Type definitions

In Julia we can also define the primitive types the way we want. So what is primitive types? These are the types which directly uses the memory sizes.

```
# primitive type  Float64 64  end  ## I don't know what is going to happen, you can take t
```

There is a package called **`Bio.jl`** which does define their own primitive types for DNA or something. So in principle you can do this. However before doing this we should think about that whether we can use already defined primitive types for our work, then there is absolutely no necessary to do this.

Now let's define our complex number

```
struct MyComplex
    real::Float64
    imag::Float64
end
```

This is actually called a **Composite type**, the things inside is like a defining characteristic of the type. They are called fields. Here we are defining our own type that is kind of using the already defined primitive type. So We can use a function **`isbits`** now to see whether this type is a deducible primitive type. The following command will print true if your type can be represented as plain bits.

```
isbits(1.0)
isbits(MyComplex(1.0, 1.0))
```

Let's try this one, where we have an abstract float as

```
struct MyComplex2
    real::AbstractFloat
    imag::Float64
end
```

```
isbits(1.0)
isbits(MyComplex2(1.0, 2))
```

You will see that we can instantiate both types, meaning if you use the type constructor you can create an object. Now for the first one the important thing is it is just like a primitive type, but not directly (that is why the name deducible!). Important thing is this will not work like a

pointer, rather this will work as if we have a primitive type. So it knows the information at the compile time that the things that are always going to be **`Float64`**. So **`MyComplex`** will just take **`128`** bits.

Now we can also define operations on these types.

```
Base.:+(a::MyComplex,b::MyComplex) = MyComplex(a.real + b.real, a.imag + b.imag)
Base.:+(a::MyComplex,b::Int) = MyComplex(a.real + b, a.imag)
Base.:+(b::Int,a::MyComplex) = MyComplex(a.real + b, a.imag)
```

Now that we have defined a type and some operations we can write some functions,

```
f(x, y) = x + y

mc1 = MyComplex(1.0, 2.0)
mc2 =  MyComplex(4.0, 2.0)

mc1 + mc2
f(mc1, mc2)
```

```
@code_warntype f(mc1, mc2)

# julia> @code_warntype f(mc1, mc2)
# Variables
#   #self#::Core.Compiler.Const(f, false)
#   x::MyComplex
#   y::MyComplex

# Body::MyComplex
# 1 ─ %1 = (x + y)::MyComplex
# └──      return %1
```

So this is nice, because it's like as if things are happening just for floating point numbers. We can also look at the code that is generating,

```
@code_llvm f(mc1, mc2)
```

If we look, then the pattern that you would find, it is doing the two operations simultaneously, this is what is known as SIMD (Single instruction, multiple data). It is faster then if we have two floating point variables and then add reals and then complex. Its somewhere like doing somekind of parallel type thing.

Parametric Types (flexibility + speed)

The problem with **MyComplex** type is it only works on 64 bit floating point numbers. What if I want to have this kind of operations for 32 bit Floats, should I define another type? NO no, not again, julia always tries to offer something so that we stop copying and pasting code. In this case it will be the **parametric type**. Well for now we have to otherwise, we have to restart the kernel, but in general, this would be the better thing to do, if you want to have a generic type. In the following **T** is a parameter.

```
struct MyParameterizedComplex{T}
  real::T
  imag::T
end
isbits(MyParameterizedComplex(1.0,1.0))
```

Note that **MyParameterizedComplex{T}** is a concrete type for every **T**: it is a shorthand form for defining a whole family of types.

Again we can define functions on them,

```
Base.:+(a::MyParameterizedComplex,b::MyParameterizedComplex) = MyParameterizedComplex(a.rea
Base.:+(a::MyParameterizedComplex,b::Int) = MyParameterizedComplex(a.real+b,a.imag)
Base.:+(b::Int,a::MyParameterizedComplex) = MyParameterizedComplex(a.real+b,a.imag)


f(MyParameterizedComplex(1.0,1.0), MyParameterizedComplex(1.0,1.0))
```

```
@code_warntype g(MyParameterizedComplex(1.0,1.0),MyParameterizedComplex(1.0,1.0))
@code_warntype g(MyParameterizedComplex(1.0f0,1.0f0),MyParameterizedComplex(1.0f0,1.0f0))
```

See that this code also automatically works and compiles efficiently for **Float32** as well:

It is important to know that if there is any piece of a type which doesn't contain type information, then it cannot be **isbits** because then it would have to be compiled in such a way that the size is not known in advance. For example:

```
struct MySlowComplex
  real
  imag
end
isbits(MySlowComplex(1.0,1.0))
```

```
Base.:+(a::MySlowComplex,b::MySlowComplex) = MySlowComplex(a.real+b.real,a.imag+b.imag)
Base.:+(a::MySlowComplex,b::Int) = MySlowComplex(a.real+b,a.imag)
```

```
Base.:+(b::Int,a::MySlowComplex) = MySlowComplex(a.real+b,a.imag)

f(MySlowComplex(1.0,1.0),MySlowComplex(1.0,1.0))
```

```
@code_warntype f(MySlowComplex(1.0,1.0),MySlowComplex(1.0,1.0))
@code_llvm f(MySlowComplex(1.0,1.0),MySlowComplex(1.0,1.0))
```

```
struct MySlowComplex2
  real::AbstractFloat
  imag::AbstractFloat
end
isbits(MySlowComplex2(1.0,1.0))
```

```
Base.:+(a::MySlowComplex2,b::MySlowComplex2) = MySlowComplex2(a.real+b.real,a.imag+b.imag)
Base.:+(a::MySlowComplex2,b::Int) = MySlowComplex2(a.real+b,a.imag)
Base.:+(b::Int,a::MySlowComplex2) = MySlowComplex2(a.real+b,a.imag)
g(MySlowComplex2(1.0,1.0),MySlowComplex2(1.0,1.0))
```

Here's the timings:

```
a = MyComplex(1.0,1.0)
b = MyComplex(2.0,1.0)
@btime f(a,b)
```

```
a = MyParameterizedComplex(1.0,1.0)
b = MyParameterizedComplex(2.0,1.0)
@btime f(a,b)
```

```
a = MySlowComplex(1.0,1.0)
b = MySlowComplex(2.0,1.0)
@btime f(a,b)
```

```
a = MySlowComplex2(1.0,1.0)
b = MySlowComplex2(2.0,1.0)
@btime f(a,b)
```

Another example

Talk about inner constructor and outer constructor.