NitroPascal User Manual

Version: 1.0
Last Updated: 2025-01-12
Website: https://nitropascal.org
GitHub: https://github.com/tinyBigGAMES/NitroPascal

Table of Contents

---

1. Introduction

1.1 What is NitroPascal?

NitroPascal is a **next-generation Pascal compiler** that combines the elegance of Object Pascal with the raw performance of C/C++. Unlike traditional Pascal compilers, NitroPascal uses a revolutionary **transpilation approach**:

```text
Pascal Source → C++ Code → Native Binary
```

By generating optimized C++20 code and leveraging the Zig compiler (which uses LLVM), NitroPascal delivers **C-level performance** while maintaining Pascal's readability and strong typing.

1.2 Why NitroPascal?

For Pascal Developers:

- ✅ Write in familiar Object Pascal syntax (Delphi-compatible)
- ✅ Achieve C/C++ performance without learning new languages
- ✅ Cross-platform deployment (Windows, Linux, macOS, and more)
- ✅ Easy C/C++ library integration

For C++ Developers:

- ✅ Higher-level language with strong typing and clarity
- ✅ Faster development with Pascal's clean syntax
- ✅ No compromise on performance
- ✅ Transparent C++ output for debugging

For Everyone:

- ✅ Modern toolchain (Zig build system)
- ✅ Zero external dependencies (bundled runtime)
- ✅ Open source (BSD-3-Clause license)
- ✅ Commercial-friendly licensing

1.3 How It Works

NitroPascal's compilation pipeline consists of four stages:

```text
┌─────────────────────────────────────────────────────────┐
│ Stage 1: Parsing                                        │
│ ─────────────────────────────────────────────────────── │
│ Pascal Source (.pas) → AST (Abstract Syntax Tree)       │
│ Parser: DelphiAST (full Object Pascal support)          │
└─────────────────────────┬───────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────┐
│ Stage 2: Code Generation                                │
│ ─────────────────────────────────────────────────────── │
│ AST → Optimized C++20 Code (.cpp/.h)                    │
│ Generator: NitroPascal CodeGen Engine                   │
│ Runtime: Custom C++ Runtime Library (RTL)               │
└─────────────────────────┬───────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────┐
│ Stage 3: Compilation                                    │
│ ─────────────────────────────────────────────────────── │
│ C++ Code → Native Object Files                          │
│ Compiler: Zig (LLVM backend)                            │
│ Orchestrator: build.zig (Zig build system)              │
└─────────────────────────┬───────────────────────────────┘
                          │
                          ▼
┌─────────────────────────────────────────────────────────┐
│ Stage 4: Linking                                        │
│ ─────────────────────────────────────────────────────── │
│ Object Files → Executable/Library                       │
│ Output: Native binary for target platform               │
└─────────────────────────────────────────────────────────┘
```

Key Insight: NitroPascal doesn't reinvent the wheel. It leverages:

- **DelphiAST** for robust Object Pascal parsing
- **LLVM** (via Zig) for world-class optimization
- **C++20** as a "portable assembly language"
- **Zig's build system** for cross-platform builds

This means you get the combined benefits of decades of compiler research and optimization, wrapped in a simple Pascal-to-native workflow.

---

2. Installation

2.1 System Requirements

Operating Systems:

- Windows 10/11 (x64)
- Linux (x64) - Ubuntu 20.04+ recommended
- macOS 10.15+ (x64 or ARM64)

Disk Space:

- ~100 MB for NitroPascal installation
- Additional space for project outputs

Memory:

- Minimum: 2 GB RAM
- Recommended: 4 GB+ RAM

2.2 Installation Steps

Windows

1. **Download** the latest release from GitHub Releases
2. **Extract** the archive to your preferred location (e.g., `C:\NitroPascal`)
3. **Add to PATH** (optional but recommended):
   - Open System Properties → Environment Variables
   - Add `C:\NitroPascal\bin` to your PATH
4. **Verify installation**:

```cmd
nitro --version
```

2.3 Bundled Components

NitroPascal includes everything you need:

- ✅ **nitro** - Command-line compiler
- ✅ **Zig** - C++ compiler (embedded)
- ✅ **Runtime Library** - NitroPascal RTL (C++)
- ✅ **DelphiAST** - Pascal parser (embedded)

No additional installations required!

---

3. Quick Start

3.1 Your First Program

Let's create a classic "Hello, World!" program:

Step 1: Create a new project

```bash
nitro init HelloWorld
```

This creates:

```text
HelloWorld/
├── src/
│   └── HelloWorld.pas      # Your main source file
├── runtime/
│   ├── runtime.cpp         # NitroPascal runtime
│   └── runtime.h
├── generated/              # Generated C++ (created on build)
└── build.zig               # Build configuration
```

Step 2: Navigate to your project

```bash
cd HelloWorld
```

Step 3: Build the project

```bash
nitro build
```

Step 4: Run your program

```bash
nitro run
```

Output:

```text
Hello world, welcome to NitroPascal!
```

3.2 Modifying Your Program

Open `src/HelloWorld.pas` in your favorite editor:

```
program HelloWorld;
begin
  WriteLn('Hello world, welcome to NitroPascal!');
end.
```

Let's make it more interesting:

```
program HelloWorld;

var
  Name: string;
  Age: Integer;

begin
  WriteLn('=== Welcome to NitroPascal! ===');
  WriteLn('');

  Write('Enter your name: ');
  ReadLn(Name);

  Write('Enter your age: ');
  ReadLn(Age);

  WriteLn('');
  WriteLn('Hello, ', Name, '!');
  WriteLn('You are ', Age, ' years old.');

  if Age >= 18 then
    WriteLn('You are an adult.')
  else
    WriteLn('You are a minor.');

  WriteLn('');
  WriteLn('Program compiled with NitroPascal');
end.
```

Rebuild and run:

```bash
nitro build
nitro run
```

3.3 Creating Different Project Types

NitroPascal supports three project templates:

Program (Executable)

```bash
nitro init MyGame --template program
```

Creates a standalone executable application.

Library (Shared/Dynamic)

```bash
nitro init MyLib --template library
```

Creates a shared library (`.dll` on Windows, `.so` on Linux, `.dylib` on macOS).

Unit (Static Library)

```bash
nitro init MyUnit --template unit
```

Creates a static library (`.lib` on Windows, `.a` on Linux/macOS).

---

4. Understanding NitroPascal

4.1 The Compilation Model

NitroPascal uses **transpilation** rather than direct compilation:

Traditional Compiler:

```text
Pascal → Assembly → Machine Code
```

NitroPascal:

```text
Pascal → C++ → Assembly → Machine Code
```

Why this approach?

