Chapter 3
Teaching KoolB to Read

Introduction:

Welcome back! Up until now we have been just dabbling with compilers, how about we buckle down and dig in? Yes, I agree, we should really get going if we want to finish this project on time. First, let us go back to our original definition of a compiler. In it, we said that a compiler takes a language (in our case, Rapid-Q style BASIC, and translates it to another language (in our case, assembly language). For the time being, I think we should just concentrate on the first part of the definition, the part about taking the language and translating it. Before we can translate the BASIC into something else, we need to be able to read it. So what does that mean? Well, when you first began to learn to read, what stages did you go through? If I remember correctly, I first learned the alphabet, then numbers. Once I knew those, I learned to recognize that certain groups of these letters and numbers meant certain things; in another words, I started to learn to read individual words. Next, I learned to understand that words arranged in a certain order formed logical thoughts (unlike individual words in random order: ate cat a mouse the) called sentences. These sentences also contained punctuation and symbols other than just letters and numbers like the semi-colon, coma, period, dash, etc that helped me understand exactly what was going on. Well, for KoolB to be able to understand and translate one language to another, he has to know how to read. The problem is: "How do we teach software how to read?"

Solving this problem takes some effort and we will use the remainder of the chapter to solve this problem. The reason for the detour with how a somebody learns how to read is that we will use the same steps to teach KoolB how to read. Before we go much further, I must break some bad news to you: those of you who like to make family trees will need to make some modifications it. Unfortunately, Mrs. Rapid-Q suffered from a massive stroke (both literally and figuratively), and was rushed to the hospital only to have the doctors pronounce her dead on arrival. For her husband Mr. C++, this was a serious blow; in fact, it ends all hope that some of Mrs. Rapid-Q's niceness might rub off on him. His friends B++, D, and Delphi all comment on how he has withdrawn himself from society to live grieving in moody isolation. Their daughter, KoolB, however, has chosen a different path to take. Instead of becoming more like his father, she desires to become as good or even better than Mrs. Rapid-Q to honor her and carry on her tradition. She also dislikes his father, who she sees as a weakling who cannot get over his grief or deal with it constructively. Consequently, she wants to learn as fast as she can, and the first thing she wants to learn is how to read. It is our job to help her achieve that goal by teaching her how to read.

So with this in mind, let us begin. Open up Dev-C++ and open 'C:\Compiler\KoolB\KoolB.dev,' our compiler project. Before we start, we need to clear up some issues from last time. I believe I left you with the following code:

```
#include <windows.h>
#include <stdio.h>

void Start(void);
void Compile(void);
void Stop(void);

DWORD StartTime;

int main(int argc, char *argv[])
{
  printf("Welcome to the KoolB Compiler by Brian C. Becker.\r\n");
  Start();
  Compile();
  Stop();
  return 0;
}

void Start(void){
  StartTime = GetTickCount();
  return;
}

void Compile(void){

  return;
}

void Stop(void){
  printf("Compile Time:\t%f", (GetTickCount()-StartTime)/1000.0);
  return;
}
```

As Pavel Minayev so kindly pointed out, several corrections need to be made. First, he said that the preferred way to declare functions was without the void in the parameters. Secondly, he pointed out that printf was bulky, slow, and not as functional as other options. So we will correct this now (copy & paste this over the current code):

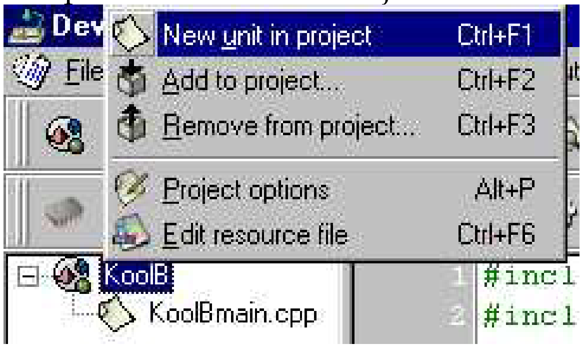
```
#include <windows.h>
#include <iostream>

void Start();
void Compile();
void Stop();

DWORD StartTime;

int main(int argc, char *argv[])
{
  cout << "Welcome to the KoolB Compiler by Brian C. Becker." << endl;
  Start();
  Compile();
  Stop();
  return 0;
}
```

```
void Start(){
  StartTime = GetTickCount();
  return;
}

void Compile(){

  return;
}

void Stop(){
  cout << "Compile Time:\t" << (GetTickCount()-StartTime)/1000.0 << endl;
  return;
}
```

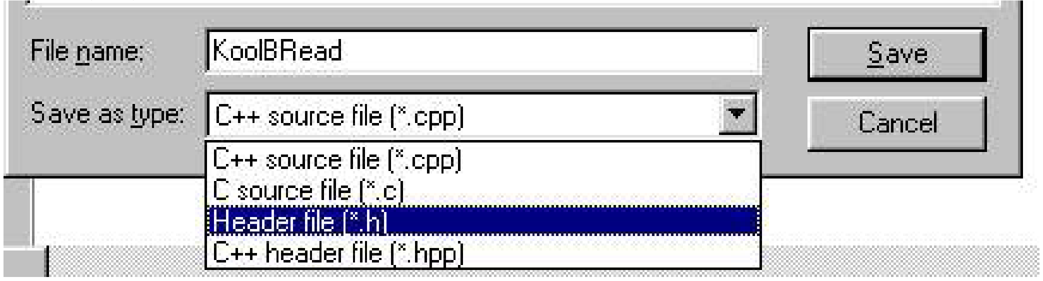
Here you can see a couple notable changes: first, on the second line (you can display the line numbers by going Options -> Environment Options in Dev-C++ and then moving to the 'Editor' tab, checking the option 'Show line numbers', and clicking 'OK') you see that I replaced #include <stdio.h> with #include <iostream>. This allows us to use the C++ style print command: cout. If you scan the code, you note that I changed all printf functions to cout. cout is just like print in Mrs. Rapid-Q (may she rest in peace) and just like printf in C++ except it has some cool features. Instead of using commas or semi-colons like Mrs. Rapid-Q, we use two less than signs: <<. This directs the what ever is right after the << to the console, whether it be a number, string, object, etc. The endl at the end of each cout statement tells the Mr. C++ to start a new line. The last thing you will notice is that I removed the void from the parameter lists (thanks, Pavel, for pointing this out). Finally, don't forget to add your own name to the compiler and any other text!

