Conquering C++

This is the first part of C++ (basics of programming and OOP)

Contents - PART I

Basic C++

Conquering C++

This is the first part of C++ (basics of programming and OOP)

Contents - PART I Basic C++

• Enumerated Data Types, Unions, Passing structures to functions

Workshop (for chapters 4, 5 and 6)

•

Test Yourself-II

•

More Questions (extra programs)

7.) Pointers

• Address of operator, Declaration and use of pointer

• Arithmetic operation on pointers, Pointers and Arrays

• Pass by value, Pass by reference (C and C++ style) and reference variables

• More on references

• Returning pointers from functions

• Memory Allocation (new and delete) and Memory Heap

• Pointers to functions, Pointer Declarations

• Pointers to Structures, Pointer to Pointer (multiple indirection)

• Pointer to constants and constant pointers, Void pointers

• Pointers to characters, Pointers Demystified (2-D arrays and pointers) Test Yourself

•

More questions (extra programs)

• Passing multidimensional arrays to functions, Functions with varying arguments

4.) Controlling Program Flow

- Loops (Iterations)
  - For loop and its variations
  - While, Do-while loop
- Decision statement
  - If...elseif..., Nested If, Conditional Operator
  - Switch Case
- Loop Control Statements
  - Break, Continue, Goto, Return

5.) Functions

- Declaration, call and definition, Parameters and arguments
- Default Arguments, Returning values from a function, Returning void, The int main( ) function
- An illustration of int main( )
- Types of functions, Using Library functions (rand, srand and time), Function overloading, An example using functions
- Recursion (calling oneself!), Inline Functions

8.) Classes and Objects

- Object Oriented Programming Concept and OOP Languages
- Class with example
- Constructors, Constructor with Parameters, Overloaded Constructors, Default Constructors and Arrays of Objects
- Scope Resolution Operator, Destructor, Objects and Functions (passing and returning objects), Initialializing an object using an object
- A Practical Use of Classes
- Data Encapsulation...Who are we hiding data from?
- More objects and classes, Private member functions (helper functions)
- Friend Function and Friend Classes, Static Class Members, Constant Objects (and the 'mutable' qualifier)
- Copy Constructor, this pointer, pointer to objects, objects in memory
- Initializer list, explicit keyword, declaration and definition, return value optimization

Workshop (for chapters 7 and 8)

- **Test Yourself-III**
- **More Questions** (extra programs)

If you are done with the basics go on to

Advanced C++ (PART - II)

8.) 4.) Controlling Program Flow

Classes and Objects

• Loops (Iterations)

• Object Oriented Programming o For loop and its variations

Concept and OOP Languages o While, Do-while loop

• Class with example

• Decision statement

• Constructors, Constructor with o If...elseif..., Nested If,

Parameters, Overloaded Conditional Operator

Constructors, Default Constructors o Switch Case and Arrays of Objects

• Loop Control Statements

• Scope Resolution Operator, o Break, Continue, Goto,

Destructor, Objects and Functions Return (passing and returning objects), Initialializing an object using an object

• A Practical Use of Classes 5.) Functions

• Data Encapsulation...Who are we hiding data from?

• Declaration, call and definition,

• More objects and classes, Private Parameters and arguments

member functions (helper functions)

• Default Arguments, Returning

• Friend Function and Friend Classes, values from a function, Returning

Static Class Members, Constant void, The int main( ) function

Objects (and the 'mutable' qualifier)

• An illustration of int main( )

• Copy Constructor, this pointer,

• Types of functions, Using Library

pointer to objects, objects in functions (rand, srand and time),

memory Function overloading, An example

• Initializer list, explicit keyword, using functions

declaration and definition, return

• Recursion (calling oneself!), Inline

value optimization Functions

Workshop (for chapters 7 and 8)

•

Test Yourself-III

•

More Questions

(extra programs)

Advanced If you are done C++ with the (PART basics go on - to

II)

Programming Languages

The earlier first section was Binary numbers. Some people said that I ought to reduce the math in this tutorial and I felt that perhaps I should start off differently. And so I've altered the sequence of sections in this first unit.

Recap of important computer concepts:

When we talk of computers, we refer to 'hardware' and 'software'. All the physical components like monitor, keyboard, mouse, modem, printer etc. fall under the hardware category. We need computers to do something useful for us but computers aren't smart enough to know what we want them to do (at least not yet!). We need to explicitly tell the computer our requirements in the form of instructions. A set of instructions for doing something useful forms a 'program'. And we refer to these programs as software (i.e. unlike hardware, software is something that we can't touch and feel).

