Universität Zürich[UZH]
Physik-Institut

Lecture 2: Introduction to basics of programming

1 Introduction to programming data types

In our previous class, we learned how to compile and write a basic `C++` code in a `Linux` environment. So far, you have set up a working platform that will serve as the foundation for the rest of this block course. What we haven't yet covered, however, are data types.
If you recall, your very last exercise and homework from last lecture was to use

```
int argc, char* argv[]
```

in your `main` code to input arguments at the execution time of your program. Obviously, this is very useful and essential to many programs. Note that the first,

```
int argc
```

stands for an integer with the *argument count* (`argc`), and the second,

```
char* argv[]
```

stands for a character *pointer* to the `array` of arguments. We will not talk about pointers today, but later. The important aspect for now is how to access and use these arguments.

```
// minmainargs.cxx
#include <iostream>
using namespace std;
int main(int argc, char *argv[]){
  cout << "Number of input arguments: " << argc << endl;
  for (int i = 0; i < argc; i++){
    cout << "argument count, argument: " << i << " , " << argv[i] << endl;
  }
  return 0;
}
```

The example above uses what is called a `for` loop, which will be discussed in detail later. For now, remember that all indices in `C/C++` start at zero, and that the first *argument* is always the program's own name. For example, if you compile and execute:

```
run_minimainargs.exe blah
```

the output will be

```
Number of input arguments: 2
argument count, argument: 0 , run_minimainargs.exe
argument count, argument: 1 , blah
```

This is not very user-friendly, because the first *argument* does not refer to what we type after the executable. We should therefore modify the code to:

```
// minmainargs_mod.cxx
#include <iostream>
using namespace std;
```

```cpp
int main(int argc, char *argv[]){
  cout << "Number of input arguments: " << argc - 1 << endl;
  for (int i = 1; i < argc; i++){
    cout << "argument count, argument: " << i << " , " << argv[i] << endl;
  }
  return 0;
}
```

If you compile and execute

```
run_minimainargs_mod.exe c++ is fun
```

the output will be

```
Number of input arguments: 3
argument count, argument: 1 , c++
argument count, argument: 2 , is
argument count, argument: 3 , fun
```

Today, we will discuss data types, `array`s, `vector`s, `while` loops, `for` loops, `if` statements as well as logical and relational operators.

1.1 Primitive data types

Types tell the computer what kind of data a variable can hold. `C` comes with a set of fundamental types built in, and `C++` adds even more through its libraries. We will start by exploring the basic types that come with `C`.

```cpp
void
```

`void` is an empty type, used for functions that do not return a value.

```cpp
int
```

`int` holds integers (e.g. $0, 1, 2, 3, ... \in \mathbb{Z}$). Integers types can be modified with keywords such as `signed`, `unsigned`, `short`, `long`, and `long long` to specify their range and representation.

```cpp
char
```

`char` represents a single character, stored internally as an integer according to a numerical encoding such as the *American Standard Code for Information Interchange* (ASCII). For example, the capital letter 'A' corresponds to the integer 65 in ASCII.

```cpp
float
```

`float` is a floating-point decimal type (e.g. $1.215, \pi, 54151.3141, ... \in \mathbb{R}$), with a precision of 6–7 digits.

```cpp
double
```

`double` is a floating point decimal type, twice the size of a `float`, providing a precision of 15–16 digits. There is also `long double`, which usually offers slighlty higher platform-dependent precision.

The standard `C++` library extends the set of primitive data types available in `C`, making programming easier and more versatile. In addition to the existing `C` primitive types, it provides:

```
bool
```

`bool` is the *Boolean* data type. It can hold only one of two values: `true` or `false`.

```
wchar_t
char16_t
char32_t
```

`wchar_t` is a *wide character* type with platform-dependent size, not tied to a specific encoding. `char16_t` and `char32_t` were introduced in `C++11` for *Unicode*, with fixed sizes of 16 and 32 bits respectively. They are intended for storing UTF-16 and UTF-32 code units, i.e. also characters beyond the basic ASCII range. However, we will not use those types in this course.

Moreover, the `C++` standard library provides additional useful *classes* that are not fundamental types though. We have already used:

```
string
```

A `string` represents a sequence of characters. In `C`, strings are typically implemented as `array`s of characters.

