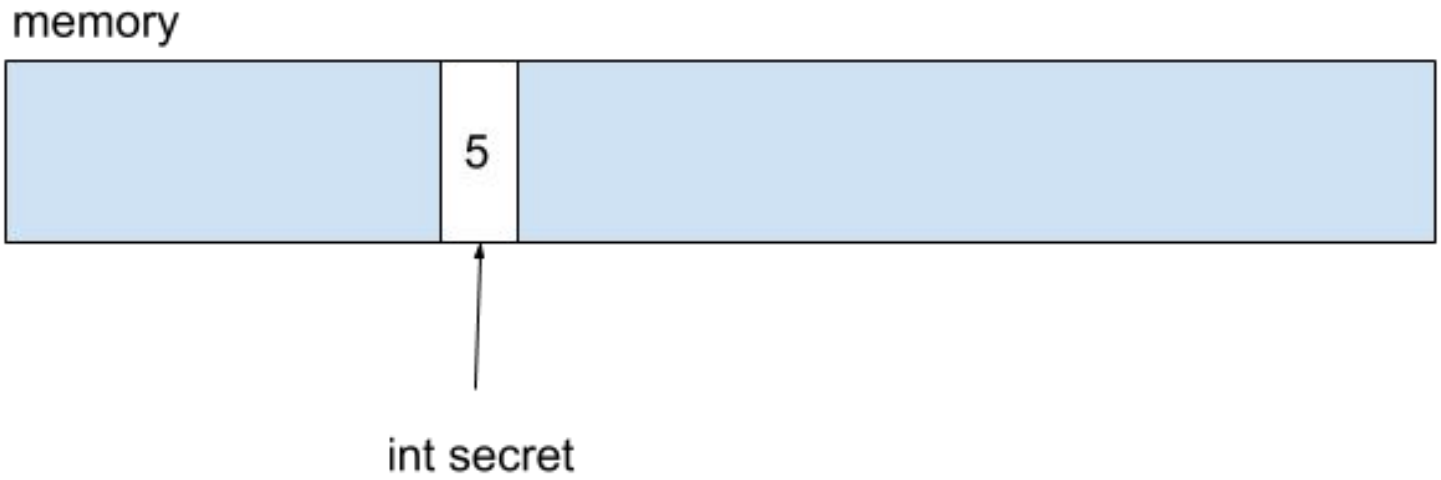
POINTERS

A bedtime story by Lesley Istead
V0.2

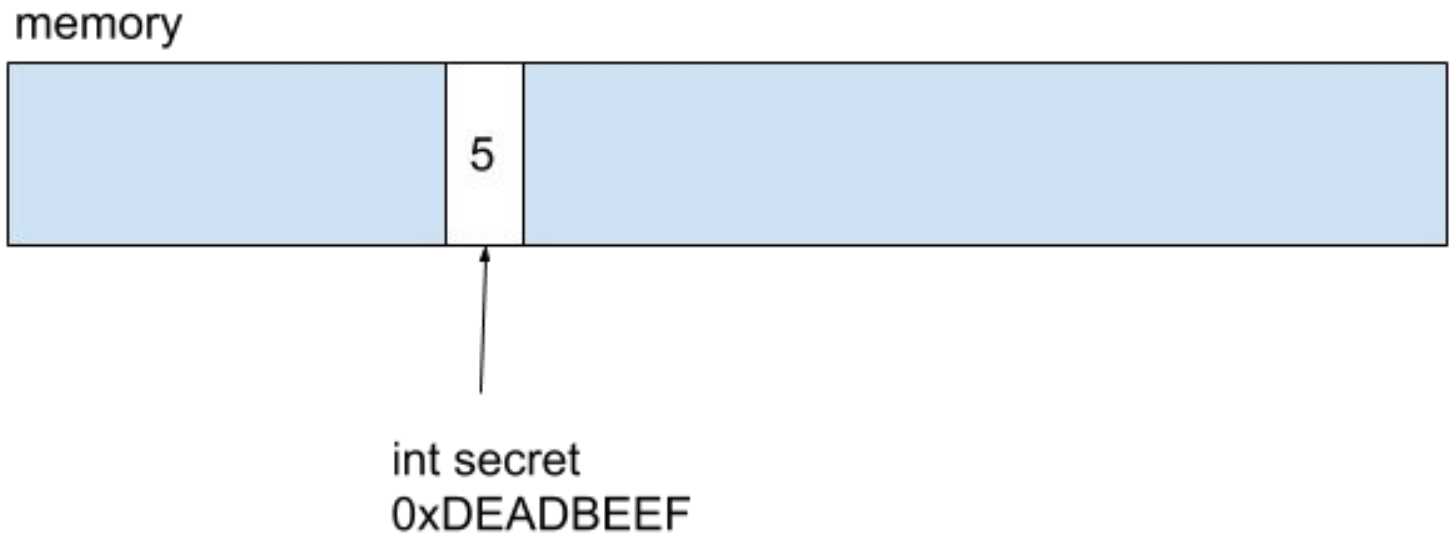
This is a variable.

```
int secret = 5;
```

Variables live in memory.



The location of the variable in memory is an address.



You can ask a variable for its address with an &.

```
&secret -> 0xDEADBEEF
```

You can print the address using %p.

```
printf(“secret’s address is %p\n”, &secret);

secret’s address is 0xDEADBEEF
```

This is a pointer.

```
int * secretPtr;
```

This is also a pointer.

```
char * monsterPtr;
```

A pointer is a type.

The value of a pointer is an address.

This pointer is the address of an integer.

```
int * secretPtr;
```

This pointer is the address of a char.

```
char * monsterPtr;
```

Pointers are variables.

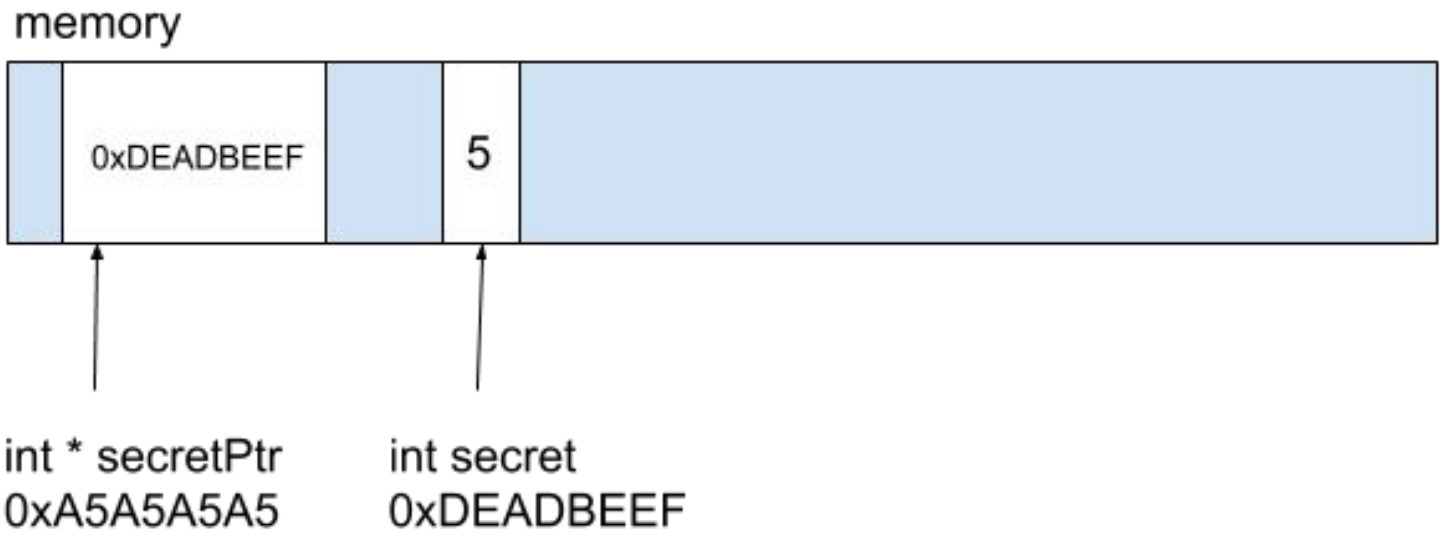
```
int * secretPtr;
```

This variable's value is the address of secret.

```
int * secretPtr = &secret;
```

Variables live in memory.

