Chapter 2: Issues and Overview

2.1 Why did C need a ++?

For application modeling. Shakespeare once explained it for houses, but it should be true of programs as well:

When we mean to build, we first survey the plot then draw the model.
...Shakespeare, King Henry IV.

2.1.1 Design Goals for C

C was designed to write Unix. C is sometimes called a "low level" language. It was created by Dennis Ritchie so that he and Kenneth Thompson could write a new operating system that they named Unix. The new language was designed to control the machine hardware (clock, registers, memory, devices) and implement input and output conversion. Thus, it was essential for C to be able to work efficiently and easily at a low level.

Ritchie and Thompson worked with small, slow machines, so they put great emphasis on creating an simple language that could be easily compiled into efficient object code. There is a direct and transparent relationship between C source code and the machine code produced by a C compiler.

Because C is a simple language, a C compiler can be much simpler than a compiler for C++ or Java. As a result, a good C compiler produces simple error comments tied to specific lines of code. Compilers for full-featured modern languages such as C++ and Java are the opposite: error comments can be hopelessly wordy and also vague. Often, they do not correctly pinpoint the erroneous line.

Ritchie never imagined that his language would leave their lab and become a dominant force in the world and the ancestor of three powerful modern languages, C++, C#, and Java. Thus, he did not worry about readability, portability, and reusability. Because of that, readability is only achieved in C by using self-discipline and adhering to strict rules of style. However, because of the clean design, C became the most portable and reusable language of its time.

In 1978, Brian Kernighan and Dennis Ritchie published "The C Programming Language", which served as the only language specification for eleven years. During that time, C and Unix became popular and widespread, and different implementations had subtle and troublesome differences. The ANSI C standard (1989) addressed this by providing a clear definition of the syntax and the meaning of C. The result was a low-level language that provides unlimited opportunity for expressing algorithms and excellent support for modular program construction. However, it provides little or no support for expressing higher-level abstractions. We can write many different efficient programs for implementing a queue in C, but we cannot express the abstraction "queue" in a clear, simple, coherent manner.

2.1.2 C++ Extends C.

C++ is an extension and adaptation of C. The designer, Bjarne. Stroustrup, originally implemented C++ as a set of macros that were translated by a preprocessor into ordinary C code. His intent was to retain efficiency and transparency and simultaneously improve the ability of the language to model abstractions. The full C language remains as a subset of C++ in the sense that anything that was legal in C is still legal in C++ (although some things have a slightly different meaning). In addition, many things that were considered "errors" in C are now legal and meaningful in C++.

Flexibility. Work on C++ began in the early days of object-oriented programming. The importance of types and type checking was understood, but this led to a style of programming in which the type of everything had to be known at compile time. (This is sometimes called "early binding"). Languages such as Lisp did not do compile-time type checking, they deferred it until run-time (late binding).

Object oriented languages operate on a middle ground. Types are not isolated from each other. Polymorphic class hierarchies can be defined with a base class and several derived subclasses. A derived class inherits data and function members from its base class. Data members no longer have unique types – they simultaneously have all the types above them in the inheritance hierarchy. A function no longer has a single unified definition – it is a collection of one or more function methods, possibly in many classes, that operate on different combinations of parameter types. An OO language checks at compile time that the arguments to each function call have the appropriate base types for at least one of the methods of the function. However, a final choice of which function method to use to execute the function call is sometimes deferred until run-time, when the subtype of the object is checked and matched against the parameter lists of the available function methods.

Thus, an OO language uses late binding for polymorphic types and early binding for simple types. This allows a combination of flexibility (where needed) and efficiency (everywhere else) that is not achievable in a language with only late binding.

Readability. C++ was no more readable than C, because C was retained as the basic vehicle for coding in C++ and is a proper subset of C++ . However, an application program, as a whole, may be much more readable in C++ than in C because of the new support for application modeling.

