POLITECHNIKA POZNAŃSKA
POZNAN UNIVERSITY OF TECHNOLOGY

Maciej Sobieraj

Lecture 2

Outline

1. **Data types, their operations and basics of flow control**
    1. **Floating-point numbers**
    2. Computer arithmetic and arithmetic operators
    3. Characters as another kind of data
    4. Controlling the flow – absolute basics
    5. Quiz

Floating-point numbers

- **Floating-point numbers in real life and in the “C” language**
- Floating-point numbers are designed to represent and to store numbers that (as a mathematician would say) have a **non-empty decimal fraction**
- They’re the numbers that have (or may have) a fractional part after the decimal point

Floating-point numbers

- *“Two and a half”* looks normal when you write it in a program, although if your native language prefers to use a comma instead of a point in the number, you should ensure that your number doesn’t contain any commas. The compiler won’t accept it, or (in very rare but possible cases) will misunderstand your intentions, as the comma itself has its own reserved meaning in the “C” language.

Floating-point numbers

- As you can probably imagine, the value of “*zero point four*” could be written in “C” as:
  - **0.4**
- Don't forget this simple rule: you can **omit** zero when it’s the only digit in front of or after the decimal point. In essence, you can write the value *0.4* as shown on the right.

Floating-point numbers

- **For example**: the value of 4.0 could be written as 4. without changing its type or value.
- **Note**: the decimal point is essential in recognizing floating-point numbers in “C”. Look at these two numbers:

4

4.0

Floating-point numbers

- For you, they might be exactly the same, but the “C” compiler sees these two numbers in a completely different way.

  4 is an *int*.
  4.0 is a double (can be easily assigned to *float*).

  We can say that the **point makes a double**. Don't forget that.

Floating-point numbers

- When you want to use any numbers that are very large or very small, you can use **scientific notation**.
- Written directly it would look like this:
  - **300000000**
- To avoid tediously writing so many zeros, physics textbooks use an abbreviated form, which you have probably already seen:
  - 3 • 10^8

Floating-point numbers

- In the “C” language, the same effect is achieved in a slightly different form – take a look:
  - **3E8**
- The letter *E* (you can also use the lower case letter *e* – it comes from the word exponent) is a concise version of the phrase *“times ten to the power of”.*
- Note:
  - the exponent (the value after the *“E”*) **has to be** an integer.
  - the base (the value in front of the *“E”*) **may or may not be** an integer.

Floating-point numbers

- A physical constant called *Planck's constant* (and denoted as *h*) has, according to the textbooks, the value of:

  6.62607 x 10-34

  If you would like to use it in a program, you would write it this way:

  6.62607E-34

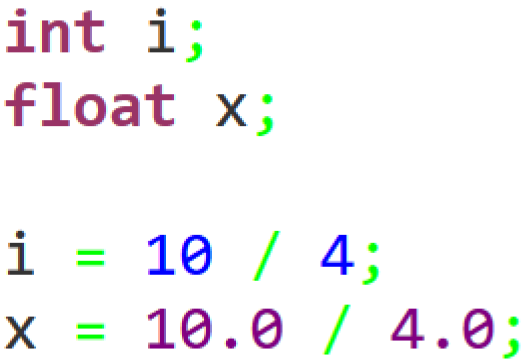
Floating-point numbers

- The declaration of float variable is done by using the keyword ***float***.

```
float PI, Field;
```

Floating-point numbers

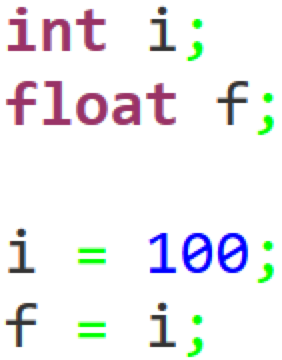
- Difference between ***int*** and ***float*** is very significant in terms of semantics



```
int i;
float x;

i = 10 / 4;
x = 10.0 / 4.0;
```

Floating-point numbers

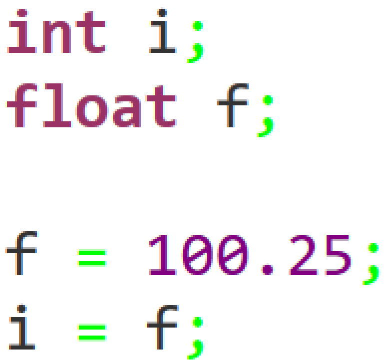
- The transformation from type ***int*** into ***float*** is always possible and feasible, but in some cases can cause a loss of accuracy.



```
int i;
float f;

i = 100;
f = i;
```

Floating-point numbers

- We can observe a loss of accuracy when we want convert ***float*** to ***int***.
- There’s another aspect of the operation: converting a ***float*** into an ***int*** is not always feasible. Integer variables (like ***floats***) have a **limited capacity**.



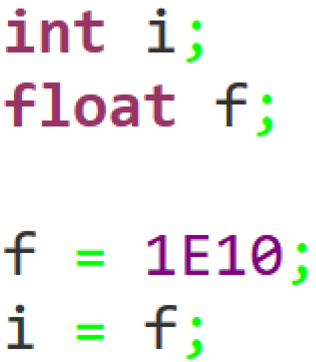
```
int i;
float f;

f = 100.25;
i = f;
```

Floating-point numbers

- if a certain type of computer uses four bytes (i.e. 32 bits) to store ***int*** values, you’re only able to use the numbers from the range of -2147483648..2147483647.

Floating-point numbers

- The *i* variable is unable to store such a large value, but it isn’t clear what will happen during the assignment.
- Certainly, a **loss of accuracy** will occur, but the value assigned to the variable *i* is not known in advance.



```
int i;
float f;

f = 1E10;
i = f;
```

Floating-point numbers

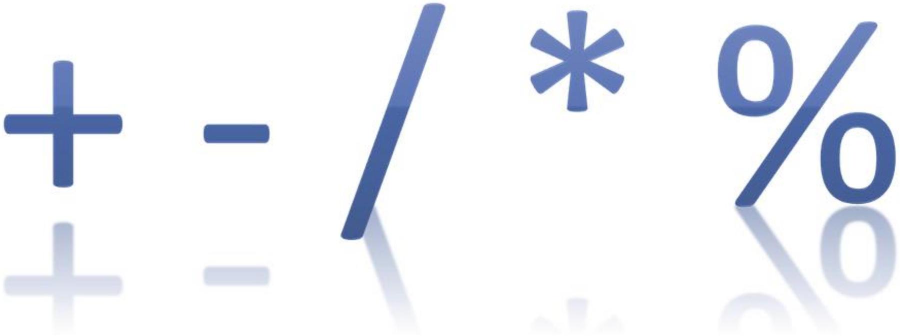
- In some systems, it may be the maximum permissible ***int*** value, while in others an error occurs.
- This is what we call an **implementation dependent issue**.

Outline

1. **Data types, their operations and basics of flow control**
    1. Floating-point numbers
    2. **Computer arithmetic and arithmetic operators**
    3. Characters as another kind of data
    4. Controlling the flow – absolute basics
    5. Quiz

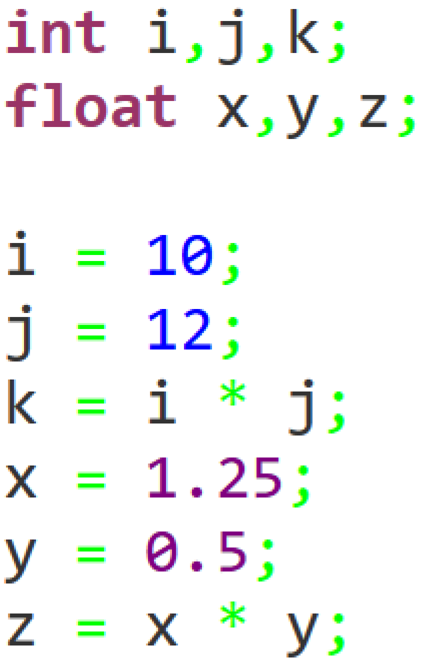
Operators

- An **operator** is a symbol of the programming language which is able to **operate** on the values.
- We’ll begin with the operators associated with widely recognizable **arithmetic operations**.



Multiplication

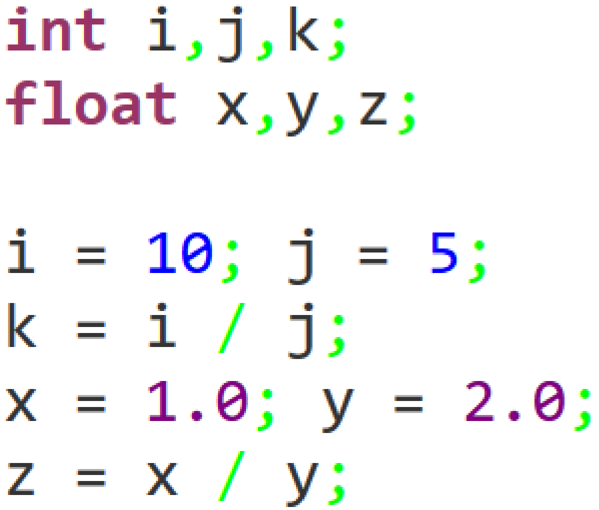
- An asterisk ("*") is a **multiplication** operator.



```
int i,j,k;
float x,y,z;

i = 10;
j = 12;
k = i * j;
x = 1.25;
y = 0.5;
z = x * y;
```