Adding the Reading Module:

Now we are ready to teach KoolB how to read. First, let's add a module specifically designed to help KoolB handle the reading of the BASIC source code. So swap to your Dev-C++ window and right-click on the KoolB project icon and then select the option 'New unit in Project':


Dev
File
New unit in project Ctrl+F1
Add to project... Ctrl+F2
Remove from project... Ctrl+F3
Project options Alt+P
Edit resource file Ctrl+F6
KoolB
KoolBmain.cpp
#incl
#incl

That will add a new file named 'Untitled1' to your project. To give it a name, go File -> Save All. It will prompt you for a name, so give it the navigate to the KoolB folder (C:\Compiler\KoolB) and name the file ‘KoolBRead’ and then select a file type of ‘Header file (*.h)’ as seen below:



File name: KoolBRead

Save as type: C++ source file (*.cpp)

Hit the ‘Save’ Button. You now have added a reading module to KoolB!

Now that we have a reading module for KoolB, we need to develop it. First, we need a reading object that will do all the hard work for us. So without further delay, here is the code to create the reading object:

```
class Reading{
 private:

 public:

};
```

As always, we will dissect this piece of code. The first line:

```
class Reading{
```

If you compare this statement to declaring other data types, you will see a great similarity. Remember how we created a variable to hold the time? We said:

```
DWORD StartTime;
```

As you compare these two declarations, you will notice that when we use Mr. C++ to create any type of data, we first need to tell him first the data type. Here the data type is class (or the declaration of an object). After the data type, Mr. C++ expects to see the name of the data. Since we are attempting to teach KoolB to read, we might as well name the object ‘Reading’, right? The curly bracket after the name tells Mr. C++ were the object starts (Just like a function). The couple of lines are related:

```
 private:

 public:
```

With Mr. C++, an object is composed of several parts. The only two we will concern ourselves with is private and public. The first section in our example is private. Compare this statement to your life: you wouldn’t want everybody prying into your private life, would you? Of course not. Similarly, sometimes we don’t want everybody to access every portion of our object. Thankfully, Mr. C++ has provided the keyword private so we can specify what stuff we want to remain secret. What kind of stuff would we want to keep a secret? Well, all sorts of stuff, but mainly variables. For example, if we have a string that contains the BASIC source-code, would we want everybody to access it? No, some stupid programmer might accidentally change it to “Dim Me As Stupid” or something! That could be disastrous! That is why we would want something private. The other keyword is public, which allows everybody to access it. This would include functions that allow us to actually read the BASIC source. An object where everything was private would be useless. The final line, which just contains an ending curly bracket and a semi-colon, ends the creation object (similar to END CREATE in Mrs. Rapid-Q).

Background Info:

Now that we have a reading object, what do we do with it? In order to answer this question, we need to go back to our original problem of teaching software to read BASIC source and converting it to ASM. For example, read the following tidbit of some of Mrs. Rapid-Q’s last code:

```
If Answer1 <> “No” OR Pi <> 3.14 Then
  Print “Error”
End If
```

How would somebody read this and understand it? First, they would break it down into smaller pieces:

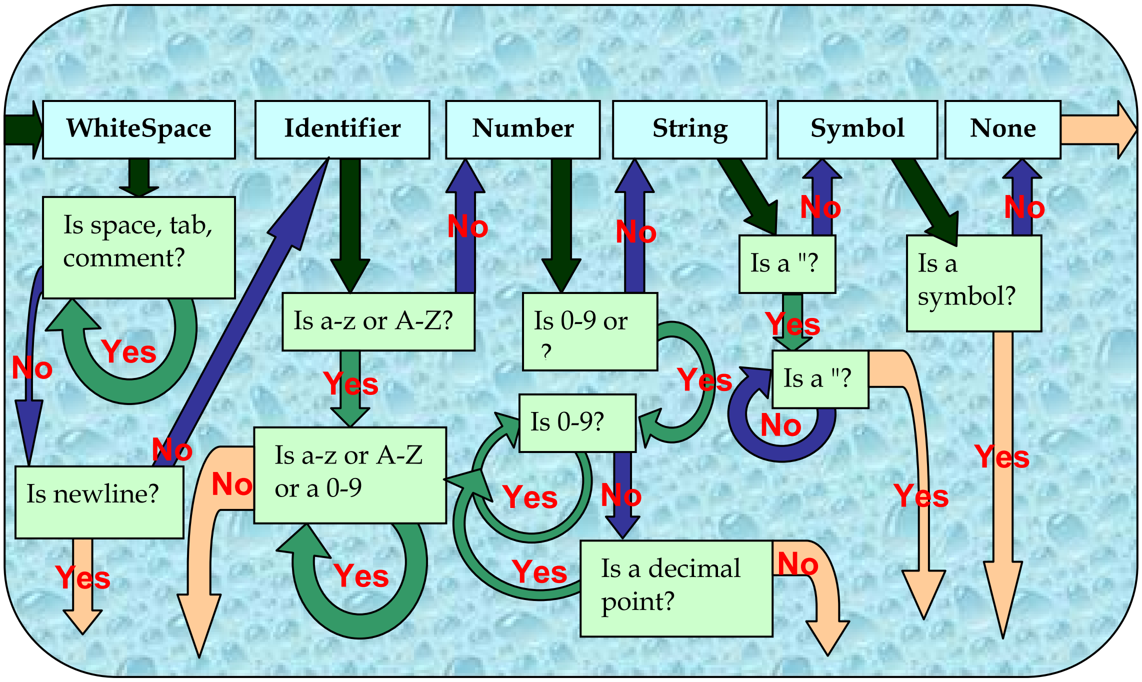
1. If A word that is part of the language; it has two parts: the condition and also an action to do if that condition is true.
2. Answer1 Also a word, but it is not part of the language, so it must be some type of data.
3. <> A symbol that is part of the language; it compares two data types to see if they are unequal.
4. “No” A string that is not part of the language; this a data type that is not stored in somewhere, we just want to use it once and then forget about it.
5. ORA word that is part of the language; it combines two comparisons, so we know there must be another comparison of two more data types.
6. Pi A word that is not part of the language, so it must be a data type. It is the beginning of the second comparison.
7. <> A symbol that is part of the language; it compares two data types to see if they are unequal.
8. Then A word that is part of the language; it separates the condition from the action of an If statement.
9. Print A word that is both part of the language and also part of the action that will take place if the condition is true.

