/\* memory \*/

/\*

\* pointers

\*/

ptr \*void = nil // points nowhere

ptr \*utf = 0x7FFFFFFF // points to 0x7FFFFFFF

// ‘&’ is the address\_of operator

\_ptr \*utf = &ptr // points to ptr

// ‘\*’ is the dereference operator

\*\_ptr = ‘x5F1D’ // here we set the value reference by \_ptr to a new value

/\*

\* getting memory from the OS

\* and returning memory to the OS

\*/

ptr \*u64 = getmem(47, u64) // get a memory block of size 47 \* 8(size of u64) bytes

retmem(ptr) // return memory pointed by ‘ptr’ to the OS

/\*

\* simple stack array

\* NOTE: these may panic if size is bigger than the available stack memory

\*/

array []s32 = {} // empty array

array []s32 // unsupported, it should be initialized at declaration

array []u64 = getmem(47, u64) // this is valid since it is initialized at declaration

array []s32 = { 14, 957, -5, 0, -4096 } // array of size 5 initialized

array [7]s32 = {} // array of size 7 initialized at 0

array [size]s32 // array of size value of ‘size’ uninitialized (garbage values)

/\*

\* even though these statements means the same

\* the compiler does bound checking on

\* ‘[]’ types and because of this it keeps

\* track of the size of the array on debug

\* this feature can be disabled at build

\*/

array []s32 = {} // better way

array \*s32 = nil;// only if you really need it this way

// You can also get OS mem with arrays!

array []u64 = getmem(47, u64)

retmem(array)

/\*

\* define iterative struct

\*/

#array

CustomArray struct {

data []u64

size u64

}