An Introduction to Programming through C++
Professor Abhiram G. Ranade
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Lecture No. 2 Part - 3
Problem Solving using Computer
Representing numbers on a computer

(Refer Slide Time: 0:41)

What we discussed

- Notion of problem solving, algorithms is old
- An algorithm specifies a sequence of operations
  - May contain arithmetic operations
  - May contain operations to be executed conditionally
  - May ask for sequence of operations to be repeated
- You already know many nice algorithms
  - Arithmetic on numbers, matrices, root finding...
  - Very useful for programming
- Programs are formal descriptions of algorithms, in specific languages.

In the previous segment we said that the notion of problem solving and algorithms is quite old and it predates computers. And that human beings have been solving numerical problems and using really sophisticated algorithms for a long long time. We said that an algorithm can contain operations which are precisely specified. They could be arithmetic operations, there can be conditional operations and there may be at some operation sequences how to be repeated. And it is important for you to remember throughout this course that you really know algorithms. You have learned algorithms throughout your schooling and what we want to do is simply translate them into programs. Translate what you know into this stylized syntax of a programming language.

(Refer Slide Time: 1:28)

How a computer solves (numerical) problems

Outline

- Digital circuits
- How numbers are represented
- Parts of a computer
- Machine language program, compilation.

So, in this segment I want to talk about how a computer works or how a computer solves numerical problems. So here is a quick outline of this segment. I am going to talk about circuits. In particular, I am going to talk about something called digital circuits. Then I will talk about how numbers are represented in circuits and then I will describe at a very higher level the parts of a computer. Then I will talk about a machine language program and what is the process called compilation that we have alluded to means.

(Refer Slide Time: 1:55)

Digital circuits: building blocks of computers

- Digital circuits: interpret electrical potential/voltage as numbers.
- Simplest convention
  - Voltage above 1 volt = number 1, Voltage between 0 and 0.2 volt = number 0
  - Circuit designed so that voltage will never be between 0.2 and 1 volt, hence no ambiguity.
- Current may also be used, e.g. current < some value represents 0...
- Charge stored on a capacitor may also denote numbers
  - Capacitor has low charge = number 0, High charge = number 1
  - Once charge is stored on a capacitor, it persists. "Memory"
- Once you can represent 0 and 1, you can represent anything:
- We can design circuits which perform arithmetic:
  - Circuit inputs: sets of voltages representing two numbers
  - Circuit outputs: set of voltages representing their sum!
  - Or product, or quotient ...

Okay. So, let us begin the digital circuits. Digital circuits are really the building blocks of computers. So, a digital circuit is simply a circuit in which we human beings interpret the voltages and currents as numbers. There are some voltages and there are some currents and we are going to interpret those as numbers. So here is a way in which we interpret, here is a simple convention. We may say that in a particular circuit, in a particular wire really, if the voltage go above 1 volt then we will think of it as the number 1. If it stays close to 0, then we will think of it as number 0.

Now, digital circuits are typically designed so that voltages will never be in between this 0.2 and 1. So, we will always be able to interpret a voltage as a number in a very unambiguous manner. So, from now on I might say something like the voltage is a 0 or the voltage represents the number 0 or there is the number 1 on this wire or there is the number 0 on this wire and that really should be thought of as that there is a voltage of 1 volt or higher than 1 volt on this wire or there is a voltage of roughly 0 volts on this wire.

We might also talk about currents. We might say that current smaller than some value represent 0, larger than some value represent 1. And we might say that the charge stored on a capacitor also denotes numbers. So, low charge may mean number 0, high charge may mean number 1. These are just conventions. Whatever the conventions we use, we have to stick to them entirely.

Now, note that capacitors are particularly interesting. Because once we store a charge on a capacitor, it can stay there, unless we connect the two end points of the capacitors. So, if the charge is stored because it can stay there, a capacitor is typically used as a memory element.

Now, I have told you how to represent 0's and 1's. But once you know how to represent 0's and 1's, you can represent anything really, and we will see this in a minute. Now, the point of all of this is we, and by that I mean electrical engineers, can design circuits which perform arithmetic in the following sense. So, we can design a circuit such that, if you feed a set of voltages to that circuit, which represents a certain set of numbers, then the circuit outputs, the voltages that come out of the circuit will precisely the numbers which represents the sum of the numbers fed on the input.

