Go is a systems programming language intended to be a general-purpose systems language, like C++. These are some notes on Go for experienced C++ programmers. This document discusses the differences between Go and C++, and says little to nothing about the similarities.

An important point to keep in mind is that there are some fundamental differences in the thought processes required to be proficient in the two respective languages. Most formidably, C++'s object model is based on classes and class hierarchies while Go's object model is based on interfaces (and is essentially flat). Consequently, C++ design patterns rarely translate verbatim to Go. To program effectively in Go, one has to consider the *problem* being solved, not the mechanisms one might use in C++ to solve the problem.

For a more general introduction to Go, see the Go Tour, How to Write Go Code and Effective Go.

For a detailed description of the Go language, see the Go spec.

Conceptual Differences

- Go does not have classes with constructors or destructors. Instead of class methods, a class inheritance hierarchy, and virtual functions, Go provides *interfaces*, which are discussed in more detail below. Interfaces are also used where C++ uses templates.
- Go provides automatic garbage collection of allocated memory. It is not necessary (or possible) to release memory explicitly. There is no need to worry about heap-allocated vs. stack-allocated storage, `new` vs. `malloc`, or `delete` vs. `delete[]` vs. `free`. There is no need to separately manage `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`, `std::auto_ptr`, and ordinary, non-smart "raw" pointers. Go's run-time system handles all of that error-prone code on the programmer's behalf.
- Go has pointers but not pointer arithmetic. Go pointers therefore more closely resemble C++ references. One cannot use a Go pointer variable to walk through the bytes of a string. Slices, discussed further below, satisfy most of the need for pointer arithmetic.
- Go is "safe" by default. Pointers cannot point to arbitrary memory, and buffer overruns result in crashes, not security exploits. The `unsafe` package lets programmers bypass some of Go's protection mechanisms where explicitly requested.
- Arrays in Go are first class values. When an array is used as a function parameter, the function receives a copy of the array, not a pointer to it. However, in practice functions often use slices for parameters; slices hold pointers to underlying arrays. Slices are discussed further below.
- Strings are provided by the language. They may not be changed once they have been created.
- Hash tables are provided by the language. They are called maps.
- Separate threads of execution, and communication channels between them, are provided by the language. This is discussed further below.
- Certain types (maps and channels, described further below) are passed by reference, not by value. That is, passing a map to a function does not copy the map, and if the function changes the map the change will be seen by the caller. In C++ terms, one can think of these as being reference types.
- Go does not use header files. Instead, each source file is part of a defined *package*. When a package defines an object (type, constant, variable, function) with a name starting with an upper case letter, that object is visible to any other file which imports that package.
- Go does not support implicit type conversion. Operations that mix different types require casts (called conversions in Go). This is true even of different user-defined aliases of the same underlying type.
- Go does not support function overloading and does not support user defined operators.
- Go does not support `const` or `volatile` qualifiers.
- Go uses `nil` for invalid pointers, where C++ uses `NULL` or simply `0` (or in C++11, `nullptr`).
- Idiomatic Go uses multiple return values to convey errors—one or more data results plus an error code—instead of sentinel values (e.g., `-1`) or structured exception handling (C++'s `try` ... `catch` and `throw`

or Go's `panic` ... `recover`).

Syntax

The declaration syntax is reversed compared to C++. You write the name followed by the type. Unlike in C++, the syntax for a type does not match the way in which the variable is used. Type declarations may be read easily from left to right. (`var v1 int` → "Variable `v1` is an `int`.")

```
//Go                        C++
var v1 int                  // int v1;
var v2 string               // const std::string v2;  (approximately)
var v3 [10]int              // int v3[10];
var v4 []int                // int* v4;  (approximately)
var v5 struct { f int }     // struct { int f; } v5;
var v6 *int                 // int* v6;  (but no pointer arithmetic)
var v7 map[string]int       // unordered_map<string, int>* v7;  (approximately)
var v8 func(a int) int      // int (*v8)(int a);
```

Declarations generally take the form of a keyword followed by the name of the object being declared. The keyword is one of `var`, `func`, `const`, or `type`. Method declarations are a minor exception in that the receiver appears before the name of the object being declared; see the discussion of interfaces.