1. **Leverage Existing Optimizers**: LLVM has decades of optimization research
2. **Cross-Platform for Free**: C++ compiles everywhere
3. **Interoperability**: Easy C/C++ library integration
4. **Maintainability**: Smaller, simpler codebase
5. **Debugging**: Inspect generated C++ when needed

4.2 The Runtime Library (RTL)

NitroPascal includes a custom **Runtime Library** written in C++20 that provides:

- **I/O Functions**: `WriteLn`, `Write`, `ReadLn`
- **String Operations**: UTF-16 strings with 1-based indexing (Delphi-compatible)
- **Control Flow Helpers**: `ForLoop`, `WhileLoop`, `RepeatUntil`
- **Type Support**: Dynamic arrays, sets, records
- **Standard Functions**: `Length`, `Copy`, `Pos`, `IntToStr`, etc.

The RTL ensures that generated C++ code behaves **exactly like Delphi Pascal**.

Example:

Pascal:

```
for i := 1 to 10 do
  WriteLn(i);
```

Generated C++:

```cpp
np::ForLoop(1, 10, [&](int i) {
    np::WriteLn(i);
});
```

The `np::ForLoop` function in the RTL guarantees Delphi semantics (inclusive range, iterator immutability).

4.3 Type Mappings

NitroPascal maps Pascal types to C++ equivalents:

Pascal Type	C++ Type	Notes
`Integer`	`int32_t`	Fixed 32-bit
`Cardinal`	`uint32_t`	Unsigned 32-bit
`Int64`	`int64_t`	Fixed 64-bit
`Byte`	`uint8_t`	Unsigned 8-bit
`Word`	`uint16_t`	Unsigned 16-bit
`Boolean`	`bool`	Native bool
`Char`	`char16_t`	UTF-16 character
`String`	`np::String`	UTF-16, 1-based indexing
`Double`	`double`	IEEE 754 double
`Single`	`float`	IEEE 754 single
`Pointer`	`void*`	Raw pointer
`array of T`	`np::DynArray<T>`	Dynamic array
`array[a..b] of T`	`std::array<T, N>`	Static array
`record`	`struct`	Plain struct
`class`	`class`	C++ class

4.4 Generated Code Structure

When you run `nitro build`, the following happens:

1. Preprocessing (Phase 0)

- Compiler directives are extracted and applied
- Build settings are configured

2. Parsing (Phase 1)

- Pascal source is parsed into JSON AST
- AST is saved to `generated/<filename>.json`

3. Code Generation (Phase 2)

- JSON AST is transformed to C++ code
- Output: `generated/<filename>.cpp` and `generated/<filename>.h`

4. Build Script Generation (Phase 3)

- `build.zig` is updated with source files and settings

5. Compilation (Phase 4)

- Zig compiles C++ code to object files
- Links with runtime library
- Output: `zig-out/bin/<projectname>` (or `.exe` on Windows)

---

5. The NitroPascal CLI

5.1 Command Reference

`nitro init <name> [options]`

Creates a new NitroPascal project.

Syntax:

```bash
nitro init <project-name> [--template <type>]
```

Options:

- `<project-name>` - Name of the project (required)
- `-t, --template <type>` - Project template: `program`, `library`, or `unit` (default: `program`)

Examples:

```bash
# Create a program (executable)
nitro init MyGame

# Create a shared library
nitro init MyLib --template library

# Create a static library
nitro init MyUnit -t unit
```

Output:

```text
Creating project: MyGame
Location: C:\Projects\MyGame

✓ Created directory structure
✓ Copied runtime files
✓ Created src/MyGame.pas
✓ Created build.zig

Project initialized successfully!

Next steps:
  cd MyGame
  nitro build
  nitro run
```

`nitro build`

Compiles the current project.

Syntax:

```bash
nitro build
```

What it does:

1. Finds entry point (`<ProjectName>.pas` or `main.pas`)
2. Preprocesses compiler directives
3. Transpiles Pascal to C++
4. Updates `build.zig` with generated sources
5. Invokes Zig compiler
6. Produces native executable/library

Output:

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

✓ Transpilation complete

✓ Updated build.zig

Building with Zig...

  [1/3] Compile C++ runtime.cpp
  [2/3] Compile C++ MyGame.cpp
  [3/3] Link executable MyGame

✓ Build completed successfully!
```

Generated Files:

- `generated/*.cpp` - Generated C++ source
- `generated/*.h` - Generated C++ headers
- `generated/*.json` - Intermediate AST (for debugging)
- `zig-out/bin/<executable>` - Final binary

`nitro run`

Executes the compiled program (programs only).

Syntax:

```bash
nitro run
```

Requirements:

- Project must be built first (`nitro build`)
- Only works with `program` template (not libraries)

Example:

```bash
nitro build
nitro run
```

Output:

```text
Running MyGame...

[your program output here]
```

`nitro clean`

Removes all generated files and build artifacts.

Syntax:

```bash
nitro clean
```

Removes:

- `generated/` - Generated C++ code and AST
- `.zig-cache/` - Zig build cache
- `zig-out/` - Build outputs

Output:

```text
Cleaning project...

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

Clean completed successfully!
```

Use case: When you want a fresh build or encounter build issues.

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

Converts C header files to Pascal units (future feature).

Syntax:

```bash
nitro convert-header <input.h> [--output <file>] [--library <name>] [--convention <type>]
```

Options:

- `<input.h>` - Input C header file (required)
- `--output <file>` - Output Pascal unit filename
- `--library <name>` - Target library name for external declarations
- `--convention <type>` - Calling convention: `cdecl` or `stdcall` (default: `cdecl`)

Example:

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

Status: 🚧 Coming in future release

`nitro version`

Displays version information.

Syntax:

```bash
nitro version
# or
nitro --version
```

Output:

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

Version 1.0.0

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

Licensed under BSD 3-Clause License
```

`nitro help`

Displays help information.

Syntax:

```bash
nitro help
# or
nitro --help
# or
nitro -h
# or
nitro (no arguments)
```

Output: Shows comprehensive help with all commands, options, and examples.

