The Square Bracket Operator, Reading

Our array class is a range-safe version of the standard C++ array. But it lacks one thing that the C++ array has -- one very useful thing: the ability to access a value at an index using square brackets, like this: `a[9]`.

Actually, C++ string objects figured out how to do this already. With a string object you can do this:

```
string s = "Hello";
cout << s[1] << endl; // outputs an 'e'
s[0] = tolower(s[0]); // makes lowercase
```

Now, we *know* that `s` is not an array, so how is it that we can do `s[0]`? The answer is that C++ lets the programmer define what its symbolic operators mean and do in the context of the objects we design! It's called "operator overloading" in object-oriented programming. It's implemented with member functions. Here's how it works:

operator[ ]

For our array class, with `Array a;` declared in the main program, if the compiler were to see `cout << a[0];`, it would recognize "a" as an object and *replace* that code with this: `cout << a.operator[ ](0);`. It looks really funny, but let's break it down. The function name is "operator[ ]" -- weird, with the [and] in the name, especially since that's not a valid identifier in C++. But it's a special reserved word in the language that's there just so we can write this function. After that, the rest is easy -- the parentheses contain the parameter, which is the index. So the prototype is:

```
int operator[ ](int) const;
```

...and the function definition is:

```
int Array::operator[ ](int index) const
{
  if (index < 0 || index >= 10)
    return 0;
  return values[index];
}
```

Look familiar? It's just Array::getAtIndex with a different name! And it really is that easy -- mostly...

operator[ ] Setter

Here's the thing. With the above we can write this in the main program:

```
cout << a[8];
```

...and it will do exactly what we hoped it would do. But we run into difficulty when we try this:

```
a[9] = 100;
```

In the first place, that would have to be a *setter*, and we wrote this as *getter*. That's easily corrected, *but* what's with the function call being on the *left side* of the equal? That's like doing this:

```
a.getAtIndex(9) = 100;
```

...and there's no way something like that could work... or is there?... The reason it won't work is that the function returns an int value, and you cannot set an int value to anything. Making it into a void function instead is even worse. But there is one thing we have not yet considered -- something else we could return from the function that would solve the problem. The function needs to return a *variable*.

And not just a copy of a variable -- that's the same as a value. It would have to return a *reference* to a variable. Remember reference variables? This is called "returning a **mutable reference**", because what's returned can be *changed* in main. Let's try that:

```
int& Array::operator[ ](int index)
{
  if (index < 0 || index >= 10)
    return 0; // hmm... this is a problem...
  return values[index];
}
```

it's going to return an *alias* for `values[index]`. And did you notice that the trailing const is gone? That's so `values[index]` can be set to something in main. We'll deal with the returned zero later, but for now, this is how main sees it:

```
  a.values[9] = 100;
```

`a[9]` is an *alias* for `a.values[9]`. We cannot do `a.values[9]` directly because `values` is a private data member. Still, the returned zero is a problem because `a[-1] = 5;` would be like saying `0 = 5;` and *that* just makes no sense at all!

A dummy Data Member

The solution is to return a variable. And not a local variable! Then `a[-1]` would be an alias for something that no longer exists, and that would end badly. The variable has to be a private data member. It works for index *in* range, so why not do the same for the index *out of* range? We just need another data member, like this:

```
  int dummy;
```

We return that instead of zero, and that solves the problem!

```
int& Array::operator[ ](int index)
{
  if (index < 0 || index >= 10)
    return dummy; // a mutable reference to a private data member
  return values[index];
}
```

It's not really necessary, but `dummy` may be initialized to zero in the constructor:

```
Array::Array()
{
  ...
  dummy = 0;
}
```

Poor `dummy` ends up taking all the hits, paying for the programmer's (or user's) mistakes. But that's okay -- that's its *job*.

operator[ ] getter *And* setter

Still, we are not quite done. What happens if we share our Array object with a function, as an immutable reference, like this:

```
bool findZero(const Array&); // prototype
```

Fine, but what if that function uses the square bracket operator to look though the array?

```
bool findZero(const Array& array)
{
  bool found = false;
  for (int i = 0; i < array.capacity() && !found; i++)
    if (array[i] == 0)
      found == true;
  return found;
}
```

Big problem! The parameter is supposed to be immutable inside the function. But the function calls a *setter*. Even though it does not use the setter to change anything, that does not matter to the compiler. It's all about the trailing const, remember?

The solution to this last problem is easier than you might have imagined -- just include *both* versions of the function -- the getter *and* the setter! The two have the same name and the same parameter list, but one has the trailing const and the other does not, and that's all the compiler needs to decide which to use where. (It uses the setter everywhere it can, but in those cases where it absolutely cannot use the setter, it uses the getter instead.)

```
class Array
{
  ...
  int dummy;

  public:
  ...
  int operator[ ](int) const;
  int& operator[ ](int);
  ...
};
```

```
int Array::operator[ ](int index) const
{
  if (index < 0 || index >= 10)
    return 0; // a copy
  return values[index]; // a copy
}

int& Array::operator[ ](int index)
{
  if (index < 0 || index >= 10)
    return dummy; // a mutable reference
  return values[index]; // a mutable reference
}
```

And about how to type the back-to-back [and] -- they are done with a space separating then in this reading. The space is optional, and only used here because the font on this page makes them run together and look like a box! Normally you'll type them without a space between them in your coding.