You can also use a keyword followed by a series of declarations in parentheses.

```
var (
    i int
    m float64
)
```

When declaring a function, you must either provide a name for each parameter or not provide a name for any parameter. (That is, C++ permits `void f(int i, int);`, but Go does not permit the analogous `func f(i int, int)`.) However, for convenience, in Go you may group several names with the same type:

```
func f(i, j, k int, s, t string)
```

A variable may be initialized when it is declared. When this is done, specifying the type is permitted but not required. When the type is not specified, the type of the variable is the type of the initialization expression.

```
var v = *p
```

See also the discussion of constants, below. If a variable is not initialized explicitly, the type must be specified. In that case it will be implicitly initialized to the type's zero value (`0`, `nil`, etc.). There are no uninitialized variables in Go.

Within a function, a short declaration syntax is available with `:=`.

```
v1 := v2 // C++11: auto v1 = v2;
```

This is equivalent to

```
var v1 = v2 // C++11: auto v1 = v2;
```

Go permits multiple assignments, which are done in parallel. That is, first all of the values on the right-hand side are computed, and then these values are assigned to the variables on the left-hand side.

```
i, j = j, i // Swap i and j.
```

Functions may have multiple return values, indicated by a list in parentheses. The returned values can be stored by assignment to a list of variables.

```
func f() (i int, j int) { ... }
v1, v2 = f()
```

Multiple return values are Go's primary mechanism for error handling:

```
result, ok := g()
if !ok {
  // Something bad happened.
  return nil
}
// Continue as normal.
…
```

or, more tersely,

```
if result, ok := g(); !ok {
  // Something bad happened.
  return nil
}
// Continue as normal.
…
```

Go code uses very few semicolons in practice. Technically, all Go statements are terminated by a semicolon. However, Go treats the end of a non-blank line as a semicolon unless the line is clearly incomplete (the exact rules are in the language specification). A consequence of this is that in some cases Go does not permit you to use a line break. For example, you may not write

```
func g()
{                   // INVALID
}
```

A semicolon will be inserted after `g()`, causing it to be a function declaration rather than a function definition. Similarly, you may not write

```
if x {
}
```

```
else {              // INVALID
}
```

A semicolon will be inserted after the `}` preceding the `else`, causing a syntax error.

Since semicolons do end statements, you may continue using them as in C++. However, that is not the recommended style. Idiomatic Go code omits unnecessary semicolons, which in practice is all of them other than the initial `for` loop clause and cases where you want several short statements on a single line.

While we're on the topic, we recommend that rather than worry about semicolons and brace placement, you format your code with the `gofmt` program. That will produce a single standard Go style, and let you worry about your code rather than your formatting. While the style may initially seem odd, it is as good as any other style, and familiarity will lead to comfort.

When using a pointer to a struct, you use `.` instead of `->`. Thus, syntactically speaking, a structure and a pointer to a structure are used in the same way.

```
type myStruct struct{ i int }
var v9 myStruct  // v9 has structure type
var p9 *myStruct // p9 is a pointer to a structure
f(v9.i, p9.i)
```

Go does not require parentheses around the condition of an `if` statement, or the expressions of a `for` statement, or the value of a `switch` statement. On the other hand, it does require curly braces around the body of an `if` or `for` statement.

```
if a < b { f() }              // Valid
if (a < b) { f() }            // Valid (condition is a parenthesized expression)
if (a < b) f()                // INVALID
for i = 0; i < 10; i++ {}     // Valid
for (i = 0; i < 10; i++) {}   // INVALID
```

Go does not have a `while` statement nor does it have a `do/while` statement. The `for` statement may be used with a single condition, which makes it equivalent to a `while` statement. Omitting the condition entirely is an endless loop.

Go permits `break` and `continue` to specify a label. The label must refer to a `for`, `switch`, or `select` statement.