5.2 Exit Codes

The `nitro` command uses standard exit codes:

Exit Code	Meaning
`0`	Success

Exit Code	Meaning
`1`	Runtime error
`2`	Invalid arguments/usage
`3`	Build/compilation failure

Use in scripts:

```bash
#!/bin/bash
nitro build
if [ $? -eq 0 ]; then
    echo "Build successful"
    nitro run
else
    echo "Build failed"
    exit 1
fi
```

---

6. Language Reference

6.1 Program Structure

Basic Program

```
program ProgramName;
begin
  // Your code here
  WriteLn('Hello, World!');
end.
```

Program with Variables

```
program Variables;

var
  X: Integer;
  Y: Double;
  Name: String;

begin
  X := 10;
  Y := 3.14;
  Name := 'NitroPascal';

  WriteLn('X = ', X);
  WriteLn('Y = ', Y:0:2);
  WriteLn('Name = ', Name);
end.
```

Program with Constants

```
program Constants;

const
  PI = 3.14159265;
  APP_NAME = 'MyApp';
  MAX_USERS = 100;

var
  Radius: Double;
  Area: Double;

begin
  Radius := 5.0;
  Area := PI * Radius * Radius;
  WriteLn('Area of circle: ', Area:0:2);
end.
```

Program with Functions

```
program Functions;

function Add(A, B: Integer): Integer;
begin
  Result := A + B;
end;

function Multiply(A, B: Integer): Integer;
begin
  Result := A * B;
end;

var
  Sum: Integer;
  Product: Integer;

begin
  Sum := Add(5, 7);
  Product := Multiply(5, 7);

  WriteLn('5 + 7 = ', Sum);
  WriteLn('5 * 7 = ', Product);
end.
```

6.2 Data Types

Integer Types

```
var
  I: Integer;      // 32-bit signed (-2,147,483,648 to 2,147,483,647)
  C: Cardinal;     // 32-bit unsigned (0 to 4,294,967,295)
  B: Byte;         // 8-bit unsigned (0 to 255)
  W: Word;         // 16-bit unsigned (0 to 65,535)
  I64: Int64;      // 64-bit signed

begin
  I := -100;
  C := 200;
  B := 255;
  W := 65535;
  I64 := 9223372036854775807;
end.
```

Floating-Point Types

```
var
  S: Single;       // 32-bit float (IEEE 754)
  D: Double;       // 64-bit float (IEEE 754)

begin
  S := 3.14;
  D := 3.141592653589793;

  WriteLn('Single: ', S:0:2);
  WriteLn('Double: ', D:0:15);
end.
```

Boolean Type

```
var
  Flag: Boolean;

begin
  Flag := True;

  if Flag then
    WriteLn('Flag is true')
  else
    WriteLn('Flag is false');
end.
```

Character and String Types

```
var
  Ch: Char;        // UTF-16 character
  Str: String;     // UTF-16 string (1-based indexing)

begin
  Ch := 'A';
  Str := 'Hello, NitroPascal!';

  WriteLn('Character: ', Ch);
  WriteLn('String: ', Str);
  WriteLn('Length: ', Length(Str));
  WriteLn('First char: ', Str[1]);  // 1-based!
end.
```

6.3 Control Flow

If-Then-Else

```
var
  Age: Integer;

begin
  Age := 25;

  if Age < 18 then
    WriteLn('Minor')
  else if Age < 65 then
    WriteLn('Adult')
  else
    WriteLn('Senior');
end.
```

For Loops

```
var
  I: Integer;

begin
  // For-to loop (ascending)
  for I := 1 to 10 do
    WriteLn('Count: ', I);

  WriteLn('');

  // For-downto loop (descending)
  for I := 10 downto 1 do
    WriteLn('Countdown: ', I);
end.
```

While Loops

```
var
  Count: Integer;

begin
  Count := 1;

  while Count <= 5 do
  begin
    WriteLn('Count: ', Count);
    Count := Count + 1;
  end;
end.
```

Repeat-Until Loops

```
var
  Count: Integer;

begin
  Count := 1;

  repeat
    WriteLn('Count: ', Count);
    Count := Count + 1;
  until Count > 5;
end.
```

Case Statements

```
var
  Day: Integer;

begin
  Day := 3;

  case Day of
    1: WriteLn('Monday');
    2: WriteLn('Tuesday');
    3: WriteLn('Wednesday');
    4: WriteLn('Thursday');
    5: WriteLn('Friday');
    6: WriteLn('Saturday');
    7: WriteLn('Sunday');
  else
    WriteLn('Invalid day');
  end;
end.
```

6.4 Procedures and Functions

Procedures

```
procedure Greet(Name: String);
begin
  WriteLn('Hello, ', Name, '!');
end;

begin
  Greet('Alice');
  Greet('Bob');
end.
```

Functions

```
function Square(X: Integer): Integer;
begin
  Result := X * X;
end;

function IsEven(N: Integer): Boolean;
begin
  Result := (N mod 2) = 0;
end;

var
  Value: Integer;

begin
  Value := Square(5);
  WriteLn('Square of 5: ', Value);

  if IsEven(10) then
    WriteLn('10 is even');
end.
```

Parameters

```
// By value (default)
procedure ByValue(X: Integer);
begin
  X := X + 1;  // Changes local copy only
end;

// By reference (var)
procedure ByReference(var X: Integer);
begin
  X := X + 1;  // Changes original variable
end;

// Const parameters (recommended for efficiency)
procedure UseConst(const Str: String);
begin
  WriteLn(Str);  // Can read but not modify
end;

var
  N: Integer;

begin
  N := 10;
  ByValue(N);
  WriteLn('After ByValue: ', N);      // Still 10

  ByReference(N);
  WriteLn('After ByReference: ', N);  // Now 11
end.
```

6.5 Records (Structures)

Basic Records

```
type
  TPoint = record
    X: Integer;
    Y: Integer;
  end;

var
  P: TPoint;

begin
  P.X := 10;
  P.Y := 20;

  WriteLn('Point: (', P.X, ', ', P.Y, ')');
end.
```

Records with Functions

```
type
  TRectangle = record
    Width: Integer;
    Height: Integer;
  end;

function CalculateArea(const R: TRectangle): Integer;
begin
  Result := R.Width * R.Height;
end;

var
  Rect: TRectangle;

begin
  Rect.Width := 10;
  Rect.Height := 20;

  WriteLn('Area: ', CalculateArea(Rect));
end.
```

6.6 Arrays

Static Arrays

```
var
  Numbers: array[0..4] of Integer;
  I: Integer;

begin
  Numbers[0] := 10;
  Numbers[1] := 20;
  Numbers[2] := 30;
  Numbers[3] := 40;
  Numbers[4] := 50;

  for I := 0 to 4 do
    WriteLn('Numbers[', I, '] = ', Numbers[I]);
end.
```

Dynamic Arrays

```
var
  Numbers: array of Integer;
  I: Integer;

begin
  SetLength(Numbers, 5);

  for I := 0 to 4 do
    Numbers[I] := I * 10;

  for I := 0 to Length(Numbers) - 1 do
    WriteLn('Numbers[', I, '] = ', Numbers[I]);
end.
```

6.7 Operators

Arithmetic Operators

```
var
  A, B: Integer;
  X, Y: Double;

begin
  A := 10;
  B := 3;

  WriteLn('Addition: ', A + B);        // 13
  WriteLn('Subtraction: ', A - B);     // 7
  WriteLn('Multiplication: ', A * B);  // 30
  WriteLn('Float Division: ', A / B);  // 3.333...
  WriteLn('Integer Division: ', A div B); // 3
  WriteLn('Modulo: ', A mod B);        // 1
end.
```