Major hardware parts in a computer:

1. **Input and Output devices:** Computer can receive input from input devices (like a keyboard, mouse or a scanner). Output devices are used by the computer to send out information to the user; example: monitor, printer, plotter etc.

2. **Arithmetic and Logical Unit (ALU):** As the name implies this unit is responsible for all arithmetic calculations. It is part of the central processing unit.

3. **Memory:** When our computer wants to work on some information, it will require some place where it can store data; where it can store the instructions; where it can store intermediate results of operations etc. Memory serves this purpose and there are different types of memory (for different requirements):

Primary memory: This memory can be quickly accessed by the computer (in technical jargon we say that this memory has low access time; i.e. time taken to access data in primary memory is less). Generally, instructions and data needed by the computer immediately for processing are placed in primary memory. Data in primary memory is not permanent. Each time we restart the computer, the memory would get refreshed.

Secondary memory/ storage: This is the place where we store all our data. It's a long-term storage device (like a hard-disk). Access times are higher but secondary memory is cheaper than primary memory.

Programming Languages

The earlier first section was Binary numbers. Some people said that I ought to reduce the math in this tutorial and I felt that perhaps I should start off differently. And so I've altered the sequence of sections in this first unit.

Recap of important computer concepts:

When we talk of computers, we refer to ‘hardware’ and ‘software’. All the physical components like monitor, keyboard, mouse, modem, printer etc. fall under the hardware category. We need computers to do something useful for us but computers aren’t smart enough to know what we want them to do (at least not yet!). We need to explicitly tell the computer our requirements in the form of instructions. A set of instructions for doing something useful forms a ‘program’. And we refer to these programs as software (i.e. unlike hardware, software is something that we can’t touch and feel).

Major hardware parts in a computer:

1. Input and Output devices: Computer can receive input from input devices (like a keyboard, mouse or a scanner). Output devices are used by the computer to send out information to the user; example: monitor, printer, plotter etc.
2. Arithmetic and Logical Unit (ALU): As the name implies this unit is responsible for all arithmetic calculations. It is part of the central processing unit.
3. Memory: When our computer wants to work on some information, it will require some place where it can store data; where it can store the instructions; where it can store intermediate results of operations etc. Memory serves this purpose and there are different types of memory (for different requirements):

Primary memory: This memory can be quickly accessed by the computer (in technical jargon we say that this memory has low access time; i.e. time taken to access data in primary memory is less). Generally, instructions and data needed by the computer immediately for processing are placed in primary memory. Data in primary memory is not permanent. Each time we restart the computer, the memory would get refreshed.

Secondary memory/ storage: This is the place where we store all our data. It’s a long-term storage device (like a hard-disk). Access times are higher but secondary memory is cheaper than primary memory.

4. **CPU (Central Processing Unit):**

We've seen computer parts to get input, store information, display output and also to perform calculations. But there needs to be someone at a higher level to control the individual units; someone to decide when to capture the input, when to send output information to the monitor etc. The CPU takes up this responsibility and is termed the 'brain' of the computer. The CPU is also called the processor (a microprocessor chip).

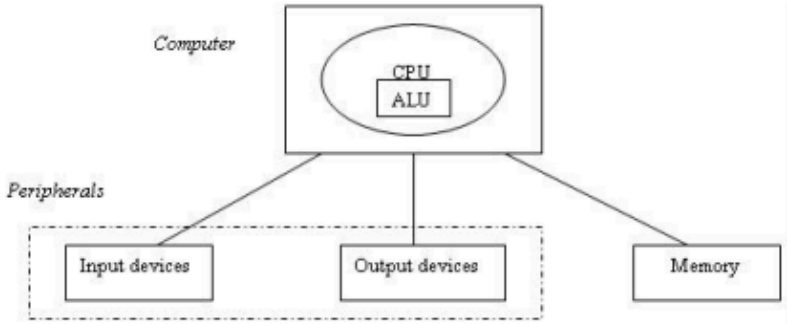

Fig. Basic components of a computer

The language computers understand:

We have a lot of languages in this world but all computers can understand only binary language. The vocabulary is very simple and small; it consists of only 2 things: 0 and 1. This is all that a computer understands. (Humans are comfortable with the decimal system - consisting of the numbers 0 to 9). We'll look at the binary system later in this chapter.