The location of the variable in memory is an address.



Variable names are for humans.

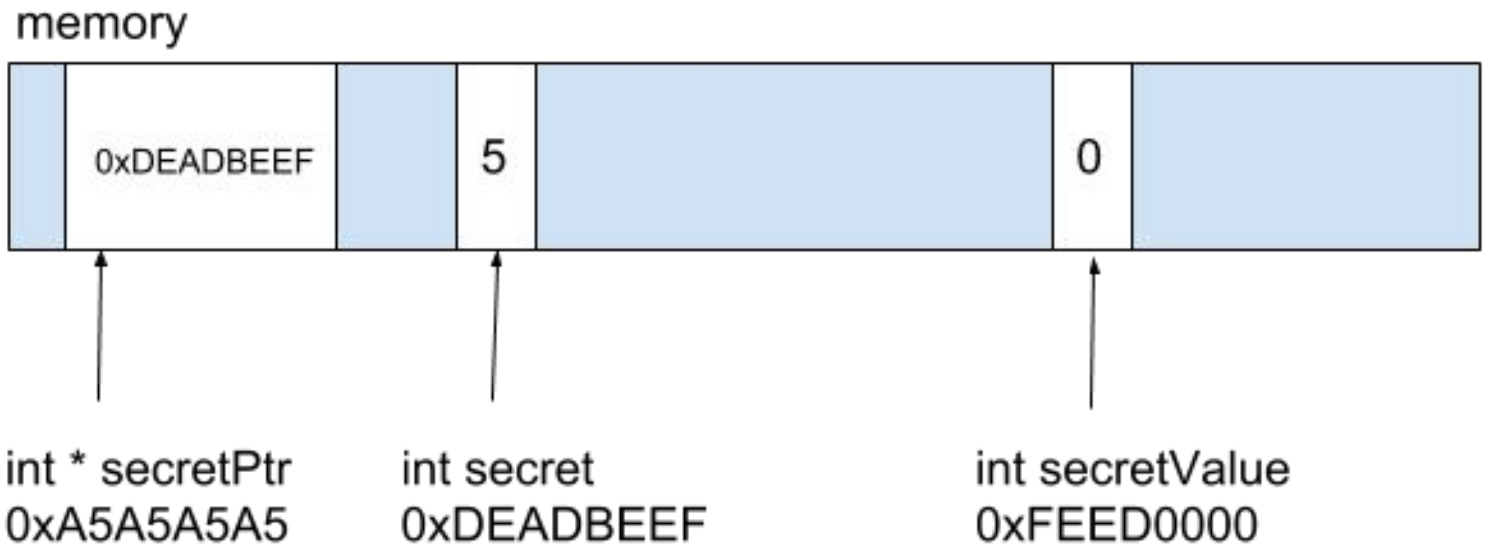
Your computer sees addresses.

human	computer
`int secret = 5;` `int * secretPtr = &secret;`	`0xDEADBEEF = 5;` `0xA5A5A5A5 = 0xDEADBEEF`

This is another variable.

```
int secretValue = 0;
```

The location of the variable in memory is an address.



The dereference or indirection operator (*) gives us the value at the specified address.

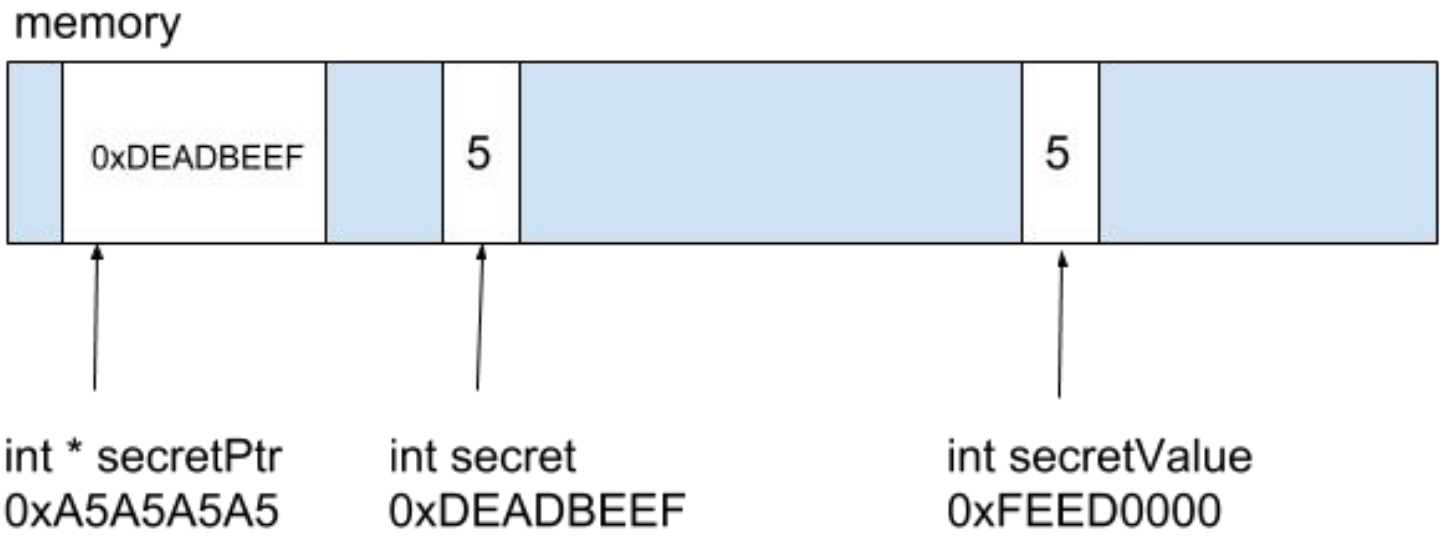
```
int secret = 5;
int * secretPtr = &secret;
int secretValue = *secretPtr;
```

You could also call the dereference or indirection operator the value at operator.

```
printf(“secret’s value is %d\n”, *secretPtr);
```

```
secret’s value is 5
```

The location of the variable in memory is an address.



Variable names are for humans.

human	computer

`int secret = 5;`
`int * secretPtr = &secret;`
`int secretValue = *secretPtr;`
	`0xDEADBEEF = 5;`
	`0xA5A5A5A5 = 0xDEADBEEF`
	`0xFEED0000 = *0xDEADBEEF ->`
	`0xFEED0000 = 5`

Remember that a pointer is just an address.