Division

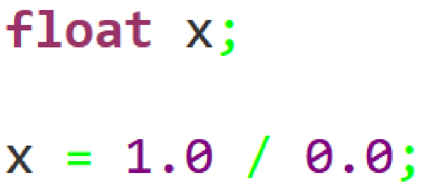
- A slash ("/") is a **divisional** operator. The value in front of the slash is a **dividend**, the value behind the slash, a **divisor**.



```
int i,j,k;
float x,y,z;

i = 10; j = 5;
k = i / j;
x = 1.0; y = 2.0;
z = x / y;
```

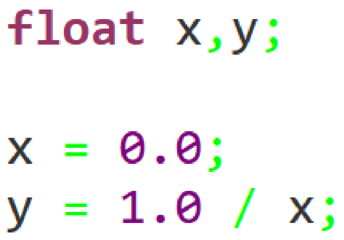
Division by zero

- As you’ve probably guessed, division by zero is strictly **forbidden.**



```
float x;

x = 1.0 / 0.0;
```

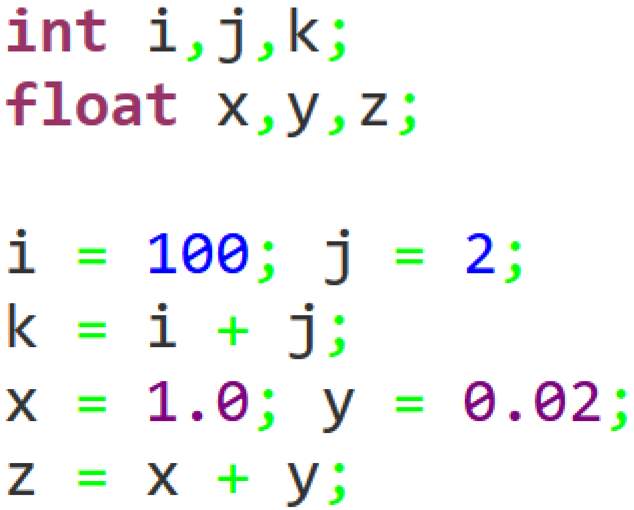
- In the following example, the compiler won't tell you anything, but when you try to execute the code it may **terminate abnormally** and produce unreliable results.



```
float x,y;

x = 0.0;
y = 1.0 / x;
```

Addition

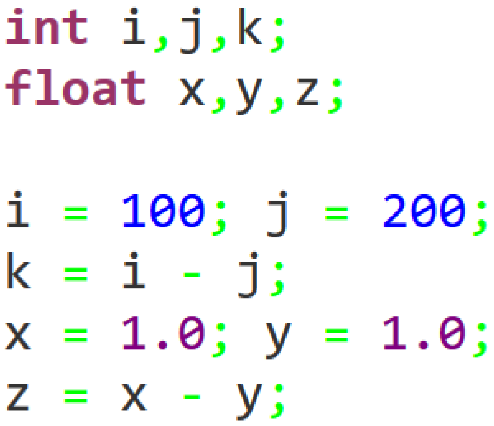
- The **addition** operator is the “+” (plus) sign, which is one that we already know from mathematics.



```
int i,j,k;
float x,y,z;

i = 100; j = 2;
k = i + j;
x = 1.0; y = 0.02;
z = x + y;
```

Subtraction

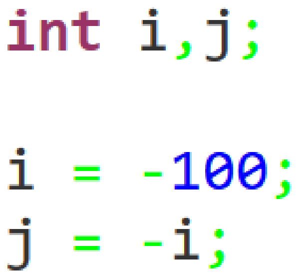
- The **subtraction** operator is obviously the “-“ (minus) sign, although you should note that this operator also has another meaning – it can change the sign of a number.
- This is a great opportunity to show you a very important distinction between **unary** and **binary** operators



```
int i,j,k;
float x,y,z;

i = 100; j = 200;
k = i - j;
x = 1.0; y = 1.0;
z = x - y;
```

Unary minus

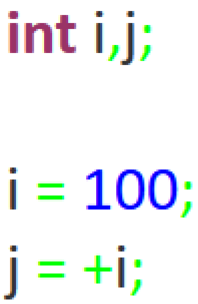
- In “subtracting” applications, the minus operator expects two arguments: the left (a **minuend** in arithmetic terms) and the right (a **subtrahend**). For this reason, the subtraction operator is considered one of the binary operators, just like the addition, multiplication and division operators.
- We used the minus operator as a **unary** operator.



```
int i,j;

i = -100;
j = -i;
```

Unary plus

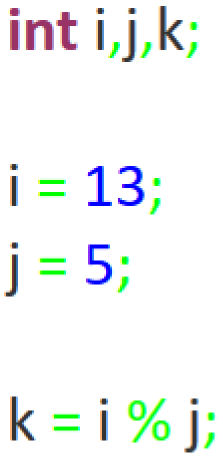
- he same dual nature is expressed by the “+” operator, which can be also used as a unary operator – its role is to **preserve** the sign.



```
int i,j;

i = 100;
j = +i;
```

Remainder

- This is quite a peculiar operator, because it has no equivalent among traditional arithmetic operators. Its graphical representation in the “C” language is the “%” (percent) character.



```
int i,j,k;

i = 13;
j = 5;

k = i % j;
```

Priorities

- The “C” language precisely defines the **priorities** of all operators and assumes that operators of a larger (higher) priority perform their operations before the operators with a lower priority. So, if we know that “*” has a higher priority than “+”, the computation of the final result is pretty obvious.

Bindings

- The **binding** of the operator determines the order of the computations performed by some operators with equal priority, put side by side in one expression.
- Most operators in the “C” language have the **left-sided binding**, which means that the calculation of the expression shown here is conducted from **left to right.**

$$2 + 3 + 5$$

List of priorities

+ -	unary
* / %	
+ -	binary

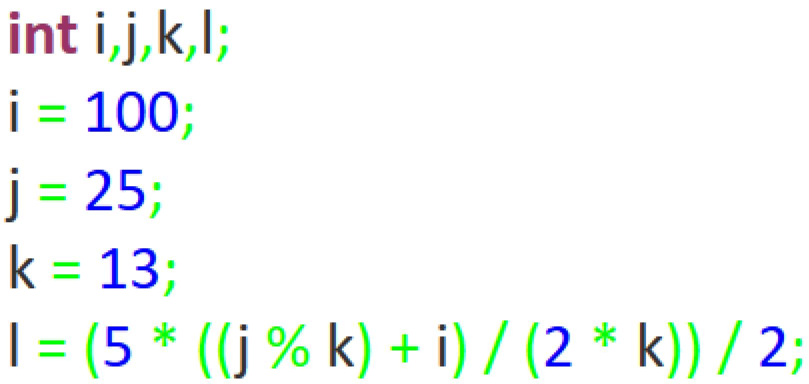
List of priorities

- Both operators ("*" and "%") have the same priority.

2 * 3 % 5

Parentheses

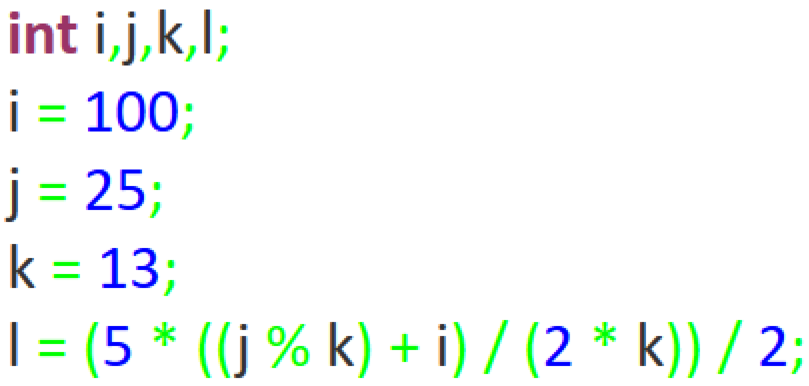
- We’re always allowed to use **parentheses**, which can change the natural order of calculation



```
int i,j,k,l;
i = 100;
j = 25;
k = 13;
l = (5 * ((j % k) + i) / (2 * k)) / 2;
```

Parentheses

- We’re always allowed to use **parentheses**, which can change the natural order of calculation



```
int i,j,k,l;
i = 100;
j = 25;
k = 13;
l = (5 * ((j % k) + i) / (2 * k)) / 2;
```

Operators continued

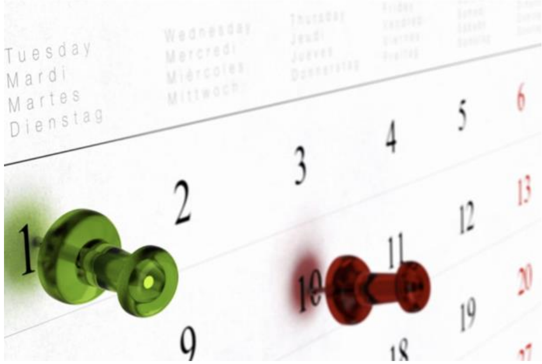
- There are some operators in the “C” language which you won’t find in the mathematics textbooks.
- Let's consider the following snippet:
  - **int** SheepCounter;
  - SheepCounter = 0;

Operators continued

- Every time a sheep runs through our thoughts we want the variable to be incremented, like this:
  - SheepCounter = SheepCounter + 1;
- Similar operations appear very frequently in typical programs so the creators of the “C “ language introduced a set of special operators for them. One is the + + (plus plus) operator. You can achieve the same effect in a shorter way:
  - SheepCounter++;

Operators continued

- Similarly, you can also decrease the value of a chosen variable. For example, if we can't wait for the holidays, our mind does the following operation every morning:
  - `DaysUntilHoliday = DaysUntilHoliday - 1;`
- We can write it in a more compact way:
  - `DaysUntilHoliday--;`

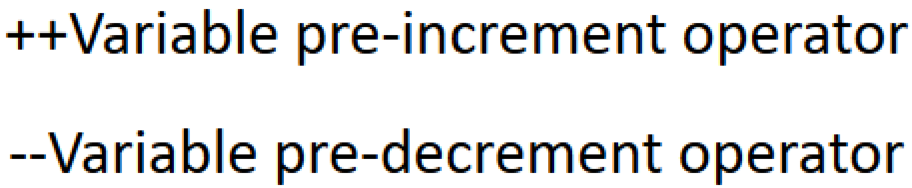


Operators continued

- Sorry, but now we have to introduce a few new words.
    - The “++” is called the **increment operator**.
    - The “--” is called the **decrement operator**.
- However, both operators can be placed in front of a variable as well (as prefix operators), like this:
    - ++SheepCounter;
    - --DaysUntilHoliday;

Pre-and post-operators and their priorities

- Operation:
    - ++Variable
    - --Variable
- Effect:
    - Increment/decrement the variable by 1 and return its value **already** increased/reduced.