Remember: A computer finally needs everything in binary language (instructions and data).

Programming Languages:

Programs are written to tell the computer to do something useful for us. It might be as simple a task as adding two numbers or as complex as transferring data between 2 computers in a network.

There are several reasons why we need programs. Imagine searching through a stack of papers to search for some telephone bill. Imagine a company maintaining its accounts (expenses and income on a day-to-day basis) - if this was done using the traditional file and paper method, someone would have to keep entering all the details in sheets of paper and then stack them into files. If, in future, someone wants to find out when a particular product was sold they would have to manually read through each

4. CPU (Central Processing Unit):

We've seen computer parts to get input, store information, display output and also to perform calculations. But there needs to be someone at a higher level to control the individual units; someone to decide when to capture the input, when to send output information to the monitor etc. The CPU takes up this responsibility and is termed the 'brain' of the computer. The CPU is also called the processor (a microprocessor chip).

The language computers understand:

We have a lot of languages in this world but all computers can understand only binary language. The vocabulary is very simple and small; it consists of only 2 things: 0 and 1. This is all that a computer understands. (Humans are comfortable with the decimal system - consisting of the numbers 0 to 9). We'll look at the binary system later in this chapter.

Remember: A computer finally needs everything in binary language (instructions and data).

Programming Languages:

Programs are written to tell the computer to do something useful for us. It might be as simple a task as adding two numbers or as complex as transferring data between 2 computers in a network.

There are several reasons why we need programs. Imagine searching through a stack of papers to search for some telephone bill. Imagine a company maintaining its accounts (expenses and income on a day-to-day basis) - if this was done using the traditional file and paper method, someone would have to keep entering all the details in sheets of paper and then stack them into files. If, in future, someone wants to find out when a particular product was sold they would have to manually read through each

and every paper in each and every file. Computers can be programmed to perform such tasks and they will do it faster. Basically, the computer is very good in performing repetitive tasks. One point to note: man created computers and man programs computers to perform certain tasks. A program is a set of instructions that can be executed by the computer. The computer will obediently follow whatever the programmer instructs it to do (as long as it understands the instructions). The downside to this is that the computer does not have any special intelligence; it is only as intelligent as it is programmed to be. For instance, a robot can be programmed to walk. If it is not programmed to detect obstacles, the robot will simply bang into obstacles because it does not have the intelligence to recognize and avoid obstacles. If the programmer provides such provisions, then the robot will be able to handle such scenarios.

With many software programs already existing, you may wonder why do we need more software; why not just buy and use what already exists? Software is usually not custom designed for the requirements of a particular company and you may not even have software for your own special requirement. For example, dentists never used to have any software specifically for their use but now you can see a range of dental software (the dentist can maintain details of each of his patients' tooth in the computer). Programs needn't be as complicated as dental software; you may want to have a little software to calculate your CGPA or to calculate your rank in class. To perform such tasks a programmer has to write a program (or a set of instructions). These instructions are written in a specific programming language and the programmer can chose to write the instruction in any one of the available languages.

Evolution of Programming Languages:

Any programming language can be categorized into one of the following categories:

- High Level
- Middle Level
- Low Level (machine and assembly languages)

Middle level languages are sometimes clubbed together under the category of high-level languages.

Machine Level Languages:

A computer has a processor and if we want the computer to do something then we need to direct instructions at the processor. The problem is that computers can understand only binary language (i.e. the language which comprises of only 1s and 0s). All instructions to the processor should be in binary and so a programmer can write programs using a series of 1s and 0s. Every

and every paper in each and every file. Computers can be programmed to perform such tasks and they will do it faster. Basically, the computer is very good in performing repetitive tasks. One point to note: man created computers and man programs computers to perform certain tasks. A program is a set of instructions that can be executed by the computer. The computer will obediently follow whatever the programmer instructs it to do (as long as it understands the instructions). The downside to this is that the computer does not have any special intelligence; it is only as intelligent as it is programmed to be. For instance, a robot can be programmed to walk. If it is not programmed to detect obstacles, the robot will simply bang into obstacles because it does not have the intelligence to recognize and avoid obstacles. If the programmer provides such provisions, then the robot will be able to handle such scenarios.

