**2025-02-20-cpp1-en**

0:03  
So hello everyone, I trust we are all here for the Introduction to Computer Systems course.

0:12  
So before we get started with the course, let me just give you a few information about how this course will be conducted and for what will you receive the credit and how the exam will be organised.

0:26  
So first of all, the most important rule of all is that on this web page, there are all the information out on this web page.

0:34  
So please go there, read it.

0:36  
There are grading rules, rules, how you submit your solutions to record X rules, how you receive your credits.

0:42  
These materials you will see here the slides, materials for your labs, everything is there.

0:47  
So please visit the site and I will actually I will go there right away so that you can take a glimpse of this web page and I'll show you the most important part.

0:58  
So first of all, yeah, you can get them just a second so you can get the slides right away.

1:10  
And most importantly, you can find the grading rules there.

1:14  
So let me start with the grading rules.

1:17  
In this course, the credit for the labs and the exam are completely separate.

1:21  
And as you will see in in future, the lectures are somewhat detached from the labs.

1:26  
So this course is somewhat awkward in this way.

1:30  
Unfortunately, that's how it works, so we cannot do anything about it.

1:34  
It's like a political decision.

1:36  
So for the labs, you need a credit and the credit is granted for two things.

1:44  
You need to do home assignments and afterwards, at the end of the semester, you need to pass the lab test.

1:50  
So that's how you get the credit.

1:52  
There are 6 assignments.

1:54  
Basically, you get one assignment on each lab session, and you need to finish it within a week if you want the points and within four weeks if you want to pass the course.

2:04  
So we were the deadlines.

2:06  
The deadlines are in record X.

2:07  
You really need to meet them, and you need to gain at least 13 points from these assignments.

2:13  
So there's a point limit you need to pass.

2:16  
If you do the all the assignments, then you can go to the test at the end.

2:20  
The test will be conducted in the labs.

2:22  
So basically you will be locked down in for two hours in labs and you you just code something on Arduino.

2:30  
That's basically it.

2:31  
One thing you should know, you can use the code, you code during the semester, during the labs.

2:37  
So if you prepare yourself well, if you do good work on your home assignments, you can take the code from home assignments and reuse it during the test.

2:47  
That's by the way, a very good idea to do so.

2:50  
This is the labs.

2:52  
Again, please beware the deadlines and beware how the submission process works.

2:56  
I just give you a very quick, quick hint.

2:59  
You can get more details from your lab teachers, but just to give you the preview, I believe some of you probably already had the labs right this year.

3:08  
It's somewhat awkward that we have labs before the lecture in the week, so it's kind of strange.

3:14  
But for the submission, there is a like a very simple process.

3:18  
You can see it fit fit only one page.

3:21  
Basically you need to submit something into record X that passes the record X test 400%.

3:27  
We don't take glimpse at your solutions that are not meeting the test requirements.

3:32  
So you really need to pass the test in record X.

3:36  
For each solution you will need the acceptance flag.

3:39  
So the teacher needs to recognise that you have submitted on time the solution and you will receive this acceptance flags.

3:47  
If you don't have this acceptance flag then something's wrong and you should ask why you don't have the acceptance flag.

3:52  
So you need 6 acceptance flags from each solution one.

3:56  
And there are two deadlines for each assignment.

4:00  
So basically there is one deadline one week after the assignment is issued, and then there's another deadline three weeks afterwards.

4:07  
So four weeks after the assignment is issued, the first deadline you need to meet if you want the points.

4:15  
So if you want to receive any points from the assignment at all, you will have to meet the first deadline.

4:19  
So you have to submit something that passes record excess 100% before the 1st deadline.

4:26  
Afterwards, after the first deadline, your solutions will be reviewed so that your lab teachers will take a look at them and manually review them.

4:34  
If they're OK from the perspective of code quality and you will get to that in the labs, then you get the point, you get additional points for code quality.

4:43  
If not, you don't get the quality codes, but you can fix the assignments later to get the quality code points.

4:50  
But you need you need to do that before the 2nd deadline.

4:54  
And also if you miss the first deadline, you are no longer eligible for points for this particular assignment.

4:59  
But you can still get the acceptance like if you meet the second deadline.

5:03  
So basically each solution needs to be done at least within four weeks after it's issued.

5:09  
If you miss these deadlines, then you are dead.

5:12  
From the perspective of the course that that's like a hard rule, they are not that difficult.

5:16  
So you can meet these requirements easily and we issue these rules because the assignments are incremental.

5:23  
You need to finish one assignment to start another.

5:25  
Well, the first one is somewhat just get you started, but from the second on you will need to do them gradually.

5:31  
So it's a good idea to force you into making them gradually over the semester.

5:38  
So that's basically it.

5:40  
Let me switch to the exam portion.

5:44  
The details about the about tests you can read on the page, but basically the test is just that you called something in the labs that that's it.

5:53  
Regarding the exam, exam is completely separate.

5:55  
So you can achieve the credit, You can get the credit and go to the exam completely separately.

6:00  
You don't need to have the credit before you attend the exam.

6:03  
It's not not mandatory and the exam is basically a also a test.

6:09  
It's incidentally also in the labs, but you are, you are not going to code in the labs.

6:13  
It's a test like you just select from multi church questions, etcetera.

6:17  
And you receive some points that you can get up to 20 points.

6:21  
The grading table is over there.

6:23  
So you can, you can see right away how the points are distributed.

6:27  
You need at least 11 points out of 20 to pass the test.

6:30  
So that's the that's the bottom line.

6:32  
And basically you have regular 3 attempts.

6:35  
It's it's like a normal, normal exam.

6:37  
There is there are no surprises there.

6:39  
OK, so this is how this course will work.

6:42  
Are there immediately any questions regarding the grading process?

6:48  
No so far.

6:49  
OK, sure.

6:51  
I have repo that's open right now.

6:53  
I'm not seeing where it would be that I would find the whether or not I got the like the check mark You can you can see it at the at the submissions.

7:05  
If you see the submission, if I expect you haven't submitted anything yet, you already have.

7:13  
Yeah.

7:13  
So the teachers will give you this acceptance check mark after they grade your, after they read your assignment.

7:20  
So they basically check it was submitted on time.

7:24  
And usually they perform a review.

7:27  
So basically when they do the review, they will also give you this assignment, this acceptance mark.

7:32  
OK, don't panic because the first reviews are due after the first deadline.

7:40  
So if you just submitted something that passes the test 100% you, you're OK for the first deadline.

7:45  
OK, that that that's true.

7:48  
Perfect.

7:49  
Any more questions before we go on?

7:53  
So this is the grading process.

7:55  
Let me switch back to the slides.

7:57  
If you cannot attend the lectures or whatever, or you want to study on your own, there are some books listed over here that you can take a glimpse at.

8:06  
Basically, yeah, they are like more broader, they they cover more broader topics than what we are going to cover in cover here.

8:13  
But they are also a good point to start if you want to study further.

8:17  
And perhaps the most important communication rule for us is that we are using mattermost.

8:22  
It's a somewhat like a Discord or Slack, but it's our own private platform.

8:26  
So we don't have, we don't have to worry about GDPR.

8:29  
So basically you need to attend.

8:31  
Sorry, you need to attend.

8:35  
I really need more, more powerful pointer.

8:38  
So you need to attend this address.

8:42  
Actually the invitations link were sent to you via e-mail.

8:46  
I just hope you got the right right links because we kind of screw up at the beginning.

8:51  
So first you got the wrong links and in the second iteration we we send you the right links for this year.

8:58  
If you cannot find them, they are also insists.

9:01  
So this the the invitation link for matter most is attached in the note in the sys schedule labels.