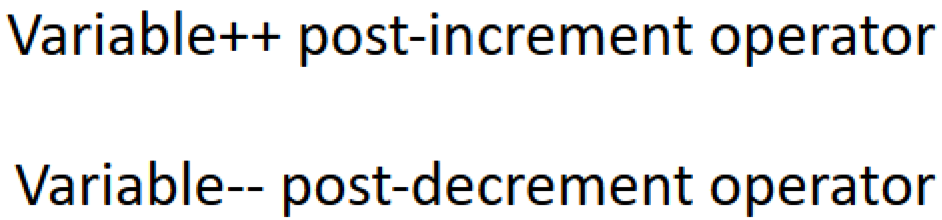


++Variable pre-increment operator

--Variable pre-decrement operator

Pre-and post-operators and their priorities

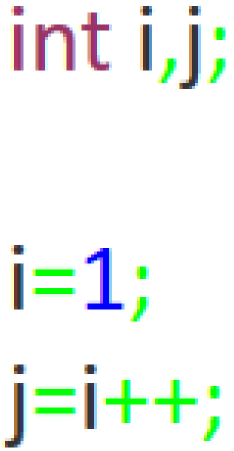
- Operation:
  - Variable++
  - Variable--
- Effect:
  - Return the original (unchanged) variable's value and then increment/decrement the variable by 1.



Variable++ post-increment operator

Variable-- post-decrement operator

Pre-and post-operators and their priorities

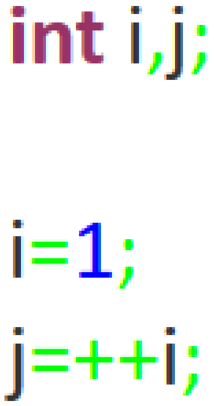


```
int i,j;

i=1;
j=i++;
```

Pre-and post-operators and their priorities

- First, the variable *i* is set to 1. In the second statement, we'll see the following steps:
  - the value of i will be taken (as we use the *post-incrementation*);
  - the variable i will be increased by 1.
- In effect, *j* will receive the value of *1* and *i* the value of *2*.

Pre-and post-operators and their priorities

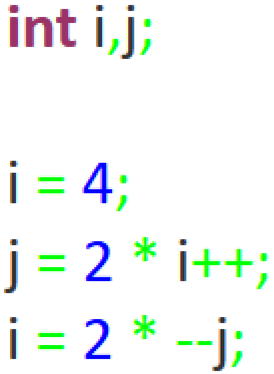


```
int i,j;

i=1;
j=++i;
```

Pre-and post-operators and their priorities

- The variable *i* is assigned with the value of *1*; next, the *i* variable is incremented and is equal to *2*; next, the increased value is assigned to the *j* variable.
- In effect, both *i* and *j* will be equal to *2*.

Pre- and post- operators

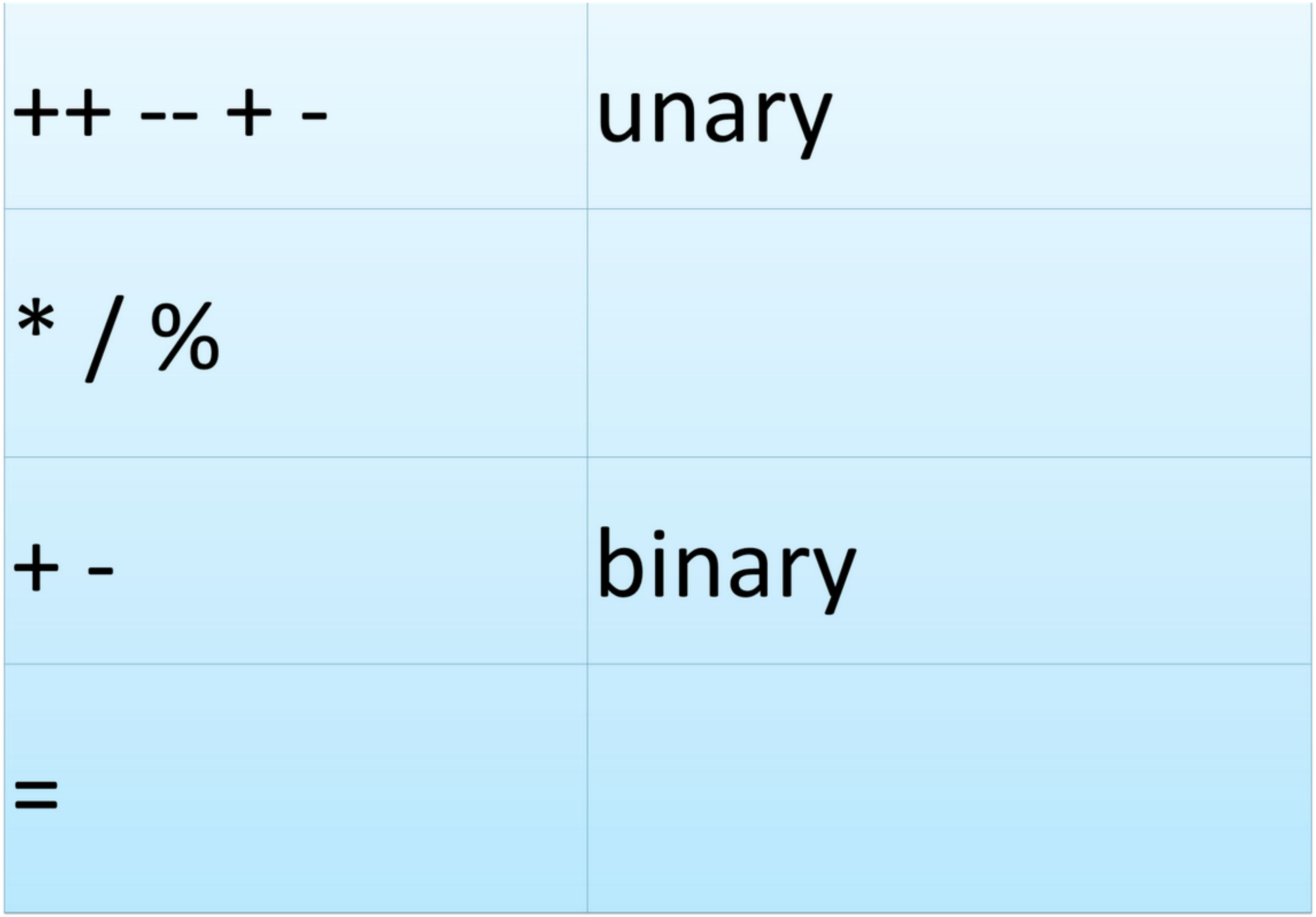


```
int i,j;

i = 4;
j = 2 * i++;
i = 2 * --j;
```

Pre- and post- operators

- Look carefully at this program. Let’s trace its execution step by step.
    - The *i* variable is assigned the value of *4*;
    - We take the original value of *i* (*4*), multiply it by *2*, assign the result (*8*) to *j* and eventually (post-)increment the *i* variable (it equals *5* now);
    - We (pre-)decrement the value of *j* (it equals 7 now); this reduced value is taken and multiplied by *2* and the result (*14*) is assigned to the variable *i*.

Pre-and post-operators and their priorities



++ -- + -	unary
* / %	
+ -	binary
=	

Shortcut operators

```
I = I * 2;
```

```
SHEEPCOUNTER = SHEEPCOUNTER + 10;
```

```
i *= 2;

SheepCounter += 10;
```

Shortcut operators

- If *op* is a **two-argument operator** (this is a very important condition!) and the operator is used in the following context:
  - variable = variable **op** expression;
- then this expression can be **simplified** as follows:
  - variable **op** = expression;

Shortcut operators

i = i + 2 * j;	i += 2 * j;
Var = Var / 2;	Var /= 2;
Rem = Rem % 10;	Rem %= 10;
j = j – (i + Var + Rem);	j -= (i + Var + Rem);

Outline

1. **Data types, their operations and basics of flow control**
    1. Floating-point numbers
    2. Computer arithmetic and arithmetic operators
    3. **Characters as another kind of data**
    4. Controlling the flow – absolute basics
    5. Quiz

Character type

- We can define a “*word*” as a **string of characters** (letters, numbers, punctuation marks, etc.)
- We dealt with these strings during the first lesson when we used the *puts* function to write some text on the computer screen.
- The problem with processing strings, though, will come back to haunt us when we start working on arrays, because in the “C” language all strings are treated as **arrays**.