Here is another pointer.

```
int ** secretPtrPtr;
```

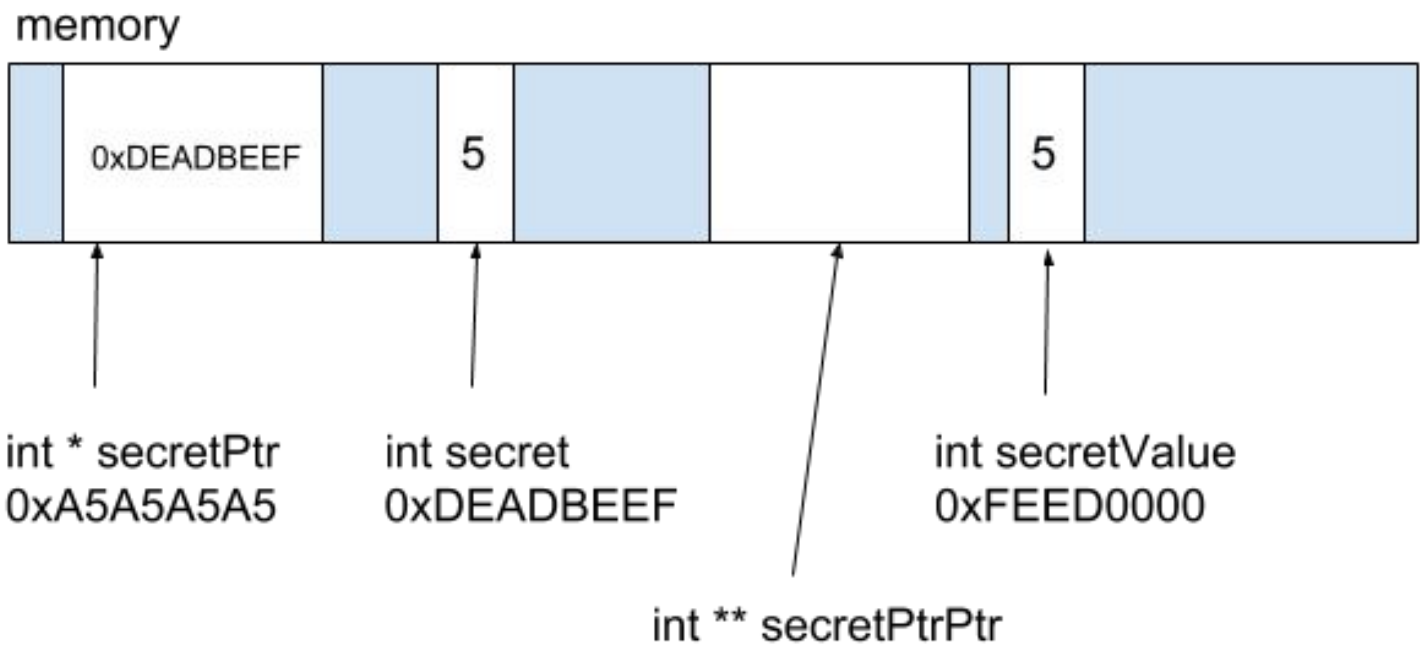
This pointers value is the address of a pointer.

```
int ** secretPtrPtr;
```

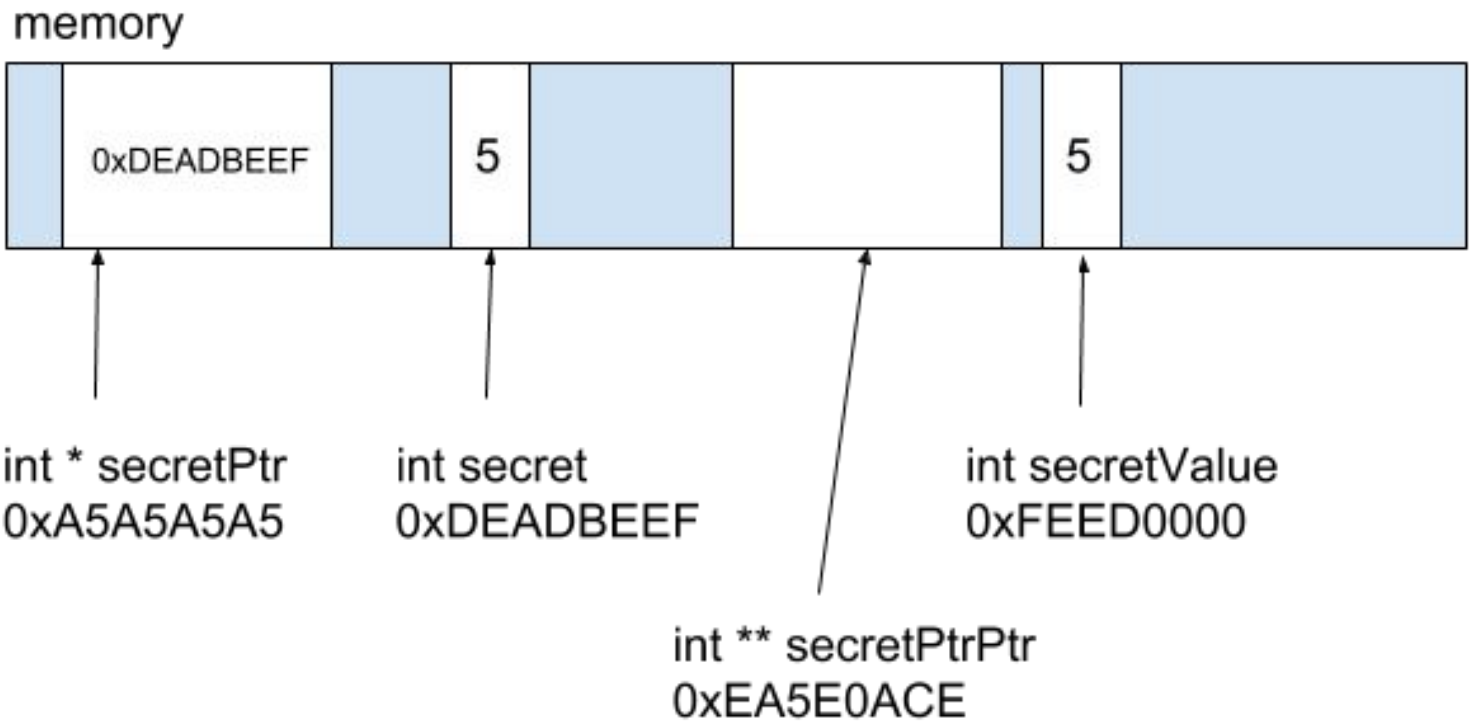
This pointer is a variable.

```
int ** secretPtrPtr;
```

Variables live in memory.



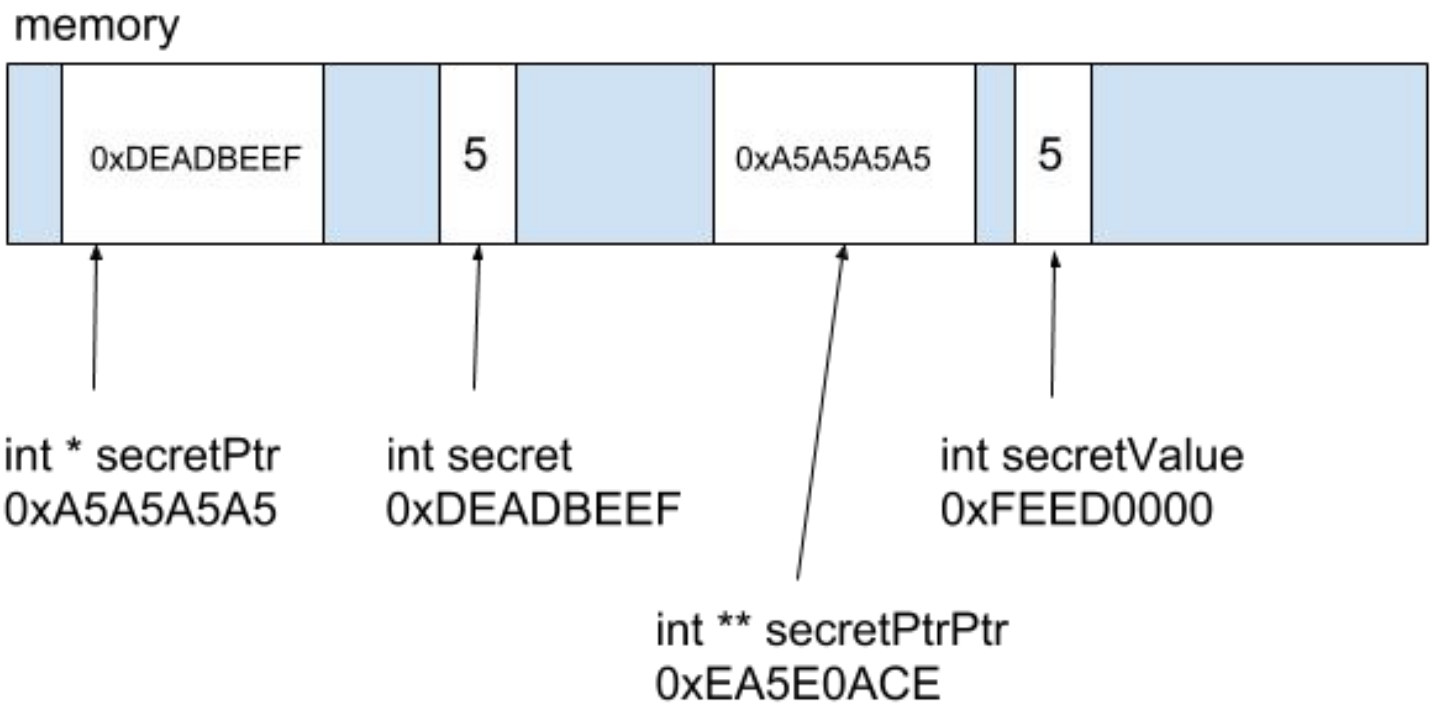
The location of a variable in memory is an address.



