**2025-02-27-cpp2-en**

0:00  
Welcome to the next lecture.

0:02  
I'm trying still to get hang of this audio visual stuff, so sorry about the delay.

0:08  
At least I found out how to turn on the lights.

0:10  
So today we will be a little more enlightened than the last time we ended about here in the erase section last week.

0:19  
So I just continue with this.

0:21  
You probably all have your labs by now, so you enter the basic C level and you need to code something into C You had to code something into C already, so you already encountered arrays.

0:33  
So some of the part would be just rehearsing.

0:36  
But this is a good place to start because arrays are very important.

0:39  
You basically need arrays or some array like containers all the time.

0:43  
So it's a really good idea to get yourself well acquainted with this.

0:47  
So arrays, as I mentioned before, are statically allocated.

0:52  
At least we will be using only the statically allocated arrays.

0:56  
It is also possible to allocate them dynamically, but dynamic allocation, dynamic memory allocation in plain C or in old C++ is somewhat tedious.

1:06  
So we decided to just skip it.

1:08  
And furthermore, you need the C and C++ only for Arduinos, mostly this this semester.

1:13  
So in Arduinos you won't be using dynamic allocation at all because it's somewhat impractical since the Arduinos very very limited memory.

1:25  
So basically we will stick with the static allocation.

1:27  
Static allocation means that you need to a priori, at the declaration time, specify how wide how large the array will be.

1:36  
So in this case I'll create an array of four elements.

1:39  
In this case, I create an array of three elements because this is inferred from the initialization when I explicitly state the number of elements.

1:49  
One important thing for arrays you need to remember right away, and you probably have from the last week, is that arrays have some base type.

1:56  
All the items of an array are of the same type.

2:00  
That's quite different from a dynamically typed languages like Python where you can have a list and you can place various data types as list items.

2:08  
You can mix them together.

2:09  
No, in C you need to get stick with single data type.

2:14  
You can also create multi dimensional arrays, but that's not that that is not that important.

2:20  
I'm using this just to demonstrate that if you use multi dimensional array it will still be alignerized in the memory, so it will be still reordered.

2:29  
So the all the elements of the array are sequentially stored in the memory.

2:34  
The important thing which will become apparent in short order when I introduce pointers is that an array is stored in a continuous block in the memory.

2:45  
So basically if you allocate 4 items for integers, you will allocate a block which has the size of integer multiplied by four and this is the block in the memory.

2:56  
A continuous block in the memory.

2:58  
It cannot be stored sparsely.

3:01  
That's one of the other, another reason why we are not introducing dynamic arrays, because if you want to relocate the dynamic array, if you want to push more items into it, you need to reallocate the memory.

3:13  
So you still have a continuous block somewhere else.

3:15  
And that's somewhat tedious if you're doing that manually and it's error prone.

3:19  
So let's just stick to fixed size arrays for the time being.

3:24  
So this was just a rehearsal from last time and let me continue with the new stuff.

3:31  
Another basic all right, some problem here.

3:37  
Another basic type that you will require for your day-to-day activities is a string.

3:42  
Accidentally or incidentally, strings are also arrays in C Basically it's just an array of characters.

3:50  
So array of the car type with small one small difference.

3:56  
Each string has a new character at the end.

4:01  
The thing is, plain C arrays don't know how large they are.

4:06  
Yeah, if you declare a static array like this, you can infer its size from how much bytes it allocates in the memory.

4:15  
If you have access to this variable, then you can somehow infer it.

4:19  
However, in case of strings, this information isn't stored anywhere, because string is usually represented only as a pointer to the beginning.

4:31  
So we need a way how to find out where the string actually ends in the memory.

4:36  
We know it's in continuous block, so we can just scan the memory, but we need to somehow find the end.

4:42  
And the programmers back in the 60s when they were designing this stuff, they decided that one way how to do this is to place an end mark at the end of the string and the end mark is the new character.

4:54  
By null character I mean a character which casts to 0 integer 0, not the 0 character.

5:02  
So there is a difference between null character and 0 digit character, right?

5:10  
Because if you cast it to int, this will be really zero.

5:13  
What?

5:13  
What will this be if you cast it to int?

5:16  
Who remembers esquitable from from heart?

5:20  
Just just curious, somebody knows that You don't have to know.

5:24  
You really don't have to memorize it.

5:25  
Just just no, incidentally, it's 48, but you don't really have to remember that.

5:34  
If you don't believe me, just look up the Esky table and you can find out the 0 digit and you will turn it into 48.

5:42  
OK, so these are these, This is how strings work.

5:46  
This is how the strings are.

5:48  
You can write strings by using double quotes.

5:52  
And if you do this literal will actually allocate a memory for how many bytes exactly?

6:04  
Because there are 5 letters here and we need one extra character for the ending 0.

6:09  
So this will look exactly like this in the memory.

6:13  
OK, if you want to find out how many characters are there, you really need to go from the beginning and scan each character until you find the 0 character and then you just happen to know how many characters you have counted.

6:26  
OK by the way, you can also declare an array of characters.

6:31  
Actually, these two are quite similar, right?

6:33  
Declaratively they are identical, they are just different identifiers and you can initialize it using the array initializer.

6:40  
But beware, this initializer doesn't take into account the ending 0.

6:45  
So if you really want to create an array of characters for some other reasons, you can.

6:50  
But beware there is no ending 0.

6:53  
So if you incidentally try to convert this into a string, it will work technically, but without the ending 0.

7:01  
So if you try to access it using regular C functions like string line Sutherland which which counts counts how many characters are there, it will it might run forever or it might run into memory which is not allocated because there is no termination character which would stop it.

7:18  
So beware when you're trying to convert arrays into strings, it's better to stick with one or the other depending on what you want to, what you want to do.

7:27  
OK, just a side note, we are talking about arrays.

7:35  
We are talking about base types and things like that.

7:38  
There is one piece of information I would like to give you right away.

7:42  
Actually, we get back to this in in the memory layout when we will talk about memory and memory allocation and stuff like that.

7:48  
But this is very important.

7:50  
So I'm giving it to right away and you just refresh it later.

7:56  
So the information I'm giving you right now is that if you have a type which takes X bytes in memory, it has to be placed in the memory.

8:06  
So the address where this element starts is divisible by X.

8:12  
In other words, divisible means that if you divide it, the modulo the reminder will be 0, right?

8:20  
So in other words, if you take like a 32 bit integer which takes 4 bytes in the memory, it must be placed on an address, it must start on an address that is divisible by 4.

8:31  
Actually this is like a thing that's enforced that's required by hardware because hardware specifies this.

8:39  
If you take a regular CPU, most of the architectures of the day will just tell you this must be done, otherwise it just blow up in your face.

8:49  
I I mean like in a software manner, not in a hardware manner.

8:53  
So you really need to do that, otherwise the code wouldn't work at all.

8:57  
And some older architectures tend to, let's say some mitigations in the hardware so they can load data which are misaligned, which are not properly aligned, but it takes longer, so it takes some extra memory operations, etcetera.

9:13  
So it's always a good idea to do do that.

9:15  
And C compiler enforces this, C compiler knows this and does this for you.

9:20  
However, sometimes you need to be cautioned because it has some side effects that you need to be aware of.

9:25  
Just to clarify what I mean by the alignment.

9:27  
So here are three examples of three values which we have stored somewhere in the memory.

9:33  
And for some reasons, because for instance they were declared as a sequence of variables, the compiler decided to place them one after another.

9:41  
And incidentally, the compiler was able to do that because the character actually character is easy.

9:47  
Character or byte, they are easy because they take one byte.

9:50  
So every every address is divisible by one, right?

9:53  
Everything is divisible.

9:54  
Every natural number is divisible by 1.

9:56  
So we don't have to bother where the character will be.

9:58  
That's one of the benefits of strings.

10:00  
Also, you can allocate strings everywhere because they are made of characters.

10:05  
However, when we take an integer, in this case 32 bit integer into account, we need to make sure that this integer is aligned to four bytes.

10:14  
And if if you bear with me, you can just take this address and divide it by 4 in your calculators.

10:20  
For for instance.

10:22  
Then you find out that this number is really indeed divisible by 4.

10:26  
So the integer can start here and double, which takes 64 byte bits.

