: int = 4; // Verbose
    #else
      LOG_LEVEL: int = 1; // Errors only
    #endif
  #else
    #ifdef BUILD_TYPE_DEBUG
      LOG_LEVEL: int = 3; // Info
    #else
      LOG_LEVEL: int = 0; // Silent
    #endif
  #endif

begin
  printf("Log Level: %d\n", LOG_LEVEL);
end.
```

Conditional Best Practices

1. Use Descriptive Symbol Names

```
// Good
#define ENABLE_LOGGING
#define TARGET_RASPBERRY_PI
#define USE_HARDWARE_ACCELERATION

// Avoid
#define X
#define FLAG1
#define A
```

2. Document Symbols at the Top

```
program MyApp;

// Compilation Configuration:
// #define DEBUG        - Enable debug output and assertions
// #define TESTING      - Include test routines
// #define USE_OPENGL   - Use OpenGL renderer (default: Vulkan)

#define DEBUG
#define TESTING

// ... rest of code
```

3. Group Related Conditionals

```
// Platform detection
#ifdef WINDOWS
  // Windows-specific code
#endif

#ifdef LINUX
  // Linux-specific code
#endif

#ifdef MACOS
  // macOS-specific code
#endif
```

4. Avoid Deep Nesting

```
// Avoid
#ifdef A
  #ifdef B
    #ifdef C
      #ifdef D
        // too deep!
      #endif
    #endif
  #endif
#endif

// Better: flatten with early returns or separate routines
```

5. Keep Conditional Blocks Small

```
// Good: small, focused conditionals
#ifdef DEBUG
  printf("Debug info\n");
#endif

// Avoid: large blocks that duplicate logic
#ifdef DEBUG
  // 100 lines of code
#else
  // 100 similar lines with minor differences
#endif
```

Conditional Compilation Characteristics

Zero Runtime Overhead

Code in false conditional blocks is **completely skipped** during parsing and never enters the AST:

```
#ifdef NEVER_DEFINED
  // This code is never parsed
  // No tokens generated
  // No AST nodes created
  // Zero runtime cost
#endif
```

Parser-Level Processing

Preprocessor directives are processed **during tokenization**:

1. Lexer encounters `#ifdef SYMBOL`
2. Checks if `SYMBOL` is defined
3. If false, skips all tokens until `#else` or `#endif`
4. Parser never sees the skipped code

Limitations

1. No Expression Evaluation

Only boolean checks (defined or not defined):

```
// NOT SUPPORTED:
#ifdef (WINDOWS and DEBUG)      // ✗ Error
#ifdef WINDOWS || LINUX         // ✗ Error
#ifdef VERSION >= 2             // ✗ Error

// Supported:
#ifdef WINDOWS                  // ✓ OK
  #ifdef DEBUG                  // ✓ OK - use nesting
    // code
  #endif
#endif
```

2. Symbols Have No Values

Symbols are boolean flags only:

```
#define VERSION 2               // ✗ Error - cannot assign value
#define VERSION                 // ✓ OK - symbol is just defined

const VERSION: int = 2;         // ✓ Use const for values
```

3. No Macro Substitution

`#define` does not support text replacement:

```
#define MAX_SIZE 1024           // ✗ Not a macro
const MAX_SIZE: int = 1024;     // ✓ Use const instead
```

4. Must Match Structure

Conditional blocks must respect language structure:

```
// ✗ Error - splits routine declaration
routine MyRoutine(
  #ifdef FEATURE_A
    const paramA: int
  #endif
): int;

// ✓ OK - complete routine conditionally compiled
#ifdef FEATURE_A
  routine MyRoutine(const paramA: int): int;
  begin
    return paramA * 2;
  end;
#endif
```

7. External Interop

C/C++ Integration

NitroPascal can call C and C++ functions via `extern` declarations.

Syntax

```
ExternDecl = "extern" ExternalSource "routine" Identifier
             "(" [ ParameterList ] ")" [ ":" Type ] [ "as" StringLiteral ] ";" .

ExternalSource = "<" Identifier ">"                    (* System header *)
               | StringLiteral                         (* Local header *)
               | "dll" StringLiteral [ CallConv ]      (* DLL/shared library *) .
```

System Headers

```
extern <stdio.h> routine printf(format: ^char; ...): int;
extern <stdlib.h> routine malloc(size: uint64): pointer;
extern <stdlib.h> routine free(ptr: pointer);
extern <math.h> routine sqrt(x: double): double;
extern <string.h> routine strlen(s: ^char): uint64;
```

Transpiles to:

```cpp
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

extern "C" {
  int printf(const char* format, ...);
  void* malloc(size_t size);
  void free(void* ptr);
  double sqrt(double x);
  size_t strlen(const char* s);
}
```

Local Headers

```
extern "MyEngine.h" routine initEngine(): bool;
extern "MyEngine.h" routine updateEngine(dt: double);
```

Transpiles to:

```cpp
#include "MyEngine.h"

extern "C" {
  bool initEngine();
  void updateEngine(double dt);
}
```

DLL/Shared Libraries

```
extern dll "user32.dll" stdcall routine MessageBoxA(
  hwnd: pointer;
  const text: ^char;
  const caption: ^char;
  type: uint
): int;

extern dll "libmylib.so" routine processData(const data: ^byte; len: uint64): int;
```

Windows:

```cpp
#include <windows.h>

__declspec(dllimport) int __stdcall MessageBoxA(
  void* hwnd,
  const char* text,
  const char* caption,
  unsigned int type
);
```

Calling Conventions

stdcall

- **Windows standard call**
- Callee cleans stack
- Common in Win32 API

```
extern dll "kernel32.dll" stdcall routine GetCurrentProcessId(): uint;
```

cdecl

- **C declaration** (default)
- Caller cleans stack
- Standard C calling convention

```
extern dll "mylib.dll" cdecl routine compute(const x: int): int;
```

fastcall

- **Fast call**
- Registers used for parameters
- Compiler-specific optimizations

```
extern dll "fastlib.dll" fastcall routine quickOp(const a, b: int): int;
```

Function Aliasing

Map NitroPascal names to different C function names:

```
extern dll "legacy.dll" routine myAdd(const a, b: int): int as "LegacyAddFunction";
```

The NitroPascal name is `myAdd`, but it calls the DLL function `LegacyAddFunction`.

Variadic Functions

```
extern <stdio.h> routine printf(format: ^char; ...): int;
extern <stdarg.h> routine vprintf(format: ^char; args: pointer): int;
```

The `...` indicates variadic parameters (like C).

Complete Interop Example

```
program SystemInterop;

// Standard C library functions
extern <stdio.h> routine printf(format: ^char; ...): int;
extern <stdlib.h> routine malloc(size: uint64): pointer;
extern <stdlib.h> routine free(ptr: pointer);
extern <string.h> routine strlen(s: ^char): uint64;
extern <string.h> routine strcpy(dest, src: ^char): ^char;

// Windows API functions
#ifdef WINDOWS
  extern dll "kernel32.dll" stdcall routine GetCurrentProcessId(): uint;
  extern dll "kernel32.dll" stdcall routine Sleep(ms: uint);
#else
  extern <unistd.h> routine getpid(): uint;
  extern <unistd.h> routine usleep(us: uint);
#endif

var
  buffer: ^char;
  message: ^char := "Hello from NitroPascal!";
  len: uint64;
  pid: uint;

begin
  // String operations
  len := strlen(message);
  printf("Message length: %llu\n", len);

  // Memory allocation
  buffer := malloc(len + 1);
  strcpy(buffer, message);
  printf("Copied: %s\n", buffer);
  free(buffer);

  // Platform-specific operations
  #ifdef WINDOWS
    pid := GetCurrentProcessId();
    printf("Process ID: %u\n", pid);
    Sleep(1000);
  #else
    pid := getpid();
    printf("Process ID: %u\n", pid);
    usleep(1000000);
  #endif

  ExitCode := 0;
end.
```

8. Architecture & Internals

Compilation Pipeline

High-Level Flow

```text
User Pascal Code (.pas files)
    ↓
[Scanner] → Discovers units and dependencies
    ↓
[Dependency Graph] → Determines build order
    ↓
[Lexer] → Tokenization + Preprocessing
    ↓
[Parser] → AST Generation
    ↓
[Symbol Table Builder] → Semantic Analysis
    ↓
[Module Resolver] → Import Resolution
    ↓
[Code Generator] → C++ Emission
    ↓
[Project Manager] → Updates build.zig
    ↓
Zig Build System → Compiles to native executable
```

Detailed Process

1. Initialization

```text
nitro init MyProject
  ↓
Creates directory structure:
  - src/
  - generated/
  - build.zig
```

2. Scanning (Module Discovery)

```text
nitro build
  ↓
Scanner.ScanDirectory("src/")
  ↓
Finds: MyProject.pas, Utils.pas, Config.pas
  ↓
For each .pas file:
  - Extract module name
  - Extract import declarations
  - Build TUnitInfo catalog
```

3. Dependency Resolution

```text
DependencyGraph.BuildGraph(units)
  ↓
Create edges from import statements
  ↓
Detect circular dependencies (error if found)
  ↓
Topological sort → build order
  ↓
Order: [Config, Utils, MyProject]
```

4. Lexical Analysis

```text
For each unit in build order:
  ↓
Lexer.Create(source, filename)
  ↓
Process preprocessor directives:
  - #define, #undef
  - #ifdef, #ifndef, #else, #endif
  ↓
Skip inactive conditional blocks
  ↓
Generate token stream
  ↓
Collect compiler directives ($target, $optimize, etc.)
```

5. Parsing

```text
Parser.Create(lexer)
  ↓
Parse based on compilation mode:
  - program → TNPProgramNode
  - module → TNPModuleNode
  - library → TNPLibraryNode
  ↓
Recursive descent parsing
  ↓
Generate Abstract Syntax Tree (AST)
```

6. Symbol Table Building