10. “Error” A string that is not part of the language; it is just being used temporarily by print.
11. End A word that is part of the language; it indicates that we are at the end of something.
12. If A word that is part of the language; it tells exactly what we are ending, which is the action part of the if statement (and therefore, the entire if statement).

When somebody reads the above code, he/she goes through these steps in several nanoseconds. I can hear you asking, “So how does this have anything to do with building a compiler?” Well, in order to have KoolB translate from one language to another, she must first be able to read and understand the BASIC source-code. To do this, first need to understand how a human reads, for once we know that, we can have KoolB just imitate us humans.

Let’s make some observations about the steps above. First, we broke everything down into manageable pieces. We then looked closely at these pieces then grouped them into categories. For instance, some were part of the language and others were not; some were numbers and others were not. Once we knew this, we could sort of predict what was coming next (Quick: what do you expect after the word DIM in a BASIC program? That’s right, you expect the name of a data type.) This is essentially what a compiler does.

KoolB needs a way to separate the BASIC source code into small, understandable pieces (let’s call them words). How are we going to separate a big chunk of text into individual words? How will KoolB know were one word starts or where a word ends? For that matter, how do we know what a word is! Well, if we go back to our example in the last section, you will see that we have a lot of different types of words. For example, we have words that are totally made up of letters, we have words that are totally made up of numbers, we have words that are made up of both, we have words that are made up symbols, and finally we have words that are composed of numbers, letters, and symbols which are enclosed by quotation marks “ and “. These definitions are a bit long, how about we simplify them? We simplify all these into identifiers, numbers, strings, symbols, and newlines. Let’s graphically define these types of words:



Wow! That looks pretty complicated, so lets analyze it (I know, I know, you are getting tired of analyzing everything). First, KoolB will start reading by looking at the first character (Don't know what a character is? Look at your keyboard and you will see a whole bunch of them. Seriously, a character can be considered any key on a keyboard, the letter 'b', the number '8', or the symbol '#') and deciding if it is white space. White space characters are just characters that produce (what else!) white space. White space characters include spaces, tabs, and comments. A newline is special in that in BASIC, the end of a line usually indicates the end of a statement, so we will have newline type of word. Comments aren't really white space to humans, but to KoolB, they are. As you can see in the above diagram, KoolB just loops through all the whitespace since it isn't important. Once the next character isn't white space, KoolB checks to see if it is the next word type: an identifier. An identifier is a word that starts off with a letter of the alphabet followed by more letters of the alphabet or digits. If KoolB finds a word like this, she knows that it is an identifier that is probably a part of the BASIC language (like DIM), a function name, etc. However, if the word does not start of with a letter of the alphabet, KoolB examines the word to see if it is a number. A number starts off with a digit or a decimal place followed by more digits or a decimal place (later, when we actually get to programming, we will need to make sure somebody doesn't put in two decimal places in a number). KoolB is patient, so if the word turns out not to be a number, KoolB checks to see if it is a string. A string starts off with a quote, can contain almost any other character, and ends with another quote. If the word is not a quote, KoolB checks to see if the word is a symbol like < or (or +. So what if the word is not one of these types? Well, in that case, KoolB would assume that it had reached the end of the BASIC source code and calls the type none.

Setting up the Structure:

Now that we know a bit of background on the subject of reading, lets turn these abstract thoughts into code. With that in mind, lets start programming. Before KoolB can start to read, she needs something to read. Just like her mother Mrs. Rapid-Q, KoolB will read BASIC source code. Since we are keeping things simple, we will just call the BASIC source code a book. With that in mind, let's fill in a skeleton of variables and functions that we will need for the Reading object. Swap to your Dev-C++ window and add the following variables under the private section:

```
    string Book;
    long   BookLength;
    long   BookMark;
    long   CurrentLine;
    string CurrentWord;
    int    TypeOfWord;
```

These are our variables for our Reading object. Here you see several different data types: string, long, and int (int is just short for integer). KoolB needs something to read, so we must provide a book for her, and since we like straightforward code (don't you just love it when people name things TemporaryVariable73?), we will name the string that contains the BASIC source code Book. KoolB also has BookMark, which keeps track of her place in the book (Ever spent 5 minutes looking for your place in a book after the bookmark fell out? We don't KoolB to take that much time trying to figure out where she last left off!). To prevent KoolB from reading onto another book, she needs to know when the book ends, which means we need a number that indicates how long the book is: BookLength. CurrentLine helps KoolB keep track of what line she is on in case she finds errors. If she happens to find errors in her book, she will want to be able to tell the programmer what line the error occurred on, just like any other proper compiler. The string CurrentWord is the result of the past several sections; it is the small chuck of text that KoolB looks at. As a side note, this word that we have been talking about has a fancy name in compiler terminology: token. So if you go on and hear stuff about tokens and tokenizing, that is the same thing as words and reading. To keep things simple, I will continue to use the name word and reading. And I guess you have already guessed what the TypeOfWord is: it holds what type of word the current word is. Now we have

The only strange data type to Mr. C++ is string. By default, Mr. C++ doesn't have a string data type; instead you have to use what is called a string object. "So where is this object, and more importantly how do I use it?" you ask. This object is part of the STL or the Standard Template Library, which is pretty complex and can be very useful to us. We won't go into details about the STL; we will just use how to use small portions of it. To use the string object, swap back to Dev-C++ and click on the KoolBMain.cpp (or whatever you named the main C++ file) icon on the left-hand side of the screen. Then add the following line near (after #include <iostream>):

```
#include <string>
```

That wasn't too hard was it? Now lets create some functions for the Reading object (in the public section since we want everybody to be able to access it):

```
Reading(){BookMark = 0; CurrentLine = 1;}
void OpenBook(string FileName);
void GetNextWord();
  bool SkipWhiteSpace();
  void GetIdentifier();
  void GetNumber();
  void GetString();
  void GetSymbol();
void CheckWordLength();
string Word();
int WordType();

enum WordTypes{EndOfLine, Identifier, Number, String, Symbol, None};
```

