# CIS 343 - Structure of Programming Languages

Nathan Bowman

Modified version of slides by: Ira Woodring

---

## Names, Binding, and scope (Follows the Sebesta Text Chapter 5)

## Overview

At their heart, imperative languages are abstractions of the hardware.

Variables, for instance, are merely representations of memory locations.

## Overview

Programming languages assign names to variables and functions.

As simple as it may seem to us, when designing a language there are complex issues involved with choosing a naming scheme.

# Names

A name is not required from the perspective of the computer; they really just exist for our benefit. Computers are perfectly happy just using addresses.

Names were one of the very early abstractions introduced in programming. When programming directly in machine code, programmers would use memory addresses directly.

Assembly language introduced the ability to refer to memory by a name rather than an address

## Names

Some issues with names:

- which characters are allowed to be used
- case sensitivity
- number of characters used
- whether reserved words are usable
- whether any "decorators" are used.

## Names

There are vastly different rules for naming depending on the language. For instance:

- Fortran allows only up to 31 characters
- C has no limit, but only the first 63 are significant (most compilers ignore this and have no limit these days).
- Java, C#, and Ada have no limit and all are significant
- C++ provides no limit, though certain implementations do.

# Names

So we need to be very careful when using a language; this could create a problem then:

```
#include <stdio.h>

int main(int argc, char** argv){
  int aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa = 42;
  int aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab = 24;

  printf("%d\n", aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa);
  printf("%d\n", aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab);
}
```

Though most (newer) implementations seem to handle this just fine.

## Names

Some of these issues may seem trivial, but we need to remember that these rules are going to either help or hinder areas such as readability and writability.

For instance, Java's insistence on case sensitivity allows us to have the following:

```
student1.toString()
student1.ToString()
student1.tostring()
student1.TOSTRING()
student1.tOsTrInG()
etc...
```

## Names

As all of those permutations *could* be valid, it is both harder to tell what the original programmer's intent was (readability), and harder to keep track of all the possible functions we may wish to call (writability).

## Names

Many languages reserve certain words and do not allow us to use them as variable names. These are called **reserved words**.

Other languages may have **keywords**. These may be context sensitive. For instance, in Fortran the word **Integer** is a keyword. It can be used to denote a variable type, or it can be used as a variable:

```
Integer Num_cars
Integer = 42
```

Note that context matters here; the first example was the keyword before a variable name; it is then a type specifier. In the second example the keyword is followed by an assignment operator meaning it is to be interpreted as a variable.

## Variables

Variables can be seen as a collection of six attributes:

- name
- address
- value
- type
- lifetime
- scope

## Address

We already discussed names so we'll next examine address.

As you can probably guess, the address is the machine address where the variable's value is stored.

## Address

We call the address of a variable its **l-value**. This is due to the fact that in most languages assignment statements put name on the left side.

A single address may be referred to by multiple variables. When this happens we say that a variable has an **alias**.

```cpp
// An alias in C++ created by using
// a reference.
int y = 42;
int & x = y;

std::cout << x << std::endl;
std::cout << y << std::endl;

y = 1701;
std::cout << x << std::endl;
```

```
42
42
1701
```

## Address

Notice that changing y changed x as well.

Aliases are a huge hit on readability, as they make programs much harder for a reader to understand. They also make it vastly harder to debug a program, meaning they hurt writability as well.

## Type

Types determine both the range of values a variable can hold, and the operations that may be performed.

We must be careful to fully understand the compiler(s) and machine(s) we are using when writing a program when it comes to type. For instance:

# Type

```
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main(int argc, char** argv){
        printf("Type\t\t\tBytes\t\tMin Value\t\tMax Value\n");
        printf("-----------------------------------------------------------------------------\n");
        printf("unsigned char\t\t%lu\t\t%d\t\t\t%d\n", sizeof(unsigned char), 0, UCHAR_MAX);
        printf("char\t\t\t%lu\t\t%d\t\t\t%d\n", sizeof(char), CHAR_MIN, CHAR_MAX);
        printf("unsigned int\t\t%lu\t\t%d\t\t\t%u\n", sizeof(unsigned int), 0, UINT_MAX);
        printf("int\t\t\t%lu\t\t%d\t\t%d\n", sizeof(int), INT_MIN, INT_MAX);
        printf("unsigned long\t\t%lu\t\t%d\t\t\t%lu\n", sizeof(unsigned long), 0, ULONG_MAX);
        printf("long\t\t\t%ld\t\t%ld\t%ld\n", sizeof(long), LONG_MIN, LONG_MAX);
        printf("unsigned long long\t%lu\t\t%d\t\t\t%llu\n", sizeof(unsigned long long), 0, ULLONG_MAX);
        printf("long long\t\t%lu\t\t%lld\t%lld\n", sizeof(long long), LLONG_MIN, LLONG_MAX);
        printf("float\t\t\t%lu\t\t%e\t\t%e\n", sizeof(float), -FLT_MAX, FLT_MAX);
        printf("double\t\t\t%lu\t\t%e\t\t%e\n", sizeof(double), -DBL_MAX, DBL_MAX);
        printf("long double\t\t%lu\t\t%Le\t\t%Le\n", sizeof(long double), -LDBL_MAX, LDBL_MAX);
}
```

# Type

On a 64-bit Intel system I get the following output:

```
//Type                   Bytes       Min Value                Max Value
//-------------------------------------------------------------------------
//unsigned char           1            0                    255
//char                    1           -128                  127
//unsigned int            4           0                     4294967295
//int                     4           -2147483648           2147483647
//unsigned long           8           0                     18446744073709551615
//long                    8           -9223372036854775808 9223372036854775807
//unsigned long long      8           0                     18446744073709551615
//long long               8           -9223372036854775808 9223372036854775807
//float                   4           -3.402823e+38         3.402823e+38
//double                  8           -1.797693e+308        1.797693e+308
//long double            16           -1.189731e+4932       1.189731e+4932
```

# Yet, on a 32-bit ARM I get this:

```
//Type                   Bytes         Min Value                 Max Value
//-------------------------------------------------------------------------
//unsigned char           1           0                     255
//char                    1           0                     255
//unsigned int            4           0                     4294967295
//int                     4           -2147483648           2147483647
//unsigned long           4           0                     4294967295
//long                    4           -2147483648           2147483647
//unsigned long long      8           0                     18446744073709551615
//long long               8           -9223372036854775808  9223372036854775807
//float                   4           -3.402823e+38         3.402823e+38
//double                  8           -1.797693e+308        1.797693e+308
//long double             8           -1.797693e+308        1.797693e+308
```

## Types

A few differences exist for the two. Many people will notice that the long double doesn't hold as high a value. In fact, a long double on the second system is the same as a double on the first.

What you may have missed though is the fact that on the ARM system a char and unsigned char produced the same values.

## Types

It turns out that on ARM based systems chars default to unsigned. This could have some fairly significant effects on our code. For instance, it is not uncommon that programmers return -1 from a function when it is unable to complete a task. Consider:

```c
#include <stdio.h>

char doSomething(int successful){
        if(successful>0){
                return 'a';
        }
        return -1;
}

int main(int argc, char** argv){
        char result = doSomething(1);
        printf("%d\n", result);
        result = doSomething(-1);
        printf("%d\n", result);
}
```

## Types

Running this on the two systems we get the output

```
97
-1
```

On the Intel system And

```
97
255
```

On the ARM.

## Value

What we store in the memory locations the variable is comprised of is the value.

We call the value the **r-value** as it is usually on the right side of an assignment statement.

## Value

Although it seems fairly straightforward, determining the value for an r-value can be complicated due to scoping rules.

## Variables

Variables can be seen as a collection of six attributes:

- name
- address
- value
- type
- lifetime
- scope

An understanding of type and lifetime requires us to first talk about binding.

As the book does, we will pause talking about variable attributes here and focus on binding.

## Binding

A binding occurs when we associate an attribute and an entity.

This does not refer strictly to a variable; we can associate a function with an attribute as well.