9:07  
If you just go to the sys to the schedule, you can see there are some notes attached to the to this lecture.

9:15  
So basically go there and if you have any questions whatsoever, go check into this channel and ask questions there.

9:22  
Beware this, this channel is shared for Czech students and English students.

9:27  
So don't be unnerved that some people might post questions in Czech there.

9:34  
It's fine.

9:35  
Don't worry.

9:35  
Just pose your questions in English.

9:37  
That's OK.

9:38  
So just go there and ask anything you like.

9:41  
If you have any specific like a questions towards me and you don't want to share them with your classmates, then just write me a personal message, that's also fine.

9:50  
So this is how the course is conducted.

9:54  
The lectures are regularly weekly so we will meet here every Thursday.

9:59  
The labs are biweekly, so every every fortnight.

10:03  
So we have only six or seven labs during the semester.

10:09  
For the labs, there's one important piece of information I need to give you right away.

10:13  
For the labs, you will require this.

10:15  
It's earned Arduino device with attached shield.

10:19  
You can either buy one if you'd like, if you're a real fan, but assumingly you don't want to buy it, so you just can borrow it in the library.

10:28  
Only the library at Mahlastrana have these.

10:30  
So you really need to go there and ask for our do you know they will know if you just give them the code of the, the, the course, they will, they will know everything you need to know.

10:39  
You need to get so you get this box and you need to bring it to your labs because you will, you will code for this device.

10:49  
And again, don't forget the deadlines.

10:51  
Many students are confused for whatever reasons, so just go to record X and beware the deadlines because that's important.

10:58  
Deadlines are strict.

11:02  
OK just a few words about the outline of this of this course.

11:08  
This course is somewhat special in the in the sense that it's like an introduction to many other courses.

11:14  
We are trying to build upon what you already already learned in last semester from Mr.

11:20  
Ajax.

11:21  
But so some some parts of this course will be just a repetition to and we try to elaborate on that.

11:28  
But we also try to introduce a few topics to other other lectures to other courses.

11:35  
First of all, we start with C++.

11:37  
Actually, this is not intended to be a language course.

11:40  
So we just take a 22 lectures about C++ to get you started so that you can code Arduino and so that you can understand the C++ examples from other parts of this course.

11:52  
That's basically it.

11:53  
But for the C++ language itself, there are two additional courses in the second year, the Basic C++ and Advanced C++.

12:00  
So for the language itself, we just defer many things to those two courses.

12:05  
After that there are some information we need to elaborate on on CPU architectures, how instructions work, etcetera.

12:13  
And hand to hand with that we talk about how memory works in in hardware.

12:19  
After that we will switch to programming languages.

12:21  
So basically we'll give give you like a brief introduction into compilers and how run runtime works and how applications are executed and the related technologies.

12:33  
Then we cover about four lectures with operating systems.

12:37  
Again, operating systems have their own course which is I think due in 3rd year.

12:44  
So we will touch only the basics.

12:46  
And finally we will spend 1 lecture on parallel programming because parallel programming is quite important nowadays, especially considering we have mostly parallel hardware.

12:56  
So we will give you the basics on that.

12:59  
So this is how this course will will go again.

13:03  
Please note that the course like the lectures and the labs are completely separate.

13:08  
We also give you some special dispensations like that.

13:12  
If you happen to like pass the exam and you fail in the labs, then we will just you don't have to retake the exam next year.

13:21  
We just rewrite the the your marks from this year to the next year.

13:26  
If you if you pass the test, if you pass the exam completely, and vice versa, if you get the labs, if you pass the credit and you don't pass, don't make the, if you don't attend or you don't pass the exam, then we just acknowledge your credit next year and you don't have to retake the labs, OK?

13:43  
Because these two are basically two different courses somewhat glued together.

13:50  
OK.

13:51  
Are there any questions regarding this course before we get started with C?

13:58  
OK, let's dive in.

14:00  
Let's start with C Oh, sorry.

14:07  
So C++ language is a language that's not that different from Python in many ways.

14:16  
So I try to relate to Python because I believe everybody, everybody here already took at least one semester in Python.

14:22  
So that's like our common ground.

14:25  
And like Python, this language is procedural, imperative structured.

14:29  
So it's not that different.

14:30  
You have the same structures like if for loops, etcetera.

14:35  
You can create functions, you can decompose your code in the same way.

14:40  
However, there are some important differences.

14:42  
The most important difference is that the C is statically typed.

14:47  
I will elaborate on that in a short while, but just to give you the first clue.

14:52  
Now, static typing means that every variable, every parameter, every function argument, every return value has to have explicit type.

15:00  
So when you're declaring variables or parameters, you are always attaching the type definition to that variable and you cannot change it in the future.

15:10  
The type is strict and as long as the variable lives, it has this only one type.

15:18  
And the second thing that goes hand to hand with the static typing is that the compiler helps us here and compiler performs various checks and verifications on these types.

15:27  
For instance, if you're performing an assignment, then the compiler will verify that the left side and the right side are compatible at the type level.

15:36  
That's very important for this type of languages, for statically typed languages, because they can eliminate a lot of programmer errors during the compile time.

15:46  
That's very helpful because if you eliminate errors at compile time, you don't get them in runtime, right?

15:51  
So it's very, very useful.

15:55  
Why are we using CC for this type?

15:57  
Of course, basically because C&C actually I'm not going to make many distinctions between C&C in this course is that they are very, this language is very close to hardware.

16:09  
It's very close to the instructions.

16:10  
You can, you can easily map a regular language constructs into the instructions that will be executed on the CPU.

16:19  
You can directly map the structures, data structures, how they will, how they will be laid out in memory.

16:26  
And that's also very important.

16:28  
So C, it's very important in the tasks where you need this level of control.

16:33  
For instance, if you're programming the core, the kernel of an operating system, you need the full control over your hardware.

16:40  
If you're programming high performance applications and you need to fine tune your optimizations, then again, you want to be in control to know which instructions will be issued to the CPU.

16:50  
Or if you're working with embedded systems, usually embedded systems also require this.

16:54  
If you're programming, I don't know, washing machines or something, then you really need to know how to code on this level.

17:02  
The price that we pay for that is explicit memory management.

17:06  
So everything you do with memory you do explicitly, you need to be in control in where and variable where a data structure becomes, becomes alive when it, when it's allocated and when it's disposed, when you you need to dispose it explicitly.

17:21  
So that's the level of control you have, but also the responsibility you need to take in this kind of language.

17:28  
And like many languages, this language is case sensitive.

17:31  
There, there is no surprise there.

17:32  
But another important difference from Python is that it ignores all white space.

17:38  
In Python you already know we have something which is called a significant white space.

17:42  
So white space matters.

17:44  
How you indent your code matters because this is the way how you create blocks, like if blocks or function blocks in C++, you don't do that.

17:53  
This way we have other constructs and the white space is completely ignored.

17:57  
However, by the way, if you already learned how to indent your code in Python, do it in C as well.

18:04  
The notion that you need to indent your code in Python isn't a bad notion.

18:09  
It's like a good idea to indent your code because it's much more readable.

18:13  
You just don't have to do it the exact way in C, but keep doing it if you already learned that in Python.

18:22  
So I told you that the most important difference between Python And C is the typing system.

18:28  
So let me elaborate on that for a while.

18:29  
OK, first of all, in Python you can rewrite the value of the variable, not only the value, but even the type of a variable.

18:41  
So if you have have a variable that is assigned an integer and sometime later it's assigned a string, that's fine with Python.

18:48  
Well, it may may not be fine with your code design.

18:50  
There are very few situations in in your life when you really want to do that.