I am sure that some of you are wondering what the heck all this stuff means, so let's go through it function by function. The first function is by far the strangest. It has the exact same name as the object, and it actually has some code to it. The reason it has the same name as the object is that it is a constructor. "A constructor," you say, "what's that?" When you first wake up in the morning, what is the first thing you do? I turn my alarm off, get out of bed, dress, and boot up the computer – almost every single day without exception. The same is true with KoolB's Reading object, it has to do some stuff before launching into the new day (or in this case, the new execution). We put all this stuff in a function that has the same name as the class. When KoolB runs, it will first create her objects (at this point, only Reading) and run the constructor. As you can see, it is the only function that has code attached to it, the reason being that it is only a little bit of code, or only two commands. Those of you not yet that familiar with Mr. C++ might be thinking, "How come he has these two instructions on the same line? In Mrs. Rapid-Q we have to put one instruction per line unless we use the : operator, and I don't see that operator." Well, Mr. C++ isn't line-driven like Mrs. Rapid-Q; instead he is semi-colon driven. Instead of putting a newline at the end of an instruction, Mr. C++ wants a semi-colon. So the code example is perfectly valid. The first instruction (BookMark = 0;) tells KoolB to put the bookmark at the very beginning of the book. The second instruction (CurrentLine = 1) tells KoolB to start counting lines at the number 1. I won't describe the rest of the functions now; instead, we will examine what each one does and how to actually program it one by one. However, the last line deserves some mention as it is not a function. The last line is:

```
enum WordTypes{EndOfLine, Identifier, Number, String, Symbol, None};
```

This creates the types of words for us. Enum just assigns each different label for the WordType a different value. For example, EndOfLine is 0, Identifier, is 1, etc. It just makes it easier to compare what type a word is. If you don't totally understand, that's OK because we will learn how to use them later on. But let's get started.

Programming the Reading Module:

I bet you can't guess what the first function, OpenBook, does. OK, I would probably lose the bet as you have already figured it out. OpenBook gets the filename to the BASIC source code and then opens the file. Mrs. Rapid-Q provides a nice component for this: QFileStream. Mr. C++ also has a file object; though it is a bit harder to use. We are now entering into a phase where we will be looking at medium sized snippets of code now, so I sure hope you brushed up on your C++ since last time (if you need it, anyhow). Without further procrastination, the C++ code:

```
void Reading::OpenBook(string FileName){
  ifstream File(FileName.c_str(), ios::in);
  long     FileStart;
  long     FileEnd;
  char *   Spoon;
  if (File.is_open() == false){
     cout << "Error! Could not open file: " << FileName << endl;
     exit(1);
  }
  FileStart = File.tellg();
  File.seekg(0, ios::end);
  FileEnd   = File.tellg();
  File.seekg(0, ios::beg);
  Spoon = new char[FileEnd-FileStart];
  File.read(Spoon, FileEnd-FileStart);
  Book        = Spoon;
  BookLength  = Book.length();
  delete[] Spoon;
  File.close();
  return;
}
```

You can copy & paste this function in your Reading file, but outside (below) the actual class. We will analyze this function in sections. The first section is the function:

```
void Reading::OpenBook(string FileName){
```

Here we have a function named OpenBook that gets a string containing the name of the file to open. The only difference between this and the other functions we have already created (remember the Start, Compile, Stop functions from Chapter 2?) is that this function is part of the Reading object. As you can see, we tell Mr. C++ this by putting Reading:: in front of the function name. The general form is <ClassName>::<FunctionName>. You will be seeing a lot of this as we add more classes. All you have to know right now is that when you see Reading::<FunctionName>, you know that the function belongs to KoolB's reading object. In the next section, we declare our variables for the function:

```
  ifstream File(FileName.c_str(), ios::in);
  long     FileStart;
  long     FileEnd;
  char *   Spoon;
```

The first line here declares a file. As I mentioned before, Mr. C++'s file component isn't nearly as nice (or friendly!) as Mrs. Rapid-Q's. If I was comparing this to Mrs. Rapid-Q, I would say that ifstream is the name of the file component just like QfileStream. Unlike in Mrs. Rapd-Q, Mr. C++ makes us include the component into our program. So go back to the file KoolBmain.cpp and in the #include section add #include <fstream> underneath all the others. Back on track, when you create a file, you don't have to use the Open command like how Mrs. Rapid-Q does it. Instead, do that right after the name of the file (which in our case is, what else, File). The first parameter is the name of the file and the second parameter tells Mr. C++ how to open it. The first parameter is FileName.c_str(). We know where FileName came from, it came from OpenBook's only parameter, but what in the world is that weird c_str() doing at the end of it? Since Mr. C++ doesn't have a string data type per say, we need a way of converting our string to something that Mr. C++ can understand. The way we do this is to put .c_str() at the end of the string. c_str() converts the string to something that Mr. C++ can understand. We won't be doing this much, but we might have to do it several more times. The second parameter is ios::in, which tells Mr. C++ we want to be reading from the file (imagine it being like Mrs. Rapid-Q's fmOpenRead). The other next two variables are numbers (longs to be specific) and will hold the starting place of the file and the ending place of the file, respectively. This way we can subtract them and get the file size (remember how we got the beginning time and ending time to get the amount of time it took for our compiler to run? This is the same concept). Finally, we have a strange data type, char *, named Spoon. char the closest thing to a string data type Mr. C++ has. That is about as much as you need to know. char can get pretty complicated, so we will just be content to learn how to use them when absolute necessary (which won't be often). Spoon will temporarily hold the contents of the file before we put it into a normal string. Think of Spoon as a spoon. Now before you assume I have lost my sanity, let me explain. What is the main function of a spoon? Isn't it to transfer some food from a bowl (or whatever) to your mouth (or whatever)? In essence, then, it is a temporary storage box for food while you lift the food from your bowl (or plate) to your mouth. Likewise, Spoon is just a storage box while we transfer the text in the file to the string Book. Next, we need to test to make sure we have opened the file successfully:

```
if (File.is_open() == false){
   cout << "Error! Could not open file: " << FileName << endl;
   exit(1);
}
```

Here we have a combination of some new stuff and old stuff. Let's get the new stuff out of the way first. The first line is an if statement, Mr. C++ style. Mr. C++ is case-sensitive, which means that IF is not the same as if, so be careful! Mr. C++ also expects the expression to evaluate to be enclosed in parenthesis. Just like a function, Mr. C++ needs to know when the if statement starts and stops, so we have to use curly brackets (here we go again). You might also notice that I used a double = sign when comparing