10:31  
Sorry, 8 bytes must be placed on an address which is divisible by 8.

10:36  
This is much more simple because 140 in hexa is is obviously divisible by 16 because it has a 0 here, right?

10:46  
We have a zero and it's 16 base math.

10:48  
So 0 means it's divisible by 16.

10:51  
So if it's divisible by 16, it must be divisible by 8 as well.

10:56  
So this is how we aligned values in the memory and why it's important will become apparent in short order when I talk about structures.

11:07  
Structures is the second most important data structure or data type compose data type in C So we have arrays for most cases, but sometimes you need to aggregate different types of values or named values.

11:22  
So basically it's also a collection of items.

11:24  
Similarly like array.

11:26  
But unlike an array, these items are not numbered, are not indexed from zero, but they are named.

11:32  
So we have a name, we have a identifier for each field in the structure.

11:37  
That's the first difference.

11:38  
And the second difference, perhaps the more important one, is that each field can have a different type.

11:43  
So you can mix types in the data structure, in the structures and it's fine.

11:47  
It's, it's designed to do that.

11:49  
So let me give you an example.

11:51  
We have a 2 dimensional point in a grid, not in a continuous space in a grid.

11:57  
So we use integers to index its position XY.

12:02  
And these two two indices obviously are logically tied together because they both belong to the same point.

12:08  
So in this case it's a very good idea to create a structure for them, because structures are designed to somehow bound together things that are logically designed to be together.

12:19  
In this case, 22 indices of the same point are logically should should be logically together.

12:28  
By the way, sometimes you can you can use other approaches how to do deal with this, but this is like the rule of thumb approach.

12:35  
If you have things that are tend to be together, it's a good idea to put them into one structure.

12:40  
If you're doing something very advanced, like in high performance computing, this doesn't have to hold, but for the beginners it's the best way how to how to deal with data.

12:50  
The important thing I'm trying to show here is that the the structures need to employ some additional measures regarding the alignment of the internal things.

13:01  
Because normally items, the fields in the structure will also be aligned in the memory one after another, and the C will preserve the order of declaration.

13:11  
So if you say this structure has X and then Y, then X will precede Y in the memory.

13:17  
That's a rule.

13:18  
OK, so if I declare a structure which has character C, double D and integer I.

13:24  
I'm sorry for the lack of imagination.

13:26  
I just want to give you a quick example so that you can infer the type from the name of the field.

13:33  
OK so in this case we have 3 fields of different types and they need to be laid down in this order.

13:40  
So C must come first, then must be D and then must be I.

13:43  
Unfortunately, if the beginning of the structure is aligned in the memory and it and it usually has to be, then C will probably be on an address which will be divisible by 8 or 16 or something like that.

13:58  
So if you want to place double properly in memory, you cannot place it immediately after C because if the beginning of the structure was aligned, then the place immediately after C will be aligned plus one, which is definitely not divisible by 8, probably not divisible by two.

14:17  
So you cannot just place it there.

14:21  
What the C what the C compiler does is that it creates additional padding.

14:25  
It just creates an empty space.

14:27  
Nothing will be there, just a spare space where no data will be stored.

14:34  
And the D field, which is a double will be shifted so it's aligned with the next alignable address.

14:41  
So it's aligned to 8.

14:43  
OK fortunately for us I is integer and integer is usually smaller in this case.

14:50  
In this example it's 32 bit, I mean 4 bytes wide, so it's smaller than the double.

14:56  
And luckily for us, if we have A8 byte word on an 8 byte alignment, the next position will start at the 16 byte alignment, which is also divisible by 4, so we can immediately after the place I.

15:09  
So the only padding will be will be.

15:11  
Here there is also an additional padding added at the end because the structure must be aligned externally to the largest item inside.

15:21  
We get to that later when we talk about the memory.

15:24  
This is just some things that you need to know that there is a big difference between array and structures in the memory layout.

15:31  
Because structure can be padded while the array is always compact.

15:35  
We don't have to pad anything in array because all the types are the same, so there are no gaps that there shouldn't be any gaps if the beginning is aligned.

15:45  
Just a technical detail, syntactical detail.

15:47  
If you're declaring a structure, it's quite easy.

15:49  
You just use the struct word and in the block of the structure you just list all the fields that are in the structure.

15:57  
When you do afterwards you can use the name of the identifier of the structure, which is in this case point 2D as a regular data type.

16:05  
So when I want to create a variable which holds this point, I just declare this variable using the point 2D as a as a regular data type as here I'm declaring integers.

16:15  
So here I'm declaring point 2D variable OK and accessing the internal property in the internal fields is done using the dot notation.

16:25  
So you just write the the array sorry sorry sorry.

16:28  
The variable that holds the the structure dot the field and this together acts as a variable.

16:37  
So you are accessing one like one Part 1 field of a larger structure like this.

16:44  
OK, so those are structures.

16:50  
Actually most of the stuff you will be doing, at least in this semester, will be handled in arrays and structures.

16:57  
So we don't need any more complex data types.

17:00  
In the end I will give you a glimpse on classes which are like extended structures which interconnect the data with the code, but you don't have to use them if you don't really want to.

17:12  
However, we still need a few things, a few additional things to make all stuff work.

17:18  
So now I'm now I'll be like skipping from a topic to topic for a while because we need to cover few distant issues.

17:27  
First one is constants.

17:28  
We already declared times, we know what variables are, but sometimes we need to create a constant, which means like something that doesn't change over time.

17:38  
The most simple constant in C is just a constant which which is created by prefixing a variable declaration with construction.

17:47  
Basically you can imagine this as a variable which cannot be changed over time.

17:53  
Why is this important?

17:54  
There it's, it's actually quite important if you're doing something more elaborate because you are telling the compiler this won't change.

18:03  
And that's important information for the compiler because when it doesn't change, it can be optimized.

18:08  
For instance, this value can be easily cached in a register if need be and and it will, it won't ever change because this, this variable cannot change.

18:16  
So it's it's always the same, it's always 42, OK.

18:19  
However, this is really like a variable.

18:22  
So you can take an address of this variable, it lives somewhere in the memory, etcetera.

18:27  
C++, Modern C++ offers a more elaborate way how to declare constants using a constant.

18:33  
Expert.

18:34  
Know that there is only slight difference, this is cons, this is cons.

18:37  
Expert.

18:38  
The behaviour is much more different from the terms how the things are from the perspective how things are compiled.

18:45  
So in this case, this value, this this variable is inlined during compilation if possible.

18:52  
What does it mean?

18:53  
It it means that it replaces all the occurrences of this variable everywhere in the code with this value.

19:04  
So it kind of integrates the value directly into the code.

19:08  
Now you can, you can think about why, why we need that?

19:12  
Why don't we put the 42 in the code right away?

19:15  
OK, There are many reasons.

19:17  
If you haven't read the Unforgivable Curse 2, I think it is.

19:21  
Then if actually, if you read the, the reasons for Unforgivable Curse 2 already, then you know the reasons.

19:27  
And the most importantly, it's much more readable if you know what what the 42 means.

19:33  
If you know it's the answer to the question of life and universe and everything, then you can just name it so that everybody who reads it knows that you mean, that's the, that's the magical number.

19:43  
That's the that's the answer.

19:45  
It's not size of my shoes, it's not my age.

19:49  
It's not anything remotely related to anything except it's the answer.

19:55  
So everybody knows what what what that means.

19:57  
OK.

19:58  
And the second reason is that if for whatever reason you want to change this value in the future and you just happen to integrate it all over your code, then you need to find all the occurrences in your code and change the variable.

20:12  
Change the value, Sorry.

20:13  
So that might be a bad thing.

20:15  
Yeah, I know this answer will never change.

20:17  
We we know that if you, if you read the book that you know, you know this won't change.

20:21  
But there are other examples where the variable where the value of this constant might change, like a pin number.

20:29  
You have an Arduino and you have a pin number where a button is attached, for instance.

20:33  
So you need some identification of this pin.

20:36  
And it's possible that sometime in the future you rewire your Arduino.

20:42  
So you attach the button to a different pin and you don't want to rewrite all the places where the pin number is in your code.

20:50  
You just want to adjust one piece of the code, one place in the code and recompile it smoothly.

20:56  
So that's another reason why we are using a constant or constant expression.

21:00  
In this case using constants and constant expressions, there is one thing that's important.