In `C++`, there are also *type qualifiers* like `const`, which modify how a type can be used. We will, however, not explicitly cover that in this course.

One of the greatest strengths of object-oriented programming is that you can create your own data types. For now, however, we will work only with the standard data types listed above. You can declare them as follows:

```
int one = 1;
char two = 't';
float three = 1234.56789;
void four();
wchar_t five = 21321;
string six = "six";
double seven = 1234.56789;
bool eight = true;
```

Also keep in mind that you do not have to specify the variables that you declare.

1.2 Arrays

When you have many objects of the same type — such as integers, characters, or other fundamental types — you can group them together into a larger collection. These are called *derived data types*. In standard `C`, one common derived type is the `array`. Below are three different ways to declare `array`s:

```
int one_to_five[5] = {1, 2, 3, 4, 5};
int not_specified[] = {5, 6, 7, 64, 365, 2, 2};
int two_dim_array[2][3];
```

You can modify the elements in an `array` after it has been declared. However, the size of the array is fixed and cannot be changed once declared. You can access an element of the `array` using the following syntax:

```
cout << one_to_five[3] << endl; // will return the fourth element, 4
cout << not_specified[2] << endl; // will return the third element, 7
cout << two_dim_array[1][2] << endl; // will return some random number since nothing has
↪  been declared here and no memory address has been assigned to this data.
```

You will encounter errors if you try to access elements outside the bounds of an `array`, for example:

```
cout << one_to_five[114] << endl; // will return some random number or give you a
↪  Segmentation fault (core dumped).
```

In the example above, encountering a

```
Segmentation fault (core dumped)
```

error can actually be helpful. If your calculation or analysis relies on a specific `array` element, a typo could silently produce incorrect results that might seem correct. Such issues are examples of undefined behaviour in programming. Developing good programming habits is the most reliable way to avoid these kinds of problems.

1.3 Vectors

In `C++`, a `vector` is a *container* provided by the *Standard Template Library* (STL). Containers are objects that store collections of other objects. To use `C++` `vector`s, you need to include the `vector` library in your code with:

```
#include <vector>
```

A `vector` is similar to an `array`, but it is declared differently and comes with many built-in *methods* (a *method* or *member function* is a function associated with a class or object). These methods allow you to increase or decrease the number of element dynamically. `vector`s also provide many safety features not present in `array`, such as bounds checking with the `.at()` method (instead of accessing an element with `[]`).

```
vector<int> v_one_to_five = {1, 2, 3, 4, 5};
cout << v_one_to_five[3] << endl; // correct: returns the fourth element, 4
// Warning: the following accesses are **undefined behavior**!
// operator[] does not perform bounds checking. The program may crash,
// return garbage values, or even appear to return 0 - its behaviour is undefined.
cout << v_one_to_five[114] << endl;
cout << v_one_to_five[5] << endl;
cout << v_one_to_five[-114] << endl;
// Safer access with bounds checking:
cout << v_one_to_five.at(3) << endl; // returns 4
// cout << v_one_to_five.at(114) << endl; // throws std::out_of_range
// cout << v_one_to_five.at(5) << endl; // throws std::out_of_range
// cout << v_one_to_five.at(-114) << endl; // throws std::out_of_range
```

Keep in mind that for handling large amounts of data efficiently, `array`s are the fastest option in `C` and `C++`. Due to their simplicity, `array`s are typically compatible with GPU programming (e.g., `OpenGL` or `NVIDIA CUDA`) directly.
The main advantage of `vector`s over `array`s is that their size can be changed dynamically after declaration, as illustrated below:

```
v_one_to_five.push_back(6);
```

Using the `push_back` method, a sixth element has now been added to the `vector` (to be accessed as `v_one_to_five[5]` or `v_one_to_five.at(5)`) and set to `6`. `vector`s also provide many other built-in functions to manipulate elements efficiently. For a complete reference, see the `C++` documentation at http://www.cplusplus.com/reference/vector/vector/.

2 If statements

The `if` statement evaluates an expression that yields either `true` or `false`. Based on this result, the program executes a specific block of code. Any of the following operators can be used to construct expressions that evaluate to `true` or `false`.

