**Brief History on C++**

**Resources**

C++ Module by D. S Malik

Docx documents too

www.course.com

Computer languages have undergone dramatic evolution since the first electronic computers were built to assist in telemetry calculations during World War II. Early on, programmers worked with the most primitive computer instructions: machine language. These instructions were represented by long strings of ones and zeroes. Soon, assemblers were invented to map machine instructions to human-readable and -manageable mnemonics, such as ADD and MOV.

In time, higher-level languages evolved, such as BASIC and COBOL. These languages let people work with something approximating words and sentences, such as Let I = 100. These instructions were translated back into machine language by interpreters and compilers. An interpreter translates a program as it reads it, turning the program instructions, or code, directly into actions. A compiler translates the code into an intermediary form. This step is called compiling, and produces an object file. The compiler then invokes a linker, which turns the object file into an executable program.

Because interpreters read the code as it is written and execute the code on the spot, interpreters are easy for the programmer to work with. Compilers, however, introduce the extra steps of compiling and linking the code, which is inconvenient. Compilers produce a program that is very fast each time it is run. However, the time-consuming task of translating the source code into machine language has already been accomplished.

Another advantage of many compiled languages like C++ is that you can distribute the executable program to people who don't have the compiler. With an interpretive language, you must have the language to run the program. For many years, the principle goal of computer programmers was to write short pieces of code that would execute quickly. The program needed to be small, because memory was expensive, and it needed to be fast, because processing power was also expensive. As computers have become smaller, cheaper, and faster, and as the cost of memory has fallen, these priorities have changed. Today the cost of a programmer's time far outweighs the cost of most of the computers in use by businesses. Well-written, easy-to-maintain code is at a premium. Easy- to maintain means that as business requirements change, the program can be extended and enhanced without great expense.

**Programs**

The word program is used in two ways: to describe individual instructions, or source code, created by the programmer, and to describe an entire piece of executable software. This distinction can cause enormous confusion, so we will try to distinguish between the source code on one hand, and the executable on the other.

**New Term:** A *program* can be defined as either a set of written instructions created by a programmer or an executable piece of software.

**Solving Problems**

The problems programmers are asked to solve have been changing. Twenty years ago, programs were created to manage large amounts of raw data. The people writing the code and the people using the program were all computer professionals. Today, computers are in use by far more people, and most know very little about how computers and programs work. Computers are tools used by people who are more interested in solving their business problems than struggling with the computer.

Ironically, in order to become easier to use for this new audience, programs have become far more sophisticated. Gone are the days when users typed in cryptic commands at esoteric prompts, only to see a stream of raw data. Today's programs use sophisticated "user-friendly interfaces," involving multiple windows, menus, dialog boxes, and the myriad of metaphors with which we've all become familiar. The programs written to support this new approach are far more complex than those written just ten years ago. As programming requirements have changed, both languages and the techniques used for writing programs have evolved. While the complete history is fascinating, this book will focus on the transformation from procedural programming to object-oriented programming.

**Procedural, Structured, and Object-Oriented Programming**

Until recently, programs were thought of as a series of procedures that acted upon data. A procedure, or function, is a set of specific instructions executed one after the other. The data was quite separate from the procedures, and the trick in programming was to keep track of which functions called which other functions, and what data was changed. To make sense of this potentially confusing situation, structured programming was created.

The principle idea behind structured programming is as simple as the idea of divide and conquer. A computer program can be thought of as consisting of a set of tasks. Any task that is too complex to be described simply would be broken down into a set of smaller component tasks, until the tasks were sufficiently small and self-contained enough that they were easily understood.

As an example, computing the average salary of every employee of a company is a rather complex task. You can, however, break it down into these sub-tasks:

**1.** Find out what each person earns.

**2.** Count how many people you have.

**3.** Total all the salaries.

**4.** Divide the total by the number of people you have.

Totaling the salaries can be broken down into

**1.** Get each employee's record.

**2.** Access the salary.

**3.** Add the salary to the running total.

**4.** Get the next employee's record.

In turn, obtaining each employee's record can be broken down into

**1.** Open the file of employees.

**2.** Go to the correct record.

**3.** Read the data from disk.

Structured programming remains an enormously successful approach for dealing with complex problems. By the late 1980s, however, some of the deficiencies of structured programming had become all too clear. First, it is natural to think of your data (employee records, for example) and what you can do with your data (sort, edit, and so on) as related ideas.

Second, programmers found themselves constantly reinventing new solutions to old problems. This is often called "reinventing the wheel," and is the opposite of re-usability. The idea behind re-usability is to build components that have known properties, and then to be able to plug them into your program as you need them. This is modelled after the hardware world--when an engineer needs a new transistor, she doesn't usually invent one, she goes to the big bin of transistors and finds one that works the way she needs it to, or perhaps modifies it. There was no similar option for a software engineer.

**New Term:** The way we are now using computers--with menus and buttons and windows--fosters a more interactive, event-driven approach to computer programming. *Event-driven* means that an event happens—the user presses a button or chooses from a menu--and the program must respond. Programs are becoming increasingly interactive, and it has become important to design for that kind of functionality.

**C++ and Object-Oriented Programming**

C++ fully supports object-oriented programming, including the four pillars of object-oriented development: encapsulation, data hiding, inheritance, and polymorphism. Encapsulation and Data Hiding When an engineer needs to add a resistor to the device she is creating, she doesn't typically build a new one from scratch. She walks over to a bin of resistors, examines the coloured bands that indicate the properties, and picks the one she needs. The resistor is a "black box" as far as the engineer is concerned--she doesn't much care how it does its work as long as it conforms to her specifications; she doesn't need to look inside the box to use it in her design.