21:08  
Sometimes you need to do enumerations.

21:10  
By by enumerations I mean you have a special type which isn't an integer, which isn't a string, which isn't anything.

21:18  
You just need to enumerate some values, enumerate states.

21:22  
You have a button and it might be released, pressed down or or being held for some time.

21:30  
So it might have us 3 states from your application point of view, you have a light in this room.

21:38  
The lights in this room might be in state they are turning on, they are being dimmed from zero to to some maximum illumination.

21:45  
They are being turned off or they are on or they are off or they may be also in some maintenance mode where they are flashing or something.

21:54  
So they are like a special cases, 5 cases, for instance, how the state of the light in this room can be described.

22:03  
Yeah, I'm making things up.

22:04  
I'm not sure if I can make them blink, but just imagine and just assigning arbitrary numbers like let's say 0 is turned off, 1 is turned on, 2 is they are being turned on continuously, 3 is they are being dimmed down or something.

22:23  
Who?

22:23  
Who would remember these constants?

22:25  
Who would remember these values?

22:27  
Yeah, you can use constants to just mark all these value, all these special values, and then use an integer to represent the state.

22:34  
But it's a bit clumsy.

22:36  
So exactly for that reason, we have a special type called enum.

22:41  
Actually in C++ in Alt C, you just use enum.

22:44  
But in modern C++ we use class enum.

22:47  
OK, so just remember the class Anum.

22:49  
It's, it's better to start with the better with the C++ way right away.

22:54  
And here you can specify identifiers for the individual states.

22:59  
So actually we don't know how they will be represented in code.

23:03  
Well, we know, but it's not important how they will be represented in code in memory.

23:07  
The important thing that is that I will refer to them in my code using these identifiers.

23:14  
So in this case it's a stopwatch, I think no, it's some something that can be running or waiting or or whatever.

23:23  
So it's some some machine that can be in the one of these three states.

23:28  
And this state can be represented by one of these constants.

23:32  
So I can use this constant and assign to it or I can compare this state to one of these constants.

23:38  
Internally, enums are translated into integers.

23:42  
So in the end, waiting would be something like 0 running will be probably one stop read probably 2.

23:49  
But that's that's not none of your business.

23:51  
That's not your concern.

23:52  
You are using only these constants so everybody can read it much more clearly.

23:57  
Because if you comparing whether the state is stopped, then you know that the machine you're working on is stopped, not that it's in the state 2 What what the hell, whatever that is.

24:10  
So these are special traits in in C++.

24:16  
Another topic that needs to be covered right away is the preprocessor.

24:21  
The preprocessor is a special tool in the compiler.

24:25  
It's like a first stage of the compiler.

24:30  
The preprocessor arranges and puts together the source code in its native text form before it gets compiled.

24:39  
So it's like a stager, it's like a, it's like a thing that arranges stuff together before the compiler can take a look at the code as a whole.

24:49  
So what it does 11 important thing that it does is it processes the include directives.

24:56  
You've probably already seen the include.

24:58  
If you where in the labs you must have seen the include because you basically cannot write anything in C without include.

25:05  
Include is something like an import in Python.

25:07  
So you know that you need to import some libraries.

25:10  
In this case you are not important libraries, you are including headers there.

25:15  
There is some difference.

25:16  
The include actually works as a copy paste tool.

25:20  
So basically there is a path after the include path either to your file or to some file header file which is somewhere in the system libraries.

25:31  
And this file is really copy pasted where the include is.

25:34  
So you really take this file and replace this.

25:39  
Sorry not you.

25:40  
The preprocessor will take this line and replace it with the contents of this file, with all the benefits and all the troubles that comes along with that.

25:52  
OK, so it's not the same as import in Python, because import in Python works on a slightly different level, but from the perspective of what you need to do in your regular code, it it works the same way.

26:05  
Another thing you can do is define macros.

26:09  
The macro is a again some identifier.

26:12  
In this case I chose a quite short identifier, only N and this is just a basic text replacement.

26:20  
So every N in the source code will be replaced with the macro value 1000 in this case.

26:28  
OK, so if I if the preprocessor encounters N anywhere in the code, it will be replaced with 1000.

26:36  
So this looks like another way how to declare constants, right?

26:40  
You can declare 1000 instead of constant Expo, which I instead of this which I which I presented in the previous slide.

26:50  
However, do not do do this.

26:53  
Use construction expert instead.

26:56  
Why?

26:57  
Because this step is done by preprocessor, so you're basically replacing the N with whatever the value of macro is before the compiler takes glimpse at your code.

27:11  
This may be useful in some very Special Situations, but in regular situations you don't want to do that.

27:17  
You really want to make sure that the compiler will take a look at the value N and for instance, perform type checking.

27:25  
Because as you can see on the previous slide, a constant expert is bound with an exact type like a variable.

27:33  
So what what the compiler does on content expert values is it performs type checking.

27:38  
If you leave that to macro preprocessor using this define, then you get rid of that.

27:45  
Sometimes it might be good if you're doing something very special or something in very low C, but normally for the regular code you don't want to do that.

27:53  
So use cons xper instead of instead of this.

27:56  
And finally, and this is for advanced users only, you can also make a conditional code.

28:01  
So you can make an if def block which kind of a covers part of code.

28:09  
And this code will be removed if this macro isn't defined.

28:13  
So if you if you defined a low debug macro, it might be empty.

28:17  
But if you define it, then this code will hold, it will remain where it is, just the the preprocessor will just remove all the things starting with hash.

28:27  
OK, but if this macro isn't there, everything between this if def and and if will be removed.

28:34  
So it's another way how to let's say compose your code on a high level.

28:38  
This might be useful for debugging.

28:40  
For instance, if you have some debugging code in debugging print outs in your code, this is the way how you can conditionalize them.

28:49  
So this is the way how you can remove them for the production version of your code.

28:53  
Actually, there are better ways how to do this in C++ is this is just to give you a glimpse.

28:58  
It might be useful if you're if you're advanced users and you want to debug Arduino.

29:04  
OK, another topic is the application entry point.

29:09  
I already mentioned that in the Hello world example.

29:12  
So you and you already need to know that because you already written some C code.

29:17  
So the code which which is executed the first the main code of the application is placed in the function named main.

29:26  
The important thing is that we have no top level statement.

29:29  
So every code in C++ must be placed in a in a function or in a method or something.

29:35  
But it cannot be as a top level statement.

29:39  
At the top level we are only declaring the functions, the structures, global variables and stuff like that.

29:48  
Important things you need to know right away about main is it has a a return value which is an integer.

29:56  
And this integer isn't just an arbitrary integer.

29:59  
This integer will be actually assigned as the exit code of your process.

30:04  
So usually if everything works fine, you should return 0 because zero is in all operating systems I know viewed as a success exit code.

30:16  
Nothing's wrong.

30:17  
0 errors, something like that.

30:21  
If you return a non 0 exit code, it's considered a failure.

30:25  
So something happens in your application and the application couldn't quit properly, couldn't be terminated, they didn't.

30:31  
They didn't finish its computation or something.

30:33  
Usually this is This is very useful if you're designing some command line tools, because command line tools somehow communicate via these exit codes.

30:43  
Normal operating systems has only one byte, like Linux has only one byte for an exit code.

30:48  
So exit code.

30:49  
Although this is an integer, the exit code must fit the range from zero to 255, otherwise it's it doesn't work.

30:58  
So just pace yourself if you want to use many exit codes.

31:02  
It doesn't give you much information to be smuggled out of your application.

31:06  
If you're if you're thinking this way is is really just an just an indicator whether the application submitted successfully.

31:14  
Or you can have some list of errors, error codes in your documentation if you wish.

31:20  
And second, second thing is the input of the application speaking in the command line terms.

31:27  
So basically, if you're designing a command line tool, you usually want it to respond to additional options, parameters, arguments that are transported that are provided via command line.

31:40  
I think you already have Linux, right?

31:42  
So in Linux you already has at least a glimpse of the command line and you need to use some command line tools.

31:50  
So this is how this works.

31:51  
You need to pass down to these tools some options or parameters, and you can receive these parameters via these two input arguments of the main function.

32:02  
This is the number of arguments passed from the command line passed from the prompt.

32:07  
And this is an array of strings.