18:55  
Because if you already have a variable and you assume that there is an integer in this variable, it might not be a good idea to just suddenly assign a string into it, right?

19:05  
But you can do it from the Python Python's perspective.

19:08  
In C you cannot.

19:12  
So if you tell the compiler that there will be an integer in this variable, there should be an integer.

19:17  
And if you try to put there anything else, it will blow up in your face at compile time.

19:23  
So the compiler will say, hey, I'm not going to compile this code.

19:27  
This code sucks because you're trying to assign a string into an integer, and I'm not happy with that.

19:32  
So no, no can do.

19:34  
And the compiler will not produce the binary, so you cannot execute this code.

19:40  
So basically if you just take a step back, one perspective how you should look at these variables is that in Python or similar languages, in dynamically typed languages that the same goes for PHP or JavaScript for instance, is that variables are just mnemonic holders, are just labels, names for the programmer that are associated with values that live in some magical heap somewhere in the memory and are automatically managed and can be anything.

20:08  
So you can relabel, you can attach a label the same label, one to the string, 1 to the string, another time to an integer, a third time to a list.

20:18  
It doesn't matter.

20:20  
In C and statically typed languages it goes also for C or Java for instance.

20:26  
The variables also allocate space.

20:29  
So the variable isn't just a label, it's label of a memory space.

20:33  
That's why we need the type, because we need to know how many bytes we need to allocate for this variable.

20:38  
If it is an integer on a 32 bit architecture, it will be probably 4 bytes.

20:43  
So this NUM variable over here will actually tell the compiler, hey you should somewhere in the memory take 4 bytes and this NUM label refers to these 4 bytes specifically forever.

20:56  
As long as this variable lives, these 4 bytes in the memory will be associated with this variable.

21:03  
So you cannot possibly put there anything else because this string will definitely won't fit for four bytes, right?

21:11  
So you cannot easily assign it there.

21:13  
Yeah, there are some situations like if you want to insert assign 1 character into this into this variable.

21:19  
It might work because a character is small enough to fit 4 bytes.

21:23  
But still, the compiler will at least warn you that you are doing something very naughty and you shouldn't be doing that.

21:30  
So basically, remember in C++ variables are directly attached to place in memory where the value of the variable lives.

21:40  
A few words about the history of the language.

21:42  
So basically C is a very old language, you know, I believe nobody here was alive back when it was this started.

21:52  
I wasn't, I assume you weren't as well.

21:56  
So it started many decades ago in Bell Labs at the actually it started in Cambridge and then it was continued in Bell Labs.

22:03  
And they just wanted to create a very low level language because back then most of the software was written in assembly language and that that was horrible.

22:10  
So they need to something very close to assembly language, so something very close to instructions.

22:15  
So they still can code things like operating systems, etcetera, but they want something more pleasant to the to the programmer.

22:22  
So they tried to create first they created the BCPL language.

22:26  
They weren't very happy about it.

22:28  
So they created the B language shorten of the BCPL and they were still very happy about it.

22:34  
So they created the C language.

22:36  
Yeah, they weren't very imaginative about the language names, right.

22:41  
The C was quite popular for some time, and as you can see, it was revised.

22:46  
The specification of the language was revised several times, and now I think the last revision is from the year 18, so it's not that old from the perspective of, let's say, this diagram.

22:57  
On the other hand, the C is not very widely used language nowadays, but it still has its place.

23:03  
For instance, as I mentioned before, when you're writing a kernel of an operating system, the kernel of of Linux is still written in C, for instance.

23:13  
However, for regular use, for normal situations, for application programming, C isn't probably your first choice.

23:22  
Actually, it's probably not your second.

23:23  
And nowadays it's probably not your last choice either, because writing something in C is a pain in the \*\*\* if you really want to do something fast and you really want to do something fancy.

23:35  
So there were several attempts to introduce modern concepts, especially the object, object oriented programming concepts into C One of these was the Objective C It was for some time propelled by Apple, but it ended quite soon, quite quite some time ago, I wanted to say.

23:55  
So there was this is language called Objective C Probably you heard about it, but I think it's it's completely dead nowadays.

24:02  
Also, no sorry.

24:05  
There was the C with classes, but it was just a middle step to something which is now called C++.

24:11  
Actually the C++ back then and the C++ of today are completely two different C plus pluses.

24:17  
The the language took a huge evolution over this time.

24:21  
So basically this C++ has nothing to do with modern C++.

24:25  
C++ 23 for instance.

24:28  
However the evolution took place, there was a huge gap right in the between 90s and like 2011.

24:36  
This was like dark ages for C++ because many, many people back then thought that C++ will die because new languages like C# and Java sprung right alongside with C++ and Java and C# are for a long time believe that they will take over the application programming.

24:55  
However, C++ kind of resurrected itself around 2011 when the C++ 11 norm specification came out and this specification was very different.

25:07  
This was like one of the turning points of this language and since that every three years there is a new specification of C++.

25:14  
So yeah, sorry, this diagram misses C++ 23, which is the currently the most recent specification.

25:21  
And each of these specifications brought something new to the language.

25:25  
So it's a very complex set of set of changes.

25:28  
I'm not going into the details, just that, you know, if you want to code in modern C++, you really don't need to.

25:34  
You really don't need to bother yourself with the old versions.

25:36  
That's something which is all already completely outdated and thanks to these new features which were introduced since C++ 11, the C++ is still competitive in many domains to C# or Java.

25:53  
Let me start with a brief introduction of the syntax.

25:56  
Actually, if you can just read this, you probably understand what this code does, right?

26:01  
Because you already know Python, so this is not, this shouldn't be that surprising.

26:06  
The first line, the include line is somewhat like import.

26:09  
You already know import in Python.

26:11  
So we need to mechanism that will import libraries or import additional dependencies.

26:16  
This is somewhat different.

26:17  
It's just like a copies header inside.

26:19  
But for all instances and purposes of this class it's like import.

26:23  
So we we are importing a module C to the IO which gives us the function for printing out strings to the standard output.

26:31  
And the function we are using here is the print F which is similar to print in in Python.

26:36  
It's just a formatted print.

26:37  
But the difference is not that important here.

26:39  
So I believe you you can read what this function, what this line does, you can notice one important difference.

26:47  
There is a column at the end, semi column at the end.

26:52  
That's one of the important differences.

26:54  
We need to place; After every statement in in C++, Yeah, just yesterday I had some talk with my friends.

27:02  
Why are we now using semicolons?

27:04  
It's it's obvious where the line ends.

27:07  
So if we have just an end of the line like Python, why we need to place semicolons there?

27:12  
Yeah, it's a very, very old custom, but you need to realize that several decades ago the compilation time was one of the critical issues.

27:21  
And placing semicolons after every statement helps the compiler to recognize where actually each statement ends.

27:30  
So it was done mostly for the purposes of faster compilation.

27:34  
Yeah, today it's it's like obsolete because we can do it fast enough without the semicolons, but it's still in the language specification.

27:44  
And perhaps the most important difference here is that you can see this is function declaration.

27:49  
You probably guessed that because this looks like a function.

27:52  
Yeah, it it likes the the prefix depth which you are used to from Python.

27:56  
This int is a return type.

27:58  
The function is declared starting with the return type, but then it's the function name and arguments which you probably could could have guessed.

28:07  
There is no top level code.

28:09  
There is there are no top level statements.

28:11  
Every code in C++ has to be placed in a function or a method.

28:15  
Imperative code of course.

28:17  
The reason for that is that C++ is like old school and you just need to decompose your code in.

28:24  
In functions there is one function special and then this is that one which is named main and the main function is the entry point of the application.

28:33  
So everybody knows that the main function is like the top level statement in Python.