We can assign the variable a value-the address of a pointer.

```
int ** secretPtrPtr = &secretPtr;
```

The location of a variable in memory is an address.



The value we assign the variable must have the matching type.

`secretPtrPtr = &secretPtr;` **OK!** `[ADDRESS OF INT *] = [ADDRESS OF INT *]`

`secretPtrPtr = &secretPtrPtr;` **NOT OK!** `[ADDRESS OF INT *] = [ADDRESS of INT **]`

`secretPtrPtr = secretPtr;` **NOT OK!** `[ADDRESS OF INT *] = [ADDRESS OF INT]`

We can get the value at the address with *.

```
printf( “the value at secretPtrPtr is %p\n”,
*secretPtrPtr );

the value at secretPtrPtr is 0xA5A5A5A5
```

We can get the value at ANY address with *.

```
printf( “the value at the address stored in
*secretPtrPtr is %d\n”, **secretPtrPtr );

the value at the address stored in
*secretPtrPtr is 5
```

Variable names are for humans.

human	computer

`int secret = 5;`
`int * secretPtr = &secret;`
`int ** secretPtrPtr = &secretPtr;`
`int secretValue = *secretPtr;`
`int * anAddress = *secretPtrPtr;`
`int aValue = **secretPtrPtr;`
	`0xDEADBEEF = 5;`
	`0xA5A5A5A5 = 0xDEADBEEF;`
	`0xEA5E0ACE = 0xA5A5A5A5;`
	`0xFEED0000 = *0xDEADBEEF = 5`
	
	`0xB4B4B4B4 = *0xA5A5A5A5 = 0xDEADBEEF`
	
	`0x12345678 = **0xA5A5A5A5 = *0xDEADBEEF = 5`

A pointer is just a type.

Its value is an address.

You can change the value of a variable.

```
int x = 18;

x = 8;
```

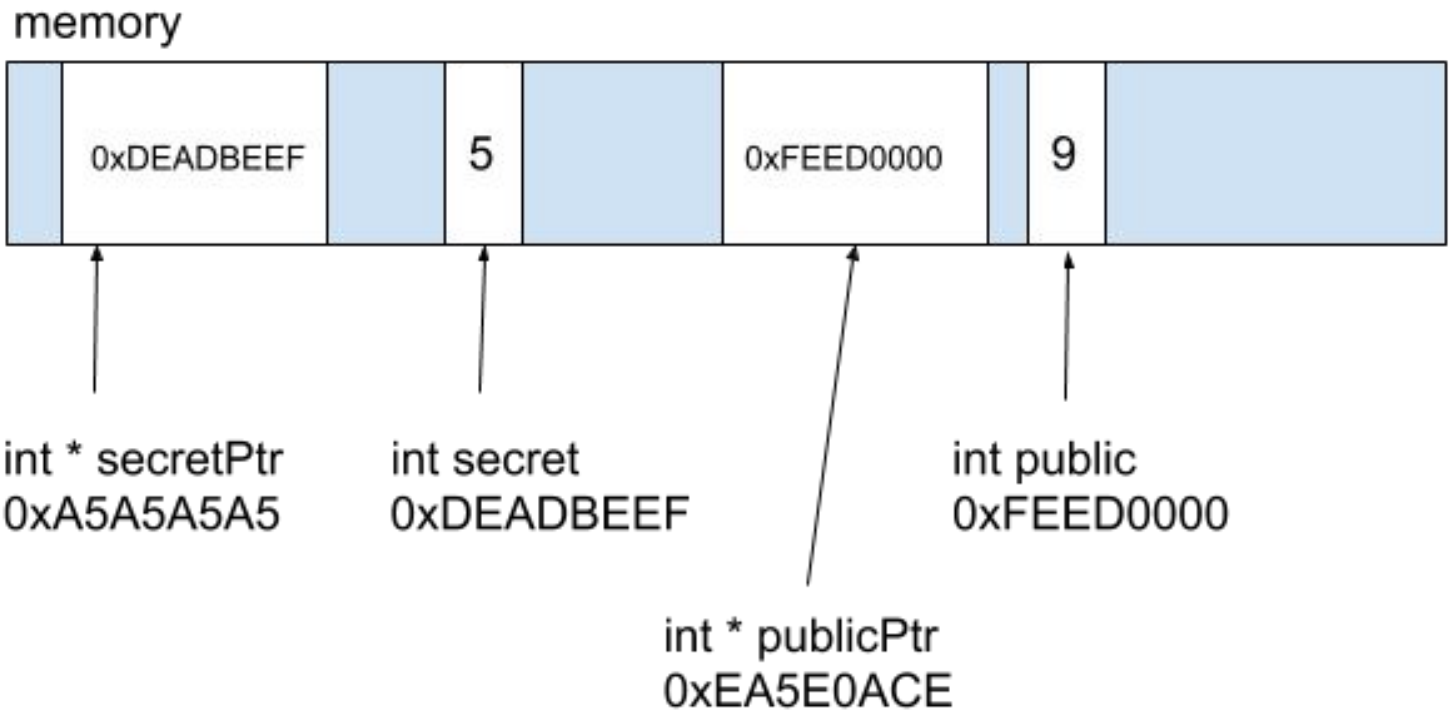
You can change the value of a pointer.

```
int public = 9;

int secret = 5;

int * secretPtr = &secret;

int * publicPtr = &public;

secretPtr = publicPtr;
```

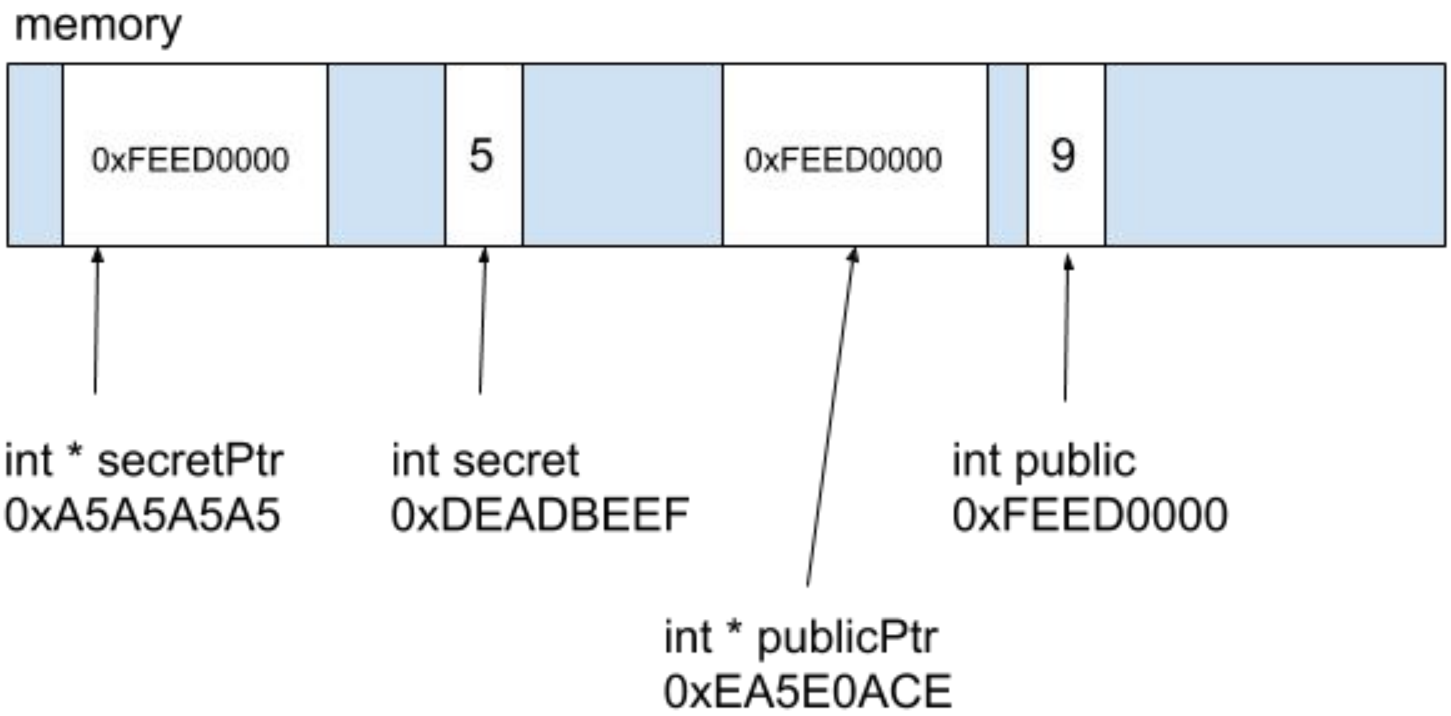
The value of a pointer is an address.