So, such circuits can be designed. It requires ingenuity, but people can do this. Not only sums, we can design circuits which perform multiplication, so they compute products, they perform quotients, they perform square roots, essentially whatever you want. So that is that is the reason why circuits work so beautifully for computation. So, we know how to represent 0's and 1's.

(Refer Slide Time: 5:50)

Standard representation for non-negative numbers

- Use many capacitors to store a single number
  - Typical number of capacitors used : 8, 16, 32, 64
- Example: To store 25 using 32 bits:
  - Represent 25 in binary.

25 Decimal = 00000000000000000000000000011001

- So store the following charge pattern (H=High, L=Low)

  LLLLLLLLLLLLLLLLLLLLLLLLLLLHHLLH
- Range stored: 0 to 11111111111111111111111111111111 = $2^{32} - 1$.
- If your numbers are likely to be larger, then use 64 bits.

So now, I am going to tell you how to represent non-negative numbers. Basically the idea is, we are going to use many capacitors or many wires to store or represent a single number okay. So,

the typical numbers of wires or capacitors we use are 8, 16, 32, 64, maybe even 128 numbers, maybe 96, 128 something like that, okay. So, how does it exactly work? So suppose I want to store 25 using 32 capacitors, or 32 wires or, in abstract terms we can say 32 bits. How do we do that?

Well, we are going to first convert 25 to binary. So, what is 25 into binary? It is this it is this last thing at the end, 11001. So, that is 25 in binary. But, we are going to use it using 32 bits. So we are going to have a total of 32 binary digits or bits over here. But of course the more significant bits are going to be all 0's. So this is the representation of 25 using the binary system. So, that is the first step that we are going to do. And if you want to store this value in a computer then we are going to store a charged pattern where H represents high or the number 1 and the L represents low or the number 0. So, we will have 32 capacitors and they will have the following pattern of charges stored.

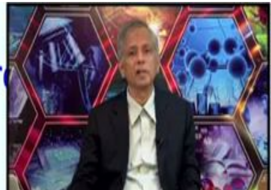
So, this capacitor will have the have low charge, low charge, low charge and essentially that will mean the number 0 will be stored this capacitor, this capacitor, this capacitor whereas some capacitors which have high charge representing the number 1, okay. Again, 2 low charges, high charge. So, this is how you will store 25 in our computer using 32 capacitors. Or, if we are going to send the data from one part of the computer to another part we will use a pattern of currents or maybe a pattern of voltages of this kind, low high depending upon whether the bit value of binary is 0 or 1.

Now, if we are restricting ourselves to using 32 bits, how large numbers can be stored? Well, we can have the pattern consisting of all 0's okay. So, all these numbers, all these bits are 0, then that will represent the number 0. Or we can have all these numbers be 1's, so that is going to be the largest number okay in binary. And in decimal that number is 2 to the power 32 minus 1. So, any number between 0 to 2 to the power 32 minus 1 can be represented using 32 bits or 32 capacitors or 32 wires. This is not a small number. This is something like about 10 digits. So, you can if you want a larger number than that, for some reason you need those numbers, then maybe you can use 64 bits, or maybe even more bits okay. Now, if you want to transmit numbers

then you have to send high or low voltages on as many wires. That is all. That is all that is needed.

(Refer Slide Time: 9:14)

Binary representation r



- Binary number $a_{n-1}a_{n-2}...a_1a_0 . a_{-1}a_{-2}...a_{-k}$
  - Example: 101.11
- Decimal value $v = \sum_i a_i 2^i$
  - $1*2^2 + 0*2^1 + 1*2^0 + 1*2^{-1} + 1*2^{-2} = 5.75$
- Converting a decimal integer v to binary
  - Divide v by 2, remainder gives a_0
  - Repeat previous step with the quotient to get a_1, a_2, ...
- Converting fraction f to binary
  - If f > 0.5, $a_{-1} = 1$

So, I have been talking about binary representation, but here is a quick revision of what binary representation really is: what binary representation is. So, binary number is a sequence of bits or binary digits. So, binary digit is 0 or 1. So, I might have a digit a bit a n minus 1, a n minus 2, a1, a0 and so on and in fact, there is a point there which now probably we should call a binary point and then the digits following that I have numbered minus 1, minus 2, minus k. In fact a subscript indicates the significance of the digit. As an example, I might have a binary number, 101.11.

