The beginning of Deitel's 10th chapter of his book on C++ works on introducing the extraction/bit-shift operator known as "<<". This operator is in a class of C++ operators known commonly as overloaded operators. These overloaded operators are created when the same symbol is necessary to perform different functions depending on the context the symbol is present in. Addition and subtraction are another two instances of overloaded operators, as they function much differently when interacting with integers as opposed to strings or floats. The array class contains a great number of overloaded operators, as well as interaction with overloaded operators outside of its scope(such as incrementing/decrementing memory addresses).

The next section of the text goes into some extensive detail on the various overloaded operators that are present as part of the string class in C++. As part of my detailed example, I will utilize references in the book to outline some of the uses of overloaded operators in the context of the string class in C++.

```
int main(int argc, char** argv) {
   string firstString = "hello";
   string secondString = "world";
   cout << firstString << " " << secondString << endl;

   string combinedString = firstString + " " + secondString;
   cout << combinedString << endl;

   string emptyString = "";
   for(int i = 0; i < combinedString.length(); i++){
      emptyString = emptyString + combinedString[i];
   }
   cout << emptyString << endl;

   cout << ("hi" == "hi") << endl;

   return 0;
}
```

which outputs

```
hello world
hello world
hello world
1
```

Four different implementations of overloaded operators are present in the code provided. The first example creates two strings, “hello” and “world” and utilizes what is normally the extraction operator “<<” for outputting the strings to console in a manner that appears like a concatenation. Our first example is bending extraction to fit the framework of a string. The next example grabs two strings, and uses the “+” operator usually defined for integer addition. By setting a new string to the addition of two strings, the “+” operator overrides it's default function to concatenate the two strings and output what is expected, “hello world”. The third example utilizes a for loop to again describe overloaded string manipulation. An empty string is generated that contains absolutely nothing. Using a for loop in conjunction with an overloaded [] operator, normally associated with the array class, we can overload the function and treat the string like an array of chars. Each character is pulled one by one and appended using another overloaded operator “+” to recreate the exact string “hello world”. The final example I have provided is a simple line with enormous implications. The line prints the result of two strings between the overloaded operator “==” which can be used for comparing primitive data types, in this example is used to determine if two strings are equal. It prints a “1” to indicate that two strings of the exact content are equal to one another.

Although not commonly discussed in early C++ context, it is possible to override the traditional operators when utilizing custom user defined data types/objects. In C++, it is not possible to create unique operators, however, all existing operators available in C++ can be

manipulated for user generated data types, allowing for customization in this regard. In C++, operator overload is not a default, or automatic process. To overload an operator, such as "+", the user must simply type operator+ or whatever operator they wish to overload. There are three operators that are unnecessary to overload due to native C++ override support. "=", "&", and "," are all unnecessary to overload, unless some explicitly special use is needed. The exceptions to custom operator overriding is the "*" pointer operator, the "::" operator and the "?" operator.

Although it is a very short section, 10.4 of Deitel's text outlines some fringe important understandings when manipulating operators in the context of of member functions. The binary operators can be manipulated and overloaded. These non-member operator functions are called "friends" due to their interaction to the root class. It is crucial to understand that an example such as operator<(const String&) const will be read as y.operator<(z) and a call like like operator < (const String&, const String&) will be read as operator<(y, z).

In conclusion, the contextual understanding of operator overloading can be viewed in two practical lights outside of the theoretical implications. The first, which is covered by the beginning of the text and this summary illustrates the importance of overloaded operators in normal coding practices. This category makes up all uses of the standard operators in interacting with C++'s primary, defined data types. The second practical implication revolves around custom data typing involved with the user, and the definition of custom operations on those data types. Operators can be overridden and redefined to match the context they are used in, but sometimes C++ provides native support for those operations or in other cases outright bans the manipulation of them. Aside from the practical, overloading must be understood in theory as a

mechanism for future-proofing languages and implementations of data structures and data types.

To undermine the potential of this technique would be a foolhardy decision.

A final working example of operator overloading in C++:

```
#include <iostream>

using namespace std;
class Animal{
   private:
      int legs;
      int eyes;
   public:
      Animal(int legamt, int eyeamt){
         legs = legamt;
         eyes = eyeamt;
      }
      int getLegs(){
         return legs;
      }
      int getEyes(){
         return eyes;
      }
      bool operator ==(Animal &an1){
         if((eyes == an1.getEyes()) && (legs == an1.getLegs())){
            return true;
         }
         else{
            return false;
         }
      }
      int operator +(Animal &an1){
         return (an1.getEyes() + eyes + an1.getLegs() + legs);
      }
};

int main()
{
   Animal animal1 = Animal(2, 3);
   Animal animal2 = Animal(4, 5);

   cout << (animal1 == animal2) << endl;
   cout << (animal1 + animal2) << endl;
```

```
    return 0;
}
```
Outputs:

0

14