Comparison Operators

```
var
  A, B: Integer;

begin
  A := 10;
  B := 20;

  WriteLn('A = B: ', A = B);   // False
  WriteLn('A <> B: ', A <> B); // True
  WriteLn('A < B: ', A < B);   // True
  WriteLn('A > B: ', A > B);   // False
  WriteLn('A <= B: ', A <= B); // True
  WriteLn('A >= B: ', A >= B); // False
end.
```

Logical Operators

```
var
  P, Q: Boolean;

begin
  P := True;
  Q := False;

  WriteLn('P and Q: ', P and Q);   // False
  WriteLn('P or Q: ', P or Q);     // True
  WriteLn('not P: ', not P);       // False
  WriteLn('P xor Q: ', P xor Q);   // True
end.
```

Bitwise Operators

```
var
  A, B: Integer;

begin
  A := 12;  // 1100 in binary
  B := 10;  // 1010 in binary

  WriteLn('A and B: ', A and B);   // 8 (1000)
  WriteLn('A or B: ', A or B);     // 14 (1110)
  WriteLn('A xor B: ', A xor B);   // 6 (0110)
  WriteLn('not A: ', not A);       // -13
  WriteLn('A shl 1: ', A shl 1);   // 24 (left shift)
  WriteLn('A shr 1: ', A shr 1);   // 6 (right shift)
end.
```

---

7. Compiler Directives

Compiler directives allow you to control build settings directly in your Pascal source code using special comments.

7.1 Directive Syntax

Directives use the format: `{$directive value}`

Rules:

- Must start with `{$`
- Case-insensitive directive names
- Values can be quoted or unquoted
- Both single and double quotes supported
- Processed before compilation (preprocessing phase)

Examples:

```
{$optimization ReleaseFast}
{$optimization "ReleaseFast"}
{$optimization 'ReleaseFast'}
```

All three formats are equivalent.

7.2 Optimization Directive

Controls the optimization level for the build.

Syntax:

```
{$optimization <mode>}
```

Modes:

Mode	Description	Use Case
`Debug`	No optimization, all safety checks	Development, debugging
`ReleaseSafe`	Optimized with safety checks	Production with runtime checks
`ReleaseFast`	Fully optimized, minimal safety	Maximum performance
`ReleaseSmall`	Optimized for binary size	Embedded systems, small binaries

Example:

```
program OptimizedApp;

{$optimization ReleaseFast}

begin
  WriteLn('Running in ReleaseFast mode');
end.
```

Generated build.zig:

```
const optimize = .ReleaseFast;
```

7.3 Target Directive

Specifies the compilation target platform.

Syntax:

```
{$target <triple>}
```

Common Targets:

Target	Platform
`native`	Current platform (default)
`x86_64-windows`	Windows 64-bit
`x86_64-linux`	Linux 64-bit
`x86_64-macos`	macOS 64-bit
`aarch64-linux`	ARM64 Linux
`aarch64-macos`	ARM64 macOS (Apple Silicon)
`wasm32-wasi`	WebAssembly

Example:

```
program CrossPlatform;

{$target x86_64-linux}

begin
  WriteLn('Compiled for Linux x86_64');
end.
```

For multiple targets, build separately:

```bash
# Edit source to set target, then build
nitro build
```

7.4 Exceptions Directive

Controls C++ exception handling.

Syntax:

```
{$exceptions on|off}
```

Values:

- `on`, `true`, `yes`, `1` - Enable exceptions (default)
- `off`, `false`, `no`, `0` - Disable exceptions

Example:

```
program NoExceptions;

{$exceptions off}

begin
  WriteLn('Compiled without exception support');
end.
```

Why disable exceptions?

- Smaller binary size
- Faster code (no exception handling overhead)
- Embedded systems compatibility

Generated C++ flag:

```cpp
const cpp_flags = [_][]const u8{
    "-std=c++20",
    "-fno-exceptions",  // Added when exceptions = off
};
```

7.5 Strip Directive

Controls debug symbol stripping.

Syntax:

```
{$strip on|off}
```

Values:

- `on`, `true`, `yes`, `1` - Strip debug symbols
- `off`, `false`, `no`, `0` - Keep debug symbols (default)

Example:

```
program StrippedBinary;

{$optimization ReleaseFast}
{$strip on}

begin
  WriteLn('Minimal binary size');
end.
```

Effect:

- `on` - Smaller binary, no debugging info
- `off` - Larger binary, full debugging support

7.6 Include Path Directive

Adds directories to the include path for C++ compilation.

Syntax:

```
{$include_path <path>}
```

Example:

```
program WithIncludes;

{$include_path "../common/include"}
{$include_path "C:/SDK/include"}

begin
  WriteLn('Custom includes loaded');
end.
```

Use case: When using custom C++ headers or libraries.

7.7 Library Path Directive

Adds directories to search for libraries during linking.

Syntax:

```
{$library_path <path>}
```

Example:

```
program WithLibraries;

{$library_path "../common/lib"}
{$library_path "C:/SDK/lib"}

begin
  WriteLn('Custom library paths set');
end.
```

7.8 Link Directive

Links against external libraries.

Syntax:

```
{$link <library>}
```

Example:

```
program WindowsApp;

{$link user32}
{$link gdi32}
{$link opengl32}

begin
  WriteLn('Linked against Windows libraries');
end.
```

Notes:

- Library names are platform-specific
- On Windows: links `user32.lib`
- On Linux: links `libuser32.so` or `libuser32.a`

7.9 Module Path Directive

Adds directories to search for Pascal modules/units.

Syntax:

```
{$module_path <path>}
```

Example:

```
program WithModules;

{$module_path "../common/units"}

begin
  WriteLn('Custom module paths set');
end.
```

Use case: When organizing units in custom directories.

7.10 Complete Example

Here's a real-world example combining multiple directives:

```
program ProductionApp;

{*
  Production Build Configuration
  - Maximum optimization
  - Stripped binary
  - Windows 64-bit target
  - External library dependencies
*}

{$optimization ReleaseFast}
{$strip on}
{$target x86_64-windows}
{$exceptions off}

{$include_path "C:/Libraries/SDL2/include"}
{$library_path "C:/Libraries/SDL2/lib"}
{$link SDL2}
{$link SDL2main}

begin
  WriteLn('Production build ready!');
  WriteLn('Optimized for maximum performance');
end.
```

Result:

- Fully optimized binary
- No debug symbols
- Compiled for Windows x64
- No exception overhead
- Linked with SDL2 library

---

8. Project Structure

8.1 Default Project Layout

When you create a project with `nitro init MyProject`, you get:

```text
MyProject/
├── src/
│   └── MyProject.pas          # Main source file
├── runtime/
│   ├── runtime.cpp            # NitroPascal runtime implementation
│   └── runtime.h              # NitroPascal runtime header
├── generated/                 # Generated files (created on build)
│   ├── MyProject.json         # AST (for debugging)
│   ├── MyProject.cpp          # Generated C++ source
│   └── MyProject.h            # Generated C++ header
├── zig-out/                   # Build output (created on build)
│   └── bin/
│       └── MyProject.exe      # Final executable (Windows)
├── .zig-cache/                # Zig build cache
└── build.zig                  # Build configuration
```

8.2 Source Directory (`src/`)

Purpose: Contains your Pascal source files.