2.1 Logical operators

The *logical operators* in `C/C++` programming are the following:

```
&&      // Logical AND - evaluates to true  if both operands are true
||      // Logical OR  - evaluates to true  if at least one operand is true
!       // Logical NOT - evaluates to true  if the condition is false
```

2.2 Relational and comparison operators

These *relational operators* compare two values and return either `true` or `false`:

```
==      // equal to
!=      // not equal to
<       // less than
>       // greater than
<=      // less than or equal to
>=      // greater than or equal to
```

2.3 Example of an if statement

Note that an `if` statement can evaluate multiple conditions. By using the logical operators `&&` (*and*) and `||` (*or*), a single `if` statement can combine any number of conditions, provided you can keep track of them.

```
if ( (one_to_five[1] == v_one_to_five[1]) && (eight == true) ){
  cout << "match and true" << endl;
}
else {
  cout << "not match or not true" << endl;
}
if (eight != true){
  cout << "false" << endl;
}
else {
  cout << "true" << endl;
}
if (3 >= 2){
  cout << "sane" << endl;
}
else {
  cout << "insane" << endl;
}
```

In the examples above, each `if` statement is followed by an `else` statement. While an `else` is not required for every `if`, it is often good practice to include it to make your conditions clearer and easier to follow.

We will now start a 60-minute practical programming session.

3 Practical Session – Part I

Extending your functions package from the previous session

1. Add a `bool` function that returns `true` or `false` based on a `double` input. Anything greater than `0.5` should return `true`; otherwise, it should return `false`.
2. Add a `bool` function that returns `true` if the sum of two integers is greater than `50`.
3. Add a `string` function that takes 5 characters as input and returns them concatenated into a single `string`.
4. Ensure that the `vector` header is properly included so that `vector` can be used.

Practicing `array`s and `vector`s

1. Create a character `array` that stores your first name and prints it to the screen.
2. Create a `vector` of characters and perform the same task as above.
3. Create an empty `vector` of characters, use the `push_back` method to insert the characters of your name, and print it to the screen.
4. Create a 3×3 integer `array` and fill it as follows:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

5. Create a 3×3 integer `vector` and fill it as follows:

$$\begin{bmatrix} 3 & 2 & 3 \\ 1 & 5 & 2 \\ 7 & 9 & 9 \end{bmatrix}$$

6. Write an `int` function that returns how many elements in the `array` and the `vector` (from tasks 4 and 5) are equal.
7. Write a `void` function that performs the addition of two 2×2 `array`s.

We will now continue with a 15-minute theory session.

4 Iterative operations

Until now, working with `array`s and `vector`s has been tedious because you've been accessing each element individually. Fortunately, there are better ways to handle them. You can instruct your computer to process one element at a time. This process is called *looping through* an `array` or `vector`. The commands used to loop through all elements are called *loops*. In `C` and `C++`, there are two main types of loops:

- `for` loop
  A traditional `for` loop specifies which element to start with and which element to end at. In standard `C`, you must use an integer index to iterate through all elements of an `array` or `vector`. In `C++`, you can also use a range-based `for` loop to iterate through all elements of an `array` or `vector` without explicitly specifying an index.

- `while` loop
  A `while` loop continues as long as a given condition remains `true`. It only stops when the condition becomes `false`. `while` loops are particularly useful in hardware applications and interactive programs because they allow a device or program to keep running until a specific command or signal is received to stop.

4.1 For loop

A standard `C` `for` loop uses the following structure:

```
int one_to_five[5] = {1, 2, 3, 4, 5};
for (int i = 0; i < 5; i++){
  cout << "For index: " << i << " element of one_to_five is:" << one_to_five[i] << endl;
}
```

As you may have noticed, a `for` loop is typically declared in the following manner:

```
for (initial_condition; end_condition; action_at_the_end_of_each_iteration){ ... }
```

In the above example, this corresponds to:

```
int i = 0   // initial_condition   -> declares a new integer i, starting from 0
i < 5       // end_condition       -> keep iterating as long as i < 5
i++         // action_at_the_end_of_each_iteration   -> count up.
```

Obviously, you can count down or use different letters as your loop index. You can also create a `for` loop inside another `for` loop — called a nested `for` loop — to handle `array`s or `vector`s with more than one dimension. An example of a two-dimensional `for` loop is shown below:

```
int two_by_two[2][2];
two_by_two[0][0] = 1;
two_by_two[0][1] = 2;
two_by_two[1][0] = 3;
two_by_two[1][1] = 4;
for (int i = 0; i < 2 ; i++){
  for(int j = 0; j < 2 ; j++){
    cout << "The position (x, y) : (" << i << ", " << j << ") holds the value: " <<
    ↪  two_by_two[i][j] << endl;
  }
}
```

In general, `for` loops can be nested infinitely, as long as the programmer can keep track of each loop.

