Introduction to C

Tutorial 3: Types and casting
Last week

- Variables
- Input and output - `printf` and `scanf`
- The Input Buffer
Agenda

• Types of variables
• Designing input and output
• Casting between types
Computing the average of numbers

```c
#include <stdio.h>

int main(void) {
    int a, b, avg, result;
    printf("Enter the first value:");
    result = scanf("%d", &a);
    if (result < 1)
        return 1;
    printf("Enter the second value:");
    result = scanf("%d", &b);
    if (result < 1)
        return 1;
    avg = (a + b) / 2;
    printf("The average is %d.\n", avg);
    return 0;
}
```
Is the program crazy?

```bash
C:\Temp>average
Enter first value: 1
Enter second value: 4
The average is 2.
C:\\Temp>
```
The source of the problem

- We said that each memory block has binary digits.
- We can interpret this block however we want (a number, letter, color, etc.)
- When we define a variable, we specify the type

**Type of variable = how the program interprets the cell**

- In our program, we have defined all of the variables using the reserved word `int`, which is short for integer (whole number).
- Therefore, the computation is executed with integer arithmetic.

\[
(1 + 4) / 2 = 5 / 2 = 2.5 = 2
\]

Result of any action between two integers results in an integer.

If non-whole number results, the part after the decimal is truncated.

Equivalent for positive numbers
Equivalent for negative numbers.
Working with fractions

- To work with fractions, define variables of type `float`.
  - The word “float” is tied to how the variable is stored in memory, called “floating point” numbers.
- For example:

```c
float a, b, avg;
a = 1;
b = 4;
avg = (a+b) / 2;
```

Average is 2.5!
In order to print or input fractions, use \texttt{\%f} with \texttt{printf} and \texttt{scanf}.

```c
float total, price;
int result;
result = scanf("\%f", &price);
if (result < 1)
    return 1;
total = price * 9;
printf("The total is \%f", total);
```
Fixing the program

• Can you fix the program so that it correctly calculates the average of 1 and 4?
Variables and types

Important to remember:

• Every variable has a type.
• Variables of different types are input and output in different ways.

A variable's type is fixed when the variable is defined, and cannot change during a program's execution.
Variable types
Variable types

• We’ve formally introduced two types of variables so far:
  • **int** – represents integer variables (whole numbers) both positive and negative.
    – For example 1, 12, -132, 56, 10,343,455.
  • **float** – can represent variables that are not whole numbers (have a fraction portion).
    – For example 1.0, 1.2, -324.32, 3.1415, 10.3e12.

• Important to remember:
  Variables of both types are stored in memory as 0’s and 1’s (bits). The difference is how the value of the variable is translated to 0’s and 1’s and vice versa.
You can determine the size of a variable (how many bytes it takes up) using the `sizeof` operator.

`sizeof` gets the name of the type or the variable as a parameter, and returns the amount of memory (in bytes) that are taken up.

- Recall 1 byte = 8 bits.

For example:

```c
int float_size = sizeof(float);
printf("float size is %d bytes.", float_size);
```
Other types

- Other than the two types we’ve covered, C has several other types.

<table>
<thead>
<tr>
<th>Range</th>
<th>Size in bytes*</th>
<th>Type of data</th>
<th>Type name</th>
</tr>
</thead>
<tbody>
<tr>
<td>-128 .. 127</td>
<td>1</td>
<td>Whole numbers</td>
<td>char</td>
</tr>
<tr>
<td>-2,147,483,648..2,147,483,647</td>
<td>4</td>
<td>Whole numbers</td>
<td>int</td>
</tr>
<tr>
<td>-2,147,483,648..2,147,483,647</td>
<td>4</td>
<td>Whole numbers</td>
<td>long</td>
</tr>
<tr>
<td>±3.4<em>10^-38 .. ±3.4</em>10^38</td>
<td>4</td>
<td>Numbers with fractions</td>
<td>float</td>
</tr>
<tr>
<td>±1.7<em>10^-308 .. ±1.7</em>10^308</td>
<td>8</td>
<td>Numbers with fractions</td>
<td>double</td>
</tr>
</tbody>
</table>

* This is the amount of memory that Code::Blocks allocates for these types. Other compilers may work differently.
Why are there duplicates?