32:11  
Yeah, this is character\* which means this is a string which holds a list array of all the arguments given to C strings.

32:24  
Beware, the first argument is always the path to the executable file path to your program how it was executed.

32:33  
OK, so this is the most important thing in in see about the main application entry point.

32:41  
And now let me start with a slightly more complex topic and that will be pointers.

32:47  
Pointers are very important and they are very special because you won't encounter pointers in other languages in in C# or in Java, pointers are hidden from you.

32:57  
You have referential types in C#.

32:59  
You probably notice that because you also have some C# already, but and there are some pointers beneath these referential types, but you cannot create them manually, you cannot directly access them.

33:13  
You you cannot handle them in a raw way.

33:15  
So only the C and C++, at least from the mainstream languages, will give you that opportunity and the opportunity to fail horribly.

33:23  
Because if you do something slightly wrong with pointers, then you usually do something very bad with your application.

33:31  
Pointer is basically a number.

33:34  
Pointer is an address.

33:35  
It's a number that stores an address.

33:38  
However, it's treated very specially in their application because it's like a really a direction, an arrow that that points to something.

33:49  
What's important is, yeah, technically we could store the addresses as regular integers, right?

33:57  
So technically we wouldn't need pointers.

34:00  
We just say, OK, this integer is, is an address and I will use it in an address manner.

34:05  
Yeah, this is how it works in assembly language.

34:08  
But in C language we want better control over the pointers.

34:13  
So the most important thing is if you just allocate int or on integer type for a pointer, it might not work because the size of the pointer depends on what architecture you are.

34:27  
If you are on a 32 bit architecture, you will get 32 bit pointers.

34:31  
If you are on a 64 bit architecture, you get 64 bit pointers.

34:35  
Then again, integer doesn't have to follow the same logic.

34:39  
Integer on a 64 bit architecture X86 is still 32 bits wide.

34:45  
So basically you have to be extra careful and you need to put extra effort to recognize how wide should be the number where you're placing the address.

34:56  
With pointers you don't have to bother, the compiler knows how wide should be the pointer, so we do automatically allocate the right amount of bytes for each pointer.

35:05  
So this is the first advantage.

35:07  
And the second advantage is a pointer has a type, so we always know what is the data to which the pointer points.

35:16  
Yeah, that's right.

35:19  
Sorry.

35:19  
Sometimes we need to sequence the words correctly.

35:23  
Basically, I cannot create something like AI can be a pointer to a student.

35:27  
OK, so I can point at any of you.

35:30  
And the benefit is that if I know that I am appointed to a student, then I always need to be pointing to a student or I can be pointing to null saying I'm not currently pointing at anyone right now.

35:43  
OK.

35:44  
And the benefit is that let's say a janitor comes comes in or, or a homeless guy from outside just came in.

35:51  
So I can't point at him because he's not a student or the the janitor at the reception, she's she's not a student.

35:59  
So I, the C++ wouldn't permit me to just take just point at her.

36:03  
And the benefit is that if I'm pointing to a student and I want, I want to pose a question like, yeah, I, I don't have any tough mathematical question at mine right away, but we'll just assume I'm, I'm asking some very tough mathematical question that, that it might be worthy asking this question, some of you, but it wouldn't probably wouldn't be worth it asking this question a homeless guy or, or the receptionist.

36:28  
OK, so this is the benefit I get that I know what I'm what I'm pointing at.

36:33  
And the CC plus plus language wouldn't allow me to point anywhere else.

36:37  
So it wouldn't allow me to take a pointer to a bad type.

36:42  
Actually this is not completely true, because I have casts, I can cast pointers to different pointers.

36:47  
So yeah, there are tricks how to manipulate this mechanism.

36:51  
But when I'm not using these tricky things, then the C++ will like protect me at least a bit from taking a pointer to a bad, bad type.

37:02  
OK, how this works?

37:07  
Yeah, I just assume I have a variable, integer variable somewhere in the memory, probably a local variable or global variable.

37:16  
It doesn't really matter, it sits somewhere in the memory.

37:19  
It takes about 32 bits, doesn't really matter it it sits somewhere in the memory.

37:25  
And then I want to take a pointer to this variable.

37:30  
So first of all, syntax, the pointer is declared using an\* in front of the identifier.

37:37  
So this end is the pointer type, is the data type to which the pointer points.

37:44  
This is a regular variable identifier.

37:47  
And this\* indicates it's a pointer.

37:51  
OK, And this & is a operator.

37:58  
Note it's a unary operator, not binary.

38:01  
It's not bitwise AND.

38:02  
It's a unary operator, and if you use it to before in front of a variable, then it takes the address of the variable.

38:15  
So instead of taking the value of V, it takes the address of VI.

38:22  
Know that V sits somewhere around the address 1234 in memory.

38:27  
This is the index of the byte where V starts in the memory.

38:33  
Just imagine the memory as a linear array.

38:35  
So this is the offset of V in the array in this array.

38:40  
So basically I'm storing value 1234 inside this PV variable.

38:49  
Pretty obvious right?

38:51  
I hope And just wanted to lay down slowly so everybody can understand and everybody can catch up.

38:59  
And the most important thing I can do with a pointer is that I can dereference it, which means I can access the value via its address OK.

39:12  
This asterisks in front of the pointer value expression if I don't use.

39:17  
If I use PV without the\* I'm using the pointer itself.

39:21  
I'm using the address itself.

39:23  
If I place\*, it means, OK, take one step further, interpret the address as an address and go to that address and give me or in this case writes to the value which is there.

39:35  
You can use it both on the left side and the right side, so you can use it for reading and writing.

39:40  
So in this case, I am, I am all writing the value of V because I'm writing to the address 1234 and I'm putting a four in there.

39:50  
OK, everybody's clear on that.

39:53  
I'm sorry.

39:54  
I'm taking this one step at a time very slowly because I really want to make sure that everybody gets every detail.

40:02  
No problems with that.

40:03  
OK, so few more things.

40:09  
Pointers are important for us in many ways.

40:12  
Some of them you haven't begin to realize it, or maybe if you work with C or C++ in in in the past you you have.

40:18  
But this is important that the.

40:23  
Arrays are basically pointers.

40:27  
Arrays are pointers in disguise.

40:30  
If you have an array, it is internally represented as a pointer to the first value.

40:37  
If you pass an array into a function, you are not copying the array as a whole.

40:43  
Well, you you can, but you need to specify it differently.

40:46  
But if you just pass it down, you're usually passing down the pointer to the beginning of the array.

40:53  
And that's very important, very important thing.

40:55  
So just imagine that you have this and this arrays, this, these two are almost equivalent.

41:05  
Yeah, actually there are some minor differences if you're accessing this variable and this variable in some manner.

41:14  
There might be there might be differences if you, for instance, take the size of these two variables, there will be different.

41:20  
I'll get to that shortly.

41:21  
But from the perspective of reading or accessing the elements, they are both the same.

41:27  
Yeah, in this case, I'm storing the whole array in here.

41:31  
In this case, I'm taking a pointer to an array which is allocated somewhere globally, but that's not that important.

41:38  
Same thing goes for four strings, right?

41:40  
So for instance if I create something like this, it will be laid down in memory as follows.

41:48  
So this is XKCD acronym.

41:52  
Actually it's not acronym, it's just you can you can Google it up if you don't know what it is.

41:58  
But this is a four letter word and this 4 letter word is declared as a string.

42:04  
So it has A5 bytes in memory because it needs the new termination at the end.

42:09  
So somewhere in memory there will be 5 bytes occupying these 5 characters and string over here will basically hold the pointer to the big to the 1st letter to the first character of this string.

42:25  
OK so this is what I what I was trying to told you from the beginning, but now with the pointers you have really taking the grasp how this works and that reveals the nakedness of of strings and also of arrays.

42:42  
If you need to know how many characters are there you need to somehow pass it along.

42:48  
In case of strings we are using the end the termination character to find out where the string ends in the memory.

42:55  
As you find out in the labs, if you need to pass down an array into a function, you also need to pass down the size of the array as an associate parameter.

43:05  
So you cannot just pass the the array itself because you wouldn't know how large the array itself is.

43:20  
The pointers in C are equipped with something which is called a pointer arithmetics.

43:27  
It allows you to compute shifts.