Portability. A portable program can be "brought up" on different kinds of computers and produce uniform results across platforms. By definition, if a language is fully portable, it does not exploit the special features of any hardware platform or operating system. It cannot rely on any particular bit-level representation of any object, operation, or device; therefore, it cannot manipulate such things. A compromise between portability and flexibility is important for real systems.

A program in C or C++ can be very portable if the programmer designs it with portability in mind and follows strict guidelines about segregating sections of code that are not portable. Skillful use of the preprocessor and conditional compilation can compensate for differences in hardware and in the system environment. However, programs written by naive programmers are usually not portable because C's most basic type, `int`, is partially undefined. Programs written for the 4-byte integer model often malfunction when compiled under 2-byte compilers and vice versa. C++ does nothing to improve this situation.

Reusability. Code that is filled with details about a particular application is not very reusable. In C, the typedef and #define commands do provide a little bit of support for creating generic code that can be tailored to a particular situation as a last step before compilation. The C libraries even include two generic functions (`qsort()` and `bsearch()`) that can be used to process an array of any base type. C++ provides much broader support for this technique and provides new type definition and type conversion facilities that make generic code easier to write.

Teamwork potential. C++ supports highly modular design and implementation and reusable components. This is ideal for team projects. The most skilled members of the group can design the project and implement any non-routine portions. Lesser-skilled programmers can implement the routine modules using the expert's classes, classes from the standard template library, and proprietary class libraries. All these people can work simultaneously, guided by defined class interfaces, to produce a complete application.

2.1.3 Modeling.

The problems of modeling are the same in C and C++. In both cases the questions are, what data objects do you need to define, how should each object work, and how do they relate to each other? A good C programmer would put the code for each type of object or activity in a different file and could use type modifiers `extern` and `static` to control visibility. A poor C programmer, however, would throw it all into one file, producing an unreadable and incomprehensible mess. Skill and style make a huge difference in C . In contrast, C++ provides classes for modeling objects and several ways to declare or define the relationship of one class to others.

What is a model? A *model of an object* is a list of the relevant facts about that object in some language. A *low level* model is a description of a particular implementation of the object, that specifies the number of parts in the object, the type of each part, and the position of each part in relation to other parts. C supports only low level models.

C++ also supports *high-level* or *abstract* models, which specify the functional properties of an object without specifying a particular representation. This high-level model must be backed up by specific low-level definitions for each abstraction before a program can be translated. However, depending on the translator used and the low-level definitions supplied, the actual number and arrangement of bytes of storage that will be used to represent the object may vary from translator to translator.

A high level model of a *process* or *function* specifies the pre- and post-conditions without specifying exactly how to get from one to the other. A low level model of a *function* is a sequence of program definitions, declarations, and statements that can be performed on objects from specific data types. This sequence must start with objects that meet the specified pre-conditions and end by achieving the post-conditions.

High level process models are not supported by the C language but do form an important element of project documentation. In contrast, C++ provides class hierarchies and virtual functions which allow the programmer to build high-level models of functionality, and later implement them.

Explicit vs. Implicit Representation. Information expressed explicitly in a program may be used by the language translator. For example, the type qualifier `const` is used liberally in well-written C++ applications. This permits the compiler to verify that the variable is never modified directly and is never passed to a function that might modify it.

A language that permits explicit communication of information must have a translator that can identify, store, organize, and utilize that information. For example, if a language permits programmers to define types and relationships among types, the translator needs to implement type tables (where type descriptions are stored), new allocation methods that use these programmer-defined descriptions, and more elaborate rules for type checking, type conversions, and type errors. This is one of the reasons why C++ translators are bigger and slower than C translators. The greater the variety and sophistication of the information that is declared, the more effort it is to translate it into low-level code that carries out the intent of the declarations.