28:37  
This is the function that will be called the 1st when you when you just execute your application, this is the function that will be called.

28:45  
OK.

28:46  
These are arguments where the command line parameters are passed to the application.

28:50  
So if you add any command line parameters like additional arguments on the, they are parsed and put into these into these variables.

28:59  
I'm not going into the details right now.

29:01  
And this is the return code.

29:02  
Here is the exit code of the application.

29:05  
You probably know or you will learn in Linux courses that every application, every proceed, every process, when it ends, it yields an exit code.

29:15  
So this is the exit code we are yielding 0 usually stands for.

29:19  
It's OK.

29:23  
So this is the basic structure of the language.

29:27  
Let me go through a few details because there are some nuances that you really need to know.

29:34  
First of all, let's start with numbers.

29:36  
Yeah, the similar numbers are.

29:37  
There are no surprises there.

29:39  
You know what the similar numbers are in C You can also write numbers in hexa format, which is prefixed with 0X, and then you can use all 16 hexa characters.

29:51  
It doesn't matter whether you're using uppercase or lowercase, it's fine both ways.

29:56  
And the same way you can use the binary prefix 0B and then you use only zeros and ones to specify your literal.

30:05  
There is one catch.

30:06  
There is also octal format, so you can specify octal numbers in eight base math and they are prefixed with zero.

30:18  
I don't know who actually come up with this, but I really want to meet that guy and tell him a thing or two because this is like a obstruction.

30:28  
If you if you ever happen to have a like an array of of these literals and you just smuggle in one octal literal inside, you just created a perfectly unmaintainable code because no one will ever spot that this particular number is octal.

30:44  
It looks almost as it's as it's that similar, right?

30:48  
The 0 prefix isn't something that stands out to.

30:52  
So please, good idea is not to use this octal format at all.

30:57  
Actually, you usually don't use it because if you're doing something very fancy, you either use hexa, or if you're doing something very low level, you might prefer to revert to binary if you're not using this SIM, of course.

31:10  
So octal is not very, very useful here.

31:12  
Yeah, floating point numbers, they are the same as in Python, no surprise there.

31:16  
Boolean constants are also all the same, they are lower case.

31:20  
Just know that you need to start them with true and false.

31:23  
Yeah, I know this is pain in the \*\*\* because when I'm switching from C like languages to Python And vice versa, I always make this mistake because you need to like, remember in which environment you are and write uppercase T or a lowercase T if you're writing true.

31:38  
Yeah, I know that it's a pain in the \*\*\* but I blame Python people for that.

31:43  
OK, string and characters.

31:45  
There is a big difference because in Python you can write strings both in quotes and in apostrophes, right?

31:52  
In single or double quotes in C&C&C there is a difference.

31:57  
Apostrophe single quote stands for a character.

32:03  
So here could be only a single letter.

32:07  
Actually there must be a single letter.

32:09  
You cannot write an empty string this way.

32:12  
You need to write exactly 1 letter inside because this is a character.

32:16  
The double quotes, normal quotes stand for string, so here can be arbitrary number of letters or even 0 letters for empty string.

32:24  
So that's kind of a pitfall Python people usually fall into in the characters and strings.

32:30  
You can use some escape sequences to place in special characters.

32:33  
For instance, if you want to place a double quote into the string, you need to prefix it with a back slash so it's inserted as a double quote, not as the end of the string.

32:44  
OK, regarding the integer types, those were literals.

32:51  
Literals like numbers you just drop in your code.

32:54  
Integer types are somewhat more perplexing in C++.

32:59  
First of all, let me start with the most basic type, int integer.

33:03  
Unfortunately integer doesn't have fixed size, meaning the size is inferred from the platform on which your code runs.

33:13  
So if you take the same code and compile it for regular X86 laptop like you you probably all have, then it will have probably 32 bits.

33:23  
If you take it and compile it for Arduino, it will have 16 bits.

33:28  
And it's possible that from some other platforms it might have a different number of bits or bytes.

33:32  
So you really don't know how large the integer is.

33:36  
The dorm says that integer should be the type which suits the computation engine you're running on the best.

33:43  
So usually it's about 32 bits, but you cannot rely on that.

33:49  
On the other hand, character is always one byte wide.

33:51  
1 character is always one byte.

33:53  
That's pretty easy.

33:55  
And these base types like care or integer can be modified with modifiers.

34:02  
These are modifiers that change the width.

34:05  
So you can have a short int, long int and long long int, which suggests the compiler that you need to narrow down or widen your number.

34:16  
But then again, the compiler will just do what they do on that specific platform, so you have no guarantees that the like the long is larger than int.

34:25  
On many platforms long and int are the same.

34:29  
Usually long long is longer than in.

34:32  
So on X86 for instance long long X64 bits long and in is 32 bits long and other modifiers are signed and unsigned.

34:42  
Sign is like a default.

34:43  
You don't have to write signed, but if you want a number that doesn't go to the negative values, you probably want to use unsigned because then you are using values from zero to the full length of the of the number.

34:56  
Just remember how these numbers are defined.

35:01  
If you have like a 32 bit number for instance, then sign integer goes from -2 to the 31 right to 02/2 to the 31 -, 1 right?

35:17  
Did I do that correctly?

35:19  
While if you Yeah, while the unsigned goes from zero to 2 to the power of 32 -, 1, all right.

35:32  
So actually you have a wider range, the upper limit if you use unsigned.

35:37  
If you don't need the lower numbers, sorry, no, you need to, you need to split the range in half.

36:01  
So instead of this may be -1 or so is it -1 No, it's not.

36:11  
So if you need both negative and positive numbers and you have 32 bit range, then you need to like take a half of it, which is 2 to the power of 3132 -, 1.

36:24  
It's half of the range that goes for the negative numbers.

36:29  
And then you need to squeeze 0 somewhere.

36:31  
So the zero is squeezed to the positive numbers.

36:35  
Some, some person decide that zero is like a positive number, so 0 towards the positive numbers.

36:41  
But there, there will be only 31 because we don't have the range.

36:45  
We spent half of the range already for the for the negative numbers.

36:51  
No, no, no problem.

36:52  
If if anything isn't clear, ask away.

36:55  
That's why we're here.

36:57  
So basically you you can switch between the sign and unsigned arithmetics using these prefixes.

37:03  
There are also some accelerated types like size T Size T is a type which is wide enough to place any number for counting.

37:11  
So basically if you're using containers or try to index arrays, size T is your is your best choice.

37:18  
Floating numbers have two formats.

37:20  
Float is usually 32 bit wide and double is 64 bit wide.

37:24  
That that's something you probably already know.

37:26  
And there are some other types.

37:28  
Yeah, bool has no surprise you.

37:29  
You know bool from from Python, but there is one special type called void.

37:34  
Void is something like somewhat like none in in Python, but not not exactly.

37:40  
Void is a placeholder.

37:42  
We we put when there is no type to specify.

37:45  
For instance, if you have a function and that function doesn't have any return type, doesn't return anything.

37:50  
It's like a procedure.

37:52  
It doesn't have any anything to output.

37:54  
Then you declare it with void, which says OK, this This is like a no type.

37:59  
Nothing is returned.

38:03  
Compiler performs for you some implicit conversions.

38:06  
For instance, if you're trying to write to take a long integer, and you're writing into the integer, that's basically a conversion.

38:15  
Because these two types are not directly compatible, they are not exactly the same or they can differ in some way.

38:24  
So basically compiler will perform all these cooperations for you if they are possible.

38:30  
And whether they are possible, there is one big rule.

38:32  
There should be no data loss.

38:34  
If there is no data loss, then there is no problem.