• If (for example) it’s possible to store numbers with fractions in only 4 bytes, why have type that allows numbers with fractions to be stored in 8 bytes?

• In other words

Why do we need double if we have float?
Why do we need int if we have char?

• (Recall that memory is finite. Can using “big variables” result in waste?)
The answer: It’s an issue of precision.

- How many decimal places can be kept in a variable of type float?
- And how about double?
- What’s the largest size of a variable that can be kept inside a char variable?
- And how about long?

The exact answer isn’t so important. The idea that there’s a difference is the important issue.
We saw that the use of scanf and printf is slightly different when using int and float (%d versus %f).

In fact, each type has its own notation in printf/scanf.

Using the wrong notation will not prevent the program from running, but **the program will print incorrect values**.

Because the computer will “decode” the bits in the wrong way.

<table>
<thead>
<tr>
<th></th>
<th>char</th>
<th>int</th>
<th>long</th>
<th>float</th>
<th>double</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>%c</code></td>
<td><code>%d</code></td>
<td><code>%ld</code></td>
<td><code>%f</code></td>
<td><code>%g</code></td>
<td></td>
</tr>
</tbody>
</table>
Type char
Decoding char slightly differently

- A variable of type char can hold one of 256 different values.
- There is a standard table that maps between the 256 values and their corresponding characters.
- For example, A is 65, B is 66, etc.
- If you print a variable of type char using `%c`, the appropriate character will be printed, and not the numeric value!
- The name of the type, “char”, is an abbreviation of character, which hints at this intended use.
ASCII Table: From chars to numbers.

<table>
<thead>
<tr>
<th>Char</th>
<th>Ascii Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>$</td>
</tr>
<tr>
<td>10</td>
<td>=</td>
</tr>
<tr>
<td>11</td>
<td>@</td>
</tr>
<tr>
<td>12</td>
<td>#</td>
</tr>
<tr>
<td>13</td>
<td>$</td>
</tr>
<tr>
<td>14</td>
<td>%</td>
</tr>
<tr>
<td>15</td>
<td>&amp;</td>
</tr>
<tr>
<td>16</td>
<td>'</td>
</tr>
<tr>
<td>17</td>
<td>(</td>
</tr>
<tr>
<td>18</td>
<td>)</td>
</tr>
<tr>
<td>19</td>
<td>*</td>
</tr>
<tr>
<td>20</td>
<td>+</td>
</tr>
<tr>
<td>21</td>
<td>-</td>
</tr>
<tr>
<td>22</td>
<td>,</td>
</tr>
<tr>
<td>23</td>
<td>.</td>
</tr>
<tr>
<td>24</td>
<td>/</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
</tr>
<tr>
<td>28</td>
<td>3</td>
</tr>
<tr>
<td>29</td>
<td>4</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
</tr>
<tr>
<td>31</td>
<td>6</td>
</tr>
<tr>
<td>32</td>
<td>7</td>
</tr>
<tr>
<td>33</td>
<td>8</td>
</tr>
<tr>
<td>34</td>
<td>9</td>
</tr>
<tr>
<td>35</td>
<td>:</td>
</tr>
<tr>
<td>36</td>
<td>;</td>
</tr>
<tr>
<td>37</td>
<td>&lt;</td>
</tr>
<tr>
<td>38</td>
<td>&gt;</td>
</tr>
<tr>
<td>39</td>
<td>?</td>
</tr>
<tr>
<td>40</td>
<td>@</td>
</tr>
<tr>
<td>41</td>
<td>A</td>
</tr>
<tr>
<td>42</td>
<td>B</td>
</tr>
<tr>
<td>43</td>
<td>C</td>
</tr>
<tr>
<td>44</td>
<td>D</td>
</tr>
<tr>
<td>45</td>
<td>E</td>
</tr>
<tr>
<td>46</td>
<td>F</td>
</tr>
<tr>
<td>47</td>
<td>G</td>
</tr>
<tr>
<td>48</td>
<td>H</td>
</tr>
<tr>
<td>49</td>
<td>I</td>
</tr>
<tr>
<td>50</td>
<td>J</td>
</tr>
<tr>
<td>51</td>
<td>K</td>
</tr>
<tr>
<td>52</td>
<td>L</td>
</tr>
<tr>
<td>53</td>
<td>M</td>
</tr>
<tr>
<td>54</td>
<td>N</td>
</tr>
<tr>
<td>55</td>
<td>O</td>
</tr>
<tr>
<td>56</td>
<td>P</td>
</tr>
<tr>
<td>57</td>
<td>Q</td>
</tr>
<tr>
<td>58</td>
<td>R</td>
</tr>
<tr>
<td>59</td>
<td>S</td>
</tr>
<tr>
<td>60</td>
<td>T</td>
</tr>
<tr>
<td>61</td>
<td>U</td>
</tr>
<tr>
<td>62</td>
<td>V</td>
</tr>
<tr>
<td>63</td>
<td>W</td>
</tr>
<tr>
<td>64</td>
<td>X</td>
</tr>
<tr>
<td>65</td>
<td>Y</td>
</tr>
<tr>
<td>66</td>
<td>Z</td>
</tr>
</tbody>
</table>