If you simply want to iterate over all elements of a container (such as an `array`, a `vector`, or others) without needing their indices explicitly, it is convenient to use a *range-based* `C++` `for` loop, which has the following syntax:

```
int one_to_five[5] = {1, 2, 3, 4, 5};
for (int entry : one_to_five){
  cout << "The present entry of one_to_five is: " << entry << endl;
}
```

For more information on `for` loops, visit the `C++` reference website https://en.cppreference.com/w/cpp/language/for.

4.2 While loop

In general, `while` loops are simpler to use because they require only a single condition. However, it is also easy to create an *infinite loop*, meaning you instruct the computer to repeat an action indefinitely without a stopping condition. An example of this can be seen below:

```
while (true){
  cout << "Keep going until interrupted by Ctrl + C from the command line" << endl;
}
```

The above is an infinite loop that will continue running until you manually stop the program. In some cases, if the program consumes too many resources, it could cause your computer to crash, requiring a hard reset (e.g., using the power button). For now, if you run the code from the `bash` command line, there are built-in safety features to help you stop such programs. One common method is pressing `Ctrl+C` (the *control* (`Ctrl`) key together with the `C` key), which sends an interrupt signal to the terminal and terminates the running program.

You can also use a `while` loop in a similar way as a `for` loop, as in the following example:

```
int i = 0;
while (i < 10){
  cout << "Counting index: " << i << endl;
  i++;
}
```

The above is equivalent to the following `for` loop:

```
for (int i = 0; i < 10 ; i++){
  cout << "Counting index: " << i << endl;
}
```

Another way to stop a `while` loop is by using an *escape condition*. We will cover escape conditions in more detail in the next lecture, but here is one example you can examine for now:

```
int i = 0;
while (true){
  i++;
  if (i == 10){
    cout << "break condition fulfilled: i == " << i << endl;
    break;
  }
}
```

The `if` statement that `break`s the `for` loop has the escape condition `(i == 10)`. For more information, visit https://en.cppreference.com/w/cpp/language/while. Let's put these new tools into practice.

We will now start a 60-minute practical programming session.

5 Practical Session – Part II

Using `for` loops and `while` loops on `array`s and `vector`s

1. Write a program that uses `cin` and a `for` loop to create a character `array` storing your first name. Then, output the name on the screen using a `for` loop.
2. Write a program that uses `cin` and a single `for` loop to input 9 elements into a 3×3 integer `array`.
3. Write a program that uses `cin` and two nested `for` loops to input 9 elements into a 3×3 integer `array`.
4. Repeat tasks 2 and 3, but this time using `vector`s instead of `array`s.
5. Re-do all of the above exercises using `while` loops instead of `for` loops.
6. Write a program that allows you to build a 4×4 matrix and returns the following:
   (a) the sum of all elements in the matrix.
   (b) the sum of the absolute values of all elements in the matrix.
   (c) the determinant of the matrix.
   (d) which elements (if any) occur more than once, along with their frequency.
7. Write a program (using either `for` or `while` loops) that allows you to create two 3×3 matrices and then
   (a) perform matrix addition.
   (b) perform matrix multiplication.
   (c) calculate the determinant of each matrix.

We will now conclude with a 15-minute question session.

6 Conclusion

Today, we covered data types and containers as well as the basic use of `if` statements, `for` loops, and `while` loops. These are fundamental concepts in all programming languages and applicable to virtually any program you write.
The most important takeaway is that you are now equipped to work with data. In the next lecture, we will put these tools into practice. If you have not completed all of today's exercises, I strongly recommend finishing them at home. Many resources are available online. Feel free to use them, but avoid direct copy-and-paste. Instead, type out the examples yourself so you fully understand each step.