The Jakt programming language

Jakt is a memory-safe systems programming language.

It currently transpiles to C++.

NOTE: The language is under heavy development.

NOTE If you're cloning to a Windows PC (not WSL), make sure that your Git client keeps the line endings as `\n`. You can set this as a global config via `git config --global core.autocrlf false`.

Usage

The transpilation to C++ requires `clang`. Make sure you have that installed.

```
jakt file.jakt
./build/file
```

Building

See here.

Goals

1. Memory safety
2. Code readability
3. Developer productivity
4. Executable performance
5. Fun!

Memory safety

The following strategies are employed to achieve memory safety:

- Automatic reference counting
- Strong typing
- Bounds checking
- No raw pointers in safe mode

In **Jakt**, there are three pointer types:

- **T** (Strong pointer to reference-counted class `T`.)
- **weak T** (Weak pointer to reference-counted class `T`. Becomes empty on pointee destruction.)
- **raw T** (Raw pointer to arbitrary type `T`. Only usable in `unsafe` blocks.)

Null pointers are not possible in safe mode, but pointers can be wrapped in `Optional`, i.e `Optional<T>` or `T?` for short.

Math safety

- Integer overflow (both signed and unsigned) is a runtime error.
- Numeric values are not automatically coerced to `int`. All casts must be explicit.

For cases where silent integer overflow is desired, there are explicit functions that provide this functionality.

Code readability

Far more time is spent reading code than writing it. For that reason, **Jakt** puts a high emphasis on readability.

Some of the features that encourage more readable programs:

- Immutable by default.
- Argument labels in call expressions (`object.function(width: 10, height: 5)`)
- Inferred `enum` scope. (You can say `Foo` instead of `MyEnum::Foo`).
- Pattern matching with `match`.
- Optional chaining (`foo?.bar?.baz` (fallible) and `foo!.bar!.baz` (infallible))
- None coalescing for optionals (`foo ?? bar` yields `foo` if `foo` has a value, otherwise `bar`)
- `defer` statements.
- Pointers are always dereferenced with `.` (never –>)
- Trailing closure parameters can be passed outside the call parentheses.
- Error propagation with `ErrorOr<T>` return type and dedicated `try` / `must` keywords.