Changing the value of a pointer does not change the value at that address.

publicPtr = &public



SECRETPTR = PUBLICPTR



Changing the value of a pointer does not change the value at that address.

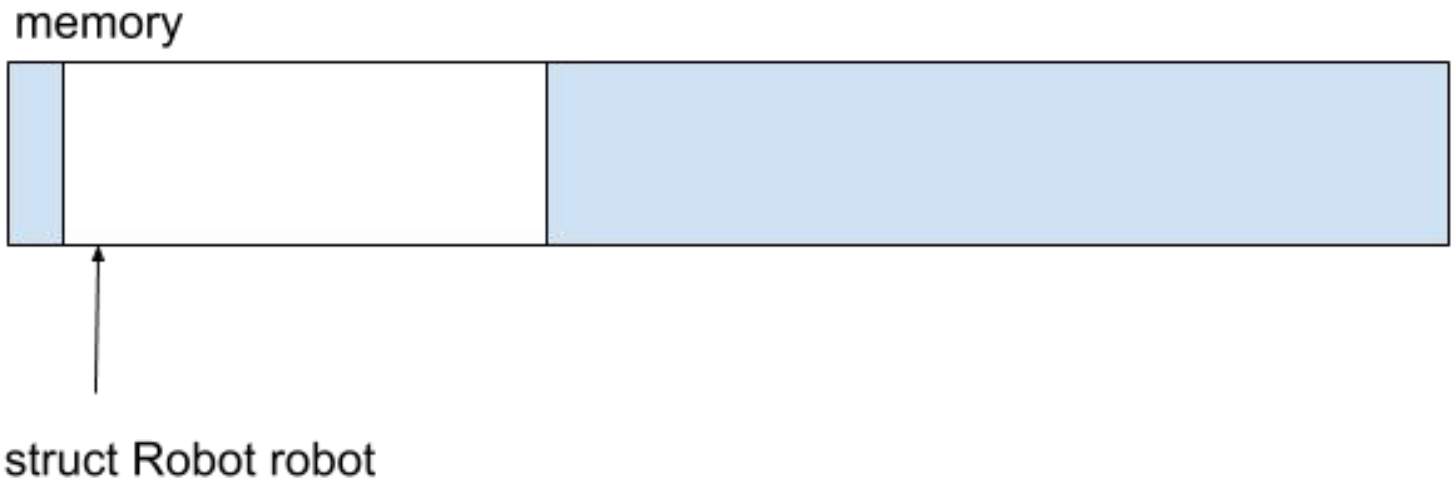
human	computer

`int public = 9;`
`int secret = 5;`
`int * secretPtr = &secret;`
`int * publicPtr = &public;`
`secretPtr = publicPtr;`
	`0xFEED0000 = 9`
	`0xDEADBEEF = 5`
	`0xA5A5A5A5 = 0xDEADBEEF`
	`0xEA5E0ACE = 0xFEED0000`
	`0xA5A5A5A5 = 0xFEED0000`

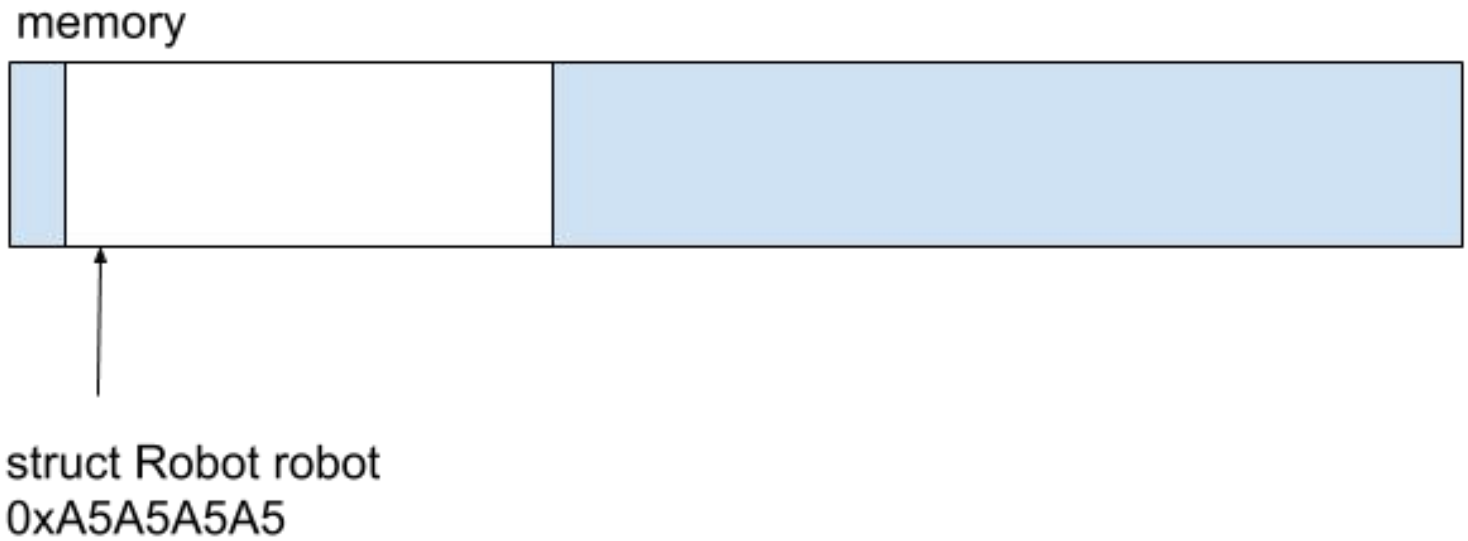
This is a variable.

```
struct Robot robot;
```

Variables live in memory.