## Binding

Binding is an abstract idea

We will consider **type binding** (which affects the type of a variable) and **storage binding** (which affects the lifetime of a variable)

First, we will discuss aspects of binding in general

Binding can occur at

- **language design time** (think of an operator)
- **language implementation time** (imagine a data type in C; it must be implemented on multiple hardwares so will have different ranges)
- **compile time** (imagine associating a variable with a type)
- **load time** (an address is given to a variable when the program is loaded into memory)
- **link time** (associating a function with an address in library)
- **run time** (giving a variable a value)

## Binding -- Static vs Dynamic

A **static** binding occurs *before runtime* and does not change while the program is running.

A binding that occurs during runtime and can change *while the program is running* is **dynamic**.

# Consider this code:

```c
#include <stdio.h>

int main(int argc, char** argv){
        static int meaning;
        meaning = 42;
        int runtimeStuff = 1701;
        printf("The meaning is %d.\n\n", meaning);
}
```

While the value 42 is not assigned statically, we can tell that the address itself is being bound at compile time by running `objdump -t` on the executable:

```
... // Lots of output...

0000000000601034 l    O .data  0000000000000004    main.meaning

... // Lots more output...
```

In the midst of all the output we find the variable `main.meaning` is given a memory location, even before the program is run. By contrast, notice that there is no address defined for runtimeStuff. This will be determined when the program is run.

## Type Binding

Recall that a variable is essentially just a spot in memory with a name (and some other attributes we are learning about)

When we bind a *type* to that variable, we specify what values it can take on and what operations can be performed on it

# Type Binding

This is your memory

```
1000: 48656c6c 6f204756 53552100 00000000
1010: 00000157 00000000 00000000 00000000
```

# Type Binding

This is your memory on type bindings

```
1000: Hello GVSU! 00000000
1010: 343 00000000 00000000 00000000
```

## Type Binding

In addition to being static or dynamic, type bindings may be **implicit** or **explicit**.

Explicit: `int x;`

Implicit: `var y = "foo";`

This implicit binding is possible in C#, but it *still static* because the type is determined before runtime

## Type Binding

Contrast with Python --

Dynamic:

```
x = 'a'
```

This looks like the implicit example, but key is that type *is not determined until runtime*

## Type Binding

Please note that implicit/explicit bindings are a subset of static bindings

Implicit vs explicit is a separate topic from static vs. dynamic bindings, though it can be easy to confuse the two

Dynamic binding can be hard to distinguish from implicit static binding -- you often need more information about the language than just the code

## Type Binding

Explicit type bindings may harm writability but aid readability.

Implicit type bindings do the opposite. In addition, they can harm reliability as they remove the ability of a compiler to detect some errors before runtime.

Dynamic binding is less reliable than static binding because no type checking can be done before runtime

## Storage Binding

In addition to type binding, another aspect of variables we consider is **storage binding**

Variables must live somewhere on the computer (in memory)

Storage binding refers to the way in which memory is associated and unassociated with variables

## Storage Binding

For a binding to occur, we first need to go through the process of **allocation**, which refers to assigning a variable an address from a pool of memory.

**Deallocation** is the returning of the memory to the pool.

```
1000: 00000000 00000000 00000000 00000000
1010: 00000000 00000000 00000000 00000000
```

```
1000: int a;   int b;   double c;
1010: 00000000 00000000 00000000 00000000
```

## Storage Binding

The **lifetime** of a variable is the time during which it is bound to a particular memory location.

## Variables

Variables can be seen as a collection of six attributes:

- name
- address
- value
- type
- **lifetime**
- scope

Still discussing "binding" -- related to a few of these attributes

## Storage Binding

**Static variables** are bound to memory cells before programs are executed, and stay bound until the program ends.

We have already seen an example of static binding earlier. We look at a similar example next.

```c
#include <stdio.h>

static const int meaning = 42;

int main(int argc, char** argv){
        int runtimeStuff = 1701;
        printf("The meaning is %d.\n\n", meaning);
}
```

# Viewing the file we can verify this binding:

```
gcc static.c
objdump -t a.out
...
0000000000002004 l  O .rodata  0000000000000004    meaning
...

objdump -x --full-contents a.out

...
Contents of section .rodata:
2000 01000200 2a000000 54686520 6d65616e  ....*...The mean
2010 696e6720 69732025 642e0a0a 00        ing is %d....

...
```

For those of you wondering, 2a in hex is 42 in decimal.

