C programming

Lecture 3: Variables, types, arithmetic expressions

Based on slides designed by Shay Artsi, Aythan Avior, Gitit Rockstein and Saher Ismir for the department of Computer Science, Technion, Israel Institute of Technology
Translated and updated by Anne Weill-Zrahia
Memory cells, addresses and names

- C allows us to store values in memory
  - Can be intermediate values computed in the course of the program, which will be later printed out
- Each memory cell has an **address** – machine language will use this address to access its contents

#1223 ← 91

Writing a constant into the address 1223

#1224 ← #1223 + 4

Read a value from cell, add a number and store the results into same
Variables

- Instead of the address, we can use a name:

```plaintext
base_grade = 91;
final_grade = base_grade + 4;
```

- `base_grade` is now the name of a variable.
- The `=` sign symbolizes assignment of a value to the variable.
Types of variables in C

- Each variable is identified by a variable name.
- Each variable has a type.
- In C we have to define variables before we use them.

```c
int num, sum;
double weight = 0.0;
char digit = '4';
```

- Definition has the following effects:
  - Allocate a memory location of appropriate size.
  - Apply a name to it.
  - Interpret the contents according to type (permitted operations).
  - Initialize it.
The int type

- int stores an integer variable
- Any integer constant in the code is an int
- An int takes up 4 bytes of memory (32 bits)
- This size gives us the domain of definition of values
  \(-2^{31}, \ldots, -2, -1, 0, 1, \ldots, 2^{31} - 1\) 4 bytes

\[2^{31} = 2,147,483,648\]
The char type

- **char** stores a character variable
- **We can print char** with `%c`
- **A char has a single quote** not a double quote.

```c
int main()
{
    char a, b;
    a = 'x';  /* Set a to the character x */
    printf("a is %c\n", a);
    b = '\n';  /* This really is one character*/
    printf("b is %c\n", b);
    return 0;
}
```

We can use it like so:
More types: Signed/unsigned, long, short

- **unsigned** means that an int or char value can only be positive. **signed** means that it can be positive or negative.
- This allows to represent more numbers:
  \[0, 1, 2, \ldots, 2^{32} - 1 = 4,294,967,295\]
- **long** means that int, float or double have more precision (and are larger) **short** means they have less precision and are smaller
- \[\text{Size}(\text{long}) \leq \text{Size}(\text{int}) \leq \text{Size}(\text{short}) < \text{Size}(\text{char})\]

```c
  short int small_no;
  Unsigned char uchar;
  long double precise_number;
  short float not_so_precise;
```
A short note about ++

- ++i means increment i then use it
- i++ means use i then increment it

```
int i = 6;
printf ("%d\n", i++); /* Prints 6 sets i to 7 */
```

Note this important difference

```
int i = 6;
printf ("%d\n", ++i); /* prints 7 and sets i to 7 */
```

It is easy to confuse yourself and others with the difference between ++i and i++ - it is best to use them only in simple ways.

All of the above also applies to --.
Casting between variables

- Recall the trouble we had dividing ints
- A cast is a way of telling one variable type to temporarily look like another.

```java
int a = 3;
int b = 4;  // Cast ints a and b to be doubles
double c;
c = (double)a/(double)b;
```

By using `(type)` in front of a variable we tell the variable to act like another type of variable. We can cast between any type. Usually, however, the only reason to cast is to stop ints being rounded by division.
Character representation

- 'a' 'A' '0' '3' '
' are constants of type int – we as programmer choose to view them as character constants.
- They are represented in the computer by codes, the most frequent of them being the ASCII code. Below it a fragment of the table for encoding:

<table>
<thead>
<tr>
<th>Characters</th>
<th>ASCIIcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a' 'b' ... 'z'</td>
<td>97 98 ... 122</td>
</tr>
<tr>
<td>'0' '9' ... '9'</td>
<td>48 49 ... 57</td>
</tr>
</tbody>
</table>

- A char variable (8 bits) is sufficient to contain any value of alphanumeric characters + some more.
- The following commands are equal:

```c
char letter = 'd';  // ==
char letter = 100;
```
## Other characters in ASCII

<table>
<thead>
<tr>
<th>Name</th>
<th>Name in C</th>
<th>Value in ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>ampersand</td>
<td>' &amp; '</td>
<td>38</td>
</tr>
<tr>
<td>asterisk</td>
<td>' * '</td>
<td>42</td>
</tr>
<tr>
<td>plus</td>
<td>' + '</td>
<td>43</td>
</tr>
<tr>
<td>backslash</td>
<td>' \ '</td>
<td>92</td>
</tr>
<tr>
<td>alert (bell)</td>
<td>' \a '</td>
<td>7</td>
</tr>
<tr>
<td>tab</td>
<td>' \t '</td>
<td>9</td>
</tr>
<tr>
<td>newline</td>
<td>' \n '</td>
<td>10</td>
</tr>
<tr>
<td>nul</td>
<td>' \0 '</td>
<td>0</td>
</tr>
<tr>
<td>*any char*</td>
<td>*ddd*</td>
<td>ddd*</td>
</tr>
</tbody>
</table>
The floating point types