In a `switch` statement, `case` labels do not fall through. You can make them fall through using the `fallthrough` keyword. This applies even to adjacent cases.

```
switch i {
case 0: // empty case body
case 1:
    f() // f is not called when i == 0!
}
```

But a `case` can have multiple values.

```
switch i {
case 0, 1:
    f() // f is called if i == 0 || i == 1.
}
```

The values in a `case` need not be constants--or even integers; any type that supports the equality comparison operator, such as strings or pointers, can be used--and if the `switch` value is omitted it defaults to `true`.

```
switch {
case i < 0:
    f1()
case i == 0:
    f2()
case i > 0:
    f3()
}
```

The `defer` statement may be used to call a function after the function containing the `defer` statement returns. `defer` often takes the place of a destructor in C++ but is associated with the calling code, not any particular class or object.

```
fd := open("filename")
defer close(fd) // fd will be closed when this function returns.
```

Operators

The `++` and `--` operators may only be used in statements, not in expressions. You cannot write `c = *p++`. `*p++` is parsed as `(*p)++`.

The operator precedence is different. As an example `4 & 3 << 1` evaluates to `0` in Go and `4` in C++.

```
Go operator precedence:
1. *   /   %  <<  >>  &  &^
2. +   -   |  ^
3. ==  !=  <  <=  >   >=
4. &&
5. ||
```

```
C++ operator precedence (only relevant operators):
1.  *    /   %
2.  +    -
3.  <<   >>
4.  <    <=  >   >=
5.  ==   !=
6.  &
7.  ^
8.  |
9.  &&
10. ||
```

Constants

In Go constants may be *untyped*. This applies even to constants named with a `const` declaration, if no type is given in the declaration and the initializer expression uses only untyped constants. A value derived from an untyped constant becomes typed when it is used within a context that requires a typed value. This permits constants to be used relatively freely without requiring general implicit type conversion.

```
var a uint
f(a + 1) // untyped numeric constant "1" becomes typed as uint
```

The language does not impose any limits on the size of an untyped numeric constant or constant expression. A limit is only applied when a constant is used where a type is required.

```
const huge = 1 << 100
f(huge >> 98)
```

Go does not support enums. Instead, you can use the special name `iota` in a single `const` declaration to get a series of increasing value. When an initialization expression is omitted for a `const`, it reuses the preceding expression.

```
const (
    red   = iota // red == 0
    blue         // blue == 1
    green        // green == 2
)
```

Types

C++ and Go provide similar, but not identical, built-in types: signed and unsigned integers of various widths, 32-bit and 64-bit floating-point numbers (real and complex), `struct`s, pointers, etc. In Go, `uint8`, `int64`, and like-named integer types are part of the language, not built on top of integers whose sizes are implementation-dependent (e.g., `long long`). Go additionally provides native `string`, `map`, and `chan` (channel) types as well as first-class arrays and slices (described below). Strings are encoded with Unicode, not ASCII.

Go is far more strongly typed than C++. In particular, there is no implicit type conversion in Go, only explicit type conversion. This provides additional safety and freedom from a class of bugs but at the cost of some additional typing. There is also no `union` type in Go, as this would enable subversion of the type system. However, a Go `interface{}` (see below) provides a type-safe alternative.

Both C++ and Go support type aliases (`typedef` in C++, `type` in Go). However, unlike C++, Go treats these as different types. Hence, the following is valid in C++:

```
// C++
typedef double position;
typedef double velocity;

position pos = 218.0;
velocity vel = -9.8;
```

```
pos += vel;
```

but the equivalent is invalid in Go without an explicit type conversion:

```
type position float64
type velocity float64

var pos position = 218.0
var vel velocity = -9.8

pos += vel // INVALID: mismatched types position and velocity
// pos += position(vel)  // Valid
```

The same is true even for unaliased types: an `int` and a `uint` cannot be combined in an expression without explicitly converting one to the other.

Go does not allow pointers to be cast to and from integers, unlike in C++. However, Go's `unsafe` package enables one to explicitly bypass this safety mechanism if necessary (e.g., for use in low-level systems code).

Slices

A slice is conceptually a struct with three fields: a pointer to an array, a length, and a capacity. Slices support the `[]` operator to access elements of the underlying array. The builtin `len` function returns the length of the slice. The builtin `cap` function returns the capacity.

Given an array, or another slice, a new slice is created via `a[i:j]`. This creates a new slice that refers to `a`, starts at index `i`, and ends before index `j`. It has length `j-i`. If `i` is omitted, the slice starts at `0`. If `j` is omitted, the slice ends at `len(a)`. The new slice refers to the same array to which `a` refers. Two implications of this statement are that ① changes made using the new slice may be seen using `a`, and ② slice creation is (intended to be) cheap; no copy needs to be made of the underlying array. The capacity of the new slice is simply the capacity of `a` minus `i`. The capacity of an array is the length of the array.