## Storage Binding

Accessing static variables can be somewhat faster than accessing non-static variables

## Storage Binding

**Elaboration** is the process that happens at run-time of allocating and binding a variable.

This applies to variables that are not statically bound (regarding storage)

## Storage Binding

**Stack-dynamic** variables are variables that have

- statically bound types
- dynamically bound storage (created when they are elaborated)

An example of these is local variables in C. Each function will have local variables; the space for them is not created until the function's memory is created. When the program executes (elaborates) the code for the variable, it is created.

# Storage Binding

```
void do_work() {
    int z;  // <-- stack-dynamic variable
    ...
}
```

Space for the variable `z` is allocated on the stack when `do_work()` is called. (Hence, stack-dynamic)

When `do_work()` returns, the memory allocated for `z` is automatically deallocated

Stack-dynamic storage binding ensures that each call to a particular function can be independent of other calls to that same function

If all variables were static variables, most useful recursion would be impossible

There can be a slight overhead to using stack-dynamic variables instead of static variables

Why can't we have recursion without these variables?

Consider this language that only has global storage:

```c
#include <stdio.h>
#include <stdlib.h>

int n;

int factorial() {
  if (n == 0) return 1;           // Base case
  n = n - 1;
  return (n * factorial());
}

int main(int argc, char** argv){
  n = atoi(argv[1]);
  printf("Factorial of %d is %d.\n", n, factorial());
}
```

This can't work; each iteration of the factorial function reduces n by 1. The final value will always be 0.

Consider n=3. The first function can't return until it calls itself, but calling itself reduces the value of n.

## Storage Binding

The stack is very tidy.

When a function is called, space is allocated for the local variables. When the function returns, the space is freed.

The heap is the Wild West

## Storage Binding

There are two types of heap-dynamic variables: explicit and implicit

**Explicit Heap-Dynamic Variables** are allocated and deallocated at run-time by the programmer.

They are useful because they can be used for dynamically sized structures

Also, sometimes a variable should outlive the function it is defined in

## Storage Binding

Downsides include the overhead of managing the heap and that pointers and references can be difficult for programmers to use correctly

```c
int* nums;
nums = (int*) malloc(50 * sizeof(int));

...

// Must give memory back at some point.
free(nums);
```

## Storage Binding

It may not look quite like C, but Java also has explicitly heap-dynamic variables

In Java, all variables that are not primitive scalars are objects, which are accessed through reference and are explicitly heap-dynamic

Java objects do not need to be (and cannot be) deallocated explicitly due to Java's garbage collection, but they are still considered explicitly heap-dynamic

## Storage Binding

**Implicit Heap-Dynamic Variables** are bound to heap storage only when they are assigned a value.

These are very flexible, but run-time overhead is high, and they don't have the ability to be error checked by the compiler.

We tend to see these in interpreted languages. Consider this Javascript code for example (from the book):

```
highs = [74, 84, 86, 90, 71];
```

## Lifetime

The **lifetime** of a variable is from the time it is bound to storage to the time it is unbound from storage.

Lifetime of a static variable is the entire run of the program

Lifetime of a stack-dynamic variable is during the function in which it is created

Lifetime of a heap-dynamic variable can vary

## Variables

Variables can be seen as a collection of six attributes:

- name
- address
- value
- type
- lifetime
- **scope**

## Scope

**Scope** is the range of statements for which a variable is visible (visible meaning it can be referenced).

A variable is **local** to the scope in which it was declared.

## Scope

**Static scope** means that the variable's scope can be determined before execution of the program.

Helps readability.

## Scope

When resolving variables the system starts with the declarations in the current subprogram. If it is not found there, it continues out to the **static parent** (the outer enclosing unit).

This continues up through other **static ancestors** until the variable can be resolved (or we run out of places to look).

## Scope

ALGOL 60 brought the oft copied idea of **blocks**, or **block structured languages**.