File.is_open() to false. That is another one of mean old Mr. C++'s bad habits. To compare two things to see if they are equal, Mr. C++ makes us use a double equal sign. Unfortunately, this causes a LOT of mistakes for programmers used to BASIC because they will sometimes accidentally assign true to the variable instead of comparing it to true! The familiar part of this is the second line. If File.is_open() is false, for some reason the file didn't open. Perhaps somebody locked it for editing, it doesn't exist, or the user mistyped the name, etc. If this happens, we print out an error saying we couldn't this file (don't forget to endl the cout statement. Also don't for get the stupid semi-colon we have to put in). Since there is no since in continuing compiling if no book exists for KoolB read, the third line exits the program. Sort of like the Mrs. Rapid-Q's end statement, except it we pass it either a zero or a one value. Zero means the program exited successfully, while one means that the program encountered errors. Since we couldn't open the BASIC file, we should pass a one. The next section gets the size of the file:

```
  FileStart = File.tellg();
  File.seekg(0, ios::end);
  FileEnd   = File.tellg();
  File.seekg(0, ios::beg);
```

Whew! That's a far cry from Mrs. Rapid-Q's File.Size method, isn't it? We see two functions here: tellg and seekg. File.tellg gets the current position of the file (like getting Mrs. Rapid-Q's File.Pos method). File.seekg sets the current position of the file (like setting Mrs. Rapid-Q's File.pos method). The first line assigns FileStart the current position in the file (since we haven't done anything, tellg gives us the beginning position of the file). The second line tells Mr. C++ to go to the end of the file (you know it's the end of the file by the weird ios::end variable). Now that the file position is at the end, we get it to store in FileEnd by calling File.tellg again. Finally, we need to put the file position at the beginning of the file so we can start read it. Mr. C++ doesn't always make things easy, does he? Now that we have the beginning and ending file positions, we can calculate the file size. Now why do we need the size of the file? So we can read the file:

```
  Spoon = new char[FileEnd-FileStart];
  File.read(Spoon, FileEnd-FileStart);
```

Remember that spoon we were talking about earlier (yes, I am still sane)? It was our analogy to store temporary stuff while we transfer it from one container to another. In this case, we have our BASIC source code in a file and the string Book. In order to transfer the stuff in container one (the file) to container two (the string), we need a spoon. We need a spoon, so let's create one. The first line does just that. It creates a new spoon the exact size of the file (imagine eating your morning cereal with a spoon the size of a bowl!). We don't really need to know what goes on internally since it deals with pointers and such. It isn't that important to the overall understanding of compilers, so for now, let's just consider it disposable spoon that we use, recycle, and forget about. Now that we have a spoon, lets use it. The second line reads the entire file (the second parameter tells Mr. C++ how many bytes to read; in our case, the entire file as designated by the difference between FileStart and FileEnd) and stores it into our temporary storage book, Spoon (the first parameter). Now that we have the entire file in the spoon, let's tell KoolB to eat it (figuratively, not literally):

```
  Book        = Spoon;
  BookLength  = Book.length();
```

The first line here transfers the contents of Spoon into our Book string. Now that we have our book, we need to get its length. Fortunately, Mr. C++ provides us with a nicely named function (for once): length. So in the second line, we assign the length of Book to BookLength. Now that we have accomplished our goal of opening the book, it is time to clean up:

```
        delete[] Spoon;
        File.close();
        return;
      }
```

The fist line recycles our disposable Spoon (we can now forget about it). The second line closes the file. The last two lines ends the function. We now have taught KoolB to go over to the bookshelf and get any book we tell him and open it. But that is only the first step. We must next teach him to break his book down into words.

The second function is GetNextWord, which KoolB uses to read the book. For our Reading module, this function probably does the most important work. It looks at the BASIC source code and reads the next word (or token) based on the characteristics we defined in the earlier section. Without further ado, here is the function using Mr. C++:

```
void Reading::GetNextWord(){
  if (SkipWhiteSpace() == false){
    return;
  }
  if (isalpha(Book[BookMark])){
    GetIdentifier();
    return;
  }
  if (isdigit(Book[BookMark]) || Book[BookMark] == '.'){
    GetNumber();
    return;
  }
  if (Book[BookMark] == '\"'){
    GetString();
    return;
  }
  if (ispunct(Book[BookMark])){
    GetSymbol();
    return;
  }
  TypeOfWord = None;
  return;
}
```

Keep breathing, its not that bad, breath in, out, in… OK, maybe I am exaggerating a bit here, but this does look pretty intimidating, especially for those who still aren't cozy-cozy with Mr. C++. Well, what do we know about this function from past experience? One, we have a function that doesn't return anything (we know that by the void keyword). Two, we have a function that is part of the Reading object. Third, we know that this function is really a sub, i.e. it doesn't take any parameters. Now let us ask a different question: what does this function do? If look carefully, you will see that it doesn't do a whole lot itself, instead it calls a bunch of other functions. The first part gets rid of all the junk:

```
  if (SkipWhiteSpace() == false){
    return;
  }
```

Here we are calling SkipWhiteSpace skip past all spaces, tabs, etc. However, we are also comparing it to false. Why are we doing that? Remember that one of our word types was the end of line? An end of the line is whitespace as well, but we don't want SkipWhiteSpace to skip that otherwise we would end up with the entire book on one line! So we tell SkipWhiteSpace to stop if we find an end of the line word and return false. When GetNextWord sees that SkipWhiteSpace has returned false, it knows that it has a word (end of the line word) and says "Well, I don't need to go any further since I already have found a word." If SkipWhiteSpace returns true, that means that GetNextWord can continue because no end of the line word was found. Now that KoolB knows that the word isn't the end of the line, it checks for the next word type:

```
  if (isalpha(Book[BookMark])){
    GetIdentifier();
    return;
  }
```

