Variables and Modifiers

During the second practice we covered what happens when we do different operations with variables of same of different type. We talked a little bit about operators' precedence and operators' associativity. We also covered type modifiers. So, let's review these things one by one and expand on them.

Variables

- **Integer division** – we concluded that when dividing values of type integer, the result of this division is also an integer. For example, if we divide 25 / 2, the result is **12**, since all values in this expression are integers, the result is also integer, but if we do 25.0 / 2, the result would be **12.5** since now there is an implicit conversion of 2 to 2.0 under the hood.
- **Adding and comparing values of type char** – We saw that values of type char are just saved as numbers in the memory. We also can find where these values come from – the ASCII table, which you can find on the internet. But what if you do not have the internet and you need two **char**s? This is also ok, because we can just use compares like **'a' < 'b'**, it may not make a lot of sense why would you make a comparison like this since everybody knows that **a** comes before **b**, but think of this example, you get a task, 2 symbols are given to the console and you have to say which one comes before the other, if they are from the English alphabet, or write the message "Invalid input" to the console if they are not from the alphabet. We can easily do that this way:

```
#include <iostream>

int main()
{
  char first, second;
  std::cout << "Please enter the two symbols: ";
  std::cin >> first >> second;
  std::cout << std::endl;

  bool firstIsLetter = first >= 'A' && first <= 'z';
  bool secondIsLetter = second >= 'A' && second <= 'z';

  if (!firstIsLetter || !secondIsLetter)
  {
    std::cout << "Invalid input." << std::endl;
    return 0;
  }

  if (first < second)
```

```
  {
    std::cout << "First comes \"" << first << "\" and then comes \"" << second << "\"." << std::endl;
  }
  else
  {
    std::cout << "First comes \"" << second << "\" and then comes \"" << first << "\"." << std::endl;
  }

  return 0;
}
```

From this example we can see that the capital letters come before the non-capital ones. This is important knowledge. If we don't have the ASCII table we can just subtract **'A' – 'a'** and see the result, if the value is negative **A** is before **a**, if it is positive **A** is after **a**.

- **Bool type possible values –** We saw that the **bool** type takes only values of 0 or 1 in memory. Having values of 0 and 1 makes it possible for us to add or subtract **bool** to and from **int**. Yes, we can do that, but being able to do something does not mean we should do it. Doing arithmetic with Boolean values is bad, we should not do it. On the other hand, we can use Boolean logic with the **bool** type. There are three operators that operate on the **bool** value.
    1. The negation operator – '!' – which reverses the value of a bool expression i.e. !true == false.

    2. The OR operator – '||' – this operator takes two arguments of type bool, if they are not bool the compilator tries to implicitly convert them as it does with **bool x = 153; // x == 1 here.** The return type of the OR operator is also a **bool**, this means we can combine values of type bool in order to create new ones. Example of the OR operator: This program tells us whether a number is a multiple of two or five.

```
#include <iostream>

int main()
{
  int number;
  std::cin >> number;

  if (number % 2 == 0 || number % 5 == 0)
  {
    std::cout << "The number is multiple of 2 or 5" << std::endl;
  }
  else
  {
    std::cout << "The number is not multiple of 2 or 5" << std::endl;
  }

  return 0;
}
```

The expression `number % 2 == 0 || number % 5 == 0` evaluates to true if either of the left or right side of the "**||**" operator evaluate to true. Possible values for **number** with which we will go to the "true" part of the if-statement are 2,4,5,6,8,10,12,14,15,…
You can check www.learncpp.com – Logical operators, the site is well done and truly helps.

3. On the other hand, we have the AND logical operator – **'&&'** which evaluates to true if BOTH of the expressions on its side are **true.**
   If we needed a check if a number is both a multiple of 2 AND 5 i.e. of 10, we could've used **&&** like this `number % 2 == 0 && number % 5 == 0`
   Possible values for **number** in which this will evaluate to true are 10, 20, 30, 40, …
   Of course, doing it like this is stupid we should just use `number % 10 == 0`**,** but for the sake of showing the AND operator we will let it slide. I highly suggest you take a look at www.learncpp.com about logical operators, they are really well explained and will deepen your understanding of the subject. In the end, in order to truly learn programming, you have to dip your hands in the code and try these things for yourself to see if you've truly learned them. There will be some code tasks at the bottom of this file, so try to do them and if you need help, ask somebody or google it.

- The last thing we talked about during the variables part was printing values of type double on the console, this is a subject for another time, so let's emphasize on the most important part – **if we need to print a double with precision of 3 numbers after the dot, we have to use both std::fixed and std::setprecision().** Also note that printing with precision is not the same as calculating with precision, we could have the number 1. 12345000.. , using it for operations with lead to right calculations regardless of **std::fixed** and **std::setprecision(),** which are used to manipulate the output stream(what we write to the console).