Space is #32
(the other missing ones are not printable)
Example: Printing characters

#include <stdio.h>
int main(void)
{
    char c = 65;
    printf("The numeric value is %d\n", c);
    printf("The character is %c\n", c);
    return 0;
}
And the output:
The numeric value is 65
The character is A

You can also print a char with a %d.
But you cannot enter a char using %d.
Using characters in a program

• A value of a **char** variable can be set to not only a number, but also a letter/character.
• **It must be in single quotes.**
  For example:

  ```
  char ch = 'B';
  ```

• What will the numerical value of **ch** be in this case?
• You can also read characters from the user using `%c` with `scanf`. Example:

  ```
  scanf("%c", &ch);
  ```
Example: Letter position in the alphabet.

- How can we determine letter position in the alphabet?

<table>
<thead>
<tr>
<th>Letter</th>
<th>‘A’</th>
<th>‘B’</th>
<th>‘C’</th>
<th>...</th>
<th>‘X’</th>
<th>‘Y’</th>
<th>‘Z’</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII number</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>...</td>
<td>88</td>
<td>89</td>
<td>90</td>
</tr>
<tr>
<td>Relative number</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>...</td>
<td>23</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

- Notice the connection between A and X

\[ 88 - 65 = 23 \]

- We can find the relative value of a letter using:

\[
\text{ASCII value of desired letter} - \text{‘A’} = \text{Relative value of desired letter}
\]
Example: Letter position in the alphabet.

```c
#include <stdio.h>

int main()
{
    char ch;
    int position;
    printf("Enter an uppercase letter: ");
    scanf("%c", &ch);
    position = ch - 'A';
    printf("Position ASCII table: %d\n", ch);
    printf("Position AlphaBet: %d\n", position);
    return 0;
}
```
Formatted input and output
What will the program print when the input is 100?

```c
#include <stdio.h>
int main(void)
{
    float per_year;
    float per_month;
    printf("How much do you earn per year? ");
    scanf("%f", &per_year);
    per_month = per_year / 12;
    printf("You earn %f per month.", per_month);
    return 0;
}
```
And the result

6 digits after the dot? No thanks.
Control over printing

- Printing with `printf` can be controlled using special values between the % sign and the letter marking the type.
- For example, when printing a float or double, you can determine how many digits are printed after the period.

```c
printf("%.2f", per_month);
```
  - Will limit the precision to two decimal places.

- Setting the minimum length of the printed number:

```c
printf("%6.2f", per_month);
```
  - If the printed number is less than 6 characters, spaces before the number will be added as needed.
And the result…

• The result of

`printf("You earn %.2f per month.\n", per_month);`

Will be:

![Image showing output](image_url)
Similar to formatted output, scanf can also receive input according to a given format.