```text
SymbolTableBuilder.Build(ast, moduleName)
  ↓
Walk AST nodes
  ↓
Create symbol entries:
  - Types
  - Constants
  - Variables
  - Routines
  ↓
Track visibility (public/private)
  ↓
Detect duplicate symbols
```

7. Code Generation

```text
CodeGenerator.Generate(ast, moduleName)
  ↓
Walk AST nodes
  ↓
Emit C++ code:
  - Header file (.h)
  - Implementation file (.cpp)
  ↓
Map Pascal constructs → C++ equivalents
  ↓
Generate files in generated/
```

8. Build Configuration

```text
ProjectManager.UpdateBuildZig()
  ↓
Read build settings from directives
  ↓
Generate build.zig:
  - List all .cpp files
  - Set target triple
  - Set optimization mode
  - Configure libraries
```

9. Native Compilation

```text
Execute: zig build
  ↓
Zig compiles C++ → object files
  ↓
Link with C++ standard library
  ↓
Link system libraries
  ↓
Output: zig-out/bin/MyProject.exe
```

AST Structure

Node Hierarchy

```text
TNPASTNode (base class)
├─ TNPProgramNode
├─ TNPModuleNode
├─ TNPLibraryNode
├─ TNPImportNode
├─ TNPExternNode
├─ TNPTypeDeclNode
├─ TNPConstDeclNode
├─ TNPVarDeclNode
├─ TNPRoutineDeclNode
├─ TNPParameterNode
├─ Type Nodes
│  ├─ TNPRecordNode
│  ├─ TNPEnumNode
│  ├─ TNPArrayNode
│  ├─ TNPPointerNode
│  ├─ TNPSubrangeNode
│  └─ TNPFunctionTypeNode
├─ Statement Nodes
│  ├─ TNPCompoundNode
│  ├─ TNPAssignmentNode
│  ├─ TNPIfNode
│  ├─ TNPWhileNode
│  ├─ TNPRepeatNode
│  ├─ TNPForNode
│  ├─ TNPCaseNode
│  ├─ TNPBreakNode
│  ├─ TNPContinueNode
│  ├─ TNPReturnNode
│  └─ TNPHaltNode
└─ Expression Nodes
   ├─ TNPBinaryOpNode
   ├─ TNPUnaryOpNode
   ├─ TNPCallNode
   ├─ TNPMethodCallNode
   ├─ TNPIndexNode
   ├─ TNPFieldAccessNode
   ├─ TNPDerefNode
   ├─ TNPTypeCastNode
   ├─ TNPIdentifierNode
   ├─ Literal Nodes
   │  ├─ TNPIntLiteralNode
   │  ├─ TNPFloatLiteralNode
   │  ├─ TNPStringLiteralNode
   │  ├─ TNPCharLiteralNode
   │  ├─ TNPBoolLiteralNode
   │  └─ TNPNilLiteralNode
   ├─ TNPArrayLiteralNode
   └─ TNPRecordLiteralNode
```

Key Node Properties

TNPProgramNode:

- `Name: string`
- `Declarations: TNPASTNodeList`
- `MainBlock: TNPASTNodeList`

TNPRoutineDeclNode:

- `RoutineName: string`
- `Parameters: TNPASTNodeList`
- `ReturnType: TNPASTNode`
- `LocalVars: TNPASTNodeList`
- `Body: TNPASTNodeList`
- `IsPublic: Boolean`

TNPBinaryOpNode:

- `Left: TNPASTNode`
- `Op: TNPTokenKind`
- `Right: TNPASTNode`

Code Generation

Emitter Architecture

The `TNPCodeGenerator` class walks the AST and emits C++ code:

```
TNPCodeGenerator = class
private
  FHeaderCode: TStringBuilder;
  FImplCode: TStringBuilder;
  FSymbols: TNPSymbolTable;
  FIndentLevel: Integer;

  procedure EmitH(const AText: string);      // Write to header
  procedure EmitCpp(const AText: string);    // Write to impl
  procedure IncIndent();
  procedure DecIndent();

public
  function Generate(const AAST: TNPASTNode; const AModuleName: string): Boolean;
  function GetHeaderCode(): string;
  function GetImplementationCode(): string;
end;
```

Translation Examples

Program → main()

Pascal:

```
program MyApp;
var x: int;
begin
  x := 42;
  ExitCode := 0;
end.
```

C++:

```cpp
#include <cstdint>

int32_t x;

int main() {
  int32_t exitCode = 0;
  x = 42;
  exitCode = 0;
  return exitCode;
}
```

Module → Header + Implementation

Pascal:

```
module Utils;

public type Point = record
  x, y: double;
end;

public routine distance(const p1, p2: Point): double;
begin
  return 0.0; // Simplified
end;

end.
```

Utils.h:

```cpp
#pragma once
#include <cstdint>

struct Point {
  double x, y;
};

double distance(Point p1, Point p2);
```

Utils.cpp:

```cpp
#include "Utils.h"

double distance(Point p1, Point p2) {
  return 0.0;
}
```

Zig Build Integration

Generated build.zig

```
const std = @import("std");

pub fn build(b: *std.Build) void {
    const target = b.resolveTargetQuery(.{
        .cpu_arch = .x86_64,
        .os_tag = .windows,
    });
    const optimize = .ReleaseFast;

    // Create module for C++ sources
    const module = b.addModule("MyProject", .{
        .target = target,
        .optimize = optimize,
        .link_libc = true,
    });

    // C++ compiler flags
    const cpp_flags = [_][]const u8{
        "-std=c++20",
        "-fno-exceptions",
    };

    // Add all generated C++ files
    module.addCSourceFile(.{
        .file = b.path("generated/MyProject.cpp"),
        .flags = &cpp_flags,
    });

    // Create executable
    const exe = b.addExecutable(.{
        .name = "MyProject",
        .root_module = module,
    });

    // Link C++ standard library
    exe.linkLibCpp();

    b.installArtifact(exe);
}
```

Build Customization

Compiler directives directly control `build.zig` generation:

```
$target "aarch64-linux-gnu"
$optimize "release_small"
$strip_symbols "on"
$link_library "m;pthread"
```

Generates:

```
const target = b.resolveTargetQuery(.{
    .cpu_arch = .aarch64,
    .os_tag = .linux,
    .abi = .gnu,
});
const optimize = .ReleaseSmall;

exe.linkSystemLibrary("m");
exe.linkSystemLibrary("pthread");
```

---

9. Complete Examples

Basic Programs

Simple Addition

```
program Addition;

extern <stdio.h> routine printf(format: ^char; ...): int;

var
  x, y, sum: int;

begin
  x := 10;
  y := 20;
  sum := x + y;
  printf("Sum: %d\n", sum);
  ExitCode := 0;
end.
```

Loops and Conditionals

```
program LoopsDemo;

extern <stdio.h> routine printf(format: ^char; ...): int;

var
  i, sum: int;

begin
  sum := 0;

  // For loop
  for i := 1 to 10 do
  begin
    sum := sum + i;
    if i mod 2 = 0 then
      printf("%d is even\n", i)
    else
      printf("%d is odd\n", i);
  end;

  printf("Sum of 1-10: %d\n", sum);
  ExitCode := 0;
end.
```

Working Features

String Manipulation

```
program StringDemo;

extern <stdio.h> routine printf(format: ^char; ...): int;

var
  greeting: string;
  name: string;
  message: string;

begin
  name := "Alice";
  greeting := "Hello, ";
  message := greeting + name + "!";

  printf("%s\n", message.c_str());
  printf("Length: %d\n", int(message.length()));

  ExitCode := 0;
end.
```

Pointer Operations

```
program PointerDemo;

extern <stdio.h> routine printf(format: ^char; ...): int;

var
  x: int;
  ptr: ^int;

begin
  x := 42;
  ptr := @x;

  printf("Value: %d\n", x);
  printf("Via pointer: %d\n", ptr^);

  ptr^ := 100;
  printf("After modification: %d\n", x);

  ExitCode := 0;
end.
```

Parameters and Functions

```
program FunctionDemo;

extern <stdio.h> routine printf(format: ^char; ...): int;

routine add(const a, b: int): int;
begin
  return a + b;
end;

routine swap(var a, b: int);
var
  temp: int;
begin
  temp := a;
  a := b;
  b := temp;
end;

var
  x, y, result: int;

begin
  x := 5;
  y := 10;

  result := add(x, y);
  printf("Sum: %d\n", result);

  printf("Before swap: x=%d, y=%d\n", x, y);
  swap(x, y);
  printf("After swap: x=%d, y=%d\n", x, y);

  ExitCode := 0;
end.
```

Conditional Compilation

```
program ConditionalDemo;

#define DEBUG
#define ENABLE_LOGGING

extern <stdio.h> routine printf(format: ^char; ...): int;

routine DoWork();
begin
  #ifdef DEBUG
    printf("[DEBUG] Starting work...\n");
  #endif

  printf("Doing important work\n");

  #ifdef DEBUG
    printf("[DEBUG] Work complete\n");
  #endif
end;

begin
  #ifdef ENABLE_LOGGING
    printf("Logging is enabled\n");
  #endif

  DoWork();

  ExitCode := 0;
end.
```

External C Libraries

File I/O Example

```
program FileIO;

extern <stdio.h> routine printf(format: ^char; ...): int;
extern <stdio.h> routine fopen(const filename, mode: ^char): pointer;
extern <stdio.h> routine fclose(file: pointer): int;
extern <stdio.h> routine fprintf(file: pointer; format: ^char; ...): int;
extern <stdio.h> routine fgets(buffer: ^char; size: int; file: pointer): ^char;

const
  BUFFER_SIZE: int = 256;

var
  file: pointer;
  buffer: array[0..255] of char;
  filename: ^char;

begin
  filename := "test.txt";