Character type

- To store and manipulate characters, the “C” language provides a special type of data. This type is called a **char**, which is an abbreviation of the word “*character*”.

```
char Character;
```

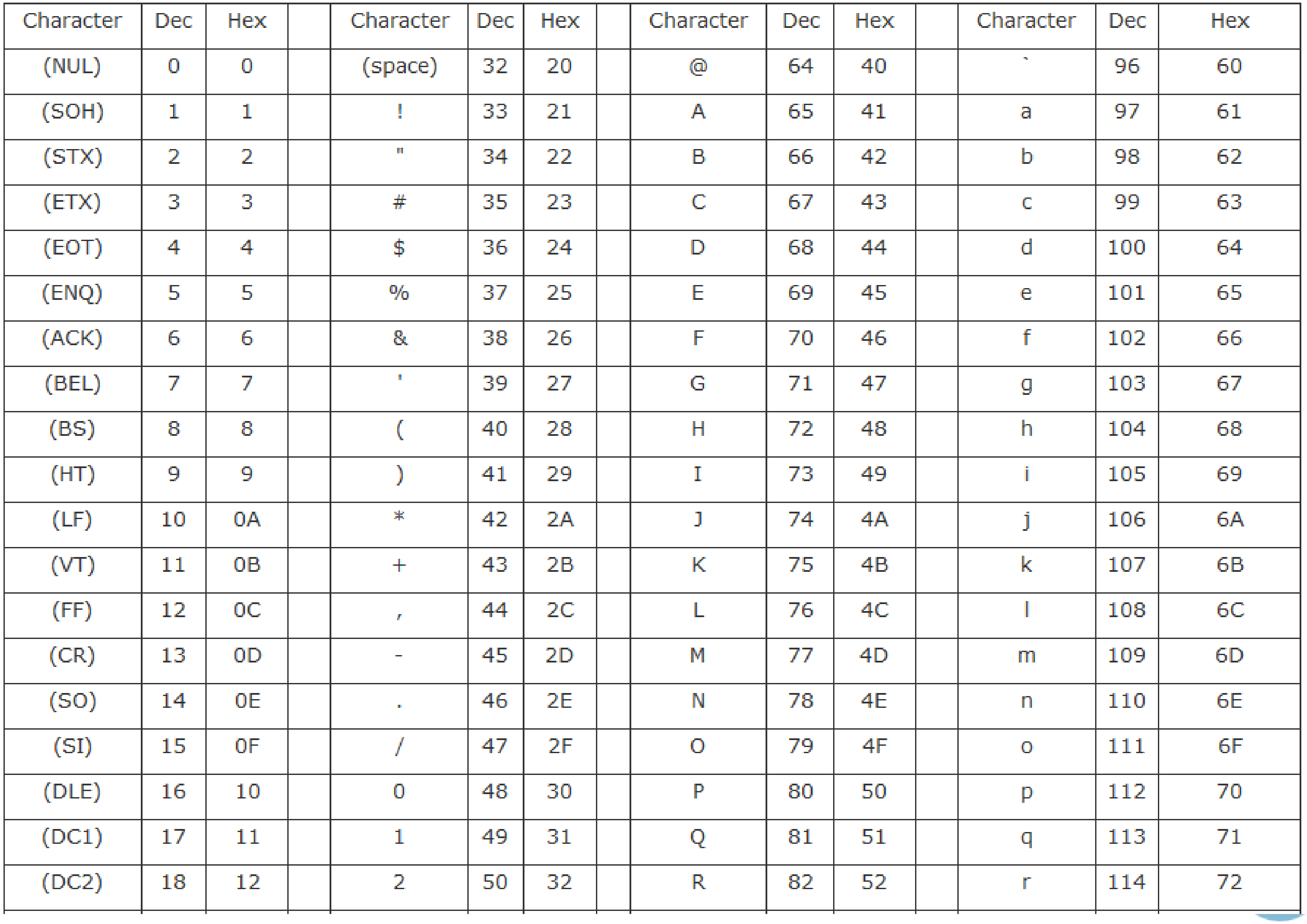
ASCII code

- Computers store characters as numbers.
- Every character used by a computer corresponds to a unique number, and vice versa.
- Many of them are invisible to humans but essential for computers. Some of these characters are called **white spaces**, while others are named **control characters**, because their purpose is to **control** the input/output devices.

ASCII code

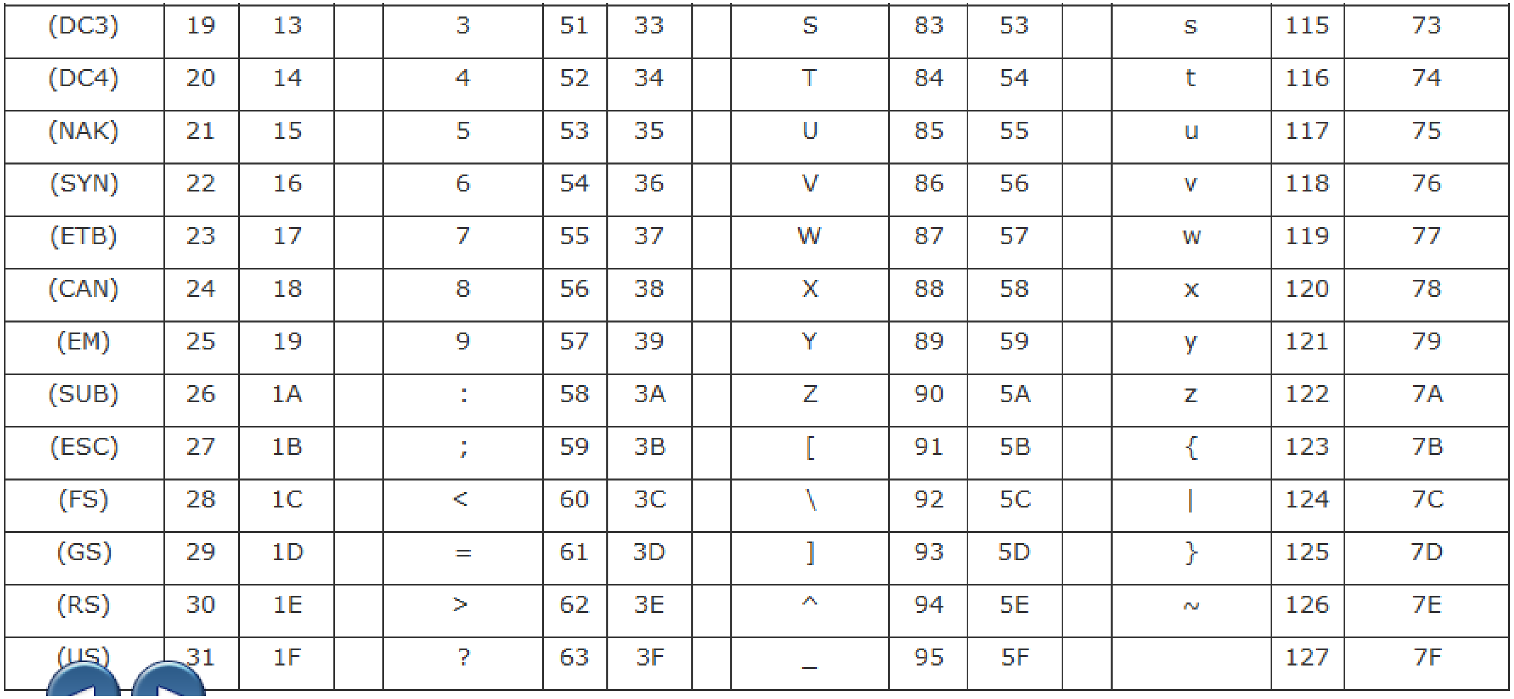
- This has created a need to introduce a universal and widely accepted standard implemented by (almost) all computers and operating systems all over the world. **ASCII** (which is a short for *American Standard Code for Information Interchange*) is the most widely used system in the world, and it's safe to assume that nearly all modern devices (like computers, printers, mobile phones, tablets, etc.) use this code.

ASCII code



Character	Dec	Hex		Character	Dec	Hex		Character	Dec	Hex		Character	Dec	Hex
(NUL)	0	0		(space)	32	20		@	64	40		`	96	60
(SOH)	1	1		!	33	21		A	65	41		a	97	61
(STX)	2	2		"	34	22		B	66	42		b	98	62
(ETX)	3	3		#	35	23		C	67	43		c	99	63
(EOT)	4	4		$	36	24		D	68	44		d	100	64
(ENQ)	5	5		%	37	25		E	69	45		e	101	65
(ACK)	6	6		&	38	26		F	70	46		f	102	66
(BEL)	7	7		'	39	27		G	71	47		g	103	67
(BS)	8	8		(	40	28		H	72	48		h	104	68
(HT)	9	9		)	41	29		I	73	49		i	105	69
(LF)	10	0A		*	42	2A		J	74	4A		j	106	6A
(VT)	11	0B		+	43	2B		K	75	4B		k	107	6B
(FF)	12	0C		,	44	2C		L	76	4C		l	108	6C
(CR)	13	0D		-	45	2D		M	77	4D		m	109	6D
(SO)	14	0E		.	46	2E		N	78	4E		n	110	6E
(SI)	15	0F		/	47	2F		O	79	4F		o	111	6F
(DLE)	16	10		0	48	30		P	80	50		p	112	70
(DC1)	17	11		1	49	31		Q	81	51		q	113	71
(DC2)	18	12		2	50	32		R	82	52		r	114	72

ASCII code



(DC3)	19	13		3	51	33		S	83	53		s	115	73
(DC4)	20	14		4	52	34		T	84	54		t	116	74
(NAK)	21	15		5	53	35		U	85	55		u	117	75
(SYN)	22	16		6	54	36		V	86	56		v	118	76
(ETB)	23	17		7	55	37		W	87	57		w	119	77
(CAN)	24	18		8	56	38		X	88	58		x	120	78
(EM)	25	19		9	57	39		Y	89	59		y	121	79
(SUB)	26	1A		:	58	3A		Z	90	5A		z	122	7A
(ESC)	27	1B		;	59	3B		[	91	5B		{	123	7B
(FS)	28	1C		<	60	3C		\	92	5C		|	124	7C
(GS)	29	1D		=	61	3D		]	93	5D		}	125	7D
(RS)	30	1E		>	62	3E		^	94	5E		~	126	7E
(US)	31	1F		?	63	3F		_	95	5F			127	7F

ASCII code

- Do you see what the code of the most common character is – the space? Yes – it’s 32.
- Now look at what the code of the lower-case letter “a” is. It’s *97*, right?
- And now let's find the upper-case “A”. Its code is *65*.
- What’s the difference between the code of “a” and “A”?
  - It’s 32. Yes, that's the code of a space.