Entry Points:

- `<ProjectName>.pas` - Primary entry point
- `main.pas` - Fallback entry point

Multi-file Projects: You can create multiple `.pas` files in `src/`:

```text
src/
├── MyProject.pas     # Main program
├── Utils.pas         # Utility unit
└── GameEngine.pas    # Game engine unit
```

Using units:

```
// Utils.pas
unit Utils;

interface

function Add(A, B: Integer): Integer;

implementation

function Add(A, B: Integer): Integer;
begin
  Result := A + B;
end;

end.
```

```
// MyProject.pas
program MyProject;

uses Utils;

begin
  WriteLn('5 + 3 = ', Add(5, 3));
end.
```

8.3 Runtime Directory (`runtime/`)

Purpose: Contains the NitroPascal Runtime Library (RTL).

Files:

- `runtime.h` - RTL interface (headers)
- `runtime.cpp` - RTL implementation

⚠️ **Important:** Do not modify these files unless you know what you're doing. They're copied from the NitroPascal installation and ensure Pascal semantics.

RTL provides:

- I/O functions (`WriteLn`, `Write`, `ReadLn`)
- String class (UTF-16, 1-based indexing)
- Control flow helpers (`ForLoop`, `WhileLoop`, etc.)
- Type conversions (`IntToStr`, `StrToInt`, etc.)
- Array/collection helpers

8.4 Generated Directory (`generated/`)

Purpose: Stores transpiled C++ code and intermediate files.

Created by: `nitro build`

Contents:

- `*.cpp` - Generated C++ source
- `*.h` - Generated C++ headers
- `*.json` - AST representation (for debugging)

Example `MyProject.cpp`:

```cpp
#include "runtime/runtime.h"

int main() {
    np::WriteLn("Hello, World!");
    return 0;
}
```

Use case for JSON: Debugging transpilation issues. You can inspect the AST to see how Pascal was parsed.

8.5 Build Output (`zig-out/`)

Purpose: Final build artifacts.

Structure:

```text
zig-out/
└── bin/
    ├── MyProject       # Linux/macOS executable
    └── MyProject.exe   # Windows executable
```

For libraries:

```text
zig-out/
└── lib/
    ├── libMyLib.so      # Linux shared library
    ├── libMyLib.dylib   # macOS shared library
    ├── MyLib.dll        # Windows DLL
    ├── libMyLib.a       # Static library (Linux/macOS)
    └── MyLib.lib        # Static library (Windows)
```

8.6 Build Configuration (`build.zig`)

Purpose: Zig build script that orchestrates C++ compilation.

Generated by: `nitro init` (initial) and `nitro build` (updated)

Example build.zig:

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

pub fn build(b: *std.Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = .ReleaseFast;

    const module = b.addModule("MyProject", .{
        .target = target,
        .optimize = optimize,
        .link_libc = true,
    });

    const cpp_flags = [_][]const u8{
        "-std=c++20",
    };

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

    // Add runtime
    module.addCSourceFile(.{
        .file = b.path("runtime/runtime.cpp"),
        .flags = &cpp_flags,
    });

    module.addIncludePath(b.path("runtime"));

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

    exe.linkLibCpp();
    b.installArtifact(exe);
}
```

⚠️ **Note:** This file is auto-generated. Manual edits will be overwritten on next build.

---

9. Build System

9.1 The Zig Build System

NitroPascal uses **Zig** as both:

1. A drop-in C++ compiler (via `zig cc`)
2. A build system orchestrator (via `build.zig`)

Why Zig?

- ✅ Cross-compilation made easy
- ✅ LLVM-based optimization
- ✅ No external dependencies
- ✅ Fast incremental builds
- ✅ Consistent across platforms

9.2 Build Targets

The Zig build system supports multiple targets:

Query available targets:

```bash
zig targets
```

Common targets:

```text
x86_64-windows-gnu
x86_64-linux-gnu
x86_64-macos
aarch64-linux-gnu
aarch64-macos
wasm32-wasi
```

Setting target in Pascal:

```
{$target x86_64-linux}
```

9.3 Build Modes

Zig supports four optimization modes (mapped from NitroPascal directives):

NitroPascal	Zig Mode	Description
`{$optimization Debug}`	`.Debug`	No optimization, full checks
`{$optimization ReleaseSafe}`	`.ReleaseSafe`	Optimized + checks
`{$optimization ReleaseFast}`	`.ReleaseFast`	Maximum speed
`{$optimization ReleaseSmall}`	`.ReleaseSmall`	Minimum size

Performance comparison (typical):

Mode	Speed	Size	Safety	Build Time
Debug	1x	Large	Full	Fast

Mode	Speed	Size	Safety	Build Time
ReleaseSafe	3-5x	Medium	Partial	Medium
ReleaseFast	5-10x	Medium	Minimal	Slow
ReleaseSmall	3-5x	Small	Minimal	Slow

9.4 Incremental Builds

Zig automatically handles incremental compilation:

First build:

```bash
nitro build
```

```text
Building with Zig...
  [1/3] Compile C++ runtime.cpp
  [2/3] Compile C++ MyProject.cpp
  [3/3] Link executable MyProject

✓ Build completed successfully!
```

Second build (no changes):

```bash
nitro build
```

```text
Building with Zig...

✓ Build completed successfully!
```

Changed source:

```bash
# Edit MyProject.pas
nitro build
```

```text
Building with Zig...
  [1/2] Compile C++ MyProject.cpp
  [2/2] Link executable MyProject

✓ Build completed successfully!
```

Note: Only changed files are recompiled!

9.5 Build Cache

Zig maintains a build cache in `.zig-cache/`:

```text
.zig-cache/
├── h/              # Header dependencies
├── o/              # Object files
└── ...
```

When to clean cache:

- Build errors after system updates
- Switching between major Zig versions
- Weird linking issues

How to clean:

```bash
nitro clean
```

This removes `.zig-cache/`, `zig-out/`, and `generated/`.

10. Advanced Topics

10.1 Inspecting Generated Code

Want to see the C++ code that NitroPascal generates?

Step 1: Build your project

```bash
nitro build
```

Step 2: Navigate to generated directory

```bash
cd generated
```

Step 3: View the C++ code

```bash
cat MyProject.cpp
# or
code MyProject.cpp  # Open in VS Code
```

Example:

Pascal:

```
program Test;

function Add(A, B: Integer): Integer;
begin
  Result := A + B;
end;

begin
  WriteLn('5 + 3 = ', Add(5, 3));
end.
```

Generated C++ (generated/Test.cpp):