With many software programs already existing, you may wonder why do we need more software; why not just buy and use what already exists? Software is usually not custom designed for the requirements of a particular company and you may not even have software for your own special requirement. For example, dentists never used to have any software specifically for their use but now you can see a range of dental software (the dentist can maintain details of each of his patients' tooth in the computer). Programs needn't be as complicated as dental software; you may want to have a little software to calculate your CGPA or to calculate your rank in class. To perform such tasks a programmer has to write a program (or a set of instructions). These instructions are written in a specific programming language and the programmer can chose to write the instruction in any one of the available languages.

Evolution of Programming Languages:

Any programming language can be categorized into one of the following categories:

- High Level
- Middle Level
- Low Level (machine and assembly languages)

Middle level languages are sometimes clubbed together under the category of high-level languages.

Machine Level Languages:

A computer has a processor and if we want the computer to do something then we need to direct instructions at the processor. The problem is that computers can understand only binary language (i.e. the language which comprises of only 1s and 0s). All instructions to the processor should be in binary and so a programmer can write programs using a series of 1s and 0s. Every

processor will understand only some instructions (this depends on how the processor was designed). An instruction to increment a variable might be:

110110100011101

Imagine having 100 such instructions in a program! The advantage is that no conversion (or translation) is needed because the computer can understand the 1s and 0s. This is called machine level language and this was used at the time computers came into existence.

The drawbacks are quite clear. By using machine language writing programs is very tedious and also prone to error (even if one bit is wrong, the entire program functionality could get altered). Trying to identify the error will also be a very tedious job (imagine reading a series of 1's and 0's trying to locate the mistake).

Assembly Languages:

Since it is very difficult for us to write instructions in binary language, assembly languages were developed. Instead of using a set of 1s and 0s for a particular instruction, the programmer could use some short abbreviations (called 'mnemonics') to write the program (for example: ADD is a mnemonic to add two numbers). Every processor has an instruction set that describes the various commands that the processor can understand. A normal instruction set will have instructions like JMP (jump), ADD (for addition) and so on. A programmer will write the program using these instructions and an assembler will convert the mnemonics into binary form so that the processor can understand it. The problem is that assembly level languages are specific to each processor. For example the 8085 (the simplest microprocessor) has an instruction set different from the 8086 microprocessor. Thus you would have to rewrite the program for each microprocessor. Another problem with assembly level programming is that the programmer needs to know details about the processor's architecture such as the internal registers where values can be stored, how many internal registers are available for use etc.

Low level languages are very close to the hardware but difficult for writing programs.

High Level Languages:

Writing larger programs in assembly language is quite difficult and hence, high-level languages were developed (like BASIC). These languages were not close to the actual computer hardware but were very close to the English language. They tried to simplify the process of programming. In these languages the programmer needn't worry about internal registers and processor architecture; instructions could be typed in almost normal English. The English-like instructions had to be converted to machine language using a compiler. One simple example of an English like command in COBOL is:

processor will understand only some instructions (this depends on how the processor was designed). An instruction to increment a variable might be:

110110100011101

Imagine having 100 such instructions in a program! The advantage is that no conversion (or translation) is needed because the computer can understand the 1s and 0s. This is called machine level language and this was used at the time computers came into existence.

The drawbacks are quite clear. By using machine language writing programs is very tedious and also prone to error (even if one bit is wrong, the entire program functionality could get altered). Trying to identify the error will also be a very tedious job (imagine reading a series of 1's and 0's trying to locate the mistake).

Assembly Languages:

Since it is very difficult for us to write instructions in binary language, assembly languages were developed. Instead of using a set of 1s and 0s for a particular instruction, the programmer could use some short abbreviations (called 'mnemonics') to write the program (for example: ADD is a mnemonic to add two numbers). Every processor has an instruction set that describes the various commands that the processor can understand. A normal instruction set will have instructions like JMP (jump), ADD (for addition) and so on. A programmer will write the program using these instructions and an assembler will convert the mnemonics into binary form so that the processor can understand it. The problem is that assembly level languages are specific to each processor. For example the 8085 (the simplest microprocessor) has an instruction set different from the 8086 microprocessor. Thus you would have to rewrite the program for each microprocessor. Another problem with assembly level programming is that the programmer needs to know details about the processor's architecture such as the internal registers where values can be stored, how many internal registers are available for use etc.

Low level languages are very close to the hardware but difficult for writing programs.