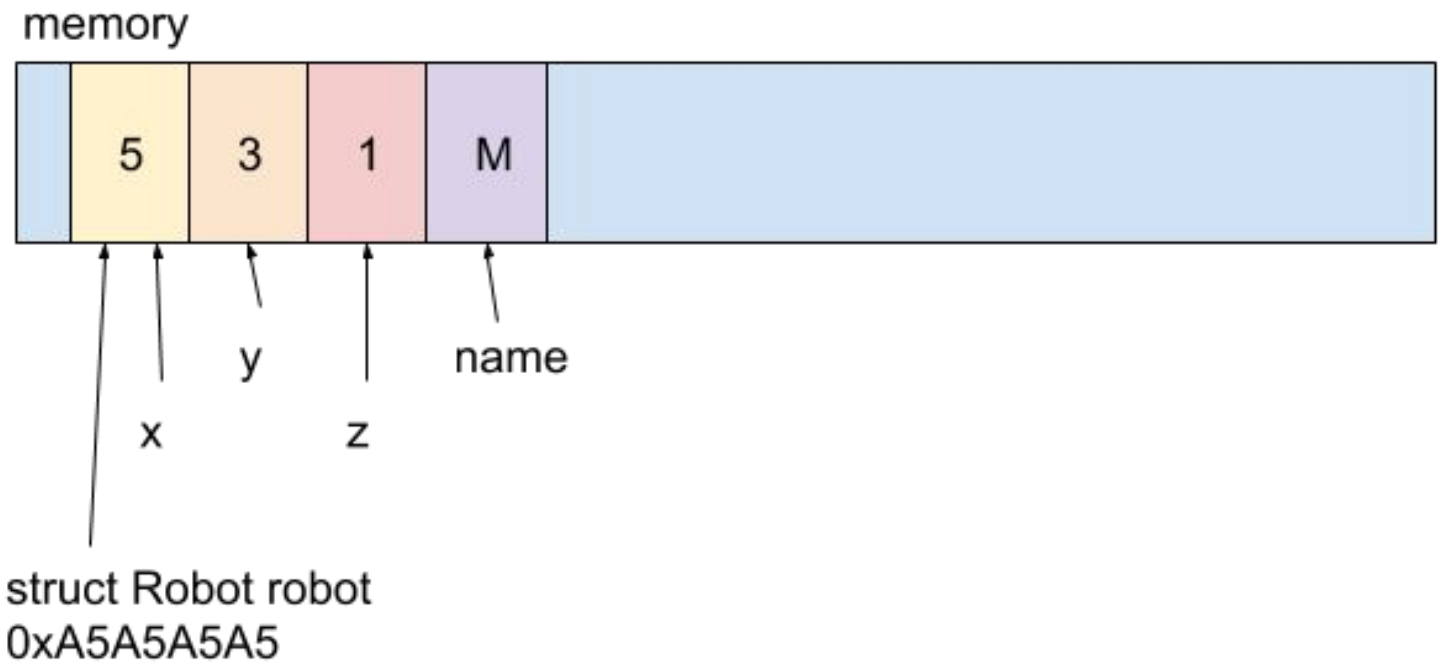
The location of a variable in memory is an address.



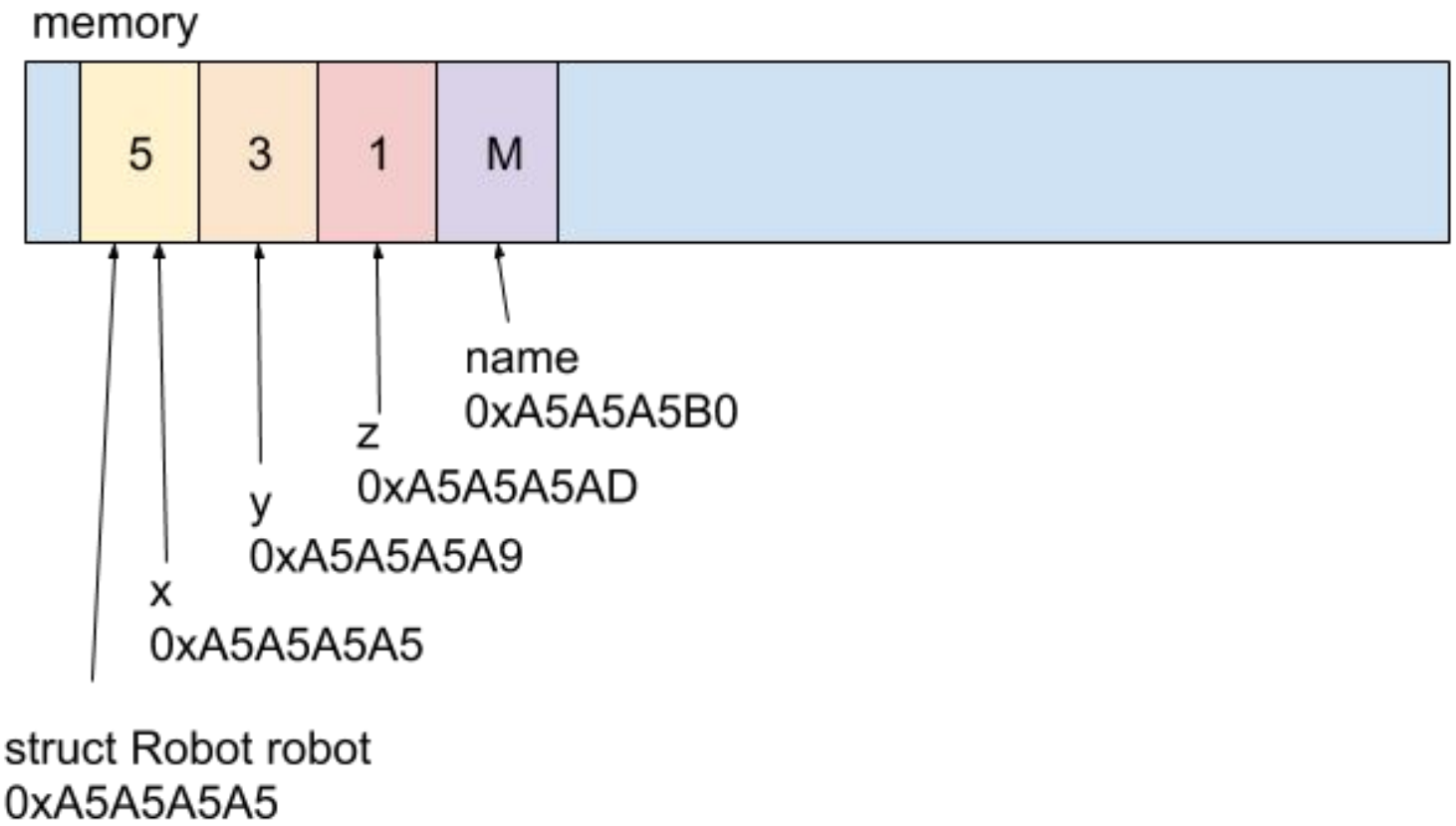
A struct has fields (variables).

```
struct Robot {
    int x;
    int y;
    int z;
    char colour;
};
```

The fields of a struct live in memory.



The location of a field in memory is an address.



A pointer is a type.

The value of a pointer is an address.

This is a pointer.

