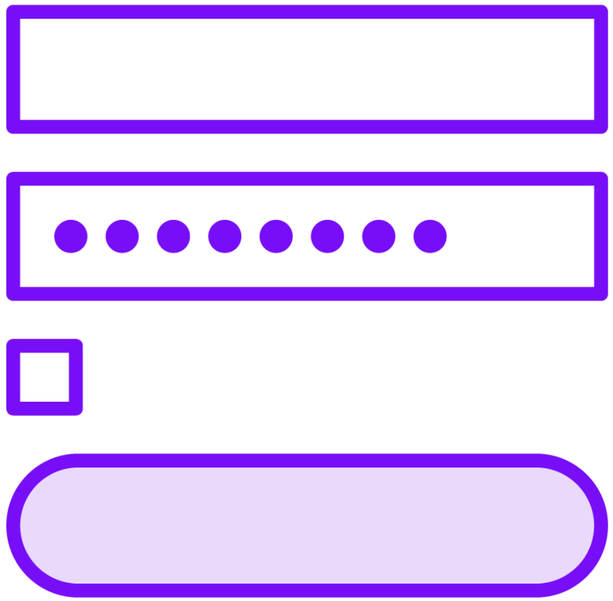
Getting Started



Kate Gregory

@ gregcons | www.gregcons.com/kateblog

Entry Point



Carbon code is in a file with a .carbon extension

- No header files

A Carbon program starts by executing a function called Run that returns an i32

```
fn Run() -> i32 {
   return 3;
}
```

You can echo this return value after running the program

To Run Carbon Programs Today



You need to be on a supported OS

- Windows people, use WSL

The only user interface we have is to print integers, and that returned integer

- Echo it after the program runs

Command Line Reminder

```
bin/carbon compile ~/samples/three.carbon

bin/carbon link  ~/samples/three.o --output=$HOME/samples/three

~/samples/three

echo $?
```



Carbon Syntax

Introducers

`name: type`

Generally resembles C++

```
let a: i32 = 0;

Core.Print(a);

a = 1;

let a: i32;
```

- Declares a value called a with a value of 0
- Puts the value of a to the terminal on a line
- ERROR: Expression is not assignable.
- ERROR: Expected `=`; `let` declaration must have an initializer.

```
fn Identity(x: i32) -> i32 {
  return x;
}
```

- x is a value inside Identity()

```
fn Bad(x: i32) -> i32 {
  x = 10;
  return x;
}
```

- ERROR: Expression is not assignable.

```
var b: i32 = 2;
b = 7;
```

- Declares an object with an initial value of 2
- Can be changed

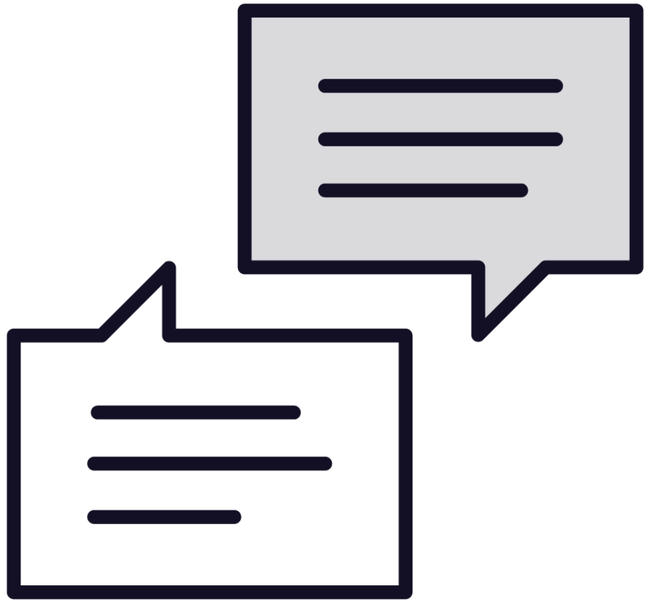
```
fn SetToFour(var x: i32) {
    x = 4;
}

fn SetToFour(x: i32*) {
  *x = 4;
}

SetToFour(&b);
```

- x is a copy, local to SetToFour(), and changing it doesn’t change what was passed to it
- Same syntax as C++ for declaring x to be a pointer, and for dereferencing it
- You can only take the address of objects, not values

