



Variables and Operators in C

Hyung-Sin Kim

Computing for Data Science

Graduate School of Data Science, Seoul National University





Variables

- **Variables** hold the values upon which a program acts
 - The most basic type of **memory object**
- A variable has a **symbolic name** instead of the storage location where its value resides
 - Programmers can focus on the program logic without concern about where to store various values
- **Compiler** generates the full set of data movement operations (to/from memory)
 - To this end, it needs to know a variable's **name** (identifier), **data type**, and **scope** (where it will be accessible)
 - The information is given by the programmer, in a variable's **declaration**





Variables – Identifiers

- Alphabet, digit, and underscore
- Case sensitive
- Variables are almost never declared in all uppercase letters
 - All uppercase letters are usually used for `#define`
- Typical ways to combine multiple words in a variable name
 - By using capitals: `wordsPerSecond`
 - By using underscore: `words_per_second`
- Giving meaningful names is important!





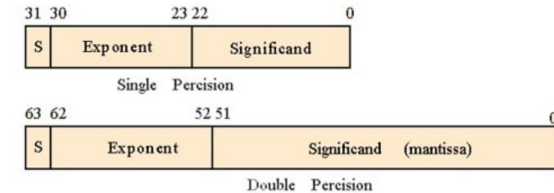
Variables – Data Type

- One bit pattern can have different meanings depending on its data type
 - 0110 0110
 - 102, if its data type is 2's complement integer
 - f, if its data type is ASCII
 - The compilers uses a variable's type information to allocate a proper amount of storage for the variable
- One operator can be performed differently at the machine level depending on the data type of its operands
 - IntegerA + IntegerB \rightarrow ADD instruction
 - FloatA + FloatB \rightarrow a set of instructions (there is no single instruction to add two floats)



Variables – Data Type

- **int** integerVariable; / int integerVariable = 10;
 - A 2's complement integer whose size depends on machines
- **char** characterVariable; / char characterVariable = 'Q';
 - A single ASCII code (8 bits)
- **float** floatVariable = 2.1; / float floatVariable = 2.1E2;
 - A single precision floating point number (32 bits)
- **double** doubleVariable = 2.1; / double doubleVariable = 2.1E2;
 - A double precision floating point number (64 bits)
- **_Bool** flag = 1; / **bool** flag = true;
 - **_Bool** takes 1 or 0, neither true nor false
 - To use bool, #include <stdbool.h> is needed - true or false (no capital T nor F!)
- Variable type is **immutable!**





Variables – Data Type

- C allows the programmer to specify larger or smaller versions of the basic types int, char, float, and double
 - long
 - short
- **unsigned int** is also provided





Variables – Scope

- The scope of a variable is the regions of the program in which the variable is “**alive**” and accessible
 - All C variables must be declared before they are used
 - The C compiler infers a variable’s scope from where it is declared within the code
- **Local** variables
 - A **block** is any subsection of a program beginning with ‘{’ and ending with ‘}’
 - If a variable is declared within a block, it is visible until the end of the block
 - The variable is **local** to the block
- **Global** variables
 - If a variable is declared outside of all blocks, it is a **global** variable that can be accessed anywhere in the program
 - WARNING: Using global variables is very **error prone**!



Variables – Initializer

- **Initializer:** Variable declaration with an initial value
 - `int a = 0;`
- It not initialized,
 - A global variable's value will become 0
 - A local variable's value will become a garbage value
- It is a standard coding practice to explicitly initialize **all local variables** in their declarations
 - To make everything predictable
- We can make a constant by using the initializer (without `#define`)
 - **const** double pi = 3.14159;





Operators

- Assignment ($A = B$)
 - Evaluate the right expression (B) and assigns it to A
- Arithmetic operators
 - *: multiplication
 - /: division
 - %: remainder
 - +: addition
 - -: subtraction
 - Same order of evaluation as in Python





Operators

- Bitwise operators
 - \sim : bitwise NOT ($\sim x$)
 - $\&$: bitwise AND ($x \& y$)
 - $|$: bitwise OR ($x | y$)
 - \wedge : bitwise XOR ($x \wedge y$)
 - \ll : left shift ($x \ll y$)
 - \gg : right shift ($x \gg y$)
- Logical operators
 - $!$: logical NOT ($!x$)
 - $\&\&$: logical AND ($x \&\& y$)
 - $||$: logical OR ($x || y$)