Character type values

- The first way allows us to specify the character itself, but **enclosed in single quotes** (apostrophes).

```
Character = 'A';
```

Character type values

- You’re not allowed to omit apostrophes under any circumstances.

```
Character = '*';
```

Character type values

- The second method consists of assigning a **non-negative integer value** that is the code of the desired character.

```
Character = 65;
```

Literal

- The literal is a symbol which **uniquely identifies its value**.
  - **Character**: this is not a literal; it’s probably a variable name; when you look at it, you cannot guess what value is currently assigned to that variable;
  - **'A'**: this is a literal; when you look at it you can immediately guess its value; you even know that it’s a literal of the *char* type;
  - **100**: this is a literal, too (of the *int* type);
  - **100.0**: this is another literal, this time of a floating point type;
  - **i + 100**: this is a combination of a variable and a literal joined together with the + operator; this structure is called an **expression**.

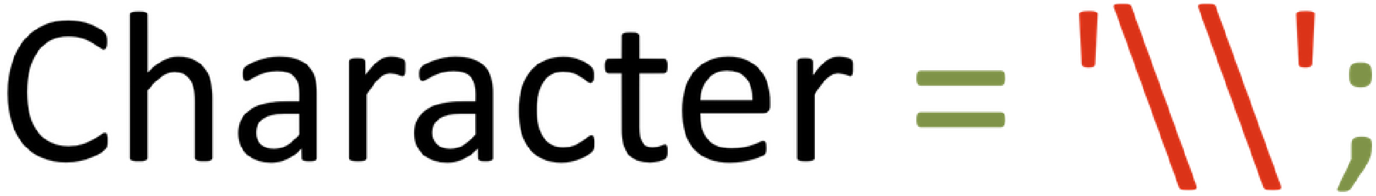
Character literals

- The “C” language uses a special convention which also extends to other characters, not only to apostrophes.
- The \ character (called *backslash*) acts as an **escape character**, because by using the \ we can escape from the normal meaning of the character that follows the slash.

```
Character = '\'';
```

Character literals

- You can also use the **escape character** to **escape** from the escape character.



```
Character = '\\';
```

Escape characters

- The “C” language allows us to escape in other circumstances too.
- \n – denotes a **transition to a new line** and is sometimes called an **LF** (**Line Feed**), as printers react to this character by pulling out the paper by one line of text.

\n

Escape characters

- \r – denotes the **return to the beginning of the line** and is sometimes called a **CR** (**Carriage Return** – “carriage” was the synonym of a “print head” in the typewriter era); printers respond to this character as if they are told to re-start printing from the left margin of the already printed line.

\r

Escape characters

- \0 (*note: the character after the backslash is a zero, not the letter O*): called **nul** (from the Latin word **nullus** – none) is a character that **does not represent any character**;

\0

Escape characters

- Now we’ll try to escape in a slightly different direction. The first example explains the variant when a backslash is followed by two or three **octal digits** (the digits from the range of 0 to 7).

```
Character = '\47';
```

Escape characters

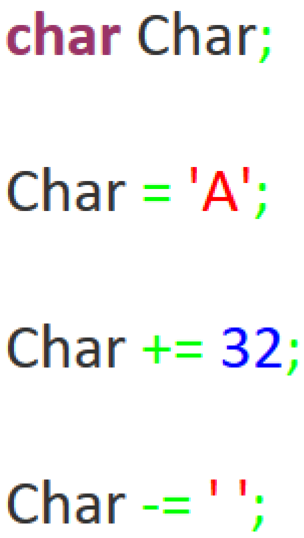
- The second escape refers to the situation when a \ is followed by the letter *X* (lower case or upper case – it doesn't matter). In this case there must be either one or two **hexadecimal digits**, which will be treated as ASCII code.

```
Character = '\x27';
```

Char values are **int** values

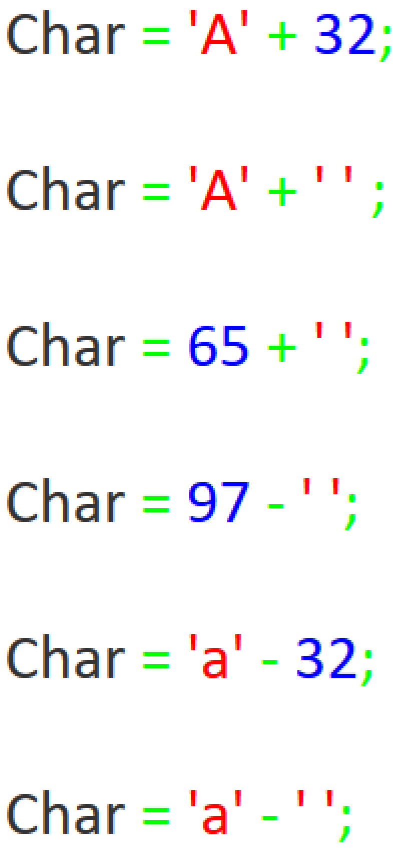
- There's an assumption in the "C" language that may seem surprising at first glance: the ***char*** type is treated as a special kind of ***int*** type. This means that:
    - You can always **assign a *char* value** to an ***int*** variable;
    - You can always **assign an *int* value** to a ***char*** variable, but if the value exceeds 255 (the top-most character code in ASCII), you must expect a loss of value;
    - The value of the *char* type can be subject to the **same operators** as the data of type *int*.

Char values are **int** values



```
char Char;

Char = 'A';

Char += 32;

Char -= ' ';
```

Char values are **int** values



```
Char = 'A' + 32;

Char = 'A' + ' ' ;

Char = 65 + ' ';

Char = 97 - ' ';

Char = 'a' - 32;

Char = 'a' - ' ';
```

Outline

1. **Data types, their operations and basics of flow control**
   1. Floating-point numbers
   2. Computer arithmetic and arithmetic operators
   3. Characters as another kind of data
   4. **Controlling the flow – absolute basics**
   5. Quiz

One who asks does not err

- Computers know only two kinds of answer: ***yes, this is true or no, this is false.***
- You will never get a response like “I don’t know” or “Probably yes, but I don’t know for sure”.
- To ask questions, the “C” language uses a set of very special operators.

Question: is x equal to y?

- Question: are two values equal?
  - To ask this question you use the == ***(equal equal)*** operator.
- Don't forget this important distinction:
  - = is an **assignment operator**
  - == is the **question *"are these values equal?"***
  - == is a **binary** operator with left-side binding. It needs two arguments and checks if they're equal.

Is x equal to y?

2 == 2

1 == 2

Is x equal to y?

- Note that we can’t know the answer if we don’t know what value is currently stored in the variable *i*.
- If the variable has been changed many times during the execution of your program, the answer to this question can be given only by the computer and only at runtime.

i == 0

Is x equal to y?

```
BlackSheepCounter == 2 * WhiteSheepCounter
```

- Due to the low priority of the == operator, this question shall be treated as equivalent to this one:
  - BlackSheepCounter == (2 * WhiteSheepCounter)

Question: is x not equal to y?

- To ask this question, we use the != (**exclamation equal**).

```
DaysUntilTheEndOfTheWorld != 0
```

Question: is x greater than y?

- You can ask this question by using the > (**greater than**) operator.

BlackSheep > WhiteSheep

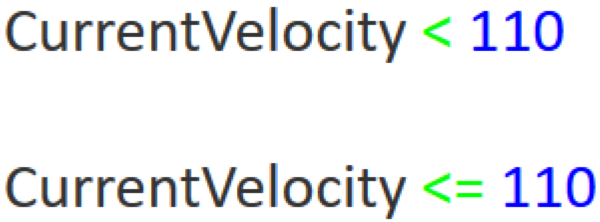
Question: is x greater than y?

- The “***greater than***” operator has another special, non-strict variant, but it’s denoted differently in classical arithmetic notation: >= (***greater than or equal***).

```
CentigradesOutside >= 0.0
```

Question: is x less than (or equal to) y?