High Level Languages:

Writing larger programs in assembly language is quite difficult and hence, high-level languages were developed (like BASIC). These languages were not close to the actual computer hardware but were very close to the English language. They tried to simplify the process of programming. In these languages the programmer needn't worry about internal registers and processor architecture; instructions could be typed in almost normal English. The English-like instructions had to be converted to machine language using a compiler. One simple example of an English like command in COBOL is:

ADD incentive TO basic GIVING salary.

The instruction is quite self-explanatory (Add 'incentive' and 'basic' and store the result in 'salary'). COBOL is also a high-level language.

As programs grew larger (for example: maintaining a record of all employees, accounts maintenance etc.), even high level languages had certain drawbacks. These languages are also referred to as unstructured form of programming. The reason they are unstructured is because when a large program is written, it becomes very difficult to analyze the program at a later date (by someone else or even by the original programmer himself). There are many reasons for this; one is the use of statements like GOTO in high-level languages. In any program some tasks will either be performed repetitively or the programmer might want the program to execute a different set of instructions if some condition is satisfied. For example in a division program, in case the denominator is 0 then the program should not divide the two numbers; instead it should display an error message. Only if the denominator is not 0 should the program divide the two numbers. This kind of programming is referred to as program flow control. In a larger program there could be various other possibilities and all this was dealt with by using the GOTO statement. If you've used BASIC you will be aware that for every instruction that you type, you have to specify a line number (example 10,20,30 and so on). GOTO will be followed by some line number (indicating the instruction that has to be executed next. Ex: 1020 GOTO 50 means that now you want the program flow to go to line number 50).

The problem with this is that in a large program it will become very difficult for anyone to understand the program logic. Imagine reading through 100 lines and then finding one GOTO statement to the 200th line. Then on the 205th line there could be another GOTO statement directing you to the 150th line. This should make it clear as to what is meant by the term 'unstructured'. This is a major problem with high level languages. Another fact is that high level languages are very far away from the actual hardware. Examples of high-level languages are BASIC, Fortran (Formula Translation), COBOL (Common Business Oriented Language) and LIST (List processing).

Thus to take advantage of high level and low level languages, middle level languages like C and C++ were developed. They were easy to use and tried to retain the advantages of low level languages (i.e. being closer to the architecture).

Introduction

ADD incentive TO basic GIVING salary.

The instruction is quite self-explanatory (Add ‘incentive’ and ‘basic’ and store the result in ‘salary’).

COBOL is also a high-level language.

As programs grew larger (for example: maintaining a record of all employees, accounts maintenance etc.), even high level languages had certain drawbacks. These languages are also referred to as unstructured form of programming. The reason they are unstructured is because when a large program is written, it becomes very difficult to analyze the program at a later date (by someone else or even by the original programmer himself). There are many reasons for this; one is the use of statements like GOTO in high-level languages. In any program some tasks will either be performed repetitively or the programmer might want the program to execute a different set of instructions if some condition is satisfied. For example in a division program, in case the denominator is 0 then the program should not divide the two numbers; instead it should display an error message. Only if the denominator is not 0 should the program divide the two numbers. This kind of programming is referred to as program flow control. In a larger program there could be various other possibilities and all this was dealt with by using the GOTO statement. If you’ve used BASIC you will be aware that for every instruction that you type, you have to specify a line number (example 10,20,30 and so on). GOTO will be followed by some line number (indicating the instruction that has to be executed next. Ex: 1020 GOTO 50 means that now you want the program flow to go to line number 50).

The problem with this is that in a large program it will become very difficult for anyone to understand the program logic. Imagine reading through 100 lines and then finding one GOTO statement to the 200

th

line. Then on the 205

th

line there could be another GOTO statement directing

you to the 150

th

line. This should make it clear as to what is meant by the term ‘unstructured’. This is a major problem with high level languages. Another fact is that high level languages are very far away from the actual hardware. Examples of high-level languages are BASIC, Fortran (Formula Translation), COBOL (Common Business Oriented Language) and LIST (List processing).

Thus to take advantage of high level and low level languages, middle level languages like C and C++ were developed. They were easy to use and tried to retain the advantages of low level languages (i.e. being closer to the architecture).