So what is the decimal equivalent of it? The decimal value simply is, whatever that digit is multiplied by 2 to the power the same i. So for here, 0 will be used and this will be 2 raise to 0, so it will be 1, but in other place whatever the other things are used. So what is the value of 101.11? Well we just have to do exactly what I said.

So, this 1 is a2. So I am going to have 1 times 2 to the power 2. This 0 is a1. So I am going to have 0 times 2 to power 1. This 1 will have value 1 times 2 to the power 0. This 1 will have value 1 times 2 to the power 1, minus 1 and 2 to the power minus 2. So, this number is really 5.75. So in this way if you give me a binary number I can convert it into a decimal number. I can convert a decimal integer v to binary and here you may know that if I divide v by 2, the remainder gives me a0 and I can repeat the previous step with the quotient that I get when I

divide by 2. So the previous quotient I take and if I divide it again, then that will give me a1. If I divide one more time, then the remainder will give me a2 and so on. Converting fraction f to binary is sort of natural. So for example, you may note that if this fraction f is bigger than 0.5 then the first binary digit or the first bit after the binary point this will have to be 1. Then you can throw that, you can subtract that value. For example, there are many ways to do this. And if what remains is bigger than 1/4th, then the next bit will also be 1. Otherwise it will be 0 and so on.

So, this is just a very quick way, a quick introduction or a quick revision of binary representation. You really do not need to know carefully what happens to fractions. But atleast, I am sure you know how to deal with integers, binary integers and this is just a quick revision of that okay.

(Refer Slide Time: 12:41)

So, going back to representing numbers. So we talked about how positive integers can be represented. But suppose, you want to represent negative numbers, negative integers, how do you do that? Basically the idea is, one sort of idea is that one of the bits is used to indicate the sign. So let us say sign bit is equal to 0 means a positive number, equal to 1 means a negative number. So if I want store minus 25, I am going to do exactly what I had earlier. So this is the representation of the 25. But the most significant bit, that the first bit in this entire number, I am

going to make it a 1. So this does not mean 2 to the power 31 okay. It mean minus 1. So this is just a matter of convention now. So we have to decide some kind of convention and we have to stick to it. So if our convention is the first bit designates the sign, rather than being a part of the number, in that case this sequence of bits is going to represent minus 25. So here, we are saying that the left most bit is a sign bit. But once we do that, we know now that the largest number is going to be 0 followed by all 1's. So this is going to be at most 2 raise to be 31 minus 1. So the range has reduced a little bit, okay. So this is going to be our largest positive number. So, I can have the negative numbers also. So if I put in a one over here then that becomes a negative number.

So the number now starts with minus 2 to 31, whole thing minus 1 it should be, 2 to the 31 minus 1. The actual representation that computers use is not exactly like this. It is something called "Two's Complement". So we are not going to discuss that and these specific details are not really important, but you should know roughly what happens okay. So that you have a good idea of what is reasonable and what is not reasonable. So we will see why we are going through all this.

(Refer Slide Time: 14:51)

int nsides;

- C++ will typically designate some 32 capacitors in your computer for the variable nsides.
- The bit pattern stored in it will be interpreted as having a sign and a magnitude.

unsigned int nsides;

- 32 capacitors will be given, but the bit pattern in it will be interpreted as 32 bit binary number.
- 10000000000000000000000000011001 will mean 2^{31}+25

So in our very first lecture you wrote this statement int nsides. So when your program executes I want to tell you what is going to happen okay. So, C++ will typically designate some 32 capacitors in your computer for storing the variable nsides. So later on when we talk about nsides, the computer will refer to the specific capacitors, and the value stored in these capacitors, the big pattern that get stored the low high charges that get stored or the 0 1 values that get stored on each charge will be interpreted as a positive or a negative value.

So say the first the first charge or the first 1 or 0 will decide whether the number is negative or positive and the rest of it will decide what the magnitude of it is. So, that is how the representation works. You get some 32 capacitors and depending upon how you decide, once and for all, how the numbers are going to be stored, the patterns of bits stored in those capacitors will be interpreted according to the convention you fixed.

Now, C++ actually allows you to write the statement and unsigned int nsides as well. So in this case you will get 32 capacitors, but now you are telling your computer that look I am never going to store a negative number in this variable. So please interpret the entire 32 bits as the magnitude of the number that I stored. In this case if you store the number 1 followed by all of these things it will really mean 2 raised to 31 what this bit position represents plus 25. Whereas, if you are stored this pattern in a variable which is just declared int then it would be interpreted as minus 25. So, how you declare, how you define that variable, what type variable you say it is will tell the computer how to interpret those, interpret the value of the bit pattern stored in that variable or stored in those capacitors.