38:37  
If you have a larger integer and you want to write in a small integer, that's fine because small integer will fit into the larger integer.

38:45  
Vice versa, it won't work if there is a possible data loss, but the conversion itself is possible.

38:52  
It will do the conversion, but it will issue a compilation warning.

38:56  
So the compiler will tell you, hey, I've noticed that you're doing this and I'm not completely certain whether this is OK.

39:03  
I will do it for you, but beware, you are you might have a mistake here.

39:08  
It's possible that this is a mistake, so double check your code.

39:11  
Normally warnings are bad.

39:14  
You shouldn't produce code that generates warnings on compilation.

39:17  
And I think from the third assignment we will enforce no warnings in in the record X.

39:21  
So just get used to the warnings and eliminate them from your code if you if you can.

39:27  
For instance, if you try to if you take a double and you try to assign it into the float, that's something which is possible because they are both floating point numbers, but there may be a data loss.

39:38  
So the compiler will allow that assignment, but will issue a warning that you shouldn't probably do this.

39:45  
At least you shouldn't do this lightly.

39:47  
You should do some checks before and you should do this explicitly to tell the compiler that this operation is fine.

39:56  
Yeah, there are some rules for this.

39:57  
For this conversions, there's something called conversion rank which computes what's the base type, etcetera.

40:04  
This, this is used in in a complex expression, but I'm going to skip that for for this time.

40:10  
OK, Sometimes these conversions needs to be done explicitly.

40:14  
You need to tell the compiler that that some operation needs to be done or that's OK to perform some of these conversions.

40:21  
For that we have casts.

40:23  
So you can cast one type into another, another type.

40:26  
It's like magic casting is easy.

40:29  
You just specify the new type in the parenthesis before the expression that needs to be casted.

40:35  
So if you have expression, or in this case just a simple variable that's also an expression, then you can convert it to another type just prefixing with the new type.

40:45  
Yeah, So we have a double and here the double is forcefully casted into a float.

40:53  
The compiler won't object to that because it's fine to convert double to float, it's technically possible.

41:00  
And also by specifying this conversion, compiler thinks, OK, the programmer really knows what he's doing or she's doing.

41:08  
So I will do that without objections.

41:12  
On the other hand, use that wisely, because it's not.

41:18  
It's not OK just to fix your warnings or errors by placing random arbitrary casts.

41:24  
You really need to like, check or notice, or sometimes even check using like if statements, whether this conversion is possible or whether it's safe within your code.

41:35  
By the way, this is how C casts work.

41:38  
In C++ the situation gets much more complicated.

41:41  
This is just a look ahead.

41:42  
I'm not going into the details right now, but just telling you that there are three types of casts in C and the things got much more mangled together.

41:51  
So I'm not going into the details regarding all the C versions and we won't need them.

41:56  
This works in the C fine as well, so use just this type of conversion and we'll be OK.

42:06  
So we talked about expressions, types and variables and now it's time to talk about variables.

42:15  
As I mentioned, variable is something that labels a place in the memory.

42:20  
Variable must be always declared using a type, and then there's the name of the variable.

42:26  
By the way, usually the variable is not initialized.

42:30  
Some variables are like global variables, but local variables are not initialized automatically.

42:35  
So it's usually a good way a good idea to initialize the variable right away because then you that there is no mistake.

42:43  
If you do this, then there will be 0 in the X variable from the from the very beginning and no one can ever make a mistake about using some arbitrary numbers for instance.

42:54  
By the way, there is also a special keyword called auto.

42:58  
You can use auto to declare variable and then the type of the variable.

43:04  
Here will be inferred, will be guessed, will be deduced from the type of the expression which is being assigned to the variable.

43:12  
OK, so sometimes it's easier, especially if you have some complex types.

43:17  
If you're working with classes or templated classes and you have some long types to write, then you can just put 4 letters there, auto, and the type will be inferred automatically.

43:27  
On the other hand, handle this with extra care because it's not a good idea to place auto just for all your variables.

43:34  
This is not Python.

43:35  
You really need to be in control what the value, what the type of the variable is.

43:41  
And sometimes, for instance here, it's fine because this is clearly double.

43:45  
Float literals are specified slightly differently, but if you just place a zero in the variable, then it's difficult to guess if I have something like this.

43:55  
If I have just a auto X = 0, what's the type of X?

44:10  
Why, Why, why?

44:13  
Why isn't it it's zero?

44:13  
Why?

44:14  
Why isn't it unsigned integer?

44:16  
Why?

44:16  
Why isn't it long?

44:18  
Why isn't it long long?

44:21  
Yeah, you're right.

44:22  
You're right.

44:22  
By the way, it will be integer, but it's difficult to guess sometimes.

44:27  
And if you just this is 0, this is fine, but you can just place a larger number here, and based on the larger number, it might try to use a larger type sometimes.

44:37  
So it's always better if I specify explicitly.

44:40  
Yeah, that's just the normal unassigned integer, and everybody's clear about that.

44:45  
By the way, writing int is shorter than writing auto, so there's no save there.

44:52  
OK.

44:53  
There is something which is called a variable scope.

44:56  
You already know that from Python on a limited basis because in Python you have local variables and global variables, right?

45:01  
You have variables within functions which are local, and then you have a global variables.

45:06  
The same thing goes in here.

45:07  
So we have local variables, but they don't exist only within a function.

45:11  
They actually exist within a block.

45:13  
So any block, anything that's placed in the curly braces, usually it's one function, but it can be just a statement, like a body of an if statement.

45:23  
So the local variable lives only within this block.

45:26  
And of course the nested blocks of this block, it goes this way.

45:31  
But once you leave the block, the local variable will be de allocated and it's no longer accessible.

45:38  
Global variables are the very same like in Python.

45:41  
But in Python you need to use the explicit global word if you want to access global variable within a function, right?

45:50  
Everybody remembers that in C you don't.

45:54  
That's why C is much more error prone when you're using global variables, because sometimes you just need to think whether the variable you are accessing right now is local or global.

46:05  
There is no distinction.

46:06  
There is no global keyword, so usually accessing global variables directly from a function is a bad idea.

46:13  
We will get to that throughout the course.

46:15  
Actually, you will get that in the labs throughout the course.

46:18  
I'm not going into that, but basically global variables are bad in many situations.

46:24  
There is also something called static local variables.

46:27  
These variables are similar to global, meaning that they live throughout the existence of the program.

46:32  
This actually they start to live when you first call a function because they are inside a function.

46:39  
But they live till the end of the application.

46:41  
They are not erased when the function call is returned.

46:46  
So the difference is that the static variable is encapsulated, is wrapped in the function, cannot be accessed from outside, which makes it much more, much less error prone, because you wouldn't accidentally or incidentally access this counter from outside.

47:03  
If you have a global variable, that's very tricky, because everybody can access the global variable.

47:07  
Someone can just accidentally or by mistake erase it, rewrite it, update it, whatever.

47:18  
Let me go quickly through the statements.

47:20  
I'm not going into too much details because I believe you know all the statements.

47:24  
So I'm just making the comparative analysis toward the Python so that just pointing out the pitfalls so you easily slip into this into the coding.

47:34  
First of all, I only mentioned this several times, so let me do this properly.

47:39  
This is a block, every statement block which needs to be compound together like a function body, like a body of a for loop, like a body of a if branch needs to be wrapped in this curly braces.

47:52  
As as I mentioned at the beginning, it's a good idea to indent the code inside this curly braces.

47:58  
So don't forget your good manners from Python.

48:01  
The thing that C doesn't enforce indentation doesn't enforce specific white space formatting doesn't mean that you shouldn't do that.