  // Write to file
  file := fopen(filename, "w");
  if file <> nil then
  begin
    fprintf(file, "Hello from NitroPascal!\n");
    fprintf(file, "Line 2\n");
    fprintf(file, "Line 3\n");
    fclose(file);
    printf("File written successfully\n");
  end;

  // Read from file
  file := fopen(filename, "r");
  if file <> nil then
  begin
    printf("File contents:\n");
    while fgets(@buffer[0], BUFFER_SIZE, file) <> nil do
      printf("%s", @buffer[0]);
    fclose(file);
  end;

  ExitCode := 0;
end.
```

Math Library Example

```
program MathDemo;

extern <stdio.h> routine printf(format: ^char; ...): int;
extern <math.h> routine sqrt(x: double): double;
extern <math.h> routine pow(base, exp: double): double;
extern <math.h> routine sin(x: double): double;
extern <math.h> routine cos(x: double): double;

const
  PI: double = 3.14159265359;

var
  radius, area, hypotenuse: double;
  angle, sinVal, cosVal: double;

begin
  // Circle area
  radius := 5.0;
  area := PI * pow(radius, 2.0);
  printf("Circle area (r=%.1f): %.2f\n", radius, area);

  // Pythagorean theorem
  hypotenuse := sqrt(pow(3.0, 2.0) + pow(4.0, 2.0));
  printf("Hypotenuse (3-4-?): %.2f\n", hypotenuse);

  // Trigonometry
  angle := PI / 4.0; // 45 degrees
  sinVal := sin(angle);
  cosVal := cos(angle);
  printf("sin(45°) = %.4f\n", sinVal);
  printf("cos(45°) = %.4f\n", cosVal);

  ExitCode := 0;
end.
```

---

10. Implementation Status

Test Results

Test Suite: 376 total tests across multiple categories
Pass Rate: All core tests passing
Last Updated: 2025-10-07

The NitroPascal test suite validates all implemented language features including:

- Lexer and parser functionality
- Type system (primitives, pointers, arrays, records, enums)
- Control flow statements
- String operations
- Pointer operations
- Parameter passing
- Conditional compilation
- Code generation

Implemented Features

✅ Core Language

Lexical Analysis

- Case-sensitive identifiers
- All 41 keywords
- Comments (single-line and multi-line)
- All literal types (integer, float, string, char, boolean, nil)
- All operators (arithmetic, logical, relational, bitwise)
- Preprocessor directives

Type System

- All primitive types (int, uint, int64, uint64, int16, uint16, byte, double, float, bool, char)
- Type aliases
- Enumerations
- Pointers and pointer operations
- String type (maps to std::string)

Compilation Modes

- Programs (executable with main())
- Modules (object files)
- Libraries (DLLs/shared libraries)

Statements

- Compound statements (begin...end)
- Assignment (:=)
- If-then-else
- While loops
- Repeat-until loops
- For loops (to/downto)
- Case statements (via if-else chains)
- Break/Continue
- Return
- Halt

Expressions

- All binary operations
- All unary operations
- Function calls
- Method calls (for strings and pointers)
- Pointer dereference (^)
- Address-of (@)
- Type casts
- String concatenation and comparison

Declarations

- Constants
- Variables
- Routines (functions/procedures)
- Types
- Import statements
- Extern declarations (C/C++ interop)

External Interop

- System headers (<header.h>)
- Local headers ("header.h")
- DLL/Shared library imports
- Calling conventions (stdcall, cdecl, fastcall)
- Variadic functions (...)
- Function aliasing (as "name")

Compiler Directives

- $target - Target platform specification
- $optimize - Optimization levels
- $exceptions - Exception handling control
- $strip_symbols - Symbol stripping
- $module_path - Module search paths
- $include_path - Include paths
- $library_path - Library search paths
- $link_library - Link libraries

Conditional Compilation

- #define / #undef
- #ifdef / #ifndef
- #else / #endif
- Nested conditionals
- Zero runtime overhead (parser-level)

Build System

- Zig integration
- Cross-compilation support
- Automatic build.zig generation
- C++ standard library linking

CLI Tools

- nitro init - Project creation
- nitro build - Compilation
- nitro run - Execution
- nitro clean - Cleanup
- nitro version - Version info
- nitro help - Help text

Types

- Array types - Full support for fixed-size arrays, multi-dimensional arrays
- Record types - Complete struct/record implementation with field access
- Subrange types - Type aliases with range documentation
- Function pointer types - Function type declarations and routine pointers

Known Issues

🟢 Loop Variable Semantics

Issue: Generated C++ declares a local loop variable that shadows any global variable.

Pascal:

```
var i: int;
for i := 1 to 5 do
  // use i
```

Generated C++:

```cpp
int32_t i;  // global, never used
for (int i = 1; i <= 5; i++) { ... }  // local shadow
```

Impact:

- Global variable with same name as loop variable is shadowed and unused
- Loop variable not accessible after loop ends
- Differs from some Pascal implementations

Status: By design - matches C++ semantics.

🟢 DLL String Return ABI

Issue: Returning `std::string` by value from DLL exports works for same-toolchain scenarios but may cause ABI issues across different compilers.

Example:

```
library StringLib;
public routine GetName(): string;
begin
  return "MyLib";
end;
end.
```

Impact:

- Works perfectly for NitroPascal-to-NitroPascal interop
- May crash or corrupt data when called from different C++ toolchains
- Cannot be easily called from C, C#, or other languages

Workaround: Use `^char` return types for DLL exports that need to be called from external code.

Status: Documented limitation.

🟢 String Indexing Bounds Checking

Issue: String indexing uses `std::string::operator[]` which has no bounds checking.

Example:

```
var s: string;
var c: char;
begin
  s := "Hello";
  c := s[100];  // undefined behavior, no runtime check
end.
```

Impact:

- Fast performance (no overhead)
- Matches C++ semantics
- Out-of-bounds access may crash, return garbage, or appear to work

Status: By design - matches C++ semantics.

🟢 Program Structure Edge Cases

Issue: Some edge cases in program structure may fail parsing:

- Empty programs with no statements
- Programs with only variable declarations
- Some routine definition patterns

Status: Minor issue, being investigated

Limitations

No Standard Library

NitroPascal has **no standard library**. All functionality comes from C/C++ via `extern` declarations.

Implications:

- No built-in I/O functions (WriteLn, ReadLn)
- No string manipulation routines
- No file handling utilities
- No math functions
- No collections

Philosophy: This is intentional - keeps the language simple and forces explicit dependencies.

Workaround: Use C standard library via `extern` declarations.

No Type Inference

All types must be explicitly declared:

```
var x: int := 42;        // ✓ OK
var y := 42;             // ✗ Error - type required
```

No Inline Variables

Variable declarations must be in `var` sections:

```
routine test();
var
  x: int;     // ✓ OK - in var section
begin
  var y: int; // ✗ Error - inline var not supported
end;
```

No Properties

Traditional Pascal properties are not supported:

```
type
  TMyClass = class
    property Value: int read FValue write SetValue;  // ✗ Not supported
  end;
```

Workaround: Use explicit getter/setter routines.

No Classes with Inheritance

Only simple structs (records) are supported. No class hierarchies, virtual methods, or inheritance.

Workaround: Use C++ classes via extern declarations.

No Generics

Template/generic types are not supported in v1.0:

```
type
  List<T> = ...;  // ✗ Not supported
```

Status: Planned for v2.0.

No Dynamic Arrays

Only fixed-size arrays are supported:

```
var
  items: array of int;  // ✗ Not supported (dynamic)
  buffer: array[0..99] of int;  // ✓ OK (fixed size)
```

Status: Dynamic arrays planned for v2.0.

Case Statements Require Integer Types

String case statements are forbidden:

```
case name of
  "Alice": ...;   // ✗ Error - strings not allowed
  "Bob": ...;
end;
```

Workaround: Use `if...else if` chains.

No Range Checking

Array and string accesses are unchecked:

```
arr[i]   // No bounds checking
s[i]     // No bounds checking
```

Status: By design - matches C++ semantics.

---

11. Appendices

Appendix A: Keyword Reference

Keywords (41 Total)

Keyword	Category	Description
`and`	Operator	Logical AND

Keyword	Category	Description
`array`	Type	Array type declaration
`begin`	Statement	Begin block
`break`	Statement	Exit loop
`case`	Statement	Case statement
`const`	Declaration	Constant declaration
`continue`	Statement	Continue loop
`div`	Operator	Integer division
`do`	Statement	Loop body
`downto`	Statement	For loop (descending)
`else`	Statement	Alternative branch
`end`	Statement	End block
`extern`	Declaration	External C function
`false`	Literal	Boolean false
`finalize`	Declaration	Library cleanup
`for`	Statement	For loop
`halt`	Statement	Exit program
`if`	Statement	Conditional
`import`	Declaration	Import module
`library`	Declaration	Library unit
`mod`	Operator	Modulo
`module`	Declaration	Module unit
`nil`	Literal	Null pointer
`not`	Operator	Logical NOT
`of`	Type	Array element type
`or`	Operator	Logical OR

Keyword	Category	Description
`program`	Declaration	Program unit
`public`	Modifier	Public visibility
`repeat`	Statement	Repeat loop
`return`	Statement	Return from routine
`routine`	Declaration	Function/procedure
`shl`	Operator	Shift left
`shr`	Operator	Shift right
`then`	Statement	If condition body
`to`	Statement	For loop (ascending)
`true`	Literal	Boolean true
`type`	Declaration	Type declaration
`until`	Statement	Repeat condition
`var`	Declaration	Variable declaration
`while`	Statement	While loop
`xor`	Operator	Logical XOR

Appendix B: Operator Precedence

Precedence Levels (Highest to Lowest)

Level 1 - Highest (Unary):

- `@` - Address-of
- `^` - Pointer dereference
- `not` - Logical NOT

Level 2 - Multiplicative:

- `*` - Multiply
- `/` - Divide (float)
- `div` - Integer divide
- `mod` - Modulo
- `shl` - Shift left
- `shr` - Shift right
- `and` - Logical AND

Level 3 - Additive:

- `+` - Add
- `-` - Subtract
- `or` - Logical OR
- `xor` - Logical XOR

Level 4 - Relational (Lowest):

- `=` - Equal
- `<>` - Not equal
- `<` - Less than
- `<=` - Less or equal
- `>` - Greater than
- `>=` - Greater or equal

Associativity

- **Left-to-right:** All binary operators
- **Right-to-left:** Unary operators

Examples