```cpp
#include "runtime/runtime.h"

int32_t Add(int32_t A, int32_t B) {
    int32_t Result;
    Result = A + B;
    return Result;
}

int main() {
    np::WriteLn("5 + 3 = ", Add(5, 3));
    return 0;
}
```

Use cases:

- Understanding how your code translates
- Debugging transpilation issues
- Learning C++ patterns
- Optimizing performance

10.2 Debugging

Debugging Pascal Source

Method 1: Output Debugging

```
program Debug;

var
  X: Integer;

begin
  X := 10;
  WriteLn('DEBUG: X = ', X);  // Simple debug output

  X := X * 2;
  WriteLn('DEBUG: After multiplication, X = ', X);
end.
```

Method 2: Conditional Compilation

```
program Debug;

{$DEFINE DEBUG}

var
  X: Integer;

begin
  X := 10;

  {$IFDEF DEBUG}
  WriteLn('DEBUG: X = ', X);
  {$ENDIF}

  X := X * 2;
end.
```

Debugging Generated C++

Step 1: Build with debug symbols

```
{$optimization Debug}
{$strip off}
```

Step 2: Compile

```bash
nitro build
```

Step 3: Debug with GDB (Linux/macOS)

```bash
gdb zig-out/bin/MyProject
```

Step 4: Debug with LLDB (macOS)

```bash
lldb zig-out/bin/MyProject
```

Step 5: Debug with Visual Studio (Windows)

1. Open `generated/MyProject.cpp` in Visual Studio
2. Set breakpoints
3. Attach to process or run under debugger

10.3 Performance Optimization

Level 1: Compiler Directives

```
program FastApp;

{$optimization ReleaseFast}
{$exceptions off}
{$strip on}

begin
  // Your code here
end.
```

Impact:

- `ReleaseFast`: 5-10x faster than Debug
- `exceptions off`: ~5-10% faster, smaller binary
- `strip on`: Smaller binary (faster loading)

Level 2: Algorithm Optimization

Avoid:

```
// Inefficient: String concatenation in loop
var
  Result: String;
  I: Integer;
begin
  Result := '';
  for I := 1 to 10000 do
    Result := Result + 'x';  // Creates new string each time!
end.
```

Prefer:

```
// Efficient: Build array then concat once
var
  Chars: array[0..9999] of Char;
  I: Integer;
begin
  for I := 0 to 9999 do
    Chars[I] := 'x';
  // Convert to string once
end.
```

Level 3: Inspect Generated C++

Check `generated/*.cpp` for optimization opportunities:

Suboptimal:

```cpp
// Unnecessary copies
void ProcessData(std::string data) {  // Copy!
    // ...
}
```

Report to NitroPascal team if you see inefficiencies!

10.4 Calling C/C++ Code

NitroPascal makes it easy to call C/C++ libraries.

Example: Calling C Math Library

Step 1: Declare external functions

```
program MathDemo;

function sqrt(x: Double): Double; external 'c' name 'sqrt';
function pow(x, y: Double): Double; external 'c' name 'pow';

var
  Result: Double;

begin
  Result := sqrt(16.0);
  WriteLn('sqrt(16) = ', Result:0:2);

  Result := pow(2.0, 10.0);
  WriteLn('2^10 = ', Result:0:0);
end.
```

Step 2: Link against math library (Linux)

```
{$link m}
```

Step 3: Build

```bash
nitro build
```

Output:

```text
sqrt(16) = 4.00
2^10 = 1024
```

10.5 Creating Libraries

Shared Library Example

Step 1: Create project

```bash
nitro init MathLib --template library
```

Step 2: Edit MathLib.pas

```
library MathLib;

function LibAdd(A: Integer; B: Integer): Integer; cdecl;
begin
  Result := A + B;
end;

function LibMultiply(A: Integer; B: Integer): Integer; stdcall;
begin
  Result := A * B;
end;

exports
  LibAdd,
  LibMultiply;

begin
end.
```

Step 3: Build

```bash
cd MathLib
nitro build
```

Output:

- Windows: `zig-out/lib/MathLib.dll`
- Linux: `zig-out/lib/libMathLib.so`
- macOS: `zig-out/lib/libMathLib.dylib`

Step 4: Use from C

```c
// main.c
#include <stdio.h>

#ifdef _WIN32
    #define EXPORT __declspec(dllimport)
#else
    #define EXPORT
#endif

EXPORT int LibAdd(int a, int b);
EXPORT int LibMultiply(int a, int b);

int main() {
    printf("5 + 3 = %d\n", LibAdd(5, 3));
    printf("5 * 3 = %d\n", LibMultiply(5, 3));
    return 0;
}
```

10.6 Cross-Compilation

NitroPascal + Zig make cross-compilation trivial.

Compile for Linux from Windows:

```
{$target x86_64-linux}
```

```bash
nitro build
```

Compile for Windows from Linux:

```
{$target x86_64-windows}
```

```bash
nitro build
```

Compile for macOS from Linux:

```
{$target x86_64-macos}
```

```bash
nitro build
```

Compile for ARM64 Linux:

```
{$target aarch64-linux}
```

```bash
nitro build
```

Compile for WebAssembly:

```
{$target wasm32-wasi}
```

```bash
nitro build
```

No cross-compilers needed! Zig handles everything.

---

11. Examples

11.1 Hello World

File: `HelloWorld.pas`

```
program HelloWorld;
begin
  WriteLn('Hello, World!');
  WriteLn('Welcome to NitroPascal!');
end.
```

Build and run:

```bash
nitro init HelloWorld
cd HelloWorld
nitro build
nitro run
```

11.2 Variables and Math

File: `MathDemo.pas`

```
program MathDemo;

var
  A, B: Integer;
  Sum, Diff, Prod: Integer;
  Quot: Double;

begin
  A := 10;
  B := 3;

  Sum := A + B;
  Diff := A - B;
  Prod := A * B;
  Quot := A / B;

  WriteLn('A = ', A);
  WriteLn('B = ', B);
  WriteLn('');
  WriteLn('Sum (A + B) = ', Sum);
  WriteLn('Difference (A - B) = ', Diff);
  WriteLn('Product (A * B) = ', Prod);
  WriteLn('Quotient (A / B) = ', Quot:0:2);
  WriteLn('Integer Division (A div B) = ', A div B);
  WriteLn('Modulo (A mod B) = ', A mod B);
end.
```

Output:

```text
A = 10
B = 3

Sum (A + B) = 13
Difference (A - B) = 7
Product (A * B) = 30
Quotient (A / B) = 3.33
Integer Division (A div B) = 3
Modulo (A mod B) = 1
```

11.3 Loops

File: `Loops.pas`

```
program Loops;

var
  I: Integer;

begin
  WriteLn('=== For Loop (1 to 5) ===');
  for I := 1 to 5 do
    WriteLn('Count: ', I);