48:10  
It's actually a good idea to do that, so the compiler will recognize these, but you, as a normal people still do the indentation.

48:20  
It's a good idea.

48:21  
Second thing, semicolons after each statement.

48:24  
I already mentioned that if statements and similar statements are different in a small way.

48:29  
The expression that's being tested in the branch is wrapped in the parenthesis and there is no column at the end.

48:37  
So that's how we know where the expression ends.

48:40  
It's it's wrapped in the parenthesis, but otherwise the if else statement is very similar to Python And return is also the same as in Python.

48:48  
There are no surprises there.

48:52  
One thing that's slightly different or not slightly not that slightly, is a switch case.

48:59  
If you have many branches, like many if elses, I mean like many if else if else if, then it's probably a better way to wrap it in something different.

49:10  
The switch case allows you to test one expression towards several possible outcomes.

49:18  
So this may be a variable or a computation that yields a number, an integer, and here are possible values that will be matched against this expression and each case will perform will have a block.

49:32  
Actually this is not a block, this is a sequence of of statements which needs to end with a break.

49:38  
The break will end the switch case and yields the control at the end of the whole switch case block.

49:44  
So basically this is like a huge if else case, a huge compound of if if else's and compiler can optimize it more slightly better than in just a simple switch case code or if else code.

50:01  
Also sometimes this might be slightly more readable because you can aggregate these cases together.

50:07  
If you don't have the break, then this case will fall out through this case.

50:10  
So case two and three will have a common, common body.

50:14  
On the other hand, one thing that I'm, let's say, emphasizing a lot is that every time you're writing a switch case, you probably want to rethink your life.

50:25  
Oh, sorry, your code because branching, heavy branching is usually bad.

50:34  
Many people, usually beginners, tend to write this if elses, like yeah, and I'm just now I'm meeting a new H case of my code which I haven't thought of before.

50:46  
So I'm just putting another if else there so we can take care of another special case.

50:52  
That's not the way it works.

50:54  
That's not the way it's supposed to work.

50:56  
It's a good idea to keep branching to minimum.

50:58  
Especially, it's a bad idea to have nested branching.

51:01  
If you have like more than two levels of nested branching, that's usually very bad.

51:06  
So if you're doing something like this, rethink your ways.

51:10  
Rethink your design.

51:12  
You probably want to do something else.

51:13  
You probably want to go about the code decomposition, about the coding another way.

51:20  
I personally find like 2 cases.

51:24  
One switch case was a good idea in C and I've been doing this for 25 years, so please take my word for it.

51:34  
Loops are quite easy.

51:36  
Yeah, they are similar to to Python.

51:39  
The while statement you already know right?

51:41  
There is an inverted statement called called do while.

51:46  
If you don't remember these two, there is a good sorry there is a good illustration 8 that will explain the difference between these two.

51:54  
So this one will perform the testing expression first.

51:58  
So it will first ask whether it's OK to execute the body.

52:02  
And if it's OK, then it executes the loop of the body, the body of the loop.

52:07  
The other one will first shoot away and perform the statement, perform the body, and then it will ask whether it was OK or not.

52:15  
Actually, it asks whether it's OK or not to continue with with that.

52:19  
So yeah, there are cases when you really want at least one iteration of this loop to to be to take place.

52:25  
So in this in that case do while might be better, but in 99% of situations the while loop is is the loop of choice.

52:35  
There is another loop called for loop and that is quite different to Python.

52:39  
In Python, the for loop is more like for each.

52:43  
You're providing a list or an iterator or a generator or something that will the for loop iterate over.

52:51  
This is usually more more suitable for the programmers because you usually want to use for loops to iterate over and container.

52:58  
You have a list and you want to visit every item in that list.

53:02  
So you basically write a for each that visits every item in the list, and the loop is invoked for every item in the list that it's perfectly clear to everybody, it's easy to use and everything.

53:15  
Unfortunately, C was designed much before this concept, much further in the past before this concept.

53:20  
So basically they go to about the four loops in another way.

53:24  
4 loop has three expressions and these expressions are performed at the beginning as a test before each loop, as at the end of each loop.

53:36  
What does that mean?

53:37  
Let me give you an example of this.

53:39  
It makes it much more clear.

53:41  
The first statement, the first expression is initialization.

53:45  
Usually you want to declare a variable which governs the loop, which is like a control variable of the loop.

53:54  
It's like if you're iterating over a list, this would be a variable that each list item is assigned in Python.

54:01  
But here we don't have that.

54:02  
So we use numbers and this is like a number that counts which iteration we are in.

54:10  
The second expression is the condition.

54:14  
This is the same as a condition in the wire loop.

54:16  
So this is tested before every loop starts.

54:20  
So we are iterating unless the I exceeds N As long as I is lesser than N we are proceeding with the iteration and the last.

54:33  
The third expression, this one is performed at the end of each loop.

54:40  
So basically this is usually an update of the governing of the main of the controlling variable.

54:46  
So in this case, we are incrementing it.

54:50  
This loop will go from zero to what would be the last value of I during the last iteration and -1 Exactly.

55:03  
Yeah, because once AI reaches N then there will be no more loops.

55:09  
By the way, the I is declared here, so it will live only within the loop.

55:17  
If you try to access I after the loop, it won't exist.

55:21  
So you can rewrite your code if you need to, but it's not a good idea.

55:25  
Just to give you an example, is there any reason why plus plus I is I plus plus?

55:31  
Yeah, there is a reason for that.

55:32  
And I will get to that when I'm talk, when I'll talk about pre and post increments.

55:37  
So yeah, there there is a small tiny reason for regular use.

55:40  
It doesn't matter.

55:41  
Yeah, but but there is one small tiny difference.

55:45  
I will mention it later.

55:48  
Sorry.

55:49  
Yeah, I want to say that you can place this integer outside the loop.

55:53  
You can, you can place it before the loop.

55:55  
Technically it's possible.

55:56  
Usually it's not a good idea to do so, but then the eye will exist after the loop.

56:01  
What would be the value of I once the loop concludes?

56:05  
When the control resumes after the loop, what would be the value of I if we declare it beforehand?

56:15  
We already established that the value of I will be n -, 1 in the last iteration, Last time this body.

56:24  
Sorry, we have some trouble with shades.

56:41  
Yeah, embrace the darkness.

56:45  
So this loop, this body will be last executed with n -, 1.

56:54  
What will be the value of I after the last loop?

57:00  
And exactly because it will be incremented at the end, so it will reach the value N then this test will take place determining that it's time to end, but the value N will remain in the variable.

57:12  
I.

57:13  
But remember this was just an example.

57:18  
I just wonder whether I will be seen on the camera when I just when I put the shades down maybe?

57:37  
I'll figure it out later.

57:38  
So that's about it.

57:41  
About the basic statements, there is one statement I haven't mentioned yet and that's go to, go to is a very sorry go to is a very special statement because it somewhat disrupts the nature of workflow of the other statements.

57:56  
So basically we can use it to just take up and go somewhere else immediately.

58:02  
In this case, if this condition holds, the go to will completely jump out of all the loops and go over here where the label is.

58:14  
This has a very few good use cases like this one if you have a heavily nested code.

58:20  
If you have many for loops inside and you want to suddenly jump out all of them, then there's no other way how to do this except for go to, because if you use constructions like break, it breaks only one loop, not all of them.

58:34  
On the other hand, if you're using go to, beware that go to is usually.

58:40  
In many cases it's a bad practice, so try to avoid go to.

58:43  
This is just that you know that go to is there.

58:46  
By the way, go to is also usually frowned upon by experienced programmers because usually you can express yourself better without go to.

58:54  
By the way, there are many parodies on go to like if you know intercolor language, it uses come from like inversion of go to.