- **double** is the basic type
- Real constants are of type **double** (unless otherwise specified)
- Other real types: **float**, **long double**
- Size(float)<size(double)<=size(long double)
- Size is 8 bit for double, 4 bit for float
- Representation is the floating point representation
  \[ \pm d d d \ldots d \cdot d d d \ldots d e \pm d d d \]
- Only integers can be exactly represented, or binary fractions, the rest of real numbers is represented approximately in memory
  
  - float: \(10e^{-38} \ldots 10e^{+38}\) 6 decimal digits, 23 bits
  - double: \(10e^{-308} \ldots 10e^{+308}\) 12 decimal digits, 52 bits
Types in this course

- Real numbers `double` is the basic type
- Integers `int`
- Positive integers `unsigned int`
Expressions

- Expressions are combinations of variables, constants and operators:

```plaintext
7
x
x = 3 + y
(x > 0) && (x < 70)
x++
found ? x : y
```

- Every arithmetic expression has a type and a value.
- Expressions can be unary, binary or ternary.
Order of computation

- Operations are written as follows $x + y$ not $+ x y$ or $x y +$
- Which of the following will be performed?

```c
int x = 2,
y = 3,
z = 4;
int w;
w = x + y * z;
```

```c
w = (x + (y * z));
w = 14;
```

```c
w = ((x + y) * z);
w = 20;
```

```c
((w = x) + y) * z);
w = 2;
```
Operators precedence

- Different operators have different precedence types
- What is in parenthesis will always be computed first
- The operation which highest precedence will be performed first

```c
int x = 2,
y = 3,
z = 4;
int w;
w = x + y * z;
```

```c
w = (x + (y * z));
w = 14;
```
The order is also determined by the associativity of the operator

```
int x = 2,
y = 3,
z = 4;
int w;
w = x - y + z;
w = x = y = z;
```

```
w = ((x - y) + z);  // w = 3;
w = (x = (y = z));  // y = 4;
x = 4;
w = 4;
```
# Partial operators table

<table>
<thead>
<tr>
<th>Class</th>
<th>Operator</th>
<th>associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>+, -, (type) sizeof(), ++, --</td>
<td>Right to left</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*, /, %</td>
<td>Left to right</td>
</tr>
<tr>
<td>Addition</td>
<td>+, -</td>
<td>Left to right</td>
</tr>
<tr>
<td>Assignment</td>
<td>=, +=, -=, *=, /=, %=</td>
<td>Right to left</td>
</tr>
</tbody>
</table>
Attention!!

- There is no definition of which function will be computed first

\[ w = \sin(x) \times \cos(x); \]

- There may be more than one possibility, according to the computer, compiler or runtime environment

```c
printf("%g\n", 0.08 - 0.5 + 0.42);
printf("%g\n", 0.08 + 0.42 - 0.5);
printf("%g\n", 0.42 - 0.5 + 0.08);
```

```
0
0
-1.38778e-17
```
## Arithmetic expressions

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>•1 + 2 * 3</td>
<td>7</td>
<td>Multiplication before addition</td>
</tr>
<tr>
<td>1 - 2 + 3</td>
<td>2</td>
<td>Add/subtract series left to right</td>
</tr>
<tr>
<td>1 - (2 + 3)</td>
<td>-4</td>
<td>Use of parenthesis</td>
</tr>
<tr>
<td>1.0/2.0*3.0</td>
<td>1.5</td>
<td>Multiply/Divide left to right</td>
</tr>
<tr>
<td>1 / 2 * 3</td>
<td>0</td>
<td>Integer divide</td>
</tr>
<tr>
<td>12 % 5</td>
<td>2</td>
<td>Remainder from int divide</td>
</tr>
<tr>
<td>5 % 12</td>
<td>5</td>
<td>Remainder from int divide</td>
</tr>
<tr>
<td>12.0 % 5.0</td>
<td>error</td>
<td>% works only for integers</td>
</tr>
</tbody>
</table>
Program to convert from Celsius to Fahrenheit

```c
#include <stdio.h>
int main()
{
    double celsius, fahrenheit;
    printf("Enter degrees Celsius: ");
    scanf("%lf", &celsius);
    fahrenheit = 32 + celsius * 9 / 5;
    printf("Degrees Fahrenheit: %lf\n", fahrenheit);
    return 0;
}
```
Working with different types

- In the program we used both int and double
- When we have a sub-expression of type int and a sub-expression of type double, the int is converted to a double

\[\text{fahrenheit} = 32 + (\text{celsius} \times \left(\frac{9}{5}\right))\]

- Calculating parenthesis can lead to an inaccurate result:

\[9 \div 5 = 1\]

\[9 \div 5 = 1.8\]
Automatic conversion between variables

- Arithmetic operations always occur between values of same type, this will also be the final type of the expression.
- If different types are represented in a computation, a conversion/promotion will take place to the most general type.

\[
\text{char } \rightarrow \text{short } \rightarrow \text{int } \rightarrow \text{long } \rightarrow \text{float } \rightarrow \text{double}
\]

#### Examples:

- \(2 + 7.5\) \(\rightarrow\) \(2.0 + 7.5\)

```c
char c;
int i;
c + i
```

\(\rightarrow\) \((\text{int})c + i\)
Automatic casting between variables

- In the presence of different types in an expression,
  1. Conversion of one of the values to the other (or both to a third type
  2. Perform operation.