What this means is that Go uses slices for some cases where C++ uses pointers. If you create a value of type `[100]byte` (an array of 100 bytes, perhaps a buffer) and you want to pass it to a function without copying it, you should declare the function parameter to have type `[]byte`, and pass a slice of the array (`a[:]` will pass the entire array). Unlike in C++, it is not necessary to pass the length of the buffer; it is efficiently accessible via `len`.

The slice syntax may also be used with a string. It returns a new string, whose value is a substring of the original string. Because strings are immutable, string slices can be implemented without allocating new storage for the slices's contents.

Making values

Go has a builtin function `new` which takes a type and allocates space on the heap. The allocated space will be zero-initialized for the type. For example, `new(int)` allocates a new int on the heap, initializes it with the value `0`, and returns its address, which has type `*int`. Unlike in C++, `new` is a function, not an operator; `new int` is a syntax error.

Perhaps surprisingly, `new` is not commonly used in Go programs. In Go taking the address of a variable is always safe and never yields a dangling pointer. If the program takes the address of a variable, it will be allocated on the heap if necessary. So these functions are equivalent:

```
type S struct { I int }

func f1() *S {
    return new(S)
}

func f2() *S {
    var s S
    return &s
}

func f3() *S {
    // More idiomatic: use composite literal syntax.
    return &S{}
}
```

In contrast, it is not safe in C++ to return a pointer to a local variable:

```
// C++
S* f2() {
  S s;
  return &s;   // INVALID -- contents can be overwritten at any time
}
```

Map and channel values must be allocated using the builtin function `make`. A variable declared with map or channel type without an initializer will be automatically initialized to `nil`. Calling `make(map[int]int)` returns a newly allocated value of type `map[int]int`. Note that `make` returns a value, not a pointer. This is consistent with the fact that map and channel values are passed by reference. Calling `make` with a map type takes an optional argument which is the expected capacity of the map. Calling `make` with a channel type takes an optional argument which sets the buffering capacity of the channel; the default is 0 (unbuffered).

The `make` function may also be used to allocate a slice. In this case it allocates memory for the underlying array and returns a slice referring to it. There is one required argument, which is the number of elements in the slice. A second, optional, argument is the capacity of the slice. For example, `make([]int, 10, 20)`. This is identical to `new([20]int)[0:10]`. Since Go uses garbage collection, the newly allocated array will be discarded sometime after there are no references to the returned slice.

Interfaces

Where C++ provides classes, subclasses and templates, Go provides interfaces. A Go interface is similar to a C++ pure abstract class: a class with no data members, with methods which are all pure virtual. However, in Go, any type which provides the methods named in the interface may be treated as an implementation of the interface. No explicitly declared inheritance is required. The implementation of the interface is entirely separate from the interface itself.

A method looks like an ordinary function definition, except that it has a *receiver*. The receiver is similar to the `this` pointer in a C++ class method.

```
type myType struct{ i int }

func (p *myType) Get() int { return p.i }
```

This declares a method `Get` associated with `myType`. The receiver is named `p` in the body of the function.

Methods are defined on named types. If you convert the value to a different type, the new value will have the methods of the new type, not the old type.

You may define methods on a builtin type by declaring a new named type derived from it. The new type is distinct from the builtin type.

```
type myInteger int

func (p myInteger) Get() int { return int(p) } // Conversion required.
func f(i int)                {}

var v myInteger

// f(v) is invalid.
// f(int(v)) is valid; int(v) has no defined methods.
```

Given this interface:

```
type myInterface interface {
    Get() int
    Set(i int)
}
```

we can make `myType` satisfy the interface by adding

```
func (p *myType) Set(i int) { p.i = i }
```

Now any function which takes `myInterface` as a parameter will accept a variable of type `*myType`.

```
func GetAndSet(x myInterface) {}
func f1() {
    var p myType
    GetAndSet(&p)
}
```

In other words, if we view `myInterface` as a C++ pure abstract base class, defining `Set` and `Get` for `*myType` made `*myType` automatically inherit from `myInterface`. A type may satisfy multiple interfaces.