59:02  
So imagine that go to is somewhat disconcerning regarding the readability of the code.

59:08  
Then come from is even more disconcerting because if you use come from, the code is even more unreadable.

59:16  
It's it's it's even more mess.

59:25  
Few words about operators.

59:29  
I think that wasn't that good idea.

59:31  
Let me try no.

1:00:01  
I think this would be a good compromise.

1:00:12  
The Sun was going from this side I believe.

1:00:15  
OK, the arithmetic operators are quite the same as in Python.

1:00:27  
Just know this is modular.

1:00:28  
I'm not sure if what's the code for modulo in Python right away.

1:00:33  
So this is modulo arithmetics.

1:00:35  
Just beware single slash is division, but double slashes are used to initiate A1 line command.

1:00:43  
So this is sometimes a little little confusing.

1:00:47  
Yeah.

1:00:47  
And these two plus plus and minus, minus are called called a pre and post increment or decrement.

1:00:54  
And there was a question What's the difference between pre and post increment or decrement?

1:01:00  
So let me give you an example.

1:01:03  
On the code previously shown, I have a plus plus I.

1:01:08  
Semantically this is equivalent to I = I + 1.

1:01:15  
Or you can write it as I + = 1 both in Python, right?

1:01:22  
That's easy.

1:01:24  
I plus plus can be also written as plus plus I can be also written as I plus plus.

1:01:32  
The difference is that this increment takes place before anything, and this increment takes place after anything.

1:01:40  
If you just use these as a statement alone statements, there is no difference semantically.

1:01:46  
But if you use these in a large expression, then the difference is that here the I is incremented first and then the incremented value of I is used in a subsequent expression.

1:02:00  
Let me demonstrate.

1:02:01  
Let's just say I use I plus plus something like y = I + + + X for instance.

1:02:10  
And here I write y = I + + + X looks almost the same.

1:02:18  
But here the I will be incremented first and then it will be added to X.

1:02:24  
Here the value of I will be taken, put aside, add it with X and afterwards the I will be incremented.

1:02:36  
That's basically the difference.

1:02:38  
Piece of advice, especially for beginners, don't write code when you need to make this distinction.

1:02:45  
So basically, if you really, really don't need to, and beginners don't need to, believe me, just use these as a separate statement and then they work the same.

1:02:54  
You don't have to make any distinctions.

1:02:56  
If you really need that, you need to remember this rule I just told you, and sometimes it's a bit bit disconcerting.

1:03:04  
So please make sure that you're happy with your code.

1:03:07  
And if you're not certain, just use these as a single statement, as a single expressions and it will work fine.

1:03:15  
Another thing are comparators.

1:03:16  
Yeah, comparators are almost the same as in Python.

1:03:20  
Beware that the comparison of equality is 2 = a single equal is used for assignments.

1:03:28  
That's like how they are distinguished.

1:03:31  
The exclamation mark is used for negation on many on many places.

1:03:39  
For instance, it can be used as a logical negation as a standalone unary operator here.

1:03:46  
When combined with equality mark, then it's it's like a negation of equality test.

1:03:52  
So this is like not equal.

1:03:53  
This is equal.

1:03:54  
This is not equal.

1:03:56  
What's perhaps the most important for us in C++ is bitwise operations.

1:04:02  
We will use them quite a lot.

1:04:03  
When you get to quite a lot, we will use them in the labs.

1:04:07  
And these bitwise operations stand for logical operations performs per each bit.

1:04:15  
For instance, the & stands for and.

1:04:22  
So if you have something like X&Y, it means that it takes these two numbers and end them bit per bit.

1:04:33  
So if you have something like 01/01 etcetera, 1100 that it will really take each bit separately and only because this is end, only the bits where both values are one will be set to one or other bits will be set to 0, right?

1:04:53  
Everybody remembers basic logical operations from the first semester.

1:04:57  
Is that so?

1:05:00  
OK, at least a few people are nodding, so I hope so.

1:05:03  
The same goes for OR.

1:05:04  
This is OR operation.

1:05:05  
The pipe is OR operation.

1:05:07  
These are somewhat more perplexing.

1:05:10  
This is bitwise negation.

1:05:12  
So this is like a flipping of all the bits, zeros to ones and ones to zeros.

1:05:18  
And this is exclusive OR or.

1:05:22  
The last two are bit shifts.

1:05:25  
Sometimes you need to move the bits in your word to the left or to the right.

1:05:30  
So for instance, if I take something like X left shift, this is like really like an arrow.

1:05:38  
Yeah, this is lesser than, lesser than, but it looks like an arrow.

1:05:40  
So it's called left shift, let's say 2.

1:05:48  
It will perform this operation that the let's just imagine that there is a 0100 something something 11101 this is the X value OK after shifting it towards left, it will all the bits will move so that there will be two zeros on the right and then I will start copying this value.

1:06:12  
So one will go here, 0 will go here, one will go here, etcetera, etcetera.

1:06:17  
And at the end, these two zeros will be the last values on the left.

1:06:23  
And these two values, these two bits will be thrown away because there is no more room in the in the in the integer for them.

1:06:30  
OK, so this is a left shift.

1:06:32  
Right shift works analogically on the other way, the other way to the right.

1:06:39  
These are actually quite important, because what is if if I take a normal integer like X and I shift it by two to the left, What arithmetical operation?

1:06:52  
This is equivalent with multiplying by 4 exactly, because shifting by one bit is multiplying by two.

1:07:04  
Shifting by two bits is multiplying by 4, two to the power of 2?

1:07:14  
By the way, there are other logical operators, two ampersands and two pipes and.

1:07:20  
The difference is these are performing these miracles on the bitwise scale on each bit in an integer.

1:07:27  
These are logical operators.

1:07:30  
They take two booleans and perform whatever logical operation they are doing.

1:07:34  
So they take true and false for instance, and the result of true and false will be in this case false.

1:07:40  
So these are to be used in a conditions.

1:07:43  
If you're writing an if statement and you need to combine 2 conditions to 1 using and or or using basic logic, then you use these.

1:07:52  
This is a negation.

1:07:53  
OK.

1:07:54  
If you're doing something low level with your bits in your, in your integers, in your words, then use these.

1:08:01  
OK, That's the distinction.

1:08:04  
OK, Pointers.

1:08:05  
Pointers are somewhat difficult.

1:08:07  
We have two operators for pointers.

1:08:12  
Yeah, it's confusing.

1:08:13  
This is an ampersand, which you already seen here and here.

1:08:18  
Yeah, at least there are two of them here.

1:08:19  
So this can be really a parsed out.

1:08:21  
You can really differentiate this this one out.

1:08:24  
But this and this are the same operators in the terms of syntax, in the terms of literals, in the terms of of tokens.

1:08:34  
So basically, if you write an ampersand, the compiler needs to distinguish whether this is this & or this one.

1:08:42  
Fortunately for us, this one is a binary operator.

1:08:45  
It needs 2 operands.

1:08:47  
This one is a unary operator.

1:08:49  
It takes only one operand on the right side.

1:08:52  
So basically this is the distinction that the compiler needs to make.

1:08:56  
And this & is to take an address of a variable.

1:09:01  
So it converts a variable name into the address in the memory.

1:09:05  
We will use it later.

1:09:06  
Actually we will use it next next week because we won't get to pointers this this session.

1:09:13  
And\* is also somewhat overloaded operator because we have\* here as a multiplication operator, right?

1:09:23  
But fortunately for us, again, this is binary operator, this is unary operator.

1:09:28  
Asterisks are used to declare and to dereference pointers.

1:09:33  
So these two I will explain in more detail later when we get to references and pointers.