The property of being a self-contained unit is called encapsulation. With encapsulation, we can accomplish data hiding. Data hiding is the highly valued characteristic that an object can be used without the user knowing or caring how it works internally. Just as you can use a refrigerator without knowing how the compressor works, you can use a well-designed object without knowing about its internal data members. Similarly, when the engineer uses the resistor, she need not know anything about the internal state of the resistor. All the properties of the resistor are encapsulated in the resistor object; they are not spread out through the circuitry. It is not necessary to understand how the resistor works in order to use it effectively. Its data is hidden inside the resistor's casing. C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. You'll see how to create classes on Day 6, "Basic Classes." Once created, a well-defined class acts as a fully encapsulated entity--it is used as a whole unit. The actual inner workings of the class should be hidden. Users of a well-defined class do not need to know how the class works; they just need to know how to use it. Inheritance and Reuse When the engineers at Acme Motors want to build a new car, they have two choices: They can start from scratch, or they can modify an existing model. Perhaps their Star model is nearly perfect, but they'd like to add a turbocharger and a six-speed transmission. The chief engineer would prefer not to start from the ground up, but rather to say, "Let's build another Star, but let's add these additional capabilities. We'll call the new model a Quasar." A

Quasar is a kind of Star, but one with new features.

C++ supports the idea of reuse through inheritance. A new type, which is an extension of an existing type, can be declared. This new subclass is said to derive from the existing type and is sometimes called a derived type. The Quasar is derived from the Star and thus inherits all its qualities, but can add to them as needed. Inheritance and its application in C++ are discussed on Day 12, "Inheritance," and Day 15, "Advanced Inheritance." Polymorphism. The new Quasar might respond differently than a Star does when you press down on the accelerator. The Quasar might engage fuel injection and a turbocharger, while the Star would simply let gasoline into its carburettor. A user, however, does not have to know about these differences. He can just "floor it," and the right thing will happen, depending on which car he's driving. C++ supports the idea that different objects do "the right thing" through what is called function polymorphism and class polymorphism. Poly means many, and morph means form. Polymorphism refers to the same name taking many forms, and is discussed on Day 10, "Advanced Functions," and Day 13, "Polymorphism."

**Development Tools**

If very few humans can (or want) to speak the machine language of the computers’ processors and software is expressed in this language, how has so much software been developed over the years?

Software can be represented by printed words and symbols that are easier for humans to manage than binary sequences. Tools exist that automatically convert a higher-level description of what is to be done into the required lower-level code. Higher-level programming languages like C++ allow programmers to express solutions to programming problems in terms that are much closer to a natural language like English. Some examples of the more popular of the hundreds of higher-level programming languages that have been devised over the past 60 years include FORTRAN, COBOL, Lisp, Haskell, C, Perl, Python, Java, and C#.

Most programmers today, especially those concerned with high-level applications, usually do not worry about the details of underlying hardware platform and its machine language. One might think that ideally such a conversion tool would accept a description in a natural language, such as English, and produce the desired executable code. This is not possible today because natural languages are quite complex compared to computer programming languages. Programs called compilers

that translate one computer language into another have been around for over 60 years, but natural language processing is still an active area of artificial intelligence research. Natural languages, as they are used by most humans, are inherently ambiguous. To understand properly all but a very limited subset of a natural language, a human (or artificially intelligent computer system) requires a vast amount of background knowledge that is beyond the capabilities of today’s software. Fortunately, programming languages provide a relatively simple structure with very strict rules for forming statements that can express a solution to any problem that can be solved by a computer.

Consider the following program fragment written in the C++ programming language:

**subtotal = 25;**

**tax = 3;**

**total = subtotal + tax;**

These three lines do not make up a complete C++ program; they are merely a piece of a program. The statements in this program fragment look similar to expressions in algebra. We see no sequence of binary digits. Three words, **subtotal**, **tax**, and **total**, called variables, are used to hold information.

Mathematicians have used variables for hundreds of years before the first digital computer was built.

In programming, a variable represents a value stored in the computer’s memory. Familiar operators (= and +) are used instead of some cryptic binary digit sequence that instructs the processor to perform the operation. Since this program is expressed in the C++ language, not machine language, it cannot be executed directly on any processor. A C++ compiler is used to translate the C++ code into machine code. The higher-level language code is called source code. The compiled machine language code is called the target code. The compiler translates the source code into the target machine language.

The beauty of higher-level languages is this: the same C++ source code can be compiled to different target platforms. The target platform must have a C++ compiler available. Minor changes in the source code may be required because of architectural differences in the platforms, but the work to move the program from one platform to another is far less than would be necessary if the program for the new platform had to be rewritten by hand in the new machine language. Just as importantly, when writing the program the human programmer is free to think about writing the solution to the problem in C++, not in a specific machine language. Programmers have a variety of tools available to enhance the software development process. Some common tools include:

• Editors. An editor allows the user to enter the program source code and save it to files. Most programming editors increase programmer productivity by using colours to highlight language features. The syntax of a language refers to the way pieces of the language are arranged to make well-formed sentences. To illustrate, the sentence

The tall boy runs quickly to the door, uses proper English syntax. By comparison, the sentence

Boy the tall runs door to quickly the, is not correct syntactically. It uses the same words as the original sentence, but their arrangement does not follow the rules of English. Similarly, programmers must follow strict syntax rules to create well-formed computer programs.

Only well-formed programs are acceptable and can be compiled and executed. Some syntax-aware

Editors can use colours or other special annotations to alert programmers of syntax errors before the program is compiled.

**Polymorphism in C++**

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.