Modifiers

So, what are modifiers? When do we use them? Why do we use them? What are they modifying? How are they modifying it? So much questions we can answer here. During the practice we talked about different data types – int, char, bool, double, float and how much bytes of memory they take. After using **sizeof()** we came to the conclusion that the text below is relatively true (depending on the machine and the compilator we're using):

1. char – 1 byte
2. bool – 1 byte
3. int – 4 bytes
4. float -4 bytes
5. double 8 bytes

Now, we said that this is not always true, but it is a good starting point which we can use, so for now we accept it as the **Ultimate Truth** about the universe. Having said only so little about types and how much memory they occupy, we can easily introduce the first two modifiers – **short** and **long**. These type modifiers change how much memory the data types take. Let's go with an example first: If we know that the variable we are creating is not going to be a big number, let's say the variable is named **age** and will be representing the current age of a person. We know that the values for this variable are going to be in the range [0, 120] or something like that. Four bytes can represent the range $[-2^{31}, 2^{31}]$, we need something smaller, some people use a char to represent this value, but we

should not do that if we do not know exactly how and why we are doing it. The **short** modifier come to the rescue and let us create this beautiful variable:

short age = 0;

You may have noticed that I just typed short instead of short int, well the compilator know what we mean so for him **short int age = 0;** is the same as **short age = 0;** Just wanted to point that out, we will use the full versions for now, at least until we get accustomed to them. But what does this modifier do? Simply put, our new **age** variable does not take 4 bytes in memory, it only takes 2 now. So the range for this variable is $[-2^{15}, 2^{15}]$. On the other hand, we can contrast the **short** modifier with the **long** modifier, which does exactly the opposite action, it extends the memory a variable takes. So we can have the variable **long int peopleOnEarth = 42;** Now this variable is in the whooping range $[-2^{63}, 2^{63}]$. That is actually huge and takes it takes 8 bytes. But what if we want an even bigger number? The other two modifiers come to the rescue here, they are **signed and unsigned**. For know we can think that all the variables we used are signed, so when we created a variable **int x = 5;** what actually has been happening was **signed int x = 5;** Signed just means that the variable can have both negative and positive values, that is why the ranges are $[-2^{number\_of\_bits-1}, 2^{number\_of\_bits-1}]$, if we want even bigger numbers we can shift this range to right and make it go like this $[0, 2^{number\_of\_bits}]$ which in the *int* case would be $[0, 2^{32}]$. We can do this with the **unsigned** modifier – this modifier changes the variables in a way that they can no longer take negative values, but on the other hand they can take twice as many positive values. We should use it when we know that the value represented by a variable does not take negative values. For example **unsigned short int alphabetSize = 26;** We know that the English alphabet won't add or remove letter in the foreseeable future. So, we could do that. There is still one more modifier which we haven't talked about, but his idea is different, he does not change how much memory a variable takes or what values it can take. The **const** modifier makes a variable take a value and never ever change it again. We use the **const** modifier with variables that we know won't change their values through the execution of our program. For example the **alphabetSize** variable actually won't be changing it's value (I hope). So we could be even more precise with it's definition and do it like this:

```
const unsigned short int alphabetSize = 26;
```

Now here we've used 3 different modifiers, each one doing its thing not conflicting with the others – the **const** modifier makes it so we cannot assign a new value to the **alphabetSize** variable later in the program, the **unsigned** modifier makes the variable only take positive values start from 0 and the **short** modifier makes the variable take 2 bytes of memory instead of 4. We finish this by reminding you that the memory a variable take is dependent on the compiler and the machine, but fortunately the C++ standard gives us some information about these sizes. The C++ standard states a char takes at least 1 byte of memory, short at least as much as char or more, int at least as much as short of more, long int/long double as much as int/double or more. In other words, we have this:

```
sizeof(char) <= sizeof(short) <= sizeof(int) <= size(long) <= sizeof(long long).
```

Coding Tasks

- Write a program that takes a whole positive number in the range [0, 150] from the console – age of a person and outputs how much **days**, **hours**, **minutes** and **seconds** have passes since that person was born. Check if the number is valid. Use constants. What types will you use for these variables? What modifiers will they have?

We haven't really touched if-statements in the practice but below are some straightforward problems, which will get us going. If you can't do them, don't worry we will explain them once again when we meet, but still, try them, google stuff, get your hands dirty, if you want to get good, you got to do the work. Play around with the code, try to break it or modify it to your tastes. Also, the problems will become a little bit more ambiguous at each step, so you have to think about what to read or write to and from the console, what kind of data types you need, etc. This will make you think and use the material above and give you freedom in your expression, you can always google, ask a friend/assistant or your grandma for feedback and help. 🤗

- Write a program that takes 2 whole numbers as an input and prints to the console which is the bigger one. What if they are real numbers? (this modification is hard and probably will require google – but you need to know it in order to get 100% on your hard homework 1)

- Write a program that checks if a number is even or odd.

- Write a program that finds the maximum of 3 whole numbers.

- Write a program that checks if a number is divisible by 3 or 5.

- Write a program that checks if a number is divisible by 3 AND 5.

- Write a program that checks if a number is divisible by the first 4 prime numbers and outputs a message foreach prime number.

- Write a program that checks whether a number is negative, zero or positive. What if the number is real? Do that with precision 3 numbers after the dot. (Also something we haven't covered – ask for help if needed)