Comments



Carbon only has // comments that go to the end of the line

They must be the only thing on the line

```
x = 4; // can't comment like this
```

The whitespace after // is not optional

Classes

Similar to C++

Can have inheritance and polymorphism

Multiple inheritance from classes without data members

No constructors

```
class Point {
   var x: i32;
   var y: i32;
}

let p: Point = {.x=1,.y=3};

a = p.y;

class Point {
   fn GetX[self: Self]() -> i32 {
      self.x =2;
      return self.x;
   }
   fn GetY[self: Self]() -> i32
       {return self.y;}
 var x: i32;
 var y: i32;
}

b = p.GetY();
```

- Note no semi colon after the brace
- Designated initializers
- Access members of the instance with .
- Methods have access to member data
- This self param is a value and cannot be modified
- ERROR: Expression is not assignable.
- Access methods with .

Naming Conventions

UpperCamelCase

types
functions and methods
compile time constants

lower_snake_case

variables
keywords
boolean literals

```
fn Make(x: i32, y: i32) -> Point{
   var p: Point;
   p.x = x;
   p.y = y;
   return p;
}
```

- Factory function

```
fn Make(x: i32, y: i32) -> Point{
   returned var p: Point;
   p.x = x;
   p.y = y;
   return var;
}
```

- ERROR: Cannot copy value of type `Point`.
- Avoiding this copy: Return Value Optimization (RVO) or Named Return Value Optimization (NRVO)
- Carbon’s copy elision is explicit

```
fn Make(x: i32, y: i32) -> Self{
   returned var p: Self;
   p.x = x;
   p.y = y;
   return var;
}
```

- The keyword Self can also be used here

```
let P2: Point = Point.Make(4,7);

let P2: auto = Point.Make(4,7);


fn Make() -> Self{
   returned var p: Self;
   p.x = 0;
   p.y = 0;
   return var;
}

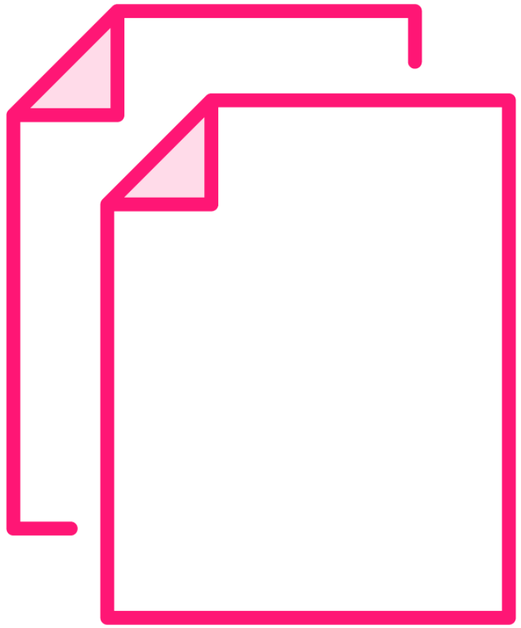
let P3: Point = Point.Make();

fn Origin() -> Self{
   returned var p: Self;
   p.x = 0;
   p.y = 0;
   return var;
}

let P3: Point = Point.Origin();
```

- Class functions are called with the class name, not an instance
- The auto keyword isn’t implemented yet, but will be
- A Make() with no arguments is like a default constructor
- Call it with parens like any other function
- But you might want to give it a more expressive name

Copying and Destructors



```
let P4: Point = p3;
```

- does not copy p3 to P4

```
var p4: Point = p;
```

- ERROR: Cannot copy value of type `Point`.

Write a Copy() or Clone() class function or method

There will be destructors

```
class Point {
  destructor [self: Self] { ... }
}
```

Packages and the Prelude

Carbon supports packages

```
import Math;
```

The prelude is automatically imported into every file

```
Core.Print(3);
```

Contains the standard types, operator overloading, etc.

Other Language Keywords and Operators



Pretty much what a C++ programmer expects

- if, while, for, …
- + - * / and comparisons

Enough is implemented to write code that calculates something

Summary

You have seen the building blocks of Carbon

- Objects (var) and values (let)
- Classes and instances, methods, and class functions
- Simple arithmetic and comparisons
- Conditionals and loops

You can compile, link, and run a carbon program

You know where the project information is and what is still being worked on