  WriteLn('');
  WriteLn('=== For Loop (5 downto 1) ===');
  for I := 5 downto 1 do
    WriteLn('Countdown: ', I);

  WriteLn('');
  WriteLn('=== While Loop ===');
  I := 1;
  while I <= 3 do
  begin
    WriteLn('While iteration: ', I);
    I := I + 1;
  end;

  WriteLn('');
  WriteLn('=== Repeat-Until Loop ===');
  I := 1;
  repeat
    WriteLn('Repeat iteration: ', I);
    I := I + 1;
  until I > 3;
end.
```

11.4 Functions

File: `Functions.pas`

```
program Functions;

function Factorial(N: Integer): Integer;
var
  I: Integer;
begin
  Result := 1;
  for I := 2 to N do
    Result := Result * I;
end;

function IsPrime(N: Integer): Boolean;
var
  I: Integer;
begin
  if N < 2 then
  begin
    Result := False;
    Exit;
  end;

  for I := 2 to N div 2 do
  begin
    if (N mod I) = 0 then
    begin
      Result := False;
      Exit;
    end;
  end;

  Result := True;
end;

var
  Num: Integer;

begin
  WriteLn('=== Factorial Examples ===');
  for Num := 1 to 10 do
    WriteLn('Factorial(', Num, ') = ', Factorial(Num));

  WriteLn('');
  WriteLn('=== Prime Numbers (1 to 20) ===');
  for Num := 1 to 20 do
    if IsPrime(Num) then
      WriteLn(Num, ' is prime');
end.
```

11.5 Records

File: `Records.pas`

```
program Records;

type
  TPoint = record
    X: Integer;
    Y: Integer;
  end;

  TRectangle = record
    TopLeft: TPoint;
    BottomRight: TPoint;
  end;

function CalculateArea(const Rect: TRectangle): Integer;
var
  Width, Height: Integer;
begin
  Width := Rect.BottomRight.X - Rect.TopLeft.X;
  Height := Rect.BottomRight.Y - Rect.TopLeft.Y;
  Result := Width * Height;
end;

procedure PrintPoint(const P: TPoint);
begin
  WriteLn('Point(', P.X, ', ', P.Y, ')');
end;

var
  P1, P2: TPoint;
  Rect: TRectangle;

begin
  P1.X := 0;
  P1.Y := 0;

  P2.X := 10;
  P2.Y := 20;

  WriteLn('Point 1:');
  PrintPoint(P1);

  WriteLn('Point 2:');
  PrintPoint(P2);

  Rect.TopLeft := P1;
  Rect.BottomRight := P2;

  WriteLn('');
  WriteLn('Rectangle area: ', CalculateArea(Rect));
end.
```

11.6 Arrays

File: `Arrays.pas`

```
program Arrays;

var
  StaticArray: array[0..4] of Integer;
  DynamicArray: array of Integer;
  I: Integer;

begin
  WriteLn('=== Static Array ===');
  StaticArray[0] := 10;
  StaticArray[1] := 20;
  StaticArray[2] := 30;
  StaticArray[3] := 40;
  StaticArray[4] := 50;

  for I := 0 to 4 do
    WriteLn('StaticArray[', I, '] = ', StaticArray[I]);

  WriteLn('');
  WriteLn('=== Dynamic Array ===');
  SetLength(DynamicArray, 5);

  for I := 0 to 4 do
    DynamicArray[I] := (I + 1) * 100;

  for I := 0 to Length(DynamicArray) - 1 do
    WriteLn('DynamicArray[', I, '] = ', DynamicArray[I]);
end.
```

11.7 Optimized Build

File: `FastApp.pas`

```
program FastApp;

{$optimization ReleaseFast}
{$exceptions off}
{$strip on}

function Fibonacci(N: Integer): Integer;
begin
  if N <= 1 then
    Result := N
  else
    Result := Fibonacci(N - 1) + Fibonacci(N - 2);
end;

var
  I: Integer;

begin
  WriteLn('=== Fibonacci Numbers (Optimized Build) ===');
  WriteLn('');

  for I := 0 to 15 do
    WriteLn('Fib(', I, ') = ', Fibonacci(I));
end.
```

Build:

```bash
nitro build
```

Note: Optimizations from directives produce significantly faster code!

---

12. Troubleshooting

12.1 Common Build Errors

Error: "File not found"

Problem:

```text
Error: File not found: src/MyProject.pas
```

Solution:

- Ensure you're in the project directory
- Check filename matches exactly (case-sensitive on Linux/macOS)
- Verify file exists: `ls src/`

Error: "Zig EXE was not found"

Problem:

```text
Error: Zig EXE was not found...
```

Solution:

- Reinstall NitroPascal from releases
- Verify Zig is bundled: Check `bin/zig` exists
- Check PATH if you moved installation

Error: "Build failed with exit code 1"

Problem:

```text
✗ Build failed!
Error: Zig build failed with exit code 1
```

Solution:

1. Check for syntax errors in Pascal code
2. Run `nitro clean` and rebuild
3. Check generated C++ code: `cat generated/*.cpp`
4. Look for Zig error messages in output

Error: "Cannot open generated file"

Problem:

```text
Error: Cannot open file 'generated/MyProject.cpp'
```

Solution:

- Generated directory may be corrupted
- Run: `nitro clean`
- Delete `generated/` manually
- Rebuild: `nitro build`

12.2 Runtime Errors

Error: "Access violation" / "Segmentation fault"

Possible causes:

1. **Array out of bounds**

```
var A: array[0..4] of Integer;
A[10] := 100;  // ERROR!
```

2. **Uninitialized pointer**

```
var P: ^Integer;
P^ := 100;  // ERROR: P not initialized!
```

3. **String index out of range**

```
var S: String;
S := 'Hello';
WriteLn(S[100]);  // ERROR!
```

Solution:

- Add bounds checking during development
- Use `{$optimization Debug}` to enable safety checks
- Validate array indices before access

Error: "Invalid memory reference"

Possible causes:

- Freeing memory twice
- Using freed memory
- Stack overflow (deep recursion)

Solution:

```
{$optimization ReleaseSafe}  // Enable safety checks
```

12.3 Getting Help

Check the Documentation

- DESIGN.md - Architecture details
- README.md - Project overview
- THIRD-PARTY.md - Dependencies

Community Support

- **Facebook Group:** NitroPascal Community
- **Discord:** Join Discord
- **Bluesky:** @tinybiggames.com

Report Bugs

- **GitHub Issues:** Open an issue

When reporting:

1. Describe the problem clearly
2. Include Pascal source code (minimal example)
3. Include error messages
4. Mention OS and NitroPascal version
5. Attach generated C++ code if relevant

---

13. FAQ

13.1 General Questions

Q: Is NitroPascal production-ready?

A: NitroPascal is under active development. While the core compiler works well, some advanced features are still being implemented. Check the GitHub repository for current status.

Q: Can I use NitroPascal for commercial projects?

A: Yes! NitroPascal is licensed under BSD-3-Clause, which allows commercial use without restrictions. You only need to include the copyright notice in distributions.

Q: Is NitroPascal compatible with Delphi code?

A: NitroPascal aims for Object Pascal compatibility and supports much of Delphi syntax. However, VCL (Visual Component Library) and Delphi-specific RTL are not supported. You can use NitroPascal for console applications, libraries, and algorithms.

Q: How fast is NitroPascal compared to Delphi/FPC?

A: NitroPascal generates C++ code optimized by LLVM, achieving performance comparable to hand-written C++. In many cases, this matches or exceeds Delphi/FPC performance, especially with `{$optimization ReleaseFast}`.

Q: Can I call C/C++ libraries from NitroPascal?

A: Yes! NitroPascal has excellent C/C++ interoperability through external declarations. See Calling C/C++ Code for examples.

Q: Does NitroPascal support GUI applications?

A: Not directly. NitroPascal focuses on console applications and libraries. However, you can:

- Create libraries that interface with C/C++ GUI frameworks
- Use external C++ GUI libraries through FFI
- Future GUI support is under consideration

Q: Which platforms does NitroPascal support?

A: NitroPascal supports any platform that Zig/LLVM support:

- Windows (x64, ARM64)
- Linux (x64, ARM64)
- macOS (x64, Apple Silicon)
- WebAssembly (WASI)
- Many embedded targets

13.2 Technical Questions

Q: Why does NitroPascal generate C++ instead of compiling directly?

A: This approach provides several benefits:

1. Leverage LLVM's world-class optimization
2. Cross-platform support for free
3. Easy C/C++ library integration
4. Smaller, simpler compiler codebase
5. Inspectable intermediate representation

See Understanding NitroPascal for details.

Q: Can I see the generated C++ code?

A: Yes! Check the `generated/` directory after building:

```bash
nitro build
cat generated/MyProject.cpp
```

Q: How do I optimize my programs?

A: Use compiler directives:

```
{$optimization ReleaseFast}
{$exceptions off}
{$strip on}
```

See Compiler Directives for all options.

Q: Does NitroPascal support generics?

A: Generic support is planned but not yet implemented. Check the GitHub repository for status.

Q: Does NitroPascal support inline assembly?

A: Not directly. However, you can:

1. Write assembly in a C++ file
2. Declare it as external in Pascal
3. Link against it

Q: Why use Zig as the C++ compiler?

A: Zig provides:

- Drop-in C/C++ compiler (via `zig cc`)
- Cross-compilation without external toolchains
- Fast, reliable builds
- LLVM-based optimization
- Consistent behavior across platforms

Q: Can I use NitroPascal without internet access?

A: Yes! All dependencies are bundled. No downloads required during build.

Q: Does NitroPascal support debugging?

A: Yes, when built with:

```
{$optimization Debug}
{$strip off}
```

You can use GDB, LLDB, or Visual Studio to debug the generated C++ code and inspect variables.

Q: How do I contribute to NitroPascal?

A: Contributions are welcome!

1. Fork the GitHub repository
2. Create a feature branch
3. Make your changes
4. Submit a pull request

Also report bugs, suggest features, or improve documentation!

Q: Where can I get support?

A: Multiple channels available:

- **Facebook:** NitroPascal Group
- **Discord:** Join Server
- **Bluesky:** @tinybiggames.com
- **GitHub:** Open Issues

Appendix A: Compiler Directive Quick Reference

Directive	Values	Default	Description
`{$optimization}`	`Debug`, `ReleaseSafe`, `ReleaseFast`, `ReleaseSmall`	`Debug`	Optimization level
`{$target}`	Target triple (e.g., `x86_64-linux`)	`native`	Compilation target
`{$exceptions}`	`on`, `off`	`on`	C++ exception handling
`{$strip}`	`on`, `off`	`off`	Strip debug symbols
`{$include_path}`	Directory path	-	Add C++ include path

Directive	Values	Default	Description
`{$library_path}`	Directory path	-	Add library search path
`{$link}`	Library name	-	Link external library
`{$module_path}`	Directory path	-	Add Pascal module path

Appendix B: Type Conversion Functions

Function	Description	Example
`IntToStr(I: Integer): String`	Integer to string	`IntToStr(42)` → `'42'`
`StrToInt(S: String): Integer`	String to integer	`StrToInt('42')` → `42`
`FloatToStr(F: Double): String`	Float to string	`FloatToStr(3.14)` → `'3.14'`
`StrToFloat(S: String): Double`	String to float	`StrToFloat('3.14')` → `3.14`

Appendix C: String Functions

Function	Description	Example
`Length(S: String): Integer`	String length	`Length('Hello')` → `5`
`Copy(S: String; Index, Count: Integer): String`	Substring (1-based)	`Copy('Hello', 1, 3)` → `'Hel'`
`Pos(Sub, S: String): Integer`	Find substring	`Pos('lo', 'Hello')` → `4`
`UpperCase(S: String): String`	Convert to uppercase	`UpperCase('hello')` → `'HELLO'`
`LowerCase(S: String): String`	Convert to lowercase	`LowerCase('HELLO')` → `'hello'`
`Trim(S: String): String`	Remove leading/trailing spaces	`Trim(' Hi ')` → `'Hi'`

Appendix D: Array Functions

Function	Description	Example
`Length(A: array of T): Integer`	Array length	`Length(arr)`
`SetLength(var A: array of T; N: Integer)`	Resize dynamic array	`SetLength(arr, 10)`

Appendix E: I/O Functions

Function	Description	Example
`Write(...)`	Output without newline	`Write('Hello')`
`WriteLn(...)`	Output with newline	`WriteLn('Hello')`
`ReadLn(var X)`	Read input line	`ReadLn(name)`

Appendix F: Glossary

AST (Abstract Syntax Tree): Internal representation of parsed source code.

Cross-compilation: Compiling for a different platform than the host.

DelphiAST: Open-source Object Pascal parser used by NitroPascal.

LLVM: Compiler infrastructure providing optimization and code generation.

RTL (Runtime Library): Standard library providing core functionality.

Transpilation: Converting source code from one language to another (Pascal → C++).

Zig: Modern programming language and toolchain used by NitroPascal for C++ compilation.

License

NitroPascal is licensed under the **BSD-3-Clause License**.

For full license text, see LICENSE.

About

NitroPascal is developed and maintained by **tinyBigGAMES™ LLC**.

- **Website:** https://nitropascal.org
- **GitHub:** https://github.com/tinyBigGAMES/NitroPascal
- **Facebook:** NitroPascal Community
- **Discord:** Join Server
- **Bluesky:** @tinybiggames.com

Thank you for using NitroPascal! 🚀

Write elegant Pascal, run blazing-fast native code.