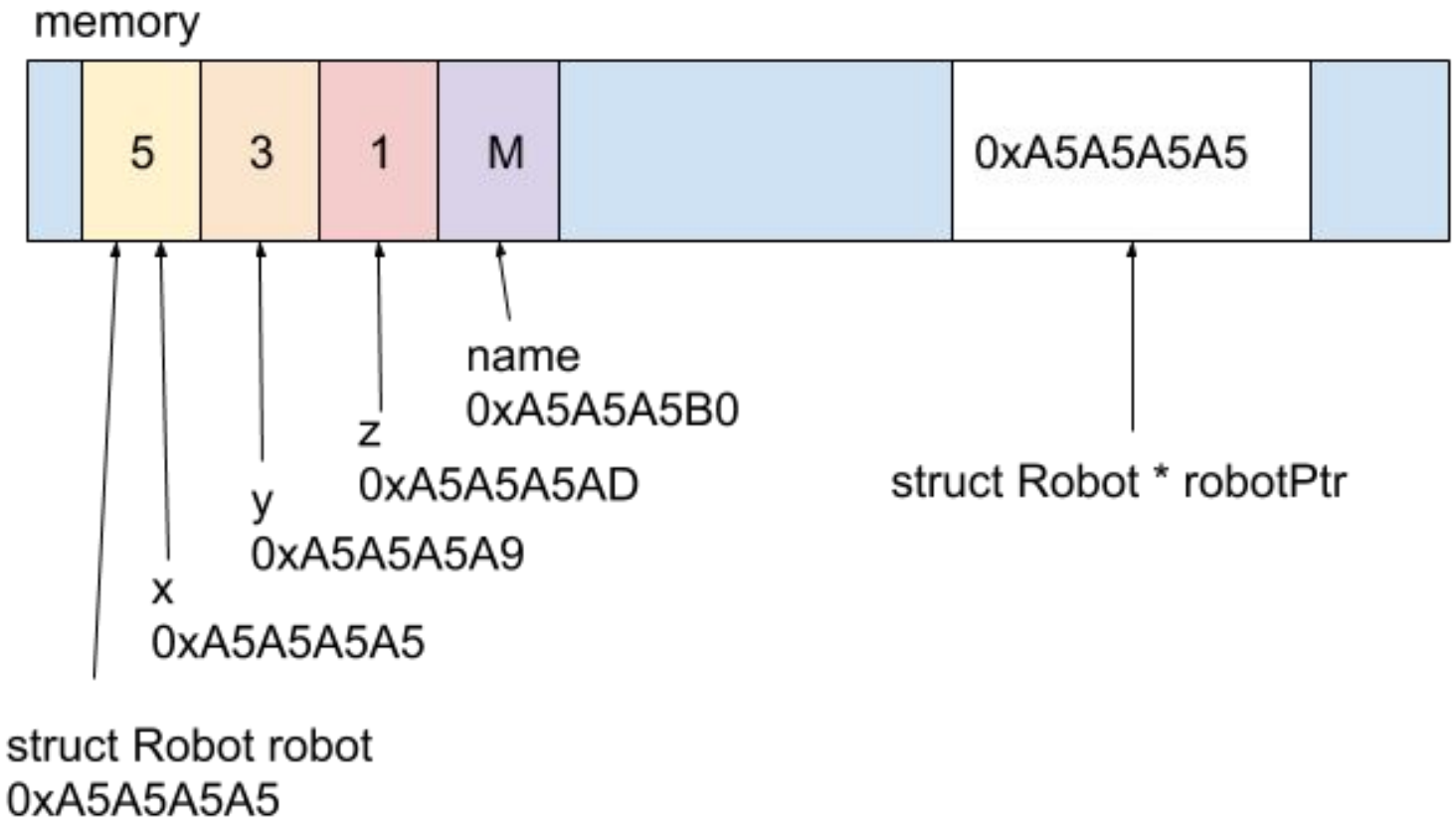
```
struct Robot * robotPtr;
```

The value of this pointer is the address of a struct Robot.

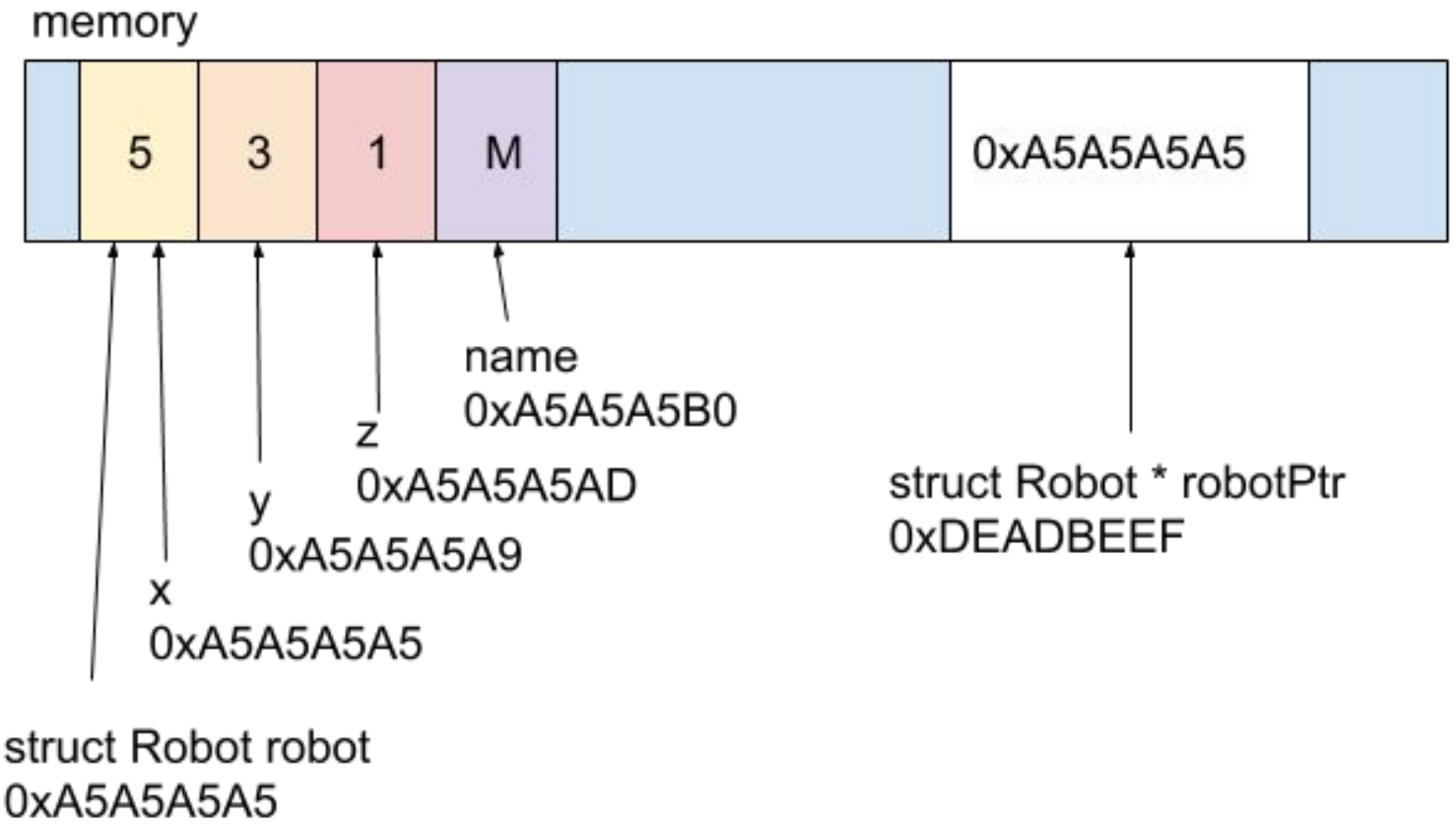
```
struct Robot * robotPtr = &robot;
```

A pointer is a variable.

Variables live in memory.



The location of a variable in memory is an address.



You can access a structs fields with the . accessor.

```
printf( “the robot’s x value is %d\n”,
robot.x );
```

```
the robot’s x value is 5
```

The value at operator gives you the value at an address.

The value at a struct address is a struct.

You can access a structs fields with the . accessor.

```
printf( “the robot’s x value is %d\n”,
(*robotPtr).x );
```

```
the robot’s x value is 5
```

Variable names are for humans.

human	computer

`struct Robot robot = {5,3,1,'M'};`
`struct Robot * robotPtr = &robot;`
`int x = robot.x;`
`int y = (*robotPtr).y;`
	`0xA5A5A5A5 = 5`
	`0xA5A5A5A9 = 3`
	`0xA5A5A5A3 = 1`
	`0xA5A5A5B0 = 'M'`
	`0xDEADBEEF = 0xA5A5A5A5`
	`0x10000000 = 5`
	`0x20000000 =`
	`(*0xA5A5A5A5)+4`
	`= 3`

Humans are lazy.

The -> symbol is a shortcut for the value at operator with structs.

```
(*robotPtr).x == robotPtr->x

(*robotPtr).colour == robotPtr->colour
```

A pointer is a type.

The value of a pointer is an address.