1:09:39  
Yeah, just to note that normal assignments doesn't have to be explained, but like in Python, we have shorthand assignments like plus, equal, minus, equal, etcetera, which combines together the explicit binary operation and assignment.

1:09:53  
So as I mentioned before, if you write something like I + = 1, it's the same as if you write I = I + 1.

1:10:01  
It's just to safe, safe typing, nothing else.

1:10:05  
One important operator that you will read right away in the labs is the size of operator size of It's very practical because it gives you at compile time number of bytes.

1:10:18  
A variable or a type or a data structure takes in the memory.

1:10:23  
So if you write something like size of int, it will tell you what is the size of integer variables on your particle architecture.

1:10:39  
So you are not completely lost in the river without a pedal.

1:10:43  
Regarding the sizes of the types you have at least something you have the size of.

1:10:47  
That will tell you what is the actual size on the architecture which is currently being used there.

1:10:54  
For the compilation of the int variable, usually it would be something like 4.

1:11:00  
On Arduino it would be two.

1:11:01  
For instance you can instead of type, you can write here a variable.

1:11:07  
So if I have a variable like I, I can just write here I and that's it.

1:11:12  
It will give me the size of this variable based on what type this variable is.

1:11:19  
And finally, last expression I want to show you is the ternary expression.

1:11:26  
Actually this is in Python as well.

1:11:28  
I'm not sure if you meant it in C it's much more compact than in Python.

1:11:33  
This is like a somewhat like a if branch, if else branch.

1:11:38  
This is the testing condition for the branching.

1:11:41  
This is the result which will be taken if the test is true.

1:11:46  
If the result of this expression is true and this will be taken.

1:11:50  
This will be yielded if the result is false.

1:11:53  
So basically you can use it to shorthand simple if else cases if you don't have to write any code or any complex code.

1:12:02  
In the if else case you just need to yield 1 value if something is true, another value if something is false.

1:12:09  
Then you can use this shorthand which is much more compact, much much more pleasant for in many situations if you just dealing with a simple simple expressions.

1:12:19  
By the way, does anyone know how to write this in Python?

1:12:25  
You meant that in Python the ternary expression.

1:12:30  
Basically, you write some something, then if The thing is true, then you write else and exactly.

1:12:41  
So you write E1 if then there is the test else.

1:12:51  
Sorry, I took a wrong piece of blackboard else E2.

1:12:56  
Yeah, that's very, very confusing.

1:12:59  
So see like which is have it slightly better and much, much shorter because you don't have to write the if and the else.

1:13:08  
OK, that's it about the expressions.

1:13:11  
And now we are finally meeting our most important nemesis of today and that are arrays.

1:13:17  
I've intentionally planned the arrays on that point because we will end with the arrays and next week I will just remind you about the arrays so we can pick up on them easily.

1:13:29  
Arrays in C have some small distinctions from lists in Python.

1:13:35  
The most important difference is an array have one specific type of items.

1:13:41  
So basically you declare a base type, the type of every element, and then you cannot deviate from this type.

1:13:49  
So if you have a list of integers, all the items array of integers, all the items are integers.

1:13:56  
In Python, if you have a list, you can put integers, floats, strings, whatever you'd like, or even nested lists as you please.

1:14:02  
You don't have to declare that beforehand.

1:14:05  
In C++, you need to say this is an array of integers and nothing else but integers can pass in This is an array.

1:14:12  
Floats.

1:14:12  
Then only floats can can be stored in.

1:14:17  
The arrays are declared using the brackets and usually you write the number inside the brackets.

1:14:23  
This way we declare something which is called statically allocated array.

1:14:29  
So we have a fixed number of items that will be stored that can be stored in this array.

1:14:35  
It's not dynamic like enlist in Python that you can just simply add more items there.

1:14:41  
It will always have just 4 items in this case.

1:14:45  
Yes, we can do this dynamically.

1:14:46  
We can do dynamic allocation, but that's much more tricky in C and we won't do it in this course.

1:14:52  
We're just deferring it to the second year.

1:14:54  
Yeah, It's not because it's like we don't want to give you that opportunity.

1:15:00  
It's just because it's too difficult to perform in C because you need to allocate and reallocate the memory manually if you're doing something like that.

1:15:07  
And we have no time for that in this course.

1:15:11  
So another way how to initialize statically allocated array is by initializer you can specify the numbers or the values that will be immediately stored into the array.

1:15:23  
And in here the size of the array.

1:15:25  
You can see that I have omitted the number that specifies the size is inferred from the initializing structure.

1:15:32  
So the compiler will deduce that since there are three integers over here, then the array should have three slots.

1:15:39  
The array should have the size of three integers.

1:15:43  
By the way, you can write the three in here if you'd like, but usually it's not a good idea if you're initializing it.

1:15:49  
It's like a redundant information.

1:15:51  
If you want to later change that, it would be more tedious because you need to change two things, the initializer list and then this number.

1:15:58  
So it's easier just to hope that the compiler will do its job and compute the right number of integers in the initializer.

1:16:07  
And we can trust compiler that at least it can do that.

1:16:11  
The arrays can be nested.

1:16:13  
This is like A2 dimensional array.

1:16:16  
Actually it's linearized in the memory.

1:16:18  
I will get to this in shorthand.

1:16:20  
This is like an initialization of two arrays of three array of three item.

1:16:26  
Sorry, two arrays of three integer arrays.

1:16:31  
Yeah, I get a little stuck in the terminology.

1:16:35  
So basically the first dimension first the dimension is size of two and the second dimension is size of three.

1:16:42  
And this array is linearized in the memory so the values are still stored one after another.

1:16:48  
This is like just a shorthand for indexing all.

1:16:52  
An array is always a compact block in the memory.

1:16:55  
That's one of one of the important things of arrays you need to learn in C, because if you have 4 integers, they will be in the memory write 1 after another.

1:17:05  
You can if you just have a pointer to the beginning of the array.

1:17:08  
You can easily compute any offset of any item position of any item just from from the offset.

1:17:16  
That's why this is done.

1:17:18  
So here the two-dimensional array is still laid down sequentially.

1:17:23  
There is just you need to remember which dimension is the first.

1:17:27  
So the increment goes from the end.

1:17:31  
1st we incrementing the last dimension when we moving towards the memory.

1:17:36  
Then once we are through with the second dimension, we will increment the first dimension.

1:17:42  
You don't have to remember that by your heart.

1:17:45  
Actually, this is like a just to show you how the linearization works.

1:17:49  
But to be completely honest, every time I'm using the statically typed aesthetically allocated 2 dimensional arrays, I need to look this up because just to make sure that I won't make a mistake.

1:17:59  
Usually we don't do that.

1:18:02  
OK, and I think this is a good place to stop right away because, yeah, next time we we pick up on arrays and I will introduce how the arrays are used for strings and other stuff.

1:18:13  
So yeah, we have some spare time, but I believe we can we can quit 5 minutes early.

1:18:19  
Are there any questions regarding this course, regarding the labs or regarding the C++ language whatsoever?

1:18:28  
Will we get any homework from the lectures or just the labs?

1:18:32  
You you want some OK, you can.

1:18:35  
So this guy wants homework so you can, you can beat him up later, OK.

1:18:40  
No, no, no, no, no.

1:18:41  
There, there will be no homeworks from, from, from lectures.

1:18:44  
The lectures are theoretical.

1:18:45  
And believe me, the only homework would be to, let's say, learn everything for the exam that that's enough.

1:18:53  
Just just do your homeworks in the labs that that will be that that will do nicely.

1:18:58  
OK.

1:18:59  
If there are no more questions, thank you for your attendance and we will meet next week.