An anonymous field may be used to implement something much like a C++ child class.

```
type myChildType struct {
    myType
    j int
}

func (p *myChildType) Get() int { p.j++; return p.myType.Get() }
```

This effectively implements `myChildType` as a child of `myType`.

```
func f2() {
    var p myChildType
    GetAndSet(&p)
}
```

The `Set` method is effectively inherited from `myType`, because methods associated with the anonymous field are promoted to become methods of the enclosing type. In this case, because `myChildType` has an anonymous field of type `myType`, the methods of `myType` also become methods of `myChildType`. In this example, the `Get` method was overridden, and the `Set` method was inherited.

This is not precisely the same as a child class in C++. When a method of an anonymous field is called, its receiver is the field, not the surrounding struct. In other words, methods on anonymous fields are not virtual functions. When you want the equivalent of a virtual function, use an interface.

A variable that has an interface type may be converted to have a different interface type using a special construct called a type assertion. This is implemented dynamically at run time, like C++ `dynamic_cast`. Unlike `dynamic_cast`, there does not need to be any declared relationship between the two interfaces.

```
type myPrintInterface interface {
    Print()
}

func f3(x myInterface) {
    x.(myPrintInterface).Print() // type assertion to myPrintInterface
}
```

The conversion to `myPrintInterface` is entirely dynamic. It will work as long as the dynamic type of x defines a `Print` method.

Because the conversion is dynamic, it may be used to implement generic programming similar to templates in C++. This is done by manipulating values of the minimal interface.

```
type Any interface{}
```

Containers may be written in terms of `Any`, but the caller must unbox using a type assertion to recover values of the contained type. As the typing is dynamic rather than static, there is no equivalent of the way that a C++ template may inline the relevant operations. The operations are fully type-checked at run time, but all operations will involve a function call.

```
type Iterator interface {
    Get() Any
    Set(v Any)
    Increment()
    Equal(arg Iterator) bool
}
```

Note that `Equal` has an argument of type `Iterator`. This does not behave like a C++ template. See the FAQ.

Function closures

In C++ versions prior to C++11, the most common way to create a function with hidden state is to use a "functor"—a class that overloads `operator()` to make instances look like functions. For example, the following code defines a `my_transform` function (a simplified version of the STL's `std::transform`) that applies a given unary operator (`op`) to each element of an array (`in`), storing the result in another array (`out`). To implement a prefix sum (i.e., {`x[0]`, `x[0]+x[1]`, `x[0]+x[1]+x[2]`, ...}) the code creates a functor (`MyFunctor`) that keeps track of the running total (`total`) and passes an instance of this functor to `my_transform`.

```
// C++
#include <iostream>
#include <cstddef>

template <class UnaryOperator>
void my_transform (size_t n_elts, int* in, int* out, UnaryOperator op)
{
  size_t i;

  for (i = 0; i < n_elts; i++)
    out[i] = op(in[i]);
}

class MyFunctor {
public:
  int total;
  int operator()(int v) {
    total += v;
    return total;
  }
  MyFunctor() : total(0) {}
};

int main (void)
{
  int data[7] = {8, 6, 7, 5, 3, 0, 9};
  int result[7];
  MyFunctor accumulate;
  my_transform(7, data, result, accumulate);

  std::cout << "Result is [ ";
  for (size_t i = 0; i < 7; i++)
```

```
    std::cout << result[i] << ' ';
  std::cout << "]\n";
  return 0;
}
```

C++11 adds anonymous ("lambda") functions, which can be stored in variables and passed to functions. They can optionally serve as closures, meaning they can reference state from parent scopes. This feature greatly simplifies `my_transform`:

```
// C++11
#include <iostream>
#include <cstddef>
#include <functional>

void my_transform (size_t n_elts, int* in, int* out, std::function<int(int)> op)
{
  size_t i;

  for (i = 0; i < n_elts; i++)
    out[i] = op(in[i]);
}

int main (void)
{
  int data[7] = {8, 6, 7, 5, 3, 0, 9};
  int result[7];
  int total = 0;
  my_transform(7, data, result, [&total] (int v) {
      total += v;
      return total;
    });

  std::cout << "Result is [ ";
  for (size_t i = 0; i < 7; i++)
    std::cout << result[i] << ' ';
  std::cout << "]\n";
  return 0;
}
```

