Language Paradigms Carol Zander

Imperative or structured/procedural programming -- Process-oriented.

Based on the von Neumann architecture where data and programs are stored in the same memory. The CPU is separate from the memory, so instructions and data must be piped from memory to the CPU. Results of operations are moved back to memory.

Because of this architecture, the central features are:

- variables (model the memory cells)
- assignment statements (based on piping operation)
- iteration form of repetition (most efficient method).

Decompose a problem algorithmically. Each module in the solution denotes a major step in the overall process.

Object-oriented -- Data-oriented.

It grew out of structured programming, but supported data abstraction which encapsulates processing with data objects, hiding access to data. Concentrates on the use of abstract data types to solve problems.

Decompose a problem by the key abstractions in the problem domain. The key abstractions become the objects. An object is a tangible entity exhibiting some well-defined behavior. Objects do things that we ask by sending them messages.

What is the difference? The algorithmic view highlights the ordering of events, and the object-oriented view emphasizes the agents that either cause action or are the subjects upon which these operations act.

Functional -- Primary means of computing is by applying functions to given parameters.

Logic programming -- Rule-based. Rules are specified in no particular order and the language implementation figures out a result.

Basics of a Typical C++ Environment (assume source code, a program, is created)

Preprocessor – Preprocessor program processes preprocessor code. It executes automatically before the compiler stage. It obeys special commands called preprocessor directives which indicate that certain manipulations are to be performed on the program before compilation. These manipulations typically consist of including other files or telling the compiler some definition code may be repeated so don't store it twice.

E.g. Usually in files defining classes, **#ifndef** tells the compiler to do what follows if the definition is not already in the symbol table under the given name (huge data structure of every identifier in your program and everything the compiler needs to know about it). The **#define** says to define it associated with the given name as follows in the code. The **#endif** terminates the #ifndef directive.

Compiler – Compiler creates object code and stores it on disk. The compiler is a special type of translator. It translates source code into machine language code (same as object code). **Errors:** Eventually compiler errors become no big deal. You leave off a semicolon, misspell, forget a parenthesis, mess up your curly braces, etc.

Linker – Linker links the object code with the libraries, creates an executable file and stores it on disk. When you refer to something in a library, the compiler leaves a *hole* and the linker must resolve the addressing; in other words, find it to create a complete executable file. **Errors:** If you get a linking error, it can't find something. Usually it's a typo. Function signatures (the header) etc., must be identical. Also, your #includes may be wrong.

Loader – Loader puts the program in memory so the CPU can take each instruction and execute it. It will likely store new data values as the program executes.

What is the difference between a compiler and interpreter? While a compiler translates high level language into machine code to later be loaded and executed, an interpreter takes the high level code and immediately carries out the instruction. There are also hybrid systems where high level code is translated into intermediate code (e.g., Java byte code) and then the instruction is performed by an interpreter.