43:30  
You can compute another pointer from 1 pointer by adding values to it.

43:35  
So it's kind of strength because basically in most languages this is not allowed.

43:40  
You can take an address of existing object, you can create an object and then you get the reference to that object.

43:47  
Yeah, you can.

43:47  
You can have a noun, so you can have some special pointer that points to where, but you cannot just shift the pointer around in the memory.

43:55  
For instance, in C#, you cannot even, at least not easily, take a glimpse on what the reference actually is.

44:03  
You cannot get to the numbers.

44:04  
In C it's.

44:05  
It's much more easier.

44:07  
But what C gives you in addition is that you can take a pointer, add a value to it, and you get another pointer.

44:14  
So you can create arbitrary pointers in your memory regardless whether this memory was allocated or not.

44:22  
Which is very handy if you know what you're doing, and very dangerous if you make a mistake, because you can easily create a pointer that points to a memory which hasn't been allocated, for instance.

44:35  
Right how this works, I will start from the from the middle this syntax which is accessing an element of an array.

44:46  
Actually it's a string, but that's not important because string is an array of characters.

44:51  
So accessing this character is actually equivalent with adding I to the pointer itself and then dereferencing it, then accessing its value.

45:04  
OK, so these two things are equivalent.

45:07  
It's your C++, I'm just taking a second so that it this message can sync In Sync in.

45:18  
So if you're accessing an element of an array, you are basically taking the pointer, adding a value to it, and then dereferencing it.

45:32  
This works fine because, well, this must work fine in any case because the pointer points to a character and character takes 1 byte in the memory.

45:42  
So if we advance a pointer by I bytes, then you will arrive to the desired character, right?

45:50  
Everybody sees that.

45:52  
But this works if you have a point, an array of integers as well.

45:58  
So if you have a normal array.

46:06  
Like this it will work as well.

46:08  
So if you write something like array +2, it will advance the pointer not by two bytes in the memory, but by two integers in the memory.

46:23  
That's the trick embedded in here and how we can do that because we know the type of the pointer.

46:30  
So thanks to that we can do pointer arithmetics without too many errors.

46:35  
Yeah, it's still very error prone, but at least we don't have to bother with finding out how what's the size of the base type of this array it's embedded inside.

46:44  
So this is advancing the pointer by two integers forward.

46:48  
And also This is why it's very important that an array is allocated as a continuous block in the memory, because if it wasn't continuous, this magic wouldn't work, right?

47:00  
You couldn't.

47:00  
You couldn't just index using this point to arithmetics, because it will break down.

47:07  
This works only because it's an compact block in the memory and you can access each element of the of that block just by computing its address using a simple multiplication and addition.

47:21  
So I'm just telling you this.

47:23  
By the way, this is one of the greatest mistakes I we usually make in this class.

47:29  
We introduce this syntax and then suddenly all the students are starting to use this syntax for accessing array elements.

47:35  
Please don't.

47:37  
This is just an example to make you understand how the pointer arithmetics work, that the pointer that the array is basically just a pointer to the beginning of the array and how the elements are accessed.

47:49  
But always use this syntax instead of this syntax.

47:53  
This is much more longer, less readable, it doesn't work in other other languages and it's it works only in low level C If you just want to switch an array for something more convenient like a vector, which is a standard container, then suddenly our code would broke break down.

48:08  
So it's better to use always the bracket syntax, not the point of the reference syntax.

48:15  
On the other hand, sometimes you can use point of the reference syntax, usually in this manner.

48:21  
This is like a piece of code that's that you can see many, many times if you just dig deep enough into some old C code or kernel functions or stuff like that.

48:32  
If you want to iterate over a string, one way how to do it is using a while loop.

48:38  
Because in the end, yeah, you can, you can rewrite it in a for loop like this one.

48:43  
But in the end, if you don't really need the index, if you really need the the pointer to the actual character, then this will access the actual character and this will advance to the next character.

48:56  
So you can iterate over the over the whole string, processing each character for instance using this while loop.

49:08  
OK, besides pointers, well, pointers are necessary if you're writing in plain C As you notice that I'm sometimes saying CC plus plus, sometimes I'm saying both.

49:22  
What?

49:22  
What we are trying to teach you here is C++, but we are focusing on the subset which basically almost covers C This doesn't.

49:31  
This is AC plus plus extension C++ feature.

49:36  
And these are references.

49:38  
References are also pointers, but they are slightly different.

49:43  
The reference pointer is not reassignable.

49:47  
Once you create a reference, it cannot point anywhere else.

49:51  
That creates a small conundrum because it's like, it's like, what?

49:56  
Why would we need that pointer?

49:58  
One of the greatest benefits of a pointer is that I can point to one student and later I can point to another student and later to another student.

50:08  
Sorry if if I'm bothering you with pointing my finger on you, I'm I'm not pointing any fingers right now.

50:12  
OK, in a literal sense.

50:16  
So basically if you have a non some, some mix up with the animation, sorry about that.

50:25  
So if you don't need, if you don't allow your pointer to be erased sign.

50:30  
If you cannot point to another example of the data type, you lose something.

50:36  
On the other hand, you get something.

50:39  
The reference is always assigned.

50:41  
So first thing is that you cannot get a null reference again.

50:46  
You can create it if you really really want to, but it's quite difficult.

50:51  
You need some castings and some some tricks to do to do that, and it's considered a bad thing.

50:56  
So normally a reference is always valid, so you know there is something at the end of that pointer, something valid this pointer points to.

51:06  
And 2nd, it stand to reason that there are situations that you really don't want to change the pointer.

51:11  
For instance, if you're using the pointer to pass something down to a function, I will get to that in short order.

51:18  
So if you need a fixed pointer that cannot be altered so it cannot be for instance altered incidentally or accidentally, then reference is your choice.

51:30  
Again I will use the same example.

51:32  
So we will have AV variable which sits somewhere in the memory.

51:35  
We already met this one so we are already acquainted with it.

51:40  
But this time I will take a reference to it.

51:43  
So instead of using esterisk which denotes pointer I am using & to declare this PV is a reference.

51:53  
Now thinking about it I may I may be I may change this to RV because PV was pointer to V so RV would be like a reference to V So sorry about that but not that important right now.

52:08  
Just to remind you when we are using pointers there were there was\* over here and there was & over here.

52:15  
So we just shift the & here and there is only the V How the reference would know that we are that we want to take the address of V?

52:26  
Well, the C++ knows because when you're assigning something to into a reference, it only can be an address.

52:34  
So basically if we can only assign addresses here, there is no point in copying the value of V we can only take the address of V So it we can save one one character from our code at least.

52:51  
And second thing which is saved here is that since the value of the reference cannot be changed, we can strip the & or\* or whatever when we are accessing the value of the PV.

53:08  
So we are reading or assigning something via reference.

53:12  
We don't have to dereference it because it's always automatically dereference.

53:17  
We cannot access the address itself easily and it doesn't make sense.

53:22  
We cannot reassign the address.

53:24  
So it doesn't make sense to assign another address here.

53:27  
So the only assignment valid here would be the assignment that rewrites the target memory, right?

53:34  
So basically we can spare another character here.

53:38  
Sorry.

53:39  
So yeah, in the end, the four is over it.

53:42  
The 8 was over it with the four in the V variable.

53:46  
So this is the same example as with pointers.

53:48  
We just save save some some writing.

53:52  
When in doubt in C++ whether to use pointer or reference, and the reference will do, then use a reference.

54:01  
Yeah, it's like a.

54:02  
Take it as a rule of a thumb if you'd like.

54:04  
So if you don't know whether to use pointers or references, then use references.

54:09  
If you cannot use reference for some reason and you really cannot make it work with reference, then you probably need the pointer practice.

54:18  
Would you ever need a reference?

54:21  
I'm not trying why you need a reference.

54:25  
I will give you a very good example in short order when I start talking about functions that that's the perfect example for reference.

54:30  
So bear with me, it will come soon.

54:38  
So this is the example.

54:39  
Actually, this is the prelude for the example I was talking about.

54:43  
The pointers and references are most useful when you try to pass something down into a function.

54:50  
Why?

54:51  
One of the important things you probably learned about functions is that if you have a function and you're passing a parameter, it's always passed by value.

55:04  
It means a copy of the parameter is passed down to the function.