(Refer Slide Time: 17:32)

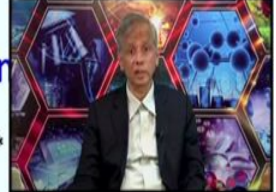
Bits, bytes, half-words,

- Bit = 1 binary "digit", (one number = 0 or 1
- byte = 8 bits
- half-word = 16 bits
- word = 32 bits
- double word = 64 bits

"one byte of memory" = memory capable of storing 8 bits = 8 capacitors.

Now, the terms bits, bytes, half-words, words are also used. I should just mention them once. A bit is 1 binary digit. So in other words just one number it can only be a 0 or 1. Byte is 8 bits. Half-word is 16 bits. Word is 32 bits and the double word is 64 bits. 1 byte of memory really means a memory capable of storing 8 bits or typically it means 8 capacitors. Of course there are ways of representing or remembering numbers without capacitors in which case it will mean some other kind of devices, maybe 8 other kinds of devices. But it is safe to think of memory as being capacitors for most part.

(Refer Slide Time: 18:32)



How do you represent real numbers? Because after all, we need to deal with real numbers. Temperatures, humidity all these things are real numbers, not integers. Here we are going to use the analog of scientific notation. So, what is the scientific notation? We write a real number as a part called a significand times 10 raised to an exponent. So for example, Avogadro's number is 6.022 into 10 raised to 23. So we are going to use exactly this on a computer as well, okay. But the significand and the exponent are in binary and so the number is going to be interpreted as significand times 2 raised to the exponent, okay instead of 10 raised to exponent, it is 2 raised to exponent. That is all. That is the reference.

So the question is, how many bits do you allocate for a significand and the exponent? So, this representation scheme called "single precision" says, that in my representation scheme I am going to store the significand using 24 bits and the exponent using 8 bits, okay. So one important point is when we are storing floating point numbers it really is in two parts. So there is a significand part and there is the exponent part and you may further note that the significand also might have a sign and the exponent might also have a sign. So, each of these things really are also in two parts okay.

Now, why are we choosing these numbers for single precision? Well turns out that these numbers are sort of good. They have been found to be good in practice, okay. And one other reason is 24 plus 8 is 32 which is one word. So, a single precision real number or a single

precision floating point number as it is called fits in one word. So I should also say that a 24 bits of significand really represent precision of 7 or 8 decimal digits, okay.

Then there is also something called “double precision”. So here you are not going to use one word but two words. So you will have 53 bits. You will dedicate 53 bits to store the significand and 11 bits to store the exponents. So you will have a bigger range in the exponents as well as more precision in the significand okay. So it is like scientific notation. How many digits you are going to write in the significand? The more digits you write, the more precise, the more accurate your representation is. And whether you are going to allow exponents to be larger is simply going to say how big numbers can you store. So it is exactly the same when it comes to binary as well, okay.

So a double precision number will fit in a double word and you should know 53 bits of significand are equivalent to 16 to 17 decimal digits. So that is really a huge amount of precision. So you, I do not think anybody really will require more precision than that. But, if they do, there are ways to get it, okay. The actual representation for single precision and double precision is more complex and it has more features as well and it is the so called “IEEE Floating Point Standard”, which your computer implements and which the C++ language also implements.

(Refer Slide Time: 22:14)

Indicative example

Let us represent Avogadro’s number 6.022×10^{23}

- Convert to binary: $1.11111110001010111 \times 2^{1001110}$
- Use 23 bits for magnitude of fraction, 1 bit for sign of fraction.
- Use 7 bits for magnitude of exponent, 1 bit for sign of exponent
- 01111111100010101110000 01001110
- Decimal point is assumed after 2nd bit.
- Inherently imprecise: fraction represented only to certain finite number of bits.
- IEEE Floating Point Representation: more complex.

So, just a very quick indicative example. So just to make these things more real, I am going to tell you how we might represent Avogadro's number. This is not exactly how Avogadro's number is represented on your computer. But this is indicative how it might be represented. So first we convert it to binary. So binary might be this, 6.022 might be this, and 10 raised to 23 might be that, so that number might be 2 raised to so many. So now, we are going to represent the significand using 23 bits of fraction okay. And 1 bit for the sign of the fraction. So 24 bits totally and 7 bits for the magnitude of exponent and 1 bit for the sign of the exponent, okay.