Introduction

C and C++

Dennis Ritchie developed C and it was quite popular. An interesting feature in C is the use of functions. The programmer could write a function for checking whether a number is odd or even and store it in a separate file. In the program, whenever it is needed to check for even numbers, the programmer could simply call that function instead of rewriting the whole code. Usually a set of commonly used functions would be stored in a separate file and that file can be included in the current project by simply using the #include <filename> syntax. Thus the current program will be able to access all functions available in 'filename'. Programs written in C were more structured compared to high level languages and another feature was the ability to create your own data types like structures. For instance if you want to create an address book program, you will need to store information like name and telephone number. The name is a string of characters while the telephone number is an integer number. Using structures you can combine both into one unit. Similarly there are many more advantages of using C.

Though C seemed to be ideal, it was not effective when the programs became even more complex (or larger). One of the problems was the use of many functions (developed by various users) which led to a clash of variable names. Though C is much more efficient than BASIC, a new concept called Object Oriented Programming seemed better than C. OOP was the basis of C++ (which was initially called 'C with classes'). C++ was developed by Bjarne Strastroup. In object oriented programming, the programmer can solve problems by breaking them down into real-life objects (it presented the programmer with an opportunity to mirror real life). What is an object? This topic is dealt with extensively in the chapter on 'Objects and Classes' but a brief introduction is provided here.

Consider the category of cars. All cars have some common features (for example all cars have four wheels, an engine, some body colour, seats etc.). Are all cars the same? Of course not. A Fiat and a Ford aren't the same but they are called as cars in general. In this example cars will form a class and Ford (or Fiat) will be an object.

Introduction

C and C++

Dennis Ritchie developed C and it was quite popular. An interesting feature in C is the use of functions. The programmer could write a function for checking whether a number is odd or even and store it in a separate file. In the program, whenever it is needed to check for even numbers, the programmer could simply call that function instead of rewriting the whole code. Usually a set of commonly used functions would be stored in a separate file and that file can be included in the current project by simply using the #include <filename> syntax. Thus the current program will be able to access all functions available in 'filename'. Programs written in C were more structured compared to high level languages and another feature was the ability to create your own data types like structures. For instance if you want to create an address book program, you will need to store information like name and telephone number. The name is a string of characters while the telephone number is an integer number. Using structures you can combine both into one unit. Similarly there are many more advantages of using C.

Though C seemed to be ideal, it was not effective when the programs became even more complex (or larger). One of the problems was the use of many functions (developed by various users) which led to a clash of variable names. Though C is much more efficient than BASIC, a new concept called Object Oriented Programming seemed better than C. OOP was the basis of C++ (which was initially called 'C with classes'). C++ was developed by Bjarne Strastroup. In object oriented programming, the programmer can solve problems by breaking them down into real-life objects (it presented the programmer with an opportunity to mirror real life). What is an object? This topic is dealt with extensively in the chapter on 'Objects and Classes' but a brief introduction is provided here.

Consider the category of cars. All cars have some common features (for example all cars have four wheels, an engine, some body colour, seats etc.). Are all cars the same? Of course not. A Fiat and a Ford aren't the same but they are called as cars in general. In this example cars will form a class and Ford (or Fiat) will be an object.

For those people who know C programming, it would be useful to know the differences between C and C++. Basically C++ includes everything present in C but the use of some C features is deprecated in C++.

- C does not have classes and objects (C does not support OOP)
- Structures in C cannot have functions.
- C does not have namespaces (namespaces are used to avoid name collisions).
- The I/O functions are entirely different in C and C++ (ex: printf( ), scanf( ) etc. are part of the C language).
- You cannot overload a function in C (i.e. you cannot have 2 functions with the same name in C).
- Better dynamic memory management operators are available in C++.
- C does not have reference variables (in C++ reference variables are used in functions).
- In C constants are defined as macros (in C++ we can make use of 'const' to declare a constant).
- Inline functions are not available in C.

Let's recap the evolution of programming languages: initially programs were written in terms of 1s and 0s (machine language). The drawback was that the process was very tedious and highly error-prone. Assembly language was developed to write programs easily (short abbreviations were used instead of 1s and 0s). To make it even simpler for programmers, high level languages were developed (instructions were more similar to regular English). As the complexity of programs increased, these languages were found to be inadequate (because they were unstructured). C was developed but even that was not capable of dealing with complex or larger programs. This led to the development of C++.

Note: Sometimes languages are divided into low level and high level only. In such a classification, C/C++ will come under high level languages.

For those people who know C programming, it would be useful to know the differences between C and C++. Basically C++ includes everything present in C but the use of some C features is deprecated in C++.