55:09  
You can modify it, but the original value wasn't modified.

55:12  
So if you have a variable which is passed as an argument of the function, yeah, it can be modified inside the function, but only the local copy is modified, not the original variable.

55:22  
Sometimes you need to, Sometimes you need to pass down a variable, and you really want the function to modify that variable.

55:33  
Yeah, one way how to do this is that the new value of that variable can be passed down from the function, right?

55:41  
So I just assume I have a function fool which takes one integer and I want to modify the value of this integer I'm passing down.

55:59  
So one way how to do deal with this is that the Foo returns the new value of the integer and then you need to use it in this special manner, something like.

56:09  
Let me use a different thing.

56:11  
A equals Foo, so you provide the old value, it's copied inside.

56:18  
New value is computed and returned as a value which is assigned to a.

56:23  
Sometimes this might be a good idea to do that, so the caller can choose whether the original value will be overwritten or whether it will be updated or whether you want to store the new value into a different variable.

56:36  
So sometimes this is a good idea how to deal with modifications.

56:41  
And furthermore, if you do it like this, then Foo might be a pure function.

56:45  
And I mentioned or maybe your teachers mentioned that pure functions are always better than normal functions because they have no side effects.

56:52  
They can be optimized better, they can be called in parallel, etcetera.

56:57  
However, there are times when you need to modify the argument that's that is being passed to the function.

57:03  
And for that we have pointers.

57:06  
So give me an example.

57:08  
This is a rotate function which gets an input structure and output structure.

57:13  
So basically we want to copy the input structure into output structure which is a point in 2D and we want to rotate this point 90° around the origin.

57:26  
So it's just a very simple operation.

57:28  
You just swept the X&Y and one of them is gets negated and that's that's the rotation by 90°.

57:35  
It's very simple.

57:37  
If you don't believe me, then just use the linear algebra you already have and you you can do it.

57:43  
And if you just fill in the 90° in all the sines and cosines in this multiplication matrix, you will get, you will get to this formula.

57:53  
The syntax which is important to us is yeah, here we are accessing the input structure and we already know that the input structures are accessed using the dot notation.

58:04  
So here we are accessing the Y&X values fields of of the input structure.

58:09  
That's fine.

58:10  
The input structure has been copied inside, so the in is a copy of whatever point we get as the input.

58:19  
However, we need to somehow pass out the new new point, so the output point is passed down as a pointer.

58:28  
This pointer needs to be the reference, so I'm using this notation.

58:32  
It's a.

58:34  
It's a dash and a greater than which visually forms an arrow.

58:41  
So it's like a dereference of a pointer to a field.

58:45  
And basically this syntax or this syntax is a shorthand for writing a pointer D reference and then applying a dot notation to access field to it.

58:56  
So it's it's we're still accessing a field of an of a structure, but since we have a pointer to the structure, that pointer needs to be dereference first so we can access the field right?

59:08  
Like we have the shorthands.

59:09  
It is, it is forecast or it's shorter.

59:13  
Some people decided it's it's like, yeah, first it's shorter and but some people decided it's like a more readable.

59:20  
It's because it's look like an arrow.

59:22  
It's it's like a dash and less than.

59:24  
So it's it's like a arrow to to emphasize that you are using a pointer to point to something.

59:30  
I don't know the etymology of this idiom, so sorry I don't have the insight and knowledge about why people decided.

59:40  
However, it it sticks in some languages.

59:42  
For instance, in Python, sorry in in PHP we we use this to access fields that that's another story because in PHP the dot was used as an operator for concatenating strings and then later when they add structure, added structures and classes they they realized, oops, we used.

1:00:01  
Dot for something else.

1:00:02  
So we really need another another token to to use to be used to access fields.

1:00:10  
So they use this from C to access field, but that's a different story.

1:00:15  
OK, so this is how we how we deal with that.

1:00:18  
What's the trouble here?

1:00:19  
Well, one trouble is that we need to explicitly, well, maybe not the trouble.

1:00:24  
One fact is that we need to explicitly distinguish accessing a simple structure and accessing a structure via pointer.

1:00:32  
On one hand, it might be good because you are explicitly stating you are accessing something via pointer.

1:00:38  
Everybody sees that it's perfectly readable.

1:00:41  
On the other hand, it's a bit tedious because every access must be carefully considered.

1:00:45  
So whether you're using a pointer or not, it's up to you.

1:00:49  
Second trouble is that some someone may incidentally slip in a null pointer.

1:00:56  
A0 pointer?

1:00:57  
That's valid.

1:00:58  
Pointer that's pointer that points to nothing.

1:01:00  
I already mentioned that in a in a sideways, but this is like a trap, because if somebody just give gives us the null pointer, the 0 pointer which doesn't point to anything.

1:01:14  
And the code gets in here because and it will because there are no conditions, nothing else.

1:01:20  
So here it will blew up in our face because the processor will try to dereference a null pointer which wouldn't work and the only thing that the processor could do is a I'm not executing this code just take it away and the the application will fail in something which is called segmentation fault.

1:01:39  
So if you ever seen the segmentation fault error, this is one way how to create it dereferencing a null pointer.

1:01:51  
This is one reason why the references are better.

1:01:54  
So we are getting to that.

1:01:56  
If we use a reference instead of a pointer, then first of all the compiler will enforce that here will be something from which we'll take the address, so something valid.

1:02:10  
We cannot create easily a new reference here, so this reference will be valid.

1:02:15  
We we can easily access it without checking.

1:02:18  
That's one benefit we get and the second benefit we get.

1:02:22  
We won't accidentally change this pointer.

1:02:25  
We really don't want to.

1:02:27  
We want that this pointer, actually this reference, sorry, always refers to the variable the caller gave us when this function was called.

1:02:39  
We we cannot change it by accident, the reference won't allow it.

1:02:44  
So This is why in this place references are better than pointers and that's one of the use cases which where the references are better than pointers.

1:02:54  
Second, very tiny benefit or maybe not benefit it's depend on a point of view, is that we use the same notation, only the dot notation to access both the regular structure and the reference structure.

1:03:07  
Again for the same reasons, there is no point in in distinguishing because we cannot possibly access the pointer itself which is beneath the reference.

1:03:16  
So the same syntax applies for the regular straight data structure and the reference data structure.

1:03:23  
Tiny, tiny advantage is when you suddenly when you pass down something as a value and later you decide you need to modify the the behavior of the function so it's get modified inside.

1:03:36  
You can just add the & here and everything changes.

1:03:40  
You don't have to rewrite the code because you use the same notation for regular structures and reference structures in the code.

1:03:46  
That's one tiny benefit.

1:03:48  
Also, it may be a pitfall because if you do it accidentally, then yeah, it might be also a pitfall.

1:03:54  
OK, So what you should learn from this if you didn't get all the details on the pointers and references and how they work.

1:04:03  
Just remember if you want to use a mutable parameter, something that is changed inside the function, you just use the reference.

1:04:13  
You just prefix it with & and it it will work fine.

1:04:16  
It will work as as easily as as as in any other language, like an out keyword in in C# language for instance.

1:04:27  
OK, I've spoken I've spoken about sizes of things for in a few places in this lecture.

1:04:35  
So let me give you an overview of a very important operator for low level programming.

1:04:43  
And this operator is called size of.

1:04:45  
It's very useful because it can take a size of things at compile time.

1:04:51  
So this is not like a regular operator which computes something at runtime and give you the number when you actually don't need it.

1:04:59  
It gives you the number, it gives the, it gives the number to the compiler.

1:05:03  
So the result of the size of function is basically treated as a constant.

1:05:09  
So it can be used to allocate arrays, it can be used to initialize constant expressions in, it can be made to make a compile time decisions when some some code is compiled in One Direction or in the other direction.

1:05:21  
Quite important in the size of operator you can specify either a type or a variable.

1:05:29  
So if you place a type there, it will give you how many bytes in memory this type normally takes.

1:05:36  
So this on most architectures will return 4.

1:05:40  
Because on most most architectures in integer has four bytes.

1:05:45  
There might be architectures with different numbers.

1:05:47  
For instance, if you try this on Arduino, you will get 2 because on Arduino integer regular integer has only two bytes.

1:05:55  
That's one of the pitfalls you should be aware straight away when you're dealing with Arduinos because yeah, the architecture is not 32 bit wide so they just use 16 bit int.