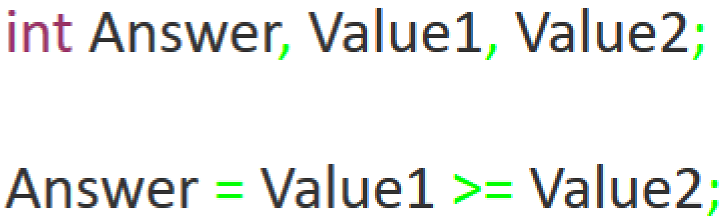
- As you’ve probably already guessed, the operators we use in this case are: the < (***less than***) operator and its non-strict sibling <= (***less than or equal***).



```
CurrentVelocity < 110

CurrentVelocity <= 110
```

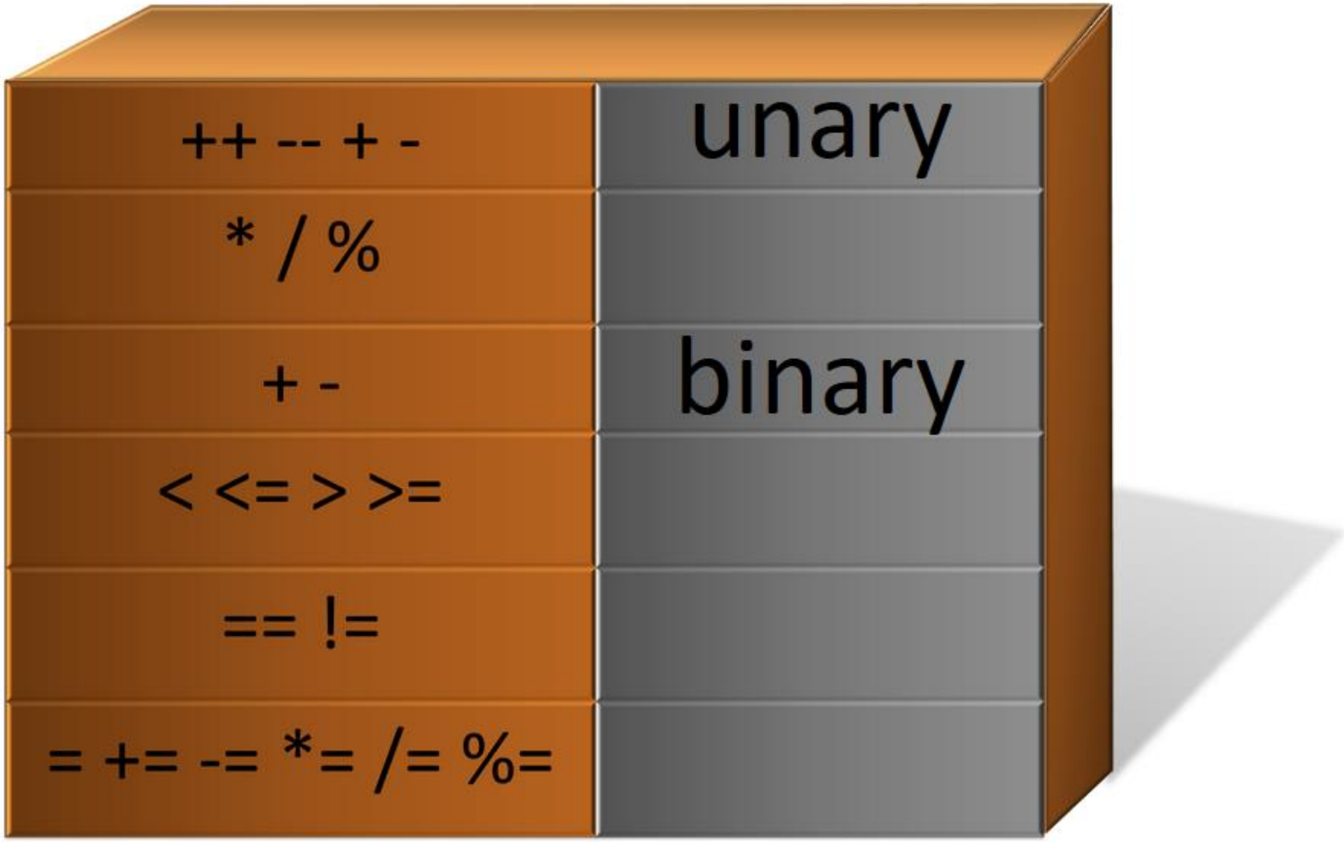
How to use the answer we got?

- What can we do with the answer we get from the computer? There are at least two possibilities: first, we can **memorize it** (store it in a variable) and make use of it later.



```
int Answer, Value1, Value2;

Answer = Value1 >= Value2;
```

The priority table – an update.



++ -- + -	unary
* / %	
+ -	binary
< <= > >=	
== !=	
= += -= *= /= %=	

Conditions and conditional executions

- We must have a mechanism which allows us to do something if a condition is met and not to do it if it isn’t.
- To make these decisions, the “C” language has a special instruction. Due to its nature and its application, it’s called a **conditional instruction** (or **conditional statement**).

```
if(true_or_not) do_this_if_true;
```

Conditions and conditional executions

```
if(true_or_not) do_this_if_true;
```

- This conditional statement consists of the following, strictly necessary, elements in this and this order only:
    - **if** keyword;
    - left (opening) parenthesis;
    - an **expression** (a **question** or an **answer**) whose value will be interpreted solely in terms of “true” (when its value is non-zero) and “false” (when it is equal to zero);
    - right (closing) parenthesis;
    - an **instruction** (only **one**, but we’ll learn how to deal with that limitation).

Conditions and conditional executions

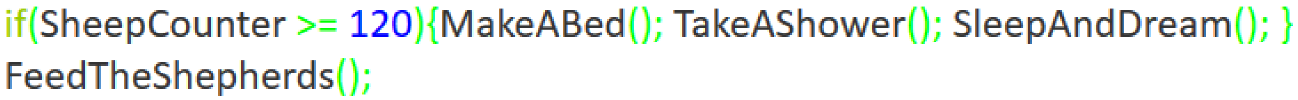
```
if(true_or_not) do_this_if_true;
```

- How does this statem*ent work?*
    - *if the* true_or_not expression enclosed inside the parentheses represents the truth (i.e. its *value is not equal to zero*), the statement behind this condition (*do_this_if_true*) **will be executed**;
    - if the *true_or_not* expression represents a **falsehood** (its value is **equal to zero**), the statement behind this condition is **omitted** and the next executed instruction will be the one that lies after the conditional statement.

Conditions and conditional executions

```
if(TheWeatherIsGood) GoForAWalk();
HaveLunch();
```

```
if(SheepCounter >= 120) SleepAndDream();
```



```
if(SheepCounter >= 120){MakeABed(); TakeAShower(); SleepAndDream(); }
FeedTheShepherds();
```

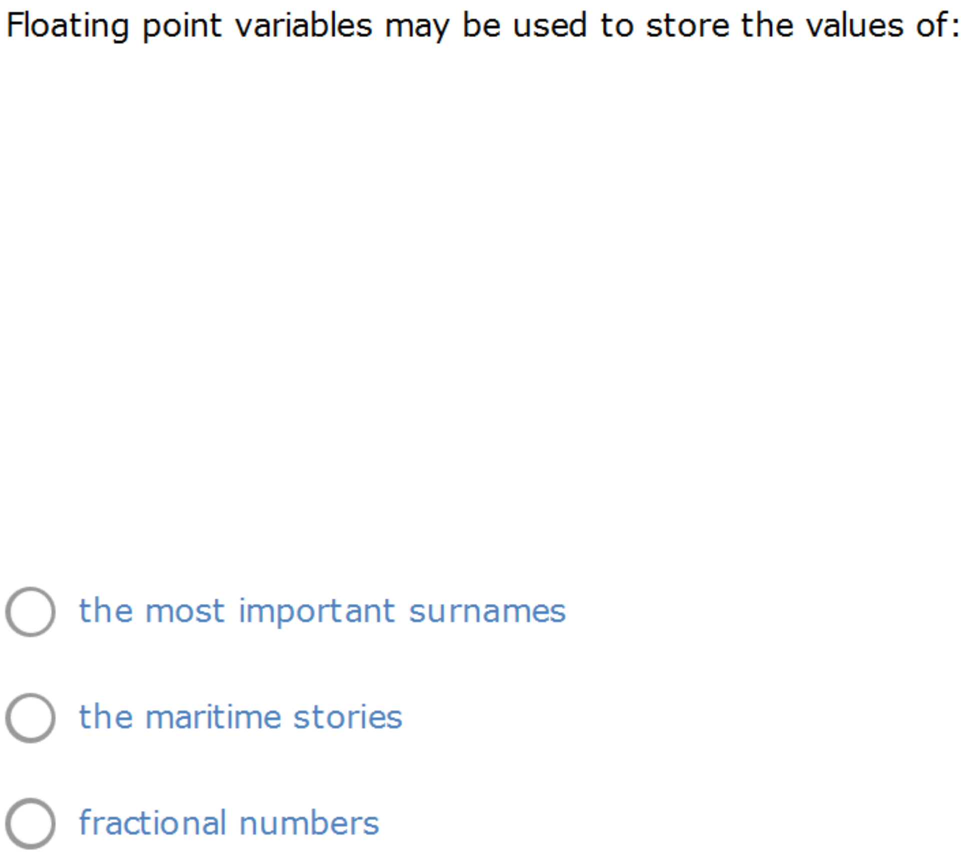
Conditions and conditional executions

```
if(SheepCounter >= 120){
MakeABed();
TakeAShower();
SleepAndDream();
}
FeedTheShepherds();
```

Outline

1. **Data types, their operations and basics of flow control**
    1. Floating-point numbers
    2. Computer arithmetic and arithmetic operators
    3. Characters as another kind of data
    4. Controlling the flow – absolute basics
    5. **Quiz**