• C does not have classes and objects (C does not support OOP)

• Structures in C cannot have functions.

• C does not have namespaces (namespaces are used to avoid name collisions).

• The I/O functions are entirely different in C and C++ (ex: printf( ), scanf( ) etc. are part of the C language).

• You cannot overload a function in C (i.e. you cannot have 2 functions with the same name in C).

• Better dynamic memory management operators are available in C++.

• C does not have reference variables (in C++ reference variables are used in functions).

• In C constants are defined as macros (in C++ we can make use of 'const' to declare a constant).

• Inline functions are not available in C.

Let's recap the evolution of programming languages: initially programs were written in terms of 1s and 0s (machine language). The drawback was that the process was very tedious and highly error-prone. Assembly language was developed to write programs easily (short abbreviations were used instead of 1s and 0s). To make it even simpler for programmers, high level languages were developed (instructions were more similar to regular English). As the complexity of programs increased, these languages were found to be inadequate (because they were unstructured). C was developed but even that was not capable of dealing with complex or larger programs. This led to the development of C++.

Note: Sometimes languages are divided into low level and high level only. In such a classification, C/C++ will come under high level languages.

Why do you need to learn C++?

There are many people who ask the question, "why should I learn C++? What use is it in my field?" It is a well-known fact that computers are used in all areas today. Programming is useful wherever computers are used because it provides you the flexibility of creating a program that suits your requirements. Even research work can be simulated on your computer if you have programming knowledge. Construction and programming may appear to be miles apart but even a civil engineer could use C++ programming. Consider the case of constructing a physical structure (like a pillar) in which the civil engineer has to decide on the diameter of the rods and the number of rods to be used. There are 2 variables in this case:

1. The number of rods needed (let's denote it as 'n') and
2. The diameter of each rod (let's call it as 'd')

The civil engineer might have to make a decision like: "Is it cost-effective for me to have 10 rods of 5cm diameter or is it better to have 8 rods of 6cm diameter?" This is just one of the simple questions he may have in his mind. There are a few related questions: "What is the best combination of number of rods, their diameters and will that combination be able to handle the maximum stress?"

Usually equations are developed for each of the factors involved. It would be much easier if the civil engineer could simply run a program and find out what is the best combination instead of manually trying out random values and arriving at a solution. This is where programming knowledge would benefit the engineer. Any person can write a good program if he has adequate knowledge about the domain (domain refers to the area for which the software is developed. In this case it is construction). Since the civil engineer has the best domain knowledge he would be able to write a program to suit his requirements if he knew programming.

Programming is applicable to almost every field- banking (for maintaining all account details as well as the transactions), educational institutions (for maintaining a database of the students), supermarkets (used for billing), libraries (to locate and search for books), medicine, electrical engineering (programs have been developed for simulating circuits) etc.

Why do you need to learn C++?

There are many people who ask the question, "why should I learn C++? What use is it in my field?" It is a well-known fact that computers are used in all areas today. Programming is useful wherever computers are used because it provides you the flexibility of creating a program that suits your requirements. Even research work can be simulated on your computer if you have programming knowledge. Construction and programming may appear to be miles apart but even a civil engineer could use C++ programming. Consider the case of constructing a physical structure (like a pillar) in which the civil engineer has to decide on the diameter of the rods and the number of rods to be used. There are 2 variables in this case:

1. The number of rods needed (let's denote it as 'n') and
2. The diameter of each rod (let's call it as 'd')

The civil engineer might have to make a decision like: "Is it cost-effective for me to have 10 rods of 5cm diameter or is it better to have 8 rods of 6cm diameter?" This is just one of the simple questions he may have in his mind. There are a few related questions: "What is the best combination of number of rods, their diameters and will that combination be able to handle the maximum stress?"

Usually equations are developed for each of the factors involved. It would be much easier if the civil engineer could simply run a program and find out what is the best combination instead of manually trying out random values and arriving at a solution. This is where programming knowledge would benefit the engineer. Any person can write a good program if he has adequate knowledge about the domain (domain refers to the area for which the software is developed. In this case it is construction). Since the civil engineer has the best domain knowledge he would be able to write a program to suit his requirements if he knew programming.

Programming is applicable to almost every field- banking (for maintaining all account details as well as the transactions), educational institutions (for maintaining a database of the students), supermarkets (used for billing), libraries (to locate and search for books), medicine, electrical engineering (programs have been developed for simulating circuits) etc.