Operators

- Assignment with arithmetic or bitwise operators
 - $x += y \longrightarrow x = x + y$
 - $x -= y \longrightarrow x = x - y$
 - $x *= y \longrightarrow x = x * y$
 - $x /= y \longrightarrow x = x / y$
 - $x \% = y \longrightarrow x = x \% y$
 - $x \& = y \longrightarrow x = x \& y$
 - $x |= y \longrightarrow x = x | y$
 - $x \wedge = y \longrightarrow x = x \wedge y$





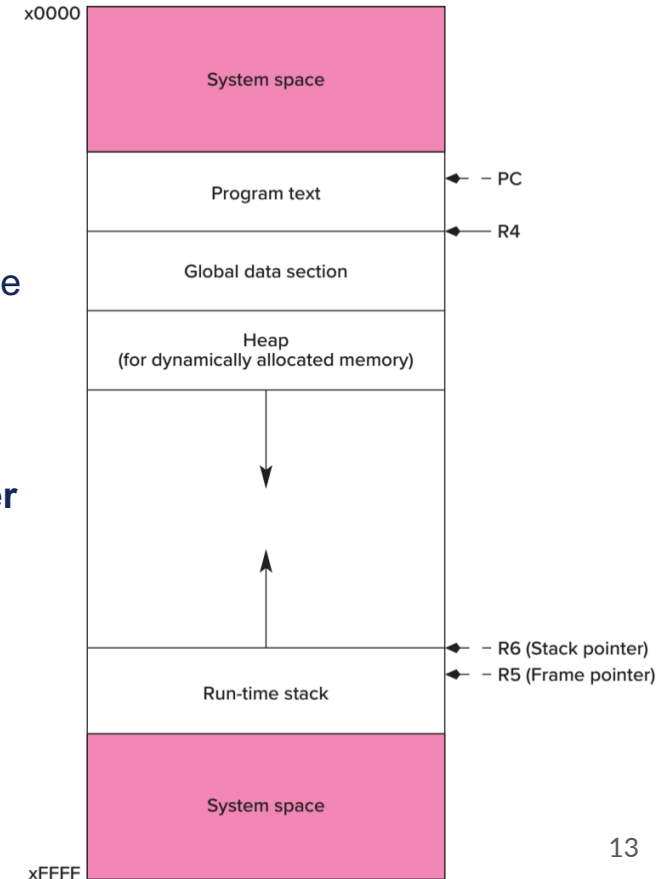
Operators

- Increment (++) / Decrement (--) operators
 - Similar to `x = x+1;` or `x = x-1;`
- Subtle difference
 - `x++`: evaluate and increment
 - `++x`: increment and evaluate
 - Ex.1) `x = 4; y = x++;`
 - `x` will be 5 and `y` will be 4
 - Ex.2) `x = 4; y = ++x;`
 - Both `x` and `y` will be 5



Memory in C

- Global variables are stored in **global data section**
 - A register (R4) contains the memory address of the beginning of the global data section (i.e., base)
 - As more global variables are stored, the offset from the base **increases**
- Local variables are stored in **run-time stack**
 - When a function is executing, the highest numbered memory address of its stack is stored in **frame pointer**
 - As more local variables in the function is stored, the offset from the frame pointer **decreases**
- Another region reserved for dynamically allocated data called **heap** (out of scope)

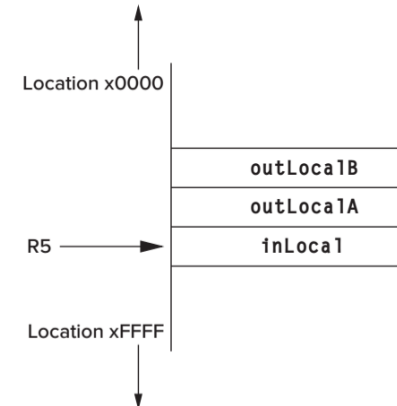


Memory in C – Example

- `#include <stdio.h>`
- `int inGlobal`
-
- `int main`
- `{`
- `int inLocal = 5;`
- `int outLocalA;`
- `int outLocalB;`
- `inGlobal = 3;`
-
- `outLocalA = inLocal * inGlobal;`
- `outLocalB = inLocal & inGlobal;`
- `printf("outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);`
- `return 0;`
- `}`

Symbol table

Identifier	Type	Location (as an offset)	Scope	Other info...
<code>inGlobal</code>	<code>int</code>	0	<code>global</code>	...
<code>inLocal</code>	<code>int</code>	0	<code>main</code>	...
<code>outLocalA</code>	<code>int</code>	-1	<code>main</code>	...
<code>outLocalB</code>	<code>int</code>	-2	<code>main</code>	...





Questions?





Thanks!