Semantic Intent A data object (variable, record, array, etc.) in a program has some intended meaning that can be known only if the programmer communicates or declares it. A programmer can try to choose a meaningful name and can supplement the code by adding comments that explain the intent, but those mechanisms communicate only to humans, not to compilers. The primary mechanism for expressing intent in most languages is the data type of an object. Some languages support more explicit declaration of intent than others. For example, C uses type `int` to represent many kinds of objects with very different semantics (numbers, truth values, characters, and bit masks). C++ is more discriminating; truth values are represented by a semantically distinct type, `bool`, and programmer-defined enumerations also create distinct types.

A program has *semantic validity* if it faithfully carries out the programmer's semantic intent. A language is badly designed to the extent that it lets the programmer write code that has no reasonable valid interpretation. A well-designed language will identify such code as an error. A strong type checking mechanism can help a programmer write a semantically valid (meaningful) program. Before applying a function to a data object, both C and C++ translators test whether the call is meaningful, that is, the function is defined for objects of the given type. An attempt to apply a function to a data object of the wrong type is identified as a semantic error at compile time, unless the data value can be converted to the correct type.

However, the type system and the rules for translating function calls are much more complex in C++ than in C for the reasons discussed in Section 2.3. For these reasons and others, achieving the first error-free compile is more difficult in C++, but the compiled code is more likely to run correctly.

2.2 Object Oriented Principles.

Classes. The term "object-oriented" has become popular, and "object-oriented" analysis, design, and implementation has been put forward as a solution to several problems that plague the software industry. OO analysis is a set of formal methods for analyzing and structuring an application from the application data's perspective, as opposed to the traditional functional or procedural point of view. The result of an OO analysis is an OO design. OO programs are built out of a collection of modules, often called *classes* that contain both function methods and data. Classes define data structures and the operations that can be performed on them.Access to all of the data and some of the method elements should only be through the defined methods of the class.

The way a language is used is more important in OO design than which language is used. C++ was designed to support OO programming[1]; it is a convenient and powerful vehicle for implementing an OO design. However, with somewhat more effort, that same OO design could also be implemented in C.[2] Similarly, a non-OO program can be written in C++.

Principles central to object-oriented programming are encapsulation, locality, coherent representation, and generic or polymorphic functions.

Encapsulation. The most fundamental OO design principle is that a class should take care of itself. Typically, a class has both data members that may be constants or variables. The class also includes function members that operate on the data members. The OO principle of encapsulation says that *only* a function inside a class should ever change the value of a data member. This is achieved by declaring member variables to be `private`. A member function can then freely use any data member, but outside functions cannot.

Functions that are intended for internal use only are also made `private`. Functions that are part of the function's published interface are made `public`. Finally, constant data members are sometimes made public.

In C++, class relationships can be declared, and the simple guidelines for `public` and `private` visibility are complicated by the introduction of `protected` visibility and of class `friendship`.

Initialization and Cleanup. One way a class takes care of itself is to define how class objects should be initialized. Initialization is done by *constructors*, which are like functions except that they have no return type. The name of the constructor is the same as the class name. A constructor is called automatically whenever a class object is declared or dynamically allocated. It uses its parameters to initialize the class's data members. A class often has multiple constructors, allowing it to be initialized from various combinations of arguments. A constructor might also dynamically allocate and parts of the class object.

Cleanup, in C++, is done by *destructors*. Each class has exactly one destructor that is called when the class object is explicitly freed or when it goes out of scope at the end of the code block that declares it. The name of the destructor is a tilde ($\sim$) followed by the class name. The job of a destructor is to free dynamically allocated parts of the class object.

Locality. The effects of an action or a declaration can be global (affecting all parts of a program) or local (affecting only nearby parts). The further the effects of an action reach in time (elapsed during execution) or in space (measured in pages of code), the more complex and harder it is to debug a program. The further an action has influence, the harder it is to remember relevant details, and the more subtle errors seem to creep into the code.