A typical Go version of `my_transform` looks a lot like the C++11 version:

```
package main

import "fmt"

func my_transform(in []int, xform func(int) int) (out []int) {
    out = make([]int, len(in))
    for idx, val := range in {
        out[idx] = xform(val)
    }
```

```
    return
}

func main() {
    data := []int{8, 6, 7, 5, 3, 0, 9}
    total := 0
    fmt.Printf("Result is %v\n", my_transform(data, func(v int) int {
        total += v
        return total
    }))
}
```

(Note that we chose to return `out` from `my_transform` rather than pass it an `out` to write to. This was an aesthetic decision; the code could have been written more like the C++ version in that regard.)

In Go, functions are always full closures, the equivalent of `[&]` in C++11. An important difference is that it is invalid in C++11 for a closure to reference a variable whose scope has gone away (as may be caused by an upward funarg—a function that returns a lambda that references local variables). In Go, this is perfectly valid.

Concurrency

Like C++11's `std::thread`, Go permits starting new threads of execution that run concurrently in a shared address space. These are called *goroutines* and are spawned using the `go` statement. While typical `std::thread` implementations launch heavyweight, operating-system threads, goroutines are implemented as lightweight, user-level threads that are multiplexed among multiple operating-system threads. Consequently, goroutines are (intended to be) cheap and can be used liberally throughout a program.

```
func server(i int) {
    for {
        fmt.Print(i)
        time.Sleep(10 * time.Second)
    }
}
go server(1)
go server(2)
```

(Note that the `for` statement in the `server` function is equivalent to a C++ `while (true)` loop.)

Function literals (which Go implements as closures) can be useful with the `go` statement.

```
var g int
go func(i int) {
    s := 0
    for j := 0; j < i; j++ {
        s += j
    }
    g = s
}(1000) // Passes argument 1000 to the function literal.
```

Like C++11, but unlike prior versions of C++, Go defines a memory model for unsynchronized accesses to memory. Although Go provides an analogue of `std::mutex` in its `sync` package, this is not the normal way to implement inter-thread communication and synchronization in Go programs. Instead, Go threads more typically communicate by message passing, which is a fundamentally different approach from locks and barriers. The Go mantra for this subject is,

> *Do not communicate by sharing memory; instead, share memory by communicating.*

That is, *channels* are used to communicate among goroutines. Values of any type (including other channels!) can be sent over a channel. Channels can be unbuffered or buffered (using a buffer length specified at channel-construction time).

Channels are first-class values; they can be stored in variables and passed to and from functions like any other value. (When supplied to functions, channels are passed by reference.) Channels are also typed: a `chan int` is different from a `chan string`.

Because they are so widely used in Go programs, channels are (intended to be) efficient and cheap. To send a value on a channel, use `<-` as a binary operator. To receive a value on a channel, use `<-` as a unary operator. Channels can be shared among multiple senders and multiple receivers and guarantee that each value sent is received by at most one receiver.

Here is an example of using a manager function to control access to a single value.

```
type Cmd struct {
    Get bool
    Val int
}

func Manager(ch chan Cmd) {
    val := 0
    for {
        c := <-ch
        if c.Get {
            c.Val = val
            ch <- c
        } else {
            val = c.Val
        }
    }
}
```

In that example the same channel is used for input and output. This is incorrect if there are multiple goroutines communicating with the manager at once: a goroutine waiting for a response from the manager might receive a request from another goroutine instead. A solution is to pass in a channel.

```
type Cmd2 struct {
    Get bool
    Val int
    Ch  chan<- int
}

func Manager2(ch <-chan Cmd2) {
```

```
    val := 0
    for {
        c := <-ch
        if c.Get {
            c.Ch <- val
        } else {
            val = c.Val
        }
    }
}
```

To use `Manager2`, given a channel to it:

```
func getFromManagedChannel(ch chan<- Cmd2) int {
    myCh := make(chan int)
    c := Cmd2{true, 0, myCh} // Composite literal syntax.
    ch <- c
    return <-myCh
}

func main() {
    ch := make(chan Cmd2)
    go Manager2(ch)
    // ... some code ...
    currentValue := getFromManagedChannel(ch)
    // ... some more code...
}
```