Here KoolB checks to see if the word type is Identifier. How does it do that? Well, we said that an identifier could start with any letter in the alphabet. Thanks to Mr. C++, we have such a function to see if an integer is part of the alphabet: isalpha. Isalpha takes only one parameter, and that is a single character in the form of its ASCII representation. "What in the world is ASCII?" you ask. ASCII just assigns each character a number. So the ASCII number for 'A' is 65. 'B' is 66, etc. Anyhow, KoolB passes isalpha Book[BookMark]. If I remember correctly, Book holds the contents of our book, and BookMark holds our current position in the book (i.e. how many characters into the book we have gone). Just like in Mrs. Rapid-Q, we can extract a single character from a string by using the StringName[X], where X is the position of the character. So we pass the current position of our BASIC source code book to isalpha and wait for it to return something. If it returns true, we know that we have the beginning of an identifier word and we call GetIdentifier, which gets an identifier for KoolB. Then KoolB can exit from the GetNextWord function by the return statement. If isalpha return false, KoolB knows that she has to keep trying:

```
if (isdigit(Book[BookMark]) || Book[BookMark] == '.'){
  GetNumber();
  return;
}
```

Bet you can't tell what KoolB is looking for here! A number, what else? However, this form is slightly different. KoolB uses isdigit to see if the beginning of the word is a number, but KoolB also has to allow for another type of number. If you aren't familiar with Mr. C++, you might be puzzled over what comes next: ||. Mr. C++ likes to confuse people (it makes him feel smarter), so he discards Mrs. Rapid-Q's keyword OR in favor of ||. So whenever you see ||, just mentally translate it to OR. If we have an OR in our expression, we need the second part of the expression. KoolB needs to allow for numbers that start with a decimal point (like .03). So we compare current position (Book[BookMark]) with a period. Keep in mind that Mr. C++ wants us to use a double equal sign. You might also notice that we use apostrophes instead of quotes around the period. Apostrophes around a single character tell Mr. C++ that we want to compare the number representation (ASCII code) to the current character. Why would we want to do this? Well, mainly because it is faster and easier to read. If the current character is either a number (meaning isdigit returned true) or the current character is a decimal point, we know that we have the beginning of a number word, so we call GetNumber and return. If the current character is not a digit, then KoolB patiently tries the next word type:

```
if (Book[BookMark] == '\"'){
  GetString();
  return;
}
```

Here we are checking to see if the current position is equal to (remember to use the double equal signs) a quotation mark, which begins a string. If the current position is a quotation mark, KoolB calls GetString read the rest of the string and then exits the function, having read a word. If you look carefully, you will see that we are using escape characters by embedding a quotation mark inside apostrophes. By now, KoolB is getting a bit discouraged with all these rejections, but since KoolB has inherited persistence from her parents, she will try one more time:

```
if (ispunct(Book[BookMark])){
  GetSymbol();
  return;
}
```

KoolB's last-ditch effort to identify the word is to see if it is a symbol like '=', '[', or '>', so she calls ispunct. If ispunct returns true, KoolB knows that she has identified a symbol, so she calls GetSymbol and then returns from the function. If KoolB finds that the current position in her book is not the start of an end-of-line, identifier, number, string, or symbol, she gives up and assumes that she has reached the end of the file:

```
  TypeOfWord = None;
  return;
}
```

Since KoolB hasn't been able to identify the word, KoolB just assumes that their isn't a word left to identify. So we assign None to TypeOfWord and return.

Now we need to make the functions used in GetNextWord work. So let us get to work the first one, GetWhiteSpace:

```
bool Reading::SkipWhiteSpace(){
  CurrentWord = "";
  while(isspace(Book[BookMark]) || Book[BookMark] == '\''){
    if (Book[BookMark] == '\''){
      BookMark++;
      while(Book[BookMark] != '\n' && BookMark < BookLength){
        BookMark++;
      }
    }
    if (Book[BookMark] == '\n'){
      CurrentWord = Book[BookMark];
      TypeOfWord  = EndOfLine;
      BookMark++;
      CurrentLine++;
      return false;
    }
    BookMark++;
  }
  return true;
}
```

Like almost all the remaining functions in the reading module, this function does a lot of confusing loops and statuses. Let us analyze it step by step:

```
bool Reading::SkipWhiteSpace(){
  CurrentWord = "";
```

Here we have a function named SkipWhiteSpace (yes, I know I like to state the obvious) that belongs to the Reading object. Unlike most other functions we have created, this one actually returns something! Yes, it returns bool, which is short for Boolean. So SkipWhiteSpace returns either true or false depending on whether or not it found an end of the line. If it did find an end of the line, SkipWhiteSpace will return false, otherwise it will return true because it skipped all the white space. If it encounters an end of the line, SkipWhiteSpace stops even though white space might exist after the end of the line. We don’t need any variables for this function, but we do need to clear out the old word from the last GetNextWord (or nothing if this is the first time GetNextWord has been called). Now we need to introduce you to another one of Mr. C++’s control statement:

```
  while(isspace(Book[BookMark]) || Book[BookMark] == '\''){
```

Yes, this is Mr. C++’s version of the while statement. Just as with the if statement, the while statement must be spelled all lowercase, the expression to evaluate must be enclosed in parenthesis, and the body of the while statement must be enclosed in curly brackets. Here we can see that we are going to loop through the body while the current position is either whitespace (Mr. C++’s isspace tells us this) or the current position is the beginning of a comment. You might notice that when we compare the Book[BookMark] to ‘, we need to use a backslash before it. Just as in Mrs. Rapid-Q we can embed a quotation mark into a string with escape chars like \”, Mr. C++ allows us to do the same thing with the apostrophe \’. To get the ASCII representation of \’, we need to put quotation marks around it, so we end up with ‘\’’, which looks sort of strange. If either is true (notice that the double || is Mr. C++’s version of Mrs. Rapid-Q’s OR), the body of the while loop runs. Well, what do we want to do if we have a comment or whitespace? We don’t need to do anything with it, so let’s just discard it by advancing our BookMark further into the book (by doing this, we effectively skip over the junk we don’t need):

```
    if (Book[BookMark] == '\''){
      BookMark++;
      while(Book[BookMark] != '\n' && BookMark < BookLength){
        BookMark++;
      }
    }
```