Quiz



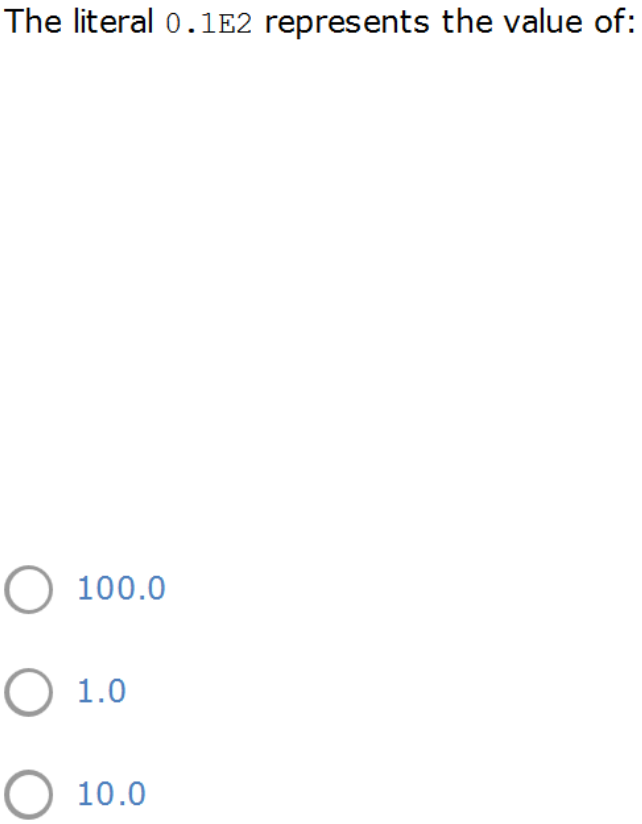
Floating point variables may be used to store the values of:

○ the most important surnames

○ the maritime stories

○ fractional numbers

Quiz



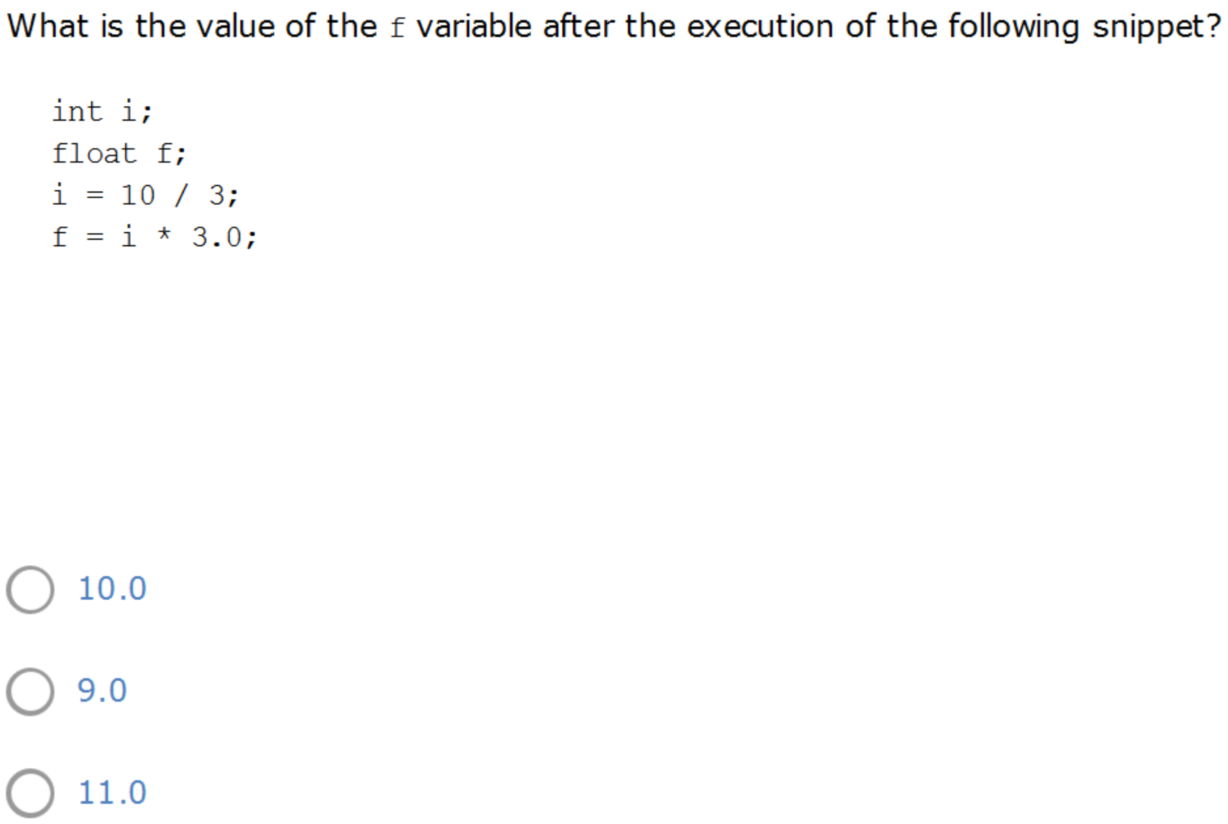
The literal `0.1E2` represents the value of:

○ 100.0

○ 1.0

○ 10.0

Quiz



What is the value of the `f` variable after the execution of the following snippet?

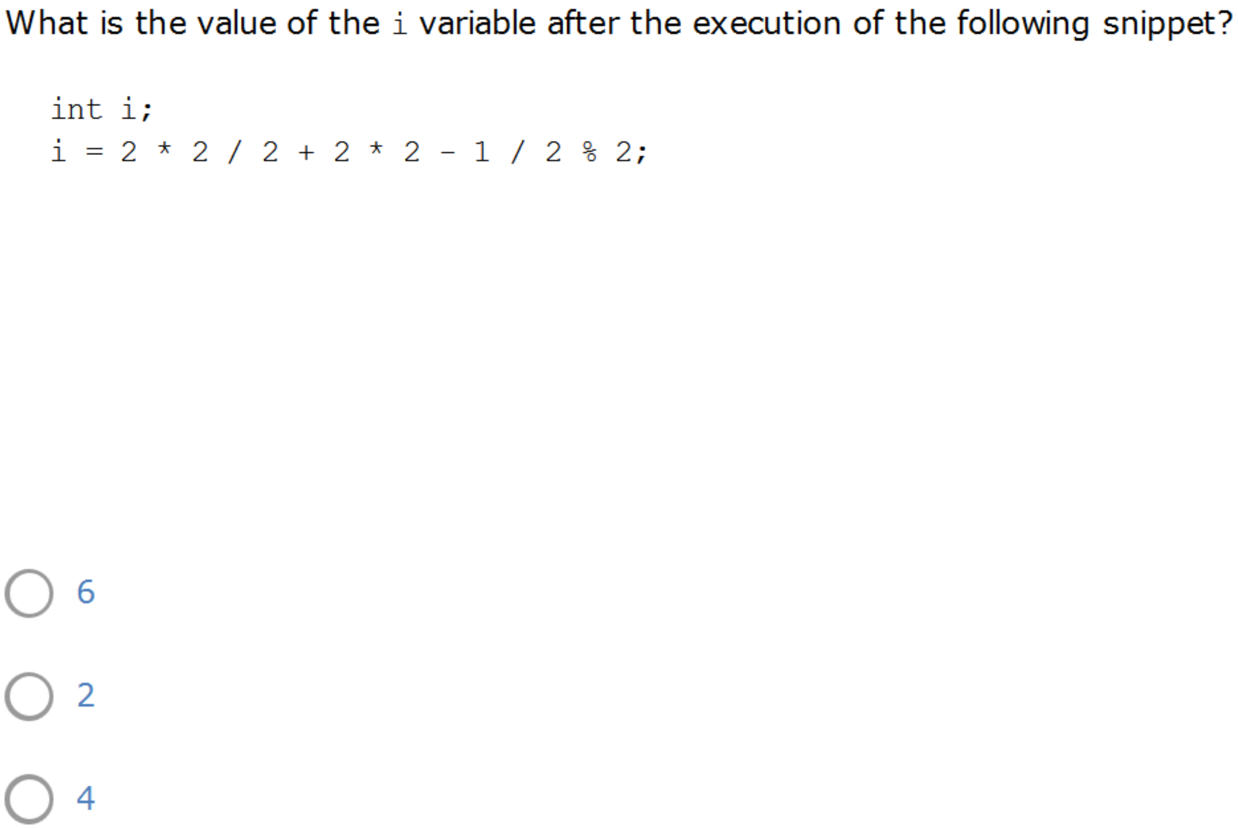
```
int i;
float f;
i = 10 / 3;
f = i * 3.0;
```

○ 10.0

○ 9.0

○ 11.0

Quiz



What is the value of the `i` variable after the execution of the following snippet?