A well-designed language supports and encourages locality. All modern languages permit functions to have local variables and minimize the need for global variables. C goes farther than many language by supporting static local variables that have the lifetime of a global object but only local visibility. OO languages go further still by introducing "private" fields in structures (class objects) that cannot be seen or changed by functions outside the class. Again, we say that the private members are *encapsulated* within the class.

Coherent vs. Diffuse Representation. A representation is *coherent* if an external entity (object, idea, or process) is represented by a single symbol in the program (a name or a pointer) so that it may be used, referenced, or manipulated as a unit. A representation is *diffuse* if various parts of the representation are known by different names, and no one name or symbol applies to the whole.

Coherence is the most important way in which object-oriented languages differ from older languages. In Pascal, for example, a programmer can declare a new data type and write a set of functions to operate on objects of that type. Taken together, the data objects and functions implement the programmer's model. However, Pascal does not provide a way to group the data and functions into a coherent package, or declare that they

[1] C++ is not the first or or only OO language. Earlier OO languages such as SIMULA-67 (from Dahl and Nygaard) and Smalltalk (from Alan Kay) embodied many OO concepts prior to Stroustrup's C++ definition in 1990.

[2] The insertion sort code example is an OO design implemented in C that illustrates locality, coherent representation, and reusable generic code.

form a meaningful module. C is a little better in this respect because separately-compiled code modules allow the programmer to group related things together and keep both functions and data objects private within the module. In an OO language, however, the class mechanisms do this and more, and provide a convenient syntax for declaring classes and class relationships.

Generic code. A big leap forward in representational power was the introduction of generic code. A generic function is one like "+" whose meaning depends on the type of the operands. Floating-point "+" and integer "+" carry out the same conceptual operation on two different representations of numbers. If we wish to define the same operation (such as "print") on five data types, C forces us to introduce five different function names. C++ lets us use one name to refer to several methods which, taken together, comprise a function. Sometimes, the same symbol is used for purposes that are only distantly related to the original. For example, the + symbol is used to concatenate strings together. (The result of `"abc" + "xyz"` is `"abcxyz"`.

(In C++ terminology, a single name is "overloaded" by giving it several meanings.) The translator decides which definition to use for each function call based on the types of the arguments in that call. This makes it much easier to build libraries of functions that can be used and reused in a variety of situations.

OO Drawbacks. Unfortunately, a language with OO capabilities is also more complex. The OO extensions in C++ make it considerably more complicated than C. It is a massive and complex language. To become an "expert" requires a much higher level of understanding in C++ than in C, and C is difficult compared to Pascal or FORTRAN. The innate complexity of the language is reflected in its translators; C++ compilers (like Java compilers) are slower and can give very confusing error comments because program semantics can be more complex.

The ease with which one can write legal but meaningless code is a hallmark characteristic of C. The C programmer can write all sorts of senseless but legal things (such as a<b<c). C++ has a better-developed system of types and type checking, which improves the situation somewhat. However C++ also provides powerful tools, such as the ability to add new definitions to old operators, that can easily be overused or misused. A good C++ programmer designs and writes code in a strictly disciplined style, following design guidelines that have evolved from experience over the years. Learning these guidelines and how to apply them is more important than learning the syntax of C++, if the goal is to produce high-quality, debugged, programs.

2.3 Important Differences

- **Comments.** Comments can begin with // and end with newline. Please use only this kind of comment to write C++ code. Then the old C-style kind of comments can be used to /* comment out */ larger sections of text during debugging.

- **Executable declarations.** Declarations can be mixed in with the code. This makes it possible to print greeting messages before processing the declarations. Why is this useful? Because C++ declarations can trigger file processing and dynamic memory allocation and it is VERY helpful to precede each major declaration with a message that will let the programmer track the progress of the program.

  When you put a declaration in a loop, the object will be allocated, initialized, and deallocated every time around the loop. This is unnecessary and inefficient for most variables but it can be useful when you need a strictly temporary object of a class type.