- White space: program expects spaces in the input and doesn’t skip over them.
- Other/additional characters: we expect them in the input and do not store them in variables.
- Special characters with %: expected values according to the type described by the character next to the %.
- If the input is not formatted properly, then the remaining invalid parts will stay in the input buffer.
Example: Reading formatted input

How do we input a date according to the form dd/mm/yyyy into integer variables d, m, and y?

```c
#include <stdio.h>
int main(void) {
    int d, m, y;
    printf("Enter the date: ");
    scanf("%d/%d/%d", &d, &m, &y);
    printf("The date in US is written: %d/%d/%d.", m, d, y);
    return 0;
}
```
Example: Inputting characters

What does each variable receive in the program?

```c
char currency;
int payment;
printf("Payment amount ($-dollars, N-NIS): ");
scanf("%d", &payment);
scanf("%c", &currency);
```

<table>
<thead>
<tr>
<th>payment</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td>currency</td>
<td>‘\n’</td>
</tr>
<tr>
<td>payment</td>
<td>1000</td>
</tr>
<tr>
<td>currency</td>
<td>‘$’</td>
</tr>
</tbody>
</table>
Example: Inputting characters (without errors)

Before receiving a single character, we must ensure that the input buffer is empty.
- Otherwise the first character in the buffer will be stored in the variable.
- Sometimes we can prevent this using formatted input.

• In the previous example, we tell scanf to skip the newline character by adding \n before the input.

```c
scanf("\n\%c", \&currency);
```
In the stdio.h library there is a function to input only a single character:

```c
int ch; ch = getchar();
```

If the input buffer is empty, then the program will wait until a character is received.
- It functions just like a scanf with only one %c.

`getchar` returns int instead of char, which allows it to return additional values in case an error occurred.
Type casting
What is type casting?

• We often want to perform a calculation based on values stored in variables of different types.
• For example, adding an integer and a float.
• Thus, one of the variables must be cast to the type of the other variable so that they can be added.
  – Or, both variables can be changed to a third type (double, for example).
• There are two types of casting
  – **Automatic casting** – casting happens without explicit intervention
  – **Explicit casting** – happens with an explicit request to cast a variable to another type.
Explicit casting

• Explicit casting is also called a cast or type cast
• Written with the form: `(type)expression`
• For example:

```java
int i = 5;
float f = 5.0;
double d = 5.9;

i = (int)f;
f = (float)d + (float)i + 2.0;
d = (double)(i * 2);
f = (float)4;
```

Even though this is a whole number, every number with a decimal point is considered as a fraction (a float or double).
Automatic casting

- Operators in C (addition, subtraction, etc.) are only for values of the same type.
- Thus, when we try to use an operator on two values of different types, an automatic cast will happen from one type to another.
- For example:
  ```
  int i = 2;
  double d = 4.5;
  double total = d + i;
  ```
  An int and double cannot be added. Thus an automatic cast happens here.
Automatic casting

• Even when you define a variable, an automatic cast can happen.

• For example:
  double d = 4.5;
  int i = d * 3.1;
  int j = 7.9;
“Type priority” in casting

- To maintain accuracy when possible and avoid lost data, some types always have priority over others.
- For example, to add an int to a char, cast the char to int and not other way around.
- Casts are done by level (hierarchy); always prefer to cast to a type that appears higher in the following hierarchy:
  
  double
  float
  long
  int
  char
• Consider the following code:

```java
int v1 = 3;
int v2 = 4;
double v3 = 7.3;
double result;
result = v3 + v1 / v2;
```

First the division is executed. It's a division between two integers, so no casting is needed. The result is 0.

Now: the addition of double and int. The int is cast to a double, the sum is a double, and it is stored in result.

No problem storing the sum into result, because both are doubles.
Constants
Defining constants using \texttt{#define}

- The \texttt{#define} instruction allows us to define constants.
- For example, the next instruction allows us to define the constant PI:
  \begin{Verbatim}
  \texttt{#define PI 3.14}
  \end{Verbatim}

- This is actually called a preprocessing instruction – before compilation the compiler replaces every “PI” in the code with 3.14.
- If we want to change the precision of PI, we simply change the \texttt{#define} instruction.

  \begin{Verbatim}
  \texttt{#define PI 3.14159}
  \end{Verbatim}
Defining constants with `#define`

```c
#include <stdio.h>
#define DOLLAR 3.62

int main()
{
    printf("239 NIS are %f $\n", DOLLAR*239);
    return 0;
}
```

- Put the `#define` in the beginning of the file, right after the `#include` (s)
- Constants typically have capital letters.

**No semi-colon after the define!**