These languages allow new scopes to be defined within executable code. For instance:

```c
int main(int argc, char** argv){
  int i=0;
  while(i<100){
    // We are in another scope here!
    int square = i * i;
  }
}
```

## Scope

**Global variables** exist outside of function definitions, but are resolvable from within functions.

In some languages we must be careful to not re-use a name of a global within some local scope. For instance, in C we could do this:

```
int num_threads;

int main(int argc, char** argv){
  // Now the int is un-resolvable!
  float num_threads = 3.14;
}
```

When you make a variable un-resolvable in this way, we say you are shadowing the variable.

# Scope

Languages with robust namespace capabilities may allow us to access the globals anyway. For instance, C++ allows us to do this:

```
#include <iostream>

int num_threads = 0;

int main(int argc, char** argv){
        float num_threads = 3.14;
        std::cout << num_threads << std::endl;
        std::cout << ::num_threads << std::endl;
}
```

## Scope

There are other methods as well. PHP uses a $GLOBALS array. We can access those globals in two ways:

```
$num_threads = 200;

function get_threads() {
  global $num_threads;
  print "Num threads = $num_threads.";
}
```

Or:

```
$num_threads = 200;

function get_threads() {
  $nt = $GLOBALS['num_threads'];
  print "Num threads = $nt.";
}
```

# Scope

Python (as is often the case...) has its own rules. In Python a global can be referenced but not assigned to in a function, unless it is declared global at the beginning. Thus, this will not work (from the book):

```
day = "Monday"

def tester():
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day

tester()
```

# Scope

However, this will (we can access):

```
day = "Monday"
def tester():
  print "The global day is:", day
tester()
```

# Scope

And this will (we declare the variable global first [now we can modify!]):

```
day = "Monday"

def tester():
  global day
  print "The global day is:", day
  day = "Tuesday"
  print "The new value of day is:", day

tester()
```

## Scope

Static scope does have issues:

- Usually allows more access than needed.
- Restricts program evolution - leading programmers to do things they shouldn't.
  - They may make more things global (Ugh!)
  - May end up with an awkward design

These issues are largely fixed by encapsulation.

## Scope

**Dynamic Scope** resolves variables not by the blocks they are in, but by the calling sequence of the program.

It is largely unseen in newer languages (but still available in some cases).

Dynamic Scope destroys readability.

Consider:

```
function big()
{
  function sub1() {
    var x = 7;
    sub2();
  }

  function sub2() {
    var y = x;
    var z = 3;
  }

  var x = 3;
  sub1();
  sub2();
}
```

Assume this is written in a language that resolves scope dynamically. The call to `sub2()` inside `sub1()` would lead to `y == 7`. The call to `sub2()` inside `big()` would lead to `y == 3`.

## Scope

Dynamic scope destroys readability completely. We cannot look at a program and tell what the value of all variables are at any given time (we don't know what the calling sequence would be; any input could change the calling sequence).

Because dynamically scoped languages cannot resolve variable information statically debugging becomes a nightmare.

## Scope vs Lifetime

Don't confuse scope and lifetime. They may appear related but aren't; scope is more of a textual relationship (or spatial), whereas lifetime deals with time.

Consider:

```
void do_stuff(){
        static int meaning = 42;
}

int main(int argc, char** argv){
        do_stuff();
}
```

The variable `meaning` is local in scope, but because it is static exists throughout the life of the program. Want proof?

Compile it, and run `objdump -t a.out`

```
...
000000000060102c l  O .data  0000000000000004     do_stuff.meanin
...
```

This shows that the variable is statically bound, but local in scope! It will exist until the program ends, but is only accessible inside the

```
do_stuff()
```

function.

# Scope

**Named Constants** are variables that are named only once. They enhance readability.

Consider:

```
const int MEANING = 42;

int main(int argc, char** argv){
  MEANING = 43;
}

clang st.c
st.c:3:10: error: read-only variable is not assignable
        MEANING = 43;
        ~~~~~~~ ^
1 error generated.
```

Additionally, if we didn't try to reassign to the constant, we can view the objdump:

```
...
00000000004005a0 g  O .rodata   0000000000000004        MEANING
...
```

Or:

```
...
Contents of section .rodata:
 400590 01000200 00000000 00000000 00000000  ................
 4005a0 2a000000
...
```