1:06:06  
If you have a data structure, meaning C structure, you can also use it here to find out how many bytes the whole structure will take.

1:06:14  
Beware that thanks to the padding, this may not be equal to the sum of the size of of each individual field in the structure, right?

1:06:23  
This may be larger because they may there might might be some internal padding.

1:06:27  
By the way, if you're not certain how the padding works, it's a good idea to just, I don't know, run replit or some other environment.

1:06:34  
Or if you have your own CC plus plus environment and just try to taking size of of a few data structures of of few C structures and try to shuffle the the fields in there so you can see how the padding works for yourself.

1:06:49  
It also works on variables, so if you just take a size of a variable it works fine.

1:06:55  
So in this case if you take a size of U variable, the compiler knows that U is actually an array of four integers.

1:07:03  
So it will give you 4 times size of an integer which will be 16 on most architectures, 8 on Arduino.

1:07:11  
OK, but and now we are getting to the difference between arrays and pointers.

1:07:17  
If you just happen to take an.

1:07:20  
Sorry this is still an array because we are not specifying the size of the array but the compiler infers it from the initializer.

1:07:29  
The initializer will take 5 + 1.

1:07:33  
Don't forget the the trailing zero 5 + 1 characters.

1:07:37  
So size of S will be 5 + 1 size of character.

1:07:41  
Size of character is always one on all architectures.

1:07:44  
It's like a rule in C++.

1:07:47  
So this will be 6 always.

1:07:49  
However, the tricky part is if you arbitrarily intermix pointers and arrays, sometimes it doesn't work as you expected.

1:08:00  
For instance, if I do the same thing here, the the only difference is that I didn't declare this as an array, but I just declared as a as a pointer.

1:08:10  
By the way, important fact, also a pitfall.

1:08:13  
This construction care refers to the character, not to the pointer.

1:08:19  
Which means I can still change where the pointer points, but I cannot change the characters to which they point, I can only read them.

1:08:26  
So this is a read only pointer, but I can still move the pointer, I just cannot modify the data to which the pointer points.

1:08:34  
OK, this is a common misconception.

1:08:37  
If you have a construction character pointer, the construction refers only to the target type, not to the\* By the way, you can have cons character cons pointer.

1:08:54  
You can have but you need to specify 2 cons in there so both the the pointer and the character are constant.

1:09:01  
You can also have a non cons non cons character constant pointer, but and I wouldn't know where to use that.

1:09:08  
But back to the example I'm trying to back to the code I'm trying to explain.

1:09:14  
So this is quite different because yeah, it will still create a string somewhere in memory, somewhere in global memory perhaps.

1:09:21  
And yeah, this will still take 6 characters, like 5 characters plus the trailing 0.

1:09:26  
But this data type is different than this.

1:09:31  
This is only a pointer.

1:09:33  
So if you take trying to take size of this variable, it will give you the size of the pointer.

1:09:39  
So it will be equivalent with the size of a character pointer, which on 64 bit architectures might be something like 8 because it's 8 byte wide.

1:09:48  
This pointer on 32 bit architecture, it will be probably 4, but it wouldn't give you the number of letters in here because it doesn't know it's just a pointer to the beginning.

1:09:59  
Here it really stores all the characters.

1:10:02  
The S variable represents the place in the memory where all the six bytes, all the six characters are stored in the memory.

1:10:10  
So that's the difference.

1:10:11  
This is just a pointer to the beginning.

1:10:13  
By the way, if you dereference it, you won't, you still won't get the right answer because it will give you the size of character.

1:10:19  
That's very unuseful information.

1:10:20  
It's it's always one, so beware this pitfall.

1:10:24  
This is also why you cannot take the size of a constant array inside a function if you just pass it down to the function.

1:10:31  
This is the pitfall you encountered in labs, right?

1:10:36  
At least one guy is nodding over there.

1:10:37  
I I just hope he's nodding and not just chilling out or listening to music.

1:10:42  
So beware this pitfall because it can bite you.

1:10:45  
By the way, if you really want to calculate how many characters are there in the string, there are functions for that.

1:10:51  
For instance, there is a standard function called sterlen string length which calculates how many bytes are there.

1:10:58  
It just traverses all the bytes until it finds a zero in the end and and gives you the number of characters it encounters.

1:11:07  
OK, a few words about C++ classes.

1:11:11  
In the end, as I told you, this course focuses mainly on the C part of the C++ language.

1:11:18  
So we are kind of focusing the subset.

1:11:21  
We are not trying to give you the bad, bad impressions.

1:11:24  
Because in regular C, well, I wouldn't say no one programs in regular CI know a guy or two, but it's rare if, unless you're doing something very low level like a programming a washing machine or programming a Linux kernel, you are not writing in C and you don't want to write in C, well, unless you're Martin Marish or someone.

1:11:47  
So basically we are not trying to give you bad impressions, but some, some pieces of this puzzle needs to be hidden till the 2nd year because we don't have time for everything.

1:11:58  
On the other hand, classes are very useful and you might want to use them in your lab assignments.

1:12:04  
So I will give you a like a quick tutorial to classes, just the stuff you need.

1:12:09  
There are way, way many more information I'm not telling you right now.

1:12:13  
OK, so classes are actually declared in the same way as structures, just you are you just using the keyword class instead of the keyword struct.

1:12:23  
And technically in C++ structures and classes are actually interchangeable.

1:12:28  
They are.

1:12:28  
They have too much in common.

1:12:30  
The only difference is in the visibility of the field.

1:12:33  
But.

1:12:33  
But you can still use structures instead of classes.

1:12:36  
Just don't.

1:12:37  
If you really want to use classes in the way that you intermix your data fields with your code, that's why we're doing classes.

1:12:45  
Then just use class, don't use structs.

1:12:49  
So the idea is that it's not not a good idea.

1:12:53  
Sorry, bad phrasing.

1:12:55  
The notion is it might not be a good idea to separate data from code when you're doing structures.

1:13:03  
You have data nicely bundled into a structure nicely packed, and then the code is somewhat separate.

1:13:10  
Let me just get back to the to this example.

1:13:14  
So yeah, I have this nice structure that represents a point in 2D and then somewhere else completely else, I have this function that somehow takes it and modifies it or produce another point or something.

1:13:28  
Not very useful.

1:13:29  
Well, the code itself is useful, but it's not very useful to get it separated, because if you get it separated, it's easy to lost track what's where.

1:13:39  
So it's always good to place the code and the data together or as as at least as close as possible.

1:13:45  
And that's what classes do because they intermit mix code and data structures, data fields.

1:13:51  
In this case.

1:13:53  
I have A1 integer in this class and I have some methods.

1:13:58  
Method is a function which is embedded in the class.

1:14:02  
It has the same syntax you you just write it in the same manner you are writing functions.

1:14:06  
So it has the return type identifier arguments, everything is the same.

1:14:11  
The difference is that the method can directly access the elements, the fields of this data structure of this class.

1:14:20  
So I can access the sum, which is some local data field from this function from this method actually.

1:14:30  
OK, so that that's the difference.

1:14:32  
I don't really understand what the third line they're doing.

1:14:35  
C I'm I'm able to getting into you're always ahead.

1:14:39  
I I like it because at least somebody's paying attention.

1:14:41  
Good for you.

1:14:42  
But please just give me a minute.

1:14:44  
I I will get to it in short order.

1:14:47  
OK, so these two are regular, regular methods.

1:14:52  
By the way, this constant says that this value, this function would not modify the contents of the class of the object of this class.

1:15:02  
OK, so this is just a detail for advanced people.

1:15:06  
And now finally I was I skipped this line.

1:15:10  
This special line is like a special function which is called a constructor.

1:15:15  
It's a one time function which is called only when the object of this class is instantiated.

1:15:21  
You already know what an object is right from from programming classes.

1:15:25  
You already make some objects from classes on in C# classes.

1:15:30  
Otherwise you couldn't code anything in C#.

1:15:32  
So I believe you already did that.

1:15:35  
So yeah, when you're creating an object from this class, this function, this constructor is called just once on the new instance, and its business is to initialise the fields for that.

1:15:49  
Yeah, you can see the constructor may have additional arguments, some don't.

1:15:54  
This is like a default constructor because it doesn't have any arguments.