Example:

\[ \text{celsius} \times 9 / 5 \]

1. 5 (int) is cast as double, perform \text{double/double}
2. 9 (int) is cast as double, \text{double*double gives a double}
### Conversion between types

**Definitions**

```c
char c;
int i;
short s;
long l;
float f;
double d;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>c + c</code></td>
<td>int</td>
</tr>
<tr>
<td><code>c + s</code></td>
<td>int</td>
</tr>
<tr>
<td><code>c - s / i</code></td>
<td>int</td>
</tr>
<tr>
<td><code>l * 2.0 - i</code></td>
<td>double</td>
</tr>
<tr>
<td><code>c + 3</code></td>
<td>int</td>
</tr>
<tr>
<td><code>c + 5.0</code></td>
<td>double</td>
</tr>
<tr>
<td><code>2 * i / l</code></td>
<td>long</td>
</tr>
<tr>
<td><code>f * 7 - i</code></td>
<td>float</td>
</tr>
<tr>
<td><code>d + c</code></td>
<td>double</td>
</tr>
<tr>
<td><code>f * d - l</code></td>
<td>double</td>
</tr>
</tbody>
</table>
Casting between variables

- Recall the trouble we had dividing ints
- A cast is a way of telling one variable type to temporarily look like another.

```java
int a = 3;
int b = 4;
double c;
c = (double)a/(double)b;
```

Cast ints a and b to be doubles

By using `(type)` in front of a variable we tell the variable to act like another type of variable. We can cast between any type. Usually, however, the only reason to cast is to stop ints being rounded by division.
Casting in the example

```c
fahrenheit =
32 + (celsius * ((double)9 / (double)5));
```

- In the example we could just cast one of the constants (why?)
- Casting a constant is not much employed, we can use instead 9.0, 5.0 etc.
Conversion through assignment

Assignment will perform a cast automatically

```
int j, k = 5, m = 5;
char c;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>j = 7.3</td>
<td>int</td>
<td>7</td>
</tr>
<tr>
<td>k += 0.3</td>
<td>int</td>
<td>5</td>
</tr>
<tr>
<td>m *= 2.5</td>
<td>int</td>
<td>12</td>
</tr>
<tr>
<td>c = 13 * 256 + 34</td>
<td>char</td>
<td>34</td>
</tr>
</tbody>
</table>
Assignements

- In expression \( x = y \) we have two variables \( x \) and \( y \) but their role is not symmetric:
  - \( y \) refers to the value in memory
  - \( X \) refers to the variable itself (address in memory)
- Expressions in C either have an address in memory or not

\[
\begin{align*}
x &= 7 & /* \text{Legal} */ \\
7 &= x & /* \text{Illegal} */ \\
x &= y + z & /* \text{Legal} */ \\
y + z &= x & /* \text{Illegal} */
\end{align*}
\]
Some simple operations for variables

- In addition to +, −, *, and / we can also use +=, −=, *=, /=, −−, and % (modulo)
- −− (subtract one) e.g. countdown--;
- += (add to a variable) e.g. a+= 5;
- −= (subtract from variable) e.g. num_living-= num_dead;
- *= (multiply a variable) e.g. no_bunnies*=2;
- /= (divide a variable) e.g. fraction/= divisor;
- \((x \% y)\) gives the remainder when \(x\) is divided by \(y\)
- remainder= \(x\%y\); (ints only)
A short note about `++`

- `++i` means increment `i` then use it
- `i++` means use `i` then increment it

```c
int i = 6;
printf("%d\n", i++);  /* Prints 6 sets `i` to 7 */
```

Note this important difference

```c
int i = 6;
printf("%d\n", ++i);  /* prints 7 and sets `i` to 7 */
```

It is easy to confuse yourself and others with the difference between `++i` and `i++` - it is best to use them only in simple ways.

All of the above also applies to `--`. 
### Examples

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Equal to</th>
<th>Value of exp is updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ exp</td>
<td>exp += 1</td>
<td>After the increment</td>
</tr>
<tr>
<td>exp++</td>
<td>exp += 1</td>
<td>Before the increment</td>
</tr>
<tr>
<td>-- exp</td>
<td>exp -= 1</td>
<td>After the decrement</td>
</tr>
<tr>
<td>exp --</td>
<td>exp -= 1</td>
<td>Before the decrement</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>X after computation</th>
<th>Y after computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x=++y</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>x=y++</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>x=--y</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>x=y--</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>