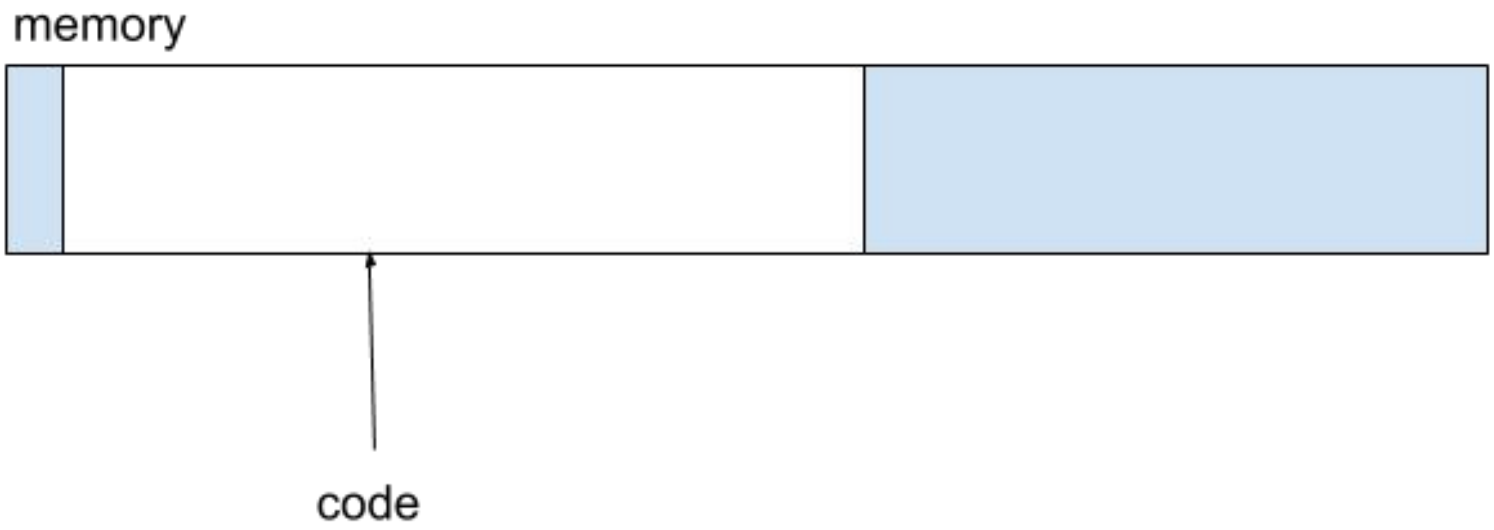
Remember that C code is for humans.

The compiler translates C into a sequence instructions for the CPU.

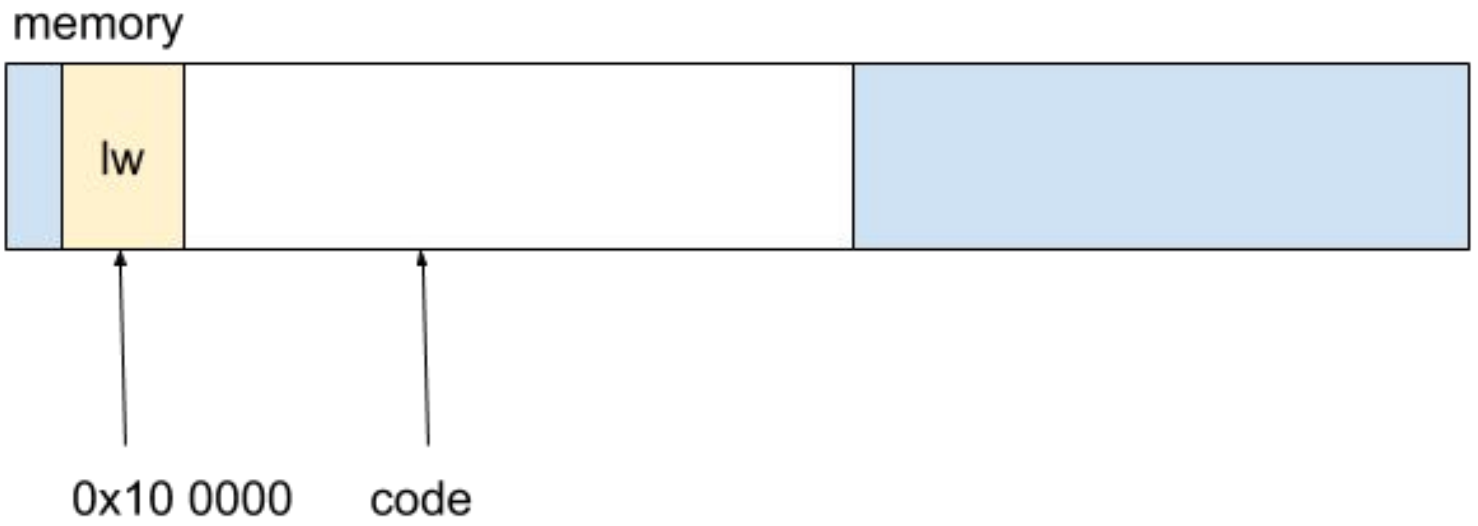
human	computer

`secret += 1;`	`lw $2, 0($1)`
	`li $3, 1`
	`add $2, $2, $3`
	`sw $2, 0($1)`

Those instructions live in memory.



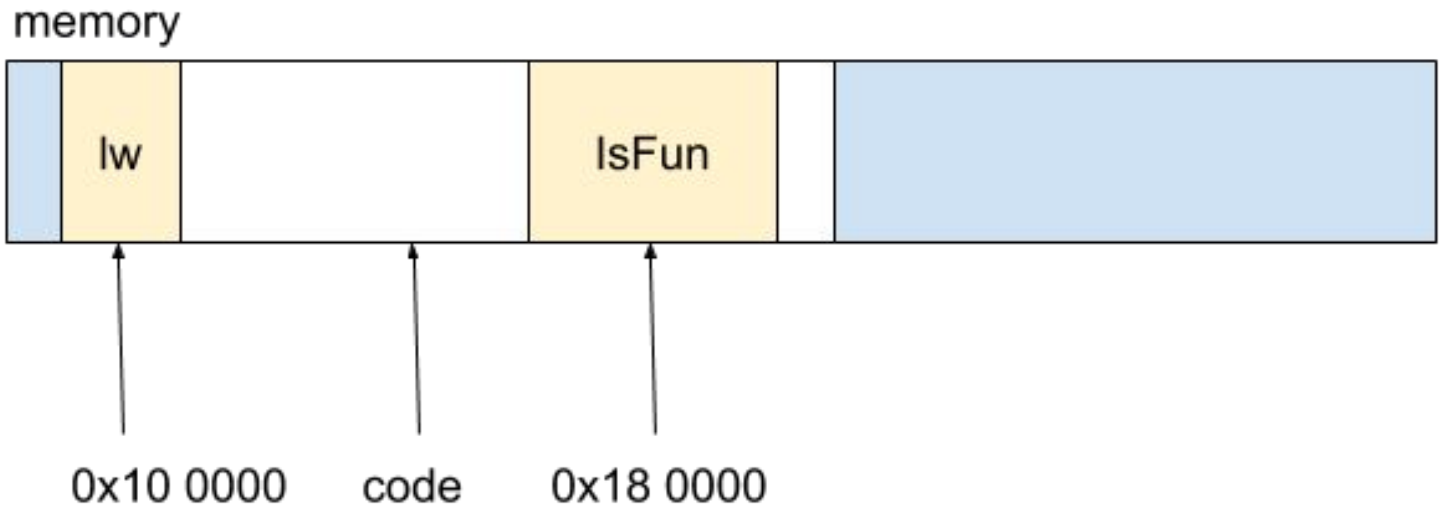
The location in memory of an instruction is an address.



A function is a labeled block of code.

```
int IsFun( int yes, int no )
{
    if ( yes ) return yes;
    return no;
}
```

The location of a function in memory is an address.



A function has an address.

The address of a function can be found with &.

```
printf( “the address of IsFun is %p\n”,
&IsFun );

the address of IsFun is 0x180000
```

This is a function pointer.

```
int (*IsFunPtr)( int, int );
```

This is also a function pointer.

```
char (*SomeMonster)(int);
```

The value of this function pointer is the address of a function that takes two ints and returns an int.

```
int (*IsFunPtr)( int, int ) = &IsFun;
```

The value of this function pointer is the address of a function that takes an int and returns a char.

```
char (*SomeMonster)(int) = &Happy;
```

We can use a function pointer to call a function.

```
int x = IsFunPtR( 1, 0 );
```

A function pointer is a type.

Function pointers can be parameters.

```
void GiveMeAFunction( int (* ring)(int, int), int a, int b )
{

    ring( a, b );

}
```

Function pointers can be returned.

```
int (* ReturnRing( int a ))(int, int)
{
    if ( a % 2 == 0 )
        return add;
    return sub;
}
```

```
(ReturnRing(1))(1, 2) => -1
```

Remember that a pointer is just an address.

Have fun with pointers!