Here is some copy/paste page info about overloading the [] operator from another version of the Gaddis text Starting out with C++ Earl Object I have used….

Overloading the [] Operator

Any C++ class can overload the array indexing operator [ ] to make its objects have array-like behavior. In fact, the vector and string library classes override [ ], enabling their objects to be indexed like arrays. For example, the following code

string str = "mad";

cout << str[0] << " ";

str[0] = 'b';

cout << str[0] << " ";

cout << str;

prints the output

  m b bad

An overloaded [ ] operator must take a single argument of any type and can return a value of any type:

ReturnType & operator[ ](inputType T)

To be consistent with how the built-in operator [ ] works, the overloaded operator should return its result by reference so that the result can be assigned to.

To give a simple example, suppose we want to write a Name class to represent the full name of a person. For such a name object, we want name[1] to be the first name and name[2] to be the last name. Furthermore, we want an attempt to index such an object with any integer other than 1 and 2 to terminate the program with an error message. The Name class is shown below.

**Contents of** Name.h

1 #include <string>

2 #include <iostream>

3 #include <cstdlib>

4

5 using namespace std;

6 class Name

7 {

8 string first\_name;

9 string last\_name;

10 void subError()

11 {

12 cout << "Index must be 1 or 2\n" ;

13 exit(1);

14 }

15 public:

16 string &operator[ ](int index)

17 {

18 switch(index)

19 {

20 case 1: return first\_name; break;

21 case 2: return last\_name; break;

22 default: subError();

23 }

24 }

25 };

Notice that the operator [ ] function takes an int parameter and returns a reference to a string variable. If name is an object of this class and k is an integer, the call name.operator[ ](k) will be equivalent to name[k]. Basically, name[1] will return a reference to the member variable first\_name, and name[2] will return a reference to last\_name. Use of this class is demonstrated in [**Program 11-12**](https://jigsaw.vitalsource.com/books/9780134379371/epub/OPS/xhtml/fileP7001011424000000000000000005ECA.xhtml?favre=brett#P700101142400000000000000000609E).

# Program 11-12

1 // This program demonstrates the use of the [ ] operator.

2 #include "name.h"

3 int main()

4 {

5 Name name;

6

7 // Set first name and last name

8 name[1] = "Joseph";

9 name[2] = "Puff";

10

11 // Access first name and last name

12 cout << name[1] << " " << name[2] << " aka Joe Blow\n";

13

14 return 0;

15 }

**Program Output**

Joseph Puff aka Joe Blow

The type of the input parameter to the [ ] operator is not limited to int. To illustrate, let us create an object that can translate English words that describe numbers in the range 0..10 to corresponding integers. If we had such an object, say trans, then trans["one"] would have the integer value 1, trans["seven"] would have the value 7, and so on.

The main idea is to have a vector

vector<string> numerals

{

"zero", "one", "two", "three", "four", "five",

"six", "seven", "eight", "nine", "ten"

};

that stores each numerical word at the corresponding integer position in the vector. For example, “zero” is stored at 0 and “seven” is stored at 7. We then write the overloaded operator

int operator[ ](string num\_str)

so that when passed a string parameter, it returns the position of its parameter in the vector. We will adopt the convention that the function returns −1 for a string that is not found in the vector. Here we have departed from the convention that the operator [ ] return a reference to a memory location that can be assigned to. The class Trans that implements this operator is demonstrated in [**Program 11-13**](https://jigsaw.vitalsource.com/books/9780134379371/epub/OPS/xhtml/fileP7001011424000000000000000005ECA.xhtml?favre=brett#P70010114240000000000000000060BA).

**Contents of** Trans.h

1 #include <string>

2 #include <vector>

3 using namespace std;

4 class Trans

5 {

6 vector<string> numerals

7 {

8 "zero", "one", "two", "three", "four", "five",

9 "six", "seven", "eight", "nine", "ten"

10 };

11 public:

12 int operator[ ](string num\_str)

13 {

14 for (int k = 0; k < numerals.size(); k++)

15 {

16 if (numerals[k] == num\_str)

17 {

18 return k;

19 }

20 }

21 return −1;

22 }

23 };

# Program 11-13

1 // This program demonstrates the array subscript []

2 // operator for the Trans class.

3 #include <cstdlib>

4 #include <iostream>

5 #include "trans.h"

6 using namespace std;

7

8 int main(int argc, char\*\* argv)

9 {

10 Trans trans;

11 cout << "seven :" << trans["seven"] << endl;

12 cout << "three :" << trans["three"] << endl;

13

14 return 0;

15 }

**Program Output**

seven :7

three :3

# 11.7 Rvalue References and Move Operations

# Concept:

An Rvalue reference denotes a temporary object that would otherwise have no name. *A* move operation transfers resources from a source object to a target object. A move operation is appropriate when the source object is temporary and is about to be destroyed.

# Lvalues and RValues

Earlier in this book, you learned that a reference is a variable that refers to the memory location of another variable. For example, in the code fragment

int x = 34;

int &lRef = x;

the identifier lRef is a reference. In a declaration, a reference is indicated by the presence of an ampersand (&) between the type and the variable’s identifier. This type of reference, which you learned about in [**Chapter 6**](https://jigsaw.vitalsource.com/books/9780134379371/epub/OPS/xhtml/fileP7001011424000000000000000003201.xhtml#P7001011424000000000000000003201), is called an *lvalue reference*. You can think of an *lvalue* as a memory location associated with a *name* that can be used to access it from *other parts of the program*. Here we interpret “name” to mean any expression that can be used to access a memory location. Thus, if arr is an array, then arr[1] and \*(arr + 1) are both considered “names” of the same memory location. In contrast, an *rvalue* is a temporary value that cannot be accessed from other parts of the program. To illustrate, consider the following program fragment:

int square(int a)

{

return a \* a;

}

int main()

{

int x = 0; // 1

x = 12; // 2

cout << x << endl; // 3

x = square(5); // 4

cout << x << endl; // 5

return 0;

}

In this program, x is an lvalue. This is because x represents a memory location that is accessed in several parts of the program, namely, lines 2, 3, 4, and 5. On the other hand, the expression square(5) is an rvalue because it represents a temporary memory location created by the compiler to hold the value returned by the function. That memory location is accessed only once, on the right-hand side of the assignment in line 4. Immediately afterward, it is deallocated and can no longer be accessed. The essence of a memory location containing an rvalue is that it *lacks a name that can be used to access it from another part of the program.*

C++ 11 introduces the concept of an *rvalue reference* to refer to a temporary object that would otherwise have no name. An rvalue reference is declared similarly to an lvalue reference, except with a double ampersand (&&). The code below uses an rvalue reference to print the square of 5 two times.

int && rRef = square(5);

cout << rRef << endl;

cout << rRef << endl;

Interestingly, declaring an rvalue reference assigns a name to a temporary memory location, making it possible to access the location from other parts of the program and transforming the temporary location into an lvalue.

An rvalue reference cannot be bound to an lvalue, so the following code will not compile.

int x = 0;

int && rRefX = x;

After the initialization statement

int && rRef1 = square(5);

the memory location containing square(5) has a name, namely, rRef1, so rRef1 itself becomes an lvalue. This means that a subsequent initialization statement

int && rRef2 = rRef1;

will not compile because the right-hand side rRef1 is no longer an rvalue. The upshot of all this is that a temporary object can have at most one lvalue reference pointing to it. If a function has an lvalue reference to a temporary object, you can be sure no other part of the program has access to the same object. This fact is important in understanding how move operations work. We discuss move operations next.