  Declarations must not be written inside one `case` of a `switch` statement unless you open a new block (using curly braces) surrounding the code for the case.

- **A new kind of *for* loop.** Starting with C++11, an additional kind of `for` loop is available. This loop syntax can be used to iterate through an array or any kind of data structure with multiple elements that is supported by the C++ Standard Template Library (STL). The loop below adds 1 to each component of an array and prints the result. In this code fragment, `ar` is an array of 10 `float`s.

```
for (float f : ar) {
    f += 1;
```

```
        cout << f;
    }
```

Each time around this loop, the name `f` gets bound to the next slot of the array. This makes it possible to both read and modify the array element, as in the above example.

- **Type identity.** In C, the type system is based on the way values are *represented*, not on what they *mean*. For example, pointers, integers, truth values, and characters are all represented by bitstrings of various lengths. Because they are represented identically, an `int` variable can be used where a `char` value is wanted, and vice versa. Truth values and integers are even more closely associated: there is no distinction at all. Because of this, one of the most powerful tools for ensuring correctness is compromised, and expressions that should cause type errors are accepted. (Example: `k < m < n`.)

  In C++, as in all modern languages, type identity is based on `meaning`, not representation. Thus, truth values and integers form distinct types. To allow backwards compatibility, automatic type coercion rules have been added. However, new programs should be written in ways that do not depend on the old features of C or the presence of automatic type conversions. For example, in C, truth values were written as 0 and 1, In C++, they are `false` and `true`.

- **Type bool.** In standard C++ , type `bool`, whose values are named `false` and `true`, is not the same type as type `int`. It is defined as a separate type along with type conversions between `bool` and `int`. The conversions will be used automatically by the compiler when a programmer uses type `int` in a context that calls for type `bool`. However, good style demands that a C++ programmer use type `bool`, not `int` and the constants true (not 1) and false (not 0), for true/false situations.

- **Enumerated types.** Enumerated type declarations were one of the last additions to the C language prior to standardization. In older versions of the language, `#define` statements were used to introduce symbolic codes, and a program might start with dozens of `#define` statements. They were tedious to read and write and gave no clue about which symbols might be related to each other. Enumerations were added to the language to provide a concise way to define sets of related symbols. For example, a program might contain error codes and category codes, all used to classify the input. Using `#define`, there is no good way to distinguish one kind of code from the other. By using two enumerations, you can give a name to each set and easily show which codes belong to it.

  In C, enumeration symbols are implemented as integers, not as a distinct, identifiable type. Because of this, the compiler does not generate type errors when they are used interchangeably, and many C programmers make no distinction. In contrast, in C++, an enumeration forms a distinct type that is not the same as type `int` or any other enumeration. Compliers *will* generate error and warning comments when they are used inappropriately.

- **Type conversions.** C provides "type cast" operators that convert a data object from one type to another. Casts are defined from any numeric type (`double, float, int, unsigned, char`) to any other numeric type, and from a pointer of any base type to a pointer of any other base type. These casts are used automatically whenever necessary:

  – When an argument type fails to match a parameter type.
  – When the expression in a return statement does not match the declared return type.
  – When the types of the left and right sides of an assignment do not match.
  – When two operands of a binary operator are different numeric types.

  These rules are the same in C++, but, in addition, the programmer can define new type casts and conversions for new classes. These operations can be applied explicitly (like a type cast) and will also be used by the compiler, as described above, to coerce types of arguments, return values, assignments, and operands.

- **Assignment.** First, the operator = is predefined for all types except arrays. Second, the behavior of the assignment operator has changed: the C++ version returns the address of the location that received the stored value. (The C version returns the value that was stored.) This makes no difference in normal usage. However, the result of some complex, nested assignments might be different.