```
// Precedence example
a + b * c         // = a + (b * c)
not x and y       // = (not x) and y
x = y and z = w   // = x = (y and (z = w))

// Parentheses for clarity
(a + b) * c
x and (y or z)
```

Appendix C: Type Mapping

Primitive Types

NitroPascal	C++	Size	Signed
`int`	`int32_t`	4	Yes
`uint`	`uint32_t`	4	No
`int64`	`int64_t`	8	Yes
`uint64`	`uint64_t`	8	No
`int16`	`int16_t`	2	Yes
`uint16`	`uint16_t`	2	No
`byte`	`uint8_t`	1	No
`double`	`double`	8	Yes
`float`	`float`	4	Yes
`bool`	`bool`	1	N/A
`char`	`char`	1	No
`pointer`	`void*`	4/8	N/A

Structured Types

NitroPascal	C++ Equivalent	Status
`string`	`std::string`	✅ Implemented
`^T`	`T*`	✅ Implemented
`array[0..N] of T`	`T[N+1]` or `std::array<T, N+1>`	✅ Implemented
`record...end`	`struct {...}`	✅ Implemented
`(value1, value2)`	`enum {...}`	✅ Implemented
`Expression..Expression`	Type alias (subrange)	✅ Implemented

Special Types

NitroPascal	C++ Equivalent	Notes
Routine parameter types	Function pointers	Via typedefs
Generic pointer	`void*`	Explicit casts required
C function pointer	`T (*)(args)`	Via extern declarations

Appendix D: EBNF Grammar

Complete Grammar

```
(* Compilation Units *)
CompilationUnit = Program | Module | Library .

Program = "program" Identifier ";"
          [ Declarations ]
          "begin"
          StatementSequence
          "end" "." .

Module = "module" Identifier ";"
         [ Declarations ]
         "end" "." .

Library = "library" Identifier ";"
          [ Declarations ]
          [ "begin" StatementSequence ]
          [ "finalize" StatementSequence ]
          "end" "." .

(* Declarations *)
Declarations = { ImportDecl | ExternDecl | TypeDecl | ConstDecl | VarDecl | RoutineDecl } .

ImportDecl = "import" Identifier ";" .

ExternDecl = "extern" ExternalSource "routine" Identifier
             "(" [ ParameterList ] ")" [ ":" Type ] [ "as" StringLiteral ] ";" .

ExternalSource = "<" Identifier ">"
               | StringLiteral
               | "dll" StringLiteral [ CallConv ] .

CallConv = "stdcall" | "cdecl" | "fastcall" .

TypeDecl = "type" Identifier "=" TypeDefinition ";" .

ConstDecl = "const" Identifier [ ":" Type ] "=" Expression ";" .

VarDecl = "var" IdentifierList ":" Type [ ":=" Expression ] ";" .

RoutineDecl = [ "public" ] "routine" Identifier
              "(" [ ParameterList ] ")" [ ":" Type ] ";"
              [ VarSection ]
              "begin"
              StatementSequence
              "end" ";" .

IdentifierList = Identifier { "," Identifier } .

VarSection = "var" { IdentifierList ":" Type ";" } .

(* Types *)
Type = TypeIdentifier
```

```
     | PointerType
     | ArrayType
     | RecordType
     | EnumType
     | SubrangeType
     | FunctionType .

TypeIdentifier = Identifier .

PointerType = "^" Type .

ArrayType = "array" "[" ArrayBounds "]" "of" Type .

ArrayBounds = Expression [ ".." Expression ] { "," Expression [ ".." Expression ] } .

RecordType = "record"
             { IdentifierList ":" Type ";" }
             "end" .

EnumType = "(" IdentifierList ")" .

SubrangeType = Expression ".." Expression .

FunctionType = "routine" "(" [ ParameterList ] ")" [ ":" Type ] .

(* Parameters *)
ParameterList = Parameter { ";" Parameter } .

Parameter = [ "const" | "var" | "out" ] IdentifierList ":" Type
          | "..." .

(* Statements *)
StatementSequence = Statement { ";" Statement } .