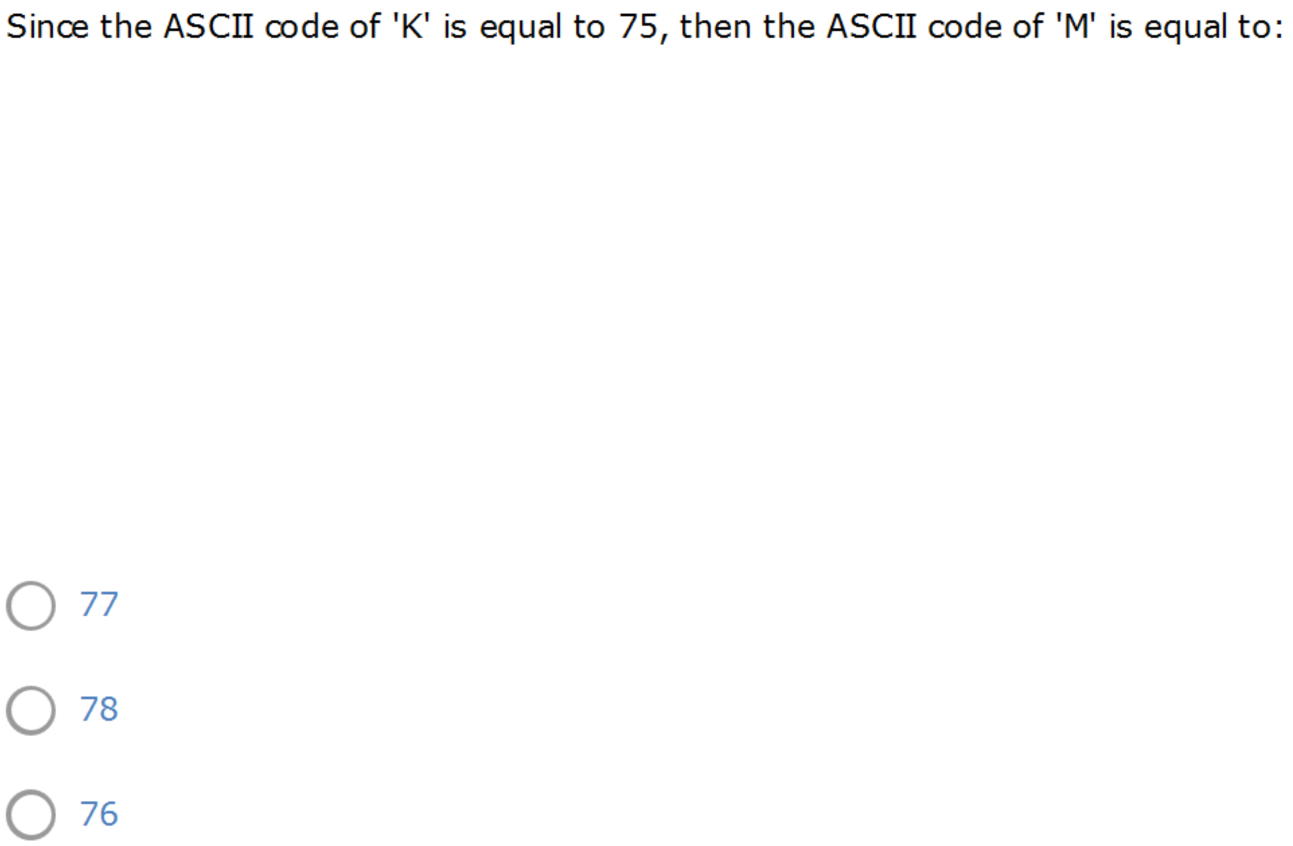
```
int i;
i = 2 * 2 / 2 + 2 * 2 - 1 / 2 % 2;
```

○ 6

○ 2

○ 4

Quiz



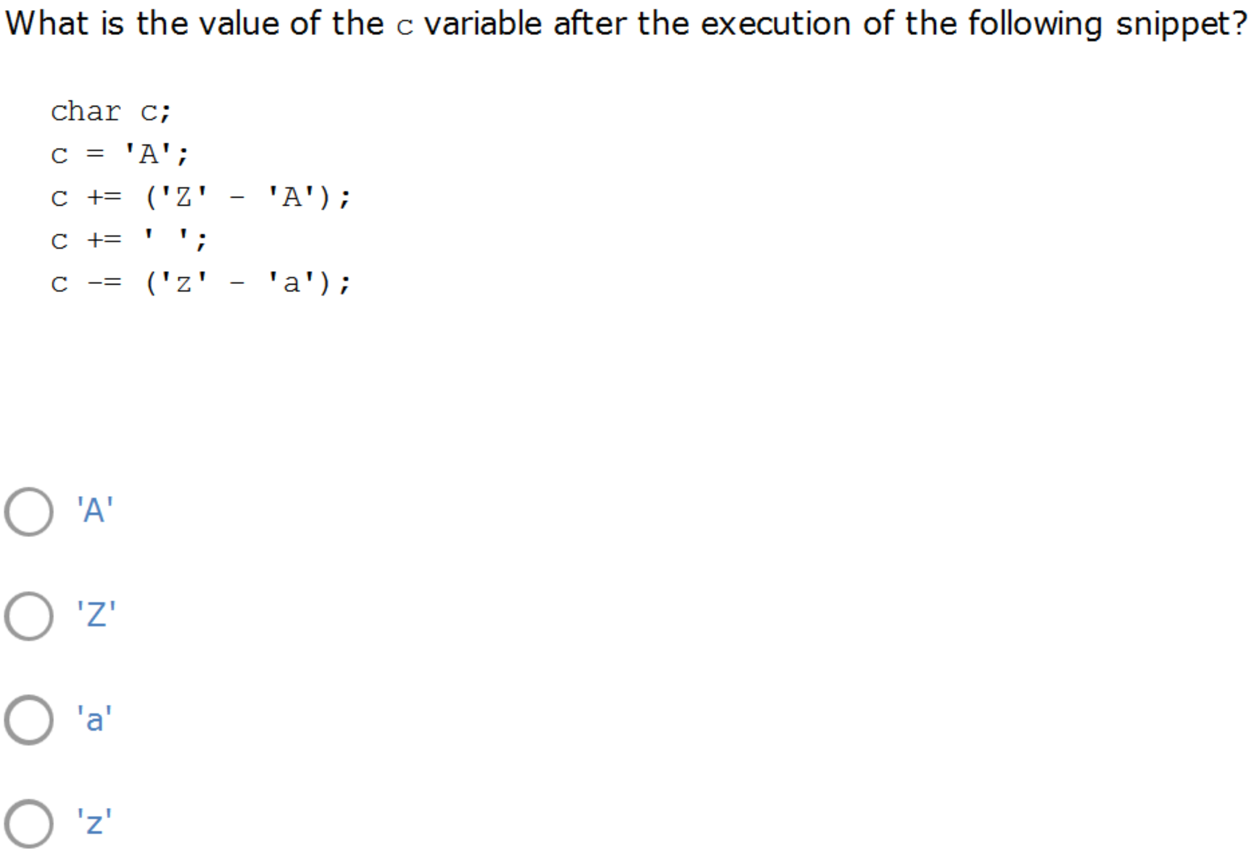
Since the ASCII code of 'K' is equal to 75, then the ASCII code of 'M' is equal to:

○ 77

○ 78

○ 76

Quiz

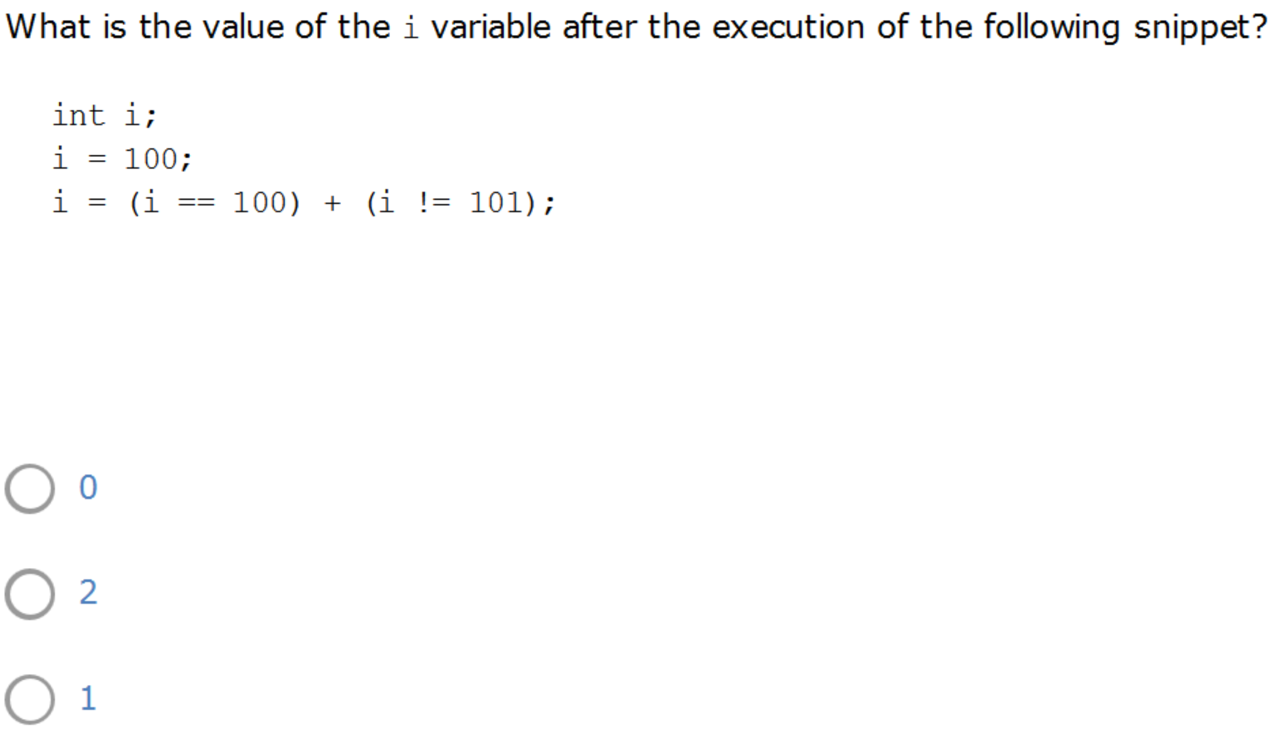


What is the value of the `c` variable after the execution of the following snippet?

```
char c;
c = 'A';
c += ('Z' - 'A');
c += ' ';
c -= ('z' - 'a');
```

- ○ 'A'
- ○ 'Z'
- ○ 'a'
- ○ 'z'

Quiz



What is the value of the `i` variable after the execution of the following snippet?

```
int i;
i = 100;
i = (i == 100) + (i != 101);
```

○ 0

○ 2

○ 1