Here we have a bigger chunk of code to analyze. Think of it as a complement on your improving C++ skills. In the first line, we compare the character at the current position to an apostrophe. If we finds that the current position is the start of a BASIC comment, we first add one (or increment) to the BookMark by using the ++ operator. That way we can then analyze the next character. In the third line, we have an another while loop inside the first while loop. This one compares the current position to a new line because that is where a BASIC comment ends. However, you notice that we aren’t comparing them with the double equal signs anymore, instead we are using !=. Now what on earth does that mean? Well, Mr. C++ uses an exclamation point (!) in place of Mrs. Rapid-Q’s NOT, so != is really NOT equal. Interesting, huh? Certainly different from <>. The next part is also something you might not be familiar with: &&. Just as || is equivalent to Mrs. Rapid-Q’s OR operand, && is equivalent to Mrs. Rapid-Q’s AND operand. The last part compares our BookMark, our current position, to BookLength, the length of the book. If BookMark is greater than BookLength, we need to stop since we cannot read past the end of the book! So why would we ever need this? Consider this, on the last line of the file, somebody puts a comment like:

```
Print “XXX”       ‘Displays XXX on the console
```

If that was the end of the book, we would try to read past it because there is no end of line, just the end of the file. This would probably cause a crash. To sum up this code fragment, while the current position is not the end of a line and we haven’t read past the length of the book, we just skip past the characters by using BookMark++. When the while statement finally gets to either the end of the line or the end of the book, we stop and continue to the next section:

```
    if (Book[BookMark] == '\n'){
      CurrentWord = Book[BookMark];
      TypeOfWord  = EndOfLine;
      BookMark++;
      CurrentLine++;
      return false;
    }
```

Here we have another if statement comparing the current position to the end of the line. We do this for a couple reasons. First, we need to check for any end of the line words (or tokens); secondly, we need to clean up after any comments. If you go back and look at the code that skips all the comments, you will notice that it doesn’t skip the end of the line, so we have to take care of it here. If the current position is the end of the line, we set CurrentWord to the current position, which contains a newline character, and the TypeOfWord to EndOfLine. We also add one to our position, BookMark, and the variable that keeps track of what line we are on, CurrentLine. Finally we return false to indicate that we have not skipped all the white space and that we have found a word. Now we finish out the function:

```
    BookMark++;
  }
  return true;
}
```

The first line here is the last part of the first while where we add the one to BookMark. We do this because we need update the current position for when the while condition runs again, comparing the current position to whitespace. When the while function finally ends because it encountered a non-whitespace character, the function returns true. Now that we have the whitespace out of the way, we GetNextWord can identify the word and call the appropriate function to deal with each particular type of word. Before we can use them, however, we need to actually program them one by one. The first one we will examine is GetIdentifier:

```
void Reading::GetIdentifier(){
  do{
    CurrentWord += toupper(Book[BookMark]);
    BookMark++;
  }while(isalpha(Book[BookMark]) || isdigit(Book[BookMark]));
  TypeOfWord  = Identifier;
}
```

Here we have a function named GetIdentifier that doesn't return anything and is part of the Reading object. For the first time, we have a variation on the while statement. Why don't we use a plain while statement? We could except we will always loop through this at least once (because a word must have a length of at least one character), and do…while loops are ideally suited to this. Here we have the keyword do followed by a curly parenthesis to contain the body of the do function. Inside the do statement we add the current character to CurrentWord, which was assigned an empty string in GetWhiteSpace, the current position. We also use Mr. C++'s function toupper, which makes each character uppercase. That makes it easier when comparing two strings (because Mr. C++ doesn't consider "For" and "FOR" the same thing). For those not familiar with the += sign, it adds the value to the current variable. For example, A += 3 is the same thing as A = A + 3; it is just one of those little niceties that Mr. C++ throws in for us to use. Once we have added the uppercase current character to the current word, we advance the current position by one by incrementing BookMark. How many times do we need to loop through and add the next character to our word? Until we encounter a character that is not part of an identifier. So while the next character is part of the alphabet or a digit (digits can be in identifiers, like My1stLong) , we loop through the while body again. When the do statement finally encounters a character that doesn't meet these conditions, it stops and assigns Identifier to TypeOfWord. Next we have to handle the getting of numbers:

```
void Reading::GetNumber(){
  bool ReachedDot = false;
  do{
    if (Book[BookMark] == '.'){
      if (ReachedDot == true){
        cout << "Error on line: " << CurrentLine << endl;
        cout << "You cannot have more than one decimal point in a number!"
             << endl;
        exit(0);
      }
      ReachedDot = true;
    }
    CurrentWord += Book[BookMark];
    BookMark++;
  }while(isdigit(Book[BookMark]) || Book[BookMark] == '.');
  if (CurrentWord.length() == 1 && ReachedDot == true){
    TypeOfWord = Symbol;
  }
  else{
    TypeOfWord  = Number;
  }
}
```

This function, GetNumber, is a bit more complex than the GetIdentifier function because we have two things to watch out for. First, numbers can contain a decimal point; simple enough, except each number can contain only ONE decimal point, which means we have to keep track of each decimal point. Second, a single dot (like to access TYPEs such as MyRect.Left) is not a number, it is a symbol (which we will cover later). So with this in mind, let's analyze the function. Unlike some of our previous functions, we need a local variable, ReachedDot to keep track of how many decimal points we have counted. We will make it a Boolean variable and set it to false at the beginning of the function because we haven't found any decimal points yet. Next we have our familiar do loop. First we have compare the current position to a decimal point. If the current position is a decimal point, we next see if we have already found a decimal point (if we have, then ReachedDot would be true). If it is, print out an error (you may notice that one of our cout statements spans a line. That is OK with Mr. C++.) and then exit. We used the exit(1) function earlier and said it just terminated our program nearly immediately. If ReachedDot is false, which means this is the first decimal point we have encountered, we skip giving the message and assign true to ReachedDot because we now have reached the decimal point. Then we add the current character to our word and loop through again. We only loop through the do statement while the current character is a digit or if the current character is a decimal point. When we finally come to a character that does meet one of our two conditions, we know we have found the end of the number. Now we have to check to see if we have a period all by itself, because then it would be a symbol and not a number. So we compare the length of our current word to one (but that doesn't cover us completely, consider this number: 1) and compare ReachedDot to true. If both are true, then we have a single period, so we assign Symbol to TypeOfWord. If either one are false (or both are false), we assign Number to TypeOfWord. Now lets teach KoolB to handle strings:

```
void Reading::GetString(){
  BookMark++;
  while (Book[BookMark] != '\"'){
    CurrentWord += Book[BookMark];
    BookMark++;
    if (BookMark > BookLength || Book[BookMark] == '\n'){
        cout << "Error on line: " << CurrentLine << endl;
        cout << "An end to the string does not exist!" << endl;
        exit(0);
      }
  }
  BookMark++;
  TypeOfWord = String;
}
```