So, 2 raised to 1001110 so that is what I have replicated over here okay and I have put a sign bit. And this pattern I have stored over here and what I get over here is this pattern okay. Okay, so there is a 0 and so these are positive numbers and that is what this is. So the decimal point is assumed after the second bit. So there is assumed to be a decimal point over here. So, this is what it is. These black bits, what are these black bits? Well, they are bits which were not there but we needed 24 bits of significand and so these bits are just padded up as 0 bits.

Notice that this presentation is inherently imprecise and the fraction is only represented to a certain finite number of bits and in fact this is not really a property of binary system. Even in the decimal system, even in the standard scientific notation this problem is there. So nothing new really, okay. And as again I will point out that the actual representation is really more complex. So this is not IEEE Floating Point. This is just something similar to IEEE Floating Point which I have given over here just to tell you what it might look like. So the actual thing is more complex.

(Refer Slide Time: 24:56)

Exercise

- Suppose I want to represent 8 digit telephone numbers. I should use ___ (Fill in 8,16,32,64) bit _____ (Fill in signed or unsigned) representation.
- Which is bigger, byte or half-word?
- What is roughly the largest number that can be represented using 64 bits?
  - Hint: $2^{10} = 1024 \approx 10^3$

So, one quick thing for you to think about, so suppose I want to represent 8 digit telephone numbers. So which representation should I use? So should I use an 8 bit representation, 16 bit representation, 32 bit representation or a 64 bit representation? And should I use unsigned or signed? So you are expected to fill in the blanks over here. So what you are supposed to do is not use too many bits because you are using memory unnecessarily, but not use too few bits either because if you use too few bits then that number will not fit there.

Then these are just question for you to check whether you are paying attention. Which is bigger, byte or half-word? What is roughly the largest number that can be represented represent using 64 bits? And I want the answer in decimal. So here you might want to note that 2 raise to 10 is 1024 which is about 10 raise to 3. So this way you can calculate what 2 raise to 64 is very easily, at least approximately.

(Refer Slide Time: 25:54)

What we discussed

- Numbers are represented by sequence of 0s and 1s
- The same sequence may mean one number as an unsigned integer, a signed integer, or a floating point number
- The capacitors only store high or low charge, they are not "aware" that the charge represents numbers.
- So long as we remember what type of number we are storing, there will be no problem.
- As a user, you don't need type/read binary numbers.
  - C++ will convert binary numbers to decimal system while printing
  - C++ will accept numbers typed in decimal by you and itself convert it to binary for use on the computer.
  - But you should know (roughly) what range of numbers can be stored in k bit unsigned/signed/floating formats

Okay so, what have we discussed in this segment? We have discussed that numbers are represented in the sequence of 0's and 1's. And we discussed the same sequence may mean one number as an unsigned integer, a different number as a signed integer and a third number as a floating point interpreted as a floating point number, single or double precision floating point number.

Now it is important to note that the capacitors only store high or low charges. They are not aware in any sense that the charge represent the numbers. So we have to keep track. So, when we write a program, we designate a memory location as having a memory a variable as having type int. So that is when we are telling the computer that look, please remember that is what we stored in this has to be interpreted as a signed integer or if we call the type unsigned int, as an unsigned integer, okay.

And there are ways to specify the types so that the computer interprets it as a 32 bit floating point number or a 64 bit floating point number. So it is our responsibility to remember what type of number we are storing and if we remember it consistently then there will be no problem. I should also point out that please do not be scared you really do not need to know binary number system. So whatever you type has to in decimal and C++ will convert it to binary.

So you do not have to type binary. Do not get scared. And C++ will also do the other things. It will convert binary numbers that it has on the computer and print out decimal numbers. So both ways it will do it. So you can only think about the familiar decimal system, okay. But then, why are we teaching you this? We are teaching you this because you need to know roughly what is going on. You need to know that whatever you are storing has finite precision and how many digits of precision you are getting roughly.

So, if you are going to want to think about how much error my calculation containing, this discussion will be important for you. Okay. So you should know roughly what range of numbers and how precisely the numbers can be stored in different formats. So that concludes this segment. We will take a break.