1:15:58  
So it's it's easy because you can create an object without passing anything to the constructor.

1:16:04  
And this weird syntax is just before the body.

1:16:07  
Yeah, I can write in the body.

1:16:09  
I can write sum equals 0 and it will work.

1:16:12  
It will initialize the sum, the sum field of this newly created object to 0.

1:16:17  
That's what I need to do, because at the beginning I haven't summed in this counter anything yet.

1:16:23  
So it's a good idea to put zero as an initial value, otherwise there might be something else.

1:16:29  
Well, actually not in this case, but I I really need to initialize the values so I'm sure what's in there.

1:16:36  
And this sum is an initializer of.

1:16:39  
This is also a constructor.

1:16:41  
It's like you're calling a constructor on the integer.

1:16:45  
Yeah, I know it's hard to accept that you can call constructors on integers, and I'm not going into the details, let's leave it for the second, second year.

1:16:54  
But this is how you initialize all the fields using this double, using this column.

1:17:00  
And then there's a list of initializers which initialize internal fields.

1:17:04  
This is like a more modern and more practical way how to do do that instead of writing sum equals 0 into the body of the constructor.

1:17:12  
By the way, there is also AD structure, which is which is written as a tilt as a wave and and the name of the class.

1:17:21  
But again, this is something we won't need in this class.

1:17:24  
So I will leave it to the to the second second term.

1:17:29  
Why are we doing classes?

1:17:30  
Well, there are two one important reason and that's it's it's called encapsulation.

1:17:37  
Sorry, my lips are getting stiff.

1:17:42  
It's called encapsulation.

1:17:45  
You may notice that I've used some very awkward keywords like public and private over here and they denote which parts of the structure are visible from outside world.

1:17:59  
So let's just say I create an instance of this structure of this class.

1:18:03  
Sorry, the easiest way how to do this would be to write this class as a type and then identifier.

1:18:12  
And since it doesn't require any additional parameters because the constructor is empty, that's it.

1:18:19  
I just declare it as a variable and it will create.

1:18:22  
It will be created on the stack.

1:18:24  
I don't have to allocate it dynamically using new.

1:18:27  
In C# you need to write something like equals new and then there is also again the name of the class, right?

1:18:33  
In C you don't.

1:18:34  
This is how we allocate the class on the stack, and this is usually how you want to do that.

1:18:41  
So I will create an instance of C and the difference is if I want to access the add method here, it's fine.

1:18:50  
I can write inst dot add and just call whatever with call it with whatever parameters I'd like.

1:19:01  
I can, however the sum is private, which means if I try to write inst dot sum, which would be a normal way how to access this field from outside, it will fail the the compiler won't compile because it would say hey, you are trying to access the internal, the private property of this class.

1:19:30  
You're not allowed to do that and that's what's called encapsulation.

1:19:35  
Some parts of the class, usually the data parts, usually all the data structures are the data fields are private and usually all or most of the methods are public.

1:19:47  
So when anyone wants to access anything in this class, it is using only the allowed methods and cannot access the private data.

1:19:58  
Why is that important?

1:19:59  
It is very important because no one can accidentally slip in and override this value.

1:20:07  
Not accidentally, not because it will be a malicious programmer who wants to ruin my code.

1:20:12  
It will prevent errors from from accessing the bad data.

1:20:17  
Only the interface.

1:20:18  
We call it the interface that the methods that are public know how to access the internal data and allow you to.

1:20:27  
To do whatever this class is designed to do without directly accessing its underlying data.

1:20:34  
One of the reason why is this important is that you need to program your.

1:20:41  
You need to design your code against a specific interface.

1:20:45  
So you need to use these functions and these are like a contracts.

1:20:48  
You you are giving some assurances that these methods will be there for you no matter how the class is implemented internally.

1:20:58  
You shouldn't know how this method is implemented internally in order to use it.

1:21:04  
And if somebody decides that it's time to replace this class with another class which would be implemented differently, for instance more efficiently, or do something slightly else just on the same interface, it's fine.

1:21:17  
That's that's why we are doing this.

1:21:19  
If you didn't take everything, don't worry.

1:21:22  
This is a concept which is which, which will be repeated in many classes.

1:21:25  
This is like a very important concept and we will build upon that later.

1:21:30  
And actually in the fifth, sorry, in the yeah, in the 5th labs, you will be introduced to the fifth unforgivable curse.

1:21:39  
And it's about encapsulation.

1:21:40  
So you will be encouraged or slightly enforced to at least slightly embrace this concept.

1:21:47  
So separate your concerns, separate your code into classes and design appropriate interface for these classes.

1:21:54  
But don't worry about it.

1:21:55  
You will be guided through this process throughout this semester.

1:21:59  
OK, so this will be this was a very brief introduction into classes.

1:22:02  
Yeah, I've, I've leave a ton of things out of this because there are so many things in C++.

1:22:08  
But let me leave that to the second, to the second semester, OK, One thing which is quite important, right away you were introduced into the courses, right?

1:22:21  
Your lab teachers explain things and you already have been explained the first course.

1:22:26  
So, yeah, why we are calling this curse is because when we call that subtle suggestions, nobody would pay attention.

1:22:33  
So that's why we choose this very harsh label for that.

1:22:38  
But basically these are rules how to design A better code.

1:22:43  
These rules will be given for to you one by one in the labs, but all the rules are already here.

1:22:50  
And I suggest strongly that you just run through them.

1:22:54  
There are motivation explanations, there are examples, there are some additional information that might be interesting to you.

1:23:01  
So it's a very good idea just to read this stuff and understand this stuff even before you get to to to it in the classes because you know it will really take some time to sing in sink in this information.

1:23:15  
So it's better that you know everything in the beginning, not if if it's passed to 1 lecture at a time.

1:23:22  
OK, so let me finish because we are nearing the end.

1:23:27  
If you still crave Python, just remember, yeah, C++ can or C can be pain in the \*\*\*.

1:23:33  
But in the end this is in what we need to code because otherwise there would be no Python, there will be no JavaScript, you know run times of these languages and compilers are written in C or C++.

1:23:44  
So this is the language we will which we will still need in the future.

1:23:48  
OK, time for a few questions.

1:23:52  
But before I get to questions I have a like a three reminders.

1:23:55  
First one, don't forget to your get your arduinos right.

1:24:00  
Remember from the next week you will require to have your Arduinos in your labs.

1:24:04  
So get to the library or buy them if you'd like, but get your arduinos.

1:24:08  
Second, I've noticed that only about 1/2 of you, well I'm not sure about about you because you are intermixed with the check group, but only about half of the students are enlisted on matter most.

1:24:20  
So if you haven't enlist in the right channels on matter most, please do because this is the only communication channel we are using.

1:24:29  
Don't send us e-mail.

1:24:30  
We won't send you emails.

1:24:31  
We will send every important piece of information on matter most.

1:24:36  
So it's a very important for you to be there.

1:24:39  
And finally, I, I'm very pleased that despite the fact that I'm trying to record these lessons, many people have come to the second, second lecture.

1:24:51  
So maybe there is no point in in emphasizing it as long as you keep coming to the lectures.

1:24:57  
I will try to record them.

1:24:59  
If I notice that most people stopped visiting the lectures, then I'm not going to lecture for an empty hall.

1:25:07  
So if I'm not lecturing, there will be no recording.

1:25:09  
So as long as you're coming, there will be recording.

1:25:12  
So just please keep coming to the to the lectures.

1:25:15  
Don't rely on the recordings.

1:25:17  
Furthermore, you may have noticed that I have some troubles at the beginning.

1:25:20  
Get it running.

1:25:21  
So it might happen that at some point it won't.

1:25:23  
It will fail.

1:25:24  
It's possible that the recordings won't be made in in a good quality, so just don't rely on them.

1:25:31  
They are just a bonus that nothing that should be should be taken too seriously.

1:25:36  
OK, we have a few more minutes for questions.

1:25:39  
2 more minutes for questions.

1:25:40  
Are there any questions regarding C or C++ before we finish?

1:25:45  
Not today.

1:25:47  
OK, If you do have some technical issues with your home assignments, don't hesitate to ask your lab teachers or me if you, if you, if your lab teacher is available.

1:25:56  
So thank you for your attention and see you next week.