Statement = Assignment
          | RoutineCall
          | IfStatement
          | CaseStatement
          | WhileStatement
          | RepeatStatement
          | ForStatement
          | BreakStatement
          | ContinueStatement
          | ReturnStatement
          | HaltStatement
          | CompoundStatement
          | [ empty ] .

Assignment = Designator ":=" Expression .

RoutineCall = Designator [ "(" [ ArgumentList ] ")" ] .

ArgumentList = Expression { "," Expression } .

CompoundStatement = "begin" StatementSequence "end" .

IfStatement = "if" Expression "then" Statement
              [ "else" Statement ] .
```

```
CaseStatement = "case" Expression "of"
                CaseElement { ";" CaseElement }
                [ "else" StatementSequence ]
                "end" .

CaseElement = CaseLabelList ":" Statement .

CaseLabelList = CaseLabel { "," CaseLabel } .

CaseLabel = Expression [ ".." Expression ] .

WhileStatement = "while" Expression "do" Statement .

RepeatStatement = "repeat" StatementSequence "until" Expression .

ForStatement = "for" Identifier ":=" Expression
               ( "to" | "downto" ) Expression
               "do" Statement .

BreakStatement = "break" .

ContinueStatement = "continue" .

ReturnStatement = "return" [ Expression ] .

HaltStatement = "halt" "(" Expression ")" .

(* Expressions *)
Expression = SimpleExpression [ RelOp SimpleExpression ] .

SimpleExpression = [ "+" | "-" ] Term { AddOp Term } .

Term = Factor { MulOp Factor } .

Factor = IntegerLiteral
       | FloatLiteral
       | StringLiteral
       | CharLiteral
       | BoolLiteral
       | NilLiteral
       | ArrayLiteral
       | RecordLiteral
       | Designator [ "(" [ ArgumentList ] ")" ]
       | "(" Expression ")"
       | "not" Factor
       | "@" Designator
       | TypeCast .

ArrayLiteral = "[" [ Expression { "," Expression } ] "]" .

RecordLiteral = "(" Identifier ":" Expression { ";" Identifier ":" Expression } ")" .

TypeCast = Type "(" Expression ")" .

Designator = Identifier { Selector } .

Selector = "." Identifier [ "(" [ ArgumentList ] ")" ]
         | "[" Expression { "," Expression } "]"
         | "^" .
```

```
RelOp = "=" | "<>" | "<" | "<=" | ">" | ">=" .

AddOp = "+" | "-" | "or" | "xor" .

MulOp = "*" | "/" | "div" | "mod" | "shl" | "shr" | "and" .

(* Literals *)
Identifier = Letter { Letter | Digit | "_" } .

IntegerLiteral = Digit { Digit }
               | "0x" HexDigit { HexDigit }
               | "0b" BinDigit { BinDigit } .

FloatLiteral = Digit { Digit } "." { Digit } [ Exponent ]
             | Digit { Digit } Exponent .

Exponent = ( "e" | "E" ) [ "+" | "-" ] Digit { Digit } .

StringLiteral = '"' { Character | EscapeSequence } '"' .

CharLiteral = "'" ( Character | EscapeSequence ) "'" .

BoolLiteral = "true" | "false" .

NilLiteral = "nil" .

EscapeSequence = "\n" | "\t" | "\r" | "\\" | "\"" | "\'" | "\0" .

Letter = "a".."z" | "A".."Z" .
Digit = "0".."9" .
HexDigit = Digit | "a".."f" | "A".."F" .
BinDigit = "0" | "1" .

(* Preprocessor *)
Preprocessor = Define | Undef | IfDef | IfNDef | Else | EndIf .

Define = "#define" Identifier .
Undef = "#undef" Identifier .
IfDef = "#ifdef" Identifier .
IfNDef = "#ifndef" Identifier .
Else = "#else" .
EndIf = "#endif" .

(* Compiler Directives *)
Directive = "$" DirectiveName StringLiteral .

DirectiveName = "target" | "optimize" | "exceptions" | "strip_symbols"
              | "module_path" | "include_path" | "library_path" | "link_library" .
```

Quick Reference Card

Common Patterns

Program Structure:

```
$target "native"
$optimize "debug"

program MyApp;

extern <stdio.h> routine printf(format: ^char; ...): int;

var
  x: int;

begin
  x := 42;
  printf("x = %d\n", x);
  ExitCode := 0;
end.
```

Module Structure:

```
module MyModule;

public type MyType = int;

public routine MyFunction(const x: int): int;
begin
  return x * 2;
end;

end.
```

Conditionals:

```
#define DEBUG

#ifdef DEBUG
  // Debug code
#else
  // Release code
#endif
```

External Functions:

```
extern <stdio.h> routine printf(format: ^char; ...): int;
extern dll "user32.dll" stdcall routine MessageBoxA(...): int;
```

Document Index

End of NitroPascal Developer Manual

For the latest updates, visit: https://github.com/tinyBigGAMES/NitroPascal