- **Operators can be extended to work with new types.** For example, the numeric operators `+, *, -, and /` are predefined for pairs of integers, pairs of floating point numbers, and mixed pairs. Now suppose you have defined a new numeric type, Complex, and wish to define arithmetic on Complex numbers. You can define an additional method for `+` to add pairs of Complex numbers, and you can define conversion functions that convert Complex numbers to other numeric types.

- **Reference parameters and return values.** C supports only *call-by-value* for simple objects and structures, and only *call-by-reference* for arrays.

  - In *call-by-value*, the value of the argument expression is copied into the parameter variable. Any changes made by the function to its parameter are local. They do not affect the original copy owned by the caller.
  - In *call-by-reference*, the address of the parameter value is passed to the function, and the parameter name becomes an alias for the caller's variable.

  When a pointer is passed as an argument in C, we can refer to it as "call-by-pointer", which is an abbreviation for "call by passing a pointer by value". This permits the function to change a value in the caller's territory. However, while similar in concept, it is not the same as *call-by-reference*, which is fully supported by C++ in addition to all the parameter passing mechanisms in C. In *call-by-reference*, which is denoted by an ampersand (`&`), an address (not a pointer) is passed to the function and the parameter name becomes an alias for the caller's variable. Like a pointer parameter, a reference parameter permits a function to change the value stored in the caller's variable. Unlike a pointer parameter, the reference parameter cannot be made to refer to a different location. Also, unlike a pointer, a `*` does not need to be written when the variable is referenced within the function. General restrictions on the use of references are:

  - A reference parameter can be passed by value as an argument to another function, but it cannot be passed by reference.
  - Sometimes reference parameters are used to avoid the time and space necessary to `copy` an entire large argument value. In that case, if it is undesirable for the function to be able to change the caller's variable, the parameter should be declared to be a `const&` type.
  - Arrays are automatically passed by reference in both C and C++. You must not (and do not need to) write the ampersand in the parameter declaration.
  - You can't set a pointer variable to point at a reference.
  - You can't create a reference to a part of a bitfield structure.

  Chapter 5 gives extensive explanation and examples of parameter passing mechanisms.

- **Classes.** Structures, as in C are still available in C++, but a C++ `struct` is not the same as a C `struct`. Instead, it is almost exactly like a C++ `class`, with privacy options and functions inside the struct. The single difference between a C++ `struct` and a C++ `class` is that the default level of protection in a C++ `struct` is public, whereas the default in a C++ `class` is private. So which should you use in your programs? Simplify. Use class consistently.

- **Function methods.** Any function can have more than one definition, as long as the list of parameter types, (the *signature*) of every definition is different. The individual definitions are called *methods* of the function. In the common terminology, such a function is called *overloaded.* I prefer not to use this term so broadly. In this course, I will distinguish among *extending*, *overriding*, and *overloading* a function.

- **I/O.** C and C++ I/O are completely different, but a program can use both kinds of I/O statements in almost any mixture. The C and C++ I/O systems both have advantages and disadvantages. For example, simple output is easier in C++ but output in columns is easier in C. Since one purpose of this course is to learn C++, please use only C++ output in the C++ programs you write for this course. Chapter 3 gives a comprehensive list and extensive examples of C++ input and output facilities.

- **Using the `C++` libraries.** To use one of the standard libraries in `C`, we write an `#include` statement of the form `#include <stdio.h>` or `#include <math.h>`. A statement of the same form can be used in `C++` . For example, the standard input/output library can be used by writing `#include <iostream.h>`. However, this form of the include statement is "old fashioned", and should not be used in new programs. Instead, you should write two lines that do the same thing:

  ```
  #include <iostream>
  using namespace std;
  ```

  The new kind of include statement still tells the preprocessor to include the headers for the iostream library, but it does not give the precise name of the file that contains those headers. It is left to the compiler to map the abstract name "iostream" onto whatever local file actually contains those headers. The second line brings the function names from the `iostream` library into your own context. Without this declaration, you would have to write `iostream::` in front of every function or constant name defined by the library.