Here we have a function named GetString from the Reading object. If you look at the previous ones, you will notice quite a difference. First, we increment the BookMark before we even start looping. Why do we do this? Well, a string is anything that comes between " and another ". But we only want the contents of the string, right? We don't want the opening “ and the closing ”, so we just skip right over it. Now we have a plain while statement that loops through the body while the current position is not the closing ” character. In the body of the while statement, we first add the current character to the word and go onto the next character. However, we have a problem to look out for. Have you ever accidentally done this:

```
Print “Hello, I forgot something here
```

What are we missing here? The ending quotation mark! We need to be able to catch these errors. So before looping back around again to get the next character, we do some comparisons. If our BookMark is greater than our BookLength, we know we have reached the end of the file. Or if the current position is the end of the line, we know that we have an error. So we dutifully print out what line the error occurred on and the reason why. Later, we might actually add an error object that would do this for us (you know how Rapid-Q prints out the line and then a little ^ to indicate where on the line the error is), but for now, a simple cout will do. When we finally encounter the closing ”, we don’t add it to the word, instead we exit from the while statement and then increment BookMark to get skip past the offending ” character. Finally, we assign String to TypeOfWord.

Now we need to add a function that gets a symbol:

```
void Reading::GetSymbol(){
  if (Book[BookMark] == '_'){
     BookMark++;
     if(SkipWhiteSpace() == true){
        cout << "Error on line: " << CurrentLine << endl;
        cout << "A newline should follow _" << endl;
        exit(1);
     }
     GetNextWord();
     return;
  }
  if (Book[BookMark] == ':'){
    CurrentWord = '\n';
    BookMark++;
    TypeOfWord = EndOfLine;
    return;
  }
  CurrentWord = Book[BookMark];
  BookMark++;
  TypeOfWord  = Symbol;
}
```

Here we have a function named GetSymbol. Ordinarily, it would be pretty simple except for one fact: several symbols mean something completely different. For example, ‘_’ means continue on the next line while ‘:’ means break this statement into two lines. So first we see if the current position is ‘_’. If it is, we skip past it and call SkipWhiteSpace to see if there is a new line after it. If SkipWhiteSpace returns true, that means it didn't encounter a new line, so we output an error and exit from the program. If a new line does follow '_', we call GetNextWord to get the first word on the next line and then return from the function. If the current position is ':', we assign a new line character to CurrentWord, increment the position in our book, tell KoolB that the TypeOfWord is EndOfLine, and return from the function. If the current position is neither of these two symbols, we just assign the current character to the CurrentWord, skip past the symbol, and assign Symbol to TypeOfWord.

I have good news for you: the rest of the functions we have to implement are a breeze from here on! The next one is CheckTokenLength:

```
void Reading::CheckWordLength(){
  if (CurrentWord.length() > 128 && TypeOfWord != String){
        cout << "Error on line: " << CurrentLine << endl;
        cout << "A word cannot exceed 128 characters." << endl;
        exit(1);
  }
  return;
}
```

Here we have a function CheckWordLength that checks to see if the current word is longer than 128 characters. Why do we need this? Well, try DIMing a variable in Rapid-Q that has a name longer than 96 charcters. What happens? Yep, Rapid-Q crashes! Also, assemblers will either crash (NASM) or give an error (FAsm) if a variable is too long. Arbitrarily, I picked 128 characters, or half a byte. FAsm only allows 255 characters, and I wanted to be sure that we could use any assembler we wanted. The other condition is that the word isn't a string, because literal strings can be longer than 128. Heck, I've had 50 KB long strings for testing purposes. If the current word is longer than 128 characters and is not a String, we print out an error and exit the program.

The next two are so easy that I will just describe them together:

```
string Reading::Word(){
  CheckWordLength();
  return CurrentWord;
}

int Reading::WordType(){
  return TypeOfWord;
}
```

Here we have two functions that are part of the Reading object: Word and WordType. The function Word checks to make sure a the current word isn't too long and then returns a string containing the current word. WordType returns an integer containing the type of word the CurrentWord is.

Testing Our Work:

Now that we have a fully functioning Reading object, lets test it out! How about we open a file, read the each word, and identify it? To start, swap back to Dev-C++ and to the KoolBmain.cpp file. First we need to include our Reading object and create it:

```
#include "Read.h"
  Reading Read;
```

Here we include the file “Read.h” (or whatever you named the Reading module). The reason we use quotation marks around the name is because this is OUR file. If we are somebody else’s file that comes with the C++ compiler, we use < and >. But since we built our own file (and you ought to be proud of that), we can use “ and “. Now let’s code some little stuff to show how off KoolB’s new reading abilities. First, modify the function

```
Compile();
```

to

```
void Compile(int argc, char *argv[]);
```

and likewise:

```
void Compile(){
```

to

```
void Compile(int argc, char *argv[]){
```

Finally, change how you call Compile in the main function:

```
  Compile();
```

to

```
  Compile(argc, argv);
```

This passes argc (number of command line parameters) and argv (the array of the actual command line parameters) to Compile. Now go to the Compile function (I told you we would fill that in) and add:

```
  if (argc != 2){
    cout << "Usage: KoolB <filename>" << endl;
    exit(1);
  }
```

First, since we expect KoolB to have only two parameters (like: "KoolB test.bas") we compare argc with 2. If argc is unequal to 2, we print out the how to use KoolB and exit the program. If argc is two, we open the file by passing the second parameter to OpenBook:

```
  Read.OpenBook(argv[1]);
```

Here we call the Read object's OpenBook function and pass argv[1] to it. Why not argv[2]? Because Mr. C++ arrays start at zero, so argv[0] is the name of the program (in our case "KoolB") while argv[1] is the name of the file to open. OpenBook handles all the errors for us, so we don't have to worry about the file not opening. Now for the rest of the function:

```
  Read.GetNextWord();
  while (Read.WordType() != Read.None){
    switch(Read.WordType()){
      case Read.Identifier:
        cout << "Got Identifier:\t\t" << Read.Word() << endl;
        break;
      case Read.Number:
        cout << "Got Number:\t\t" << Read.Word() << endl;
        break;
      case Read.String:
        cout << "Got String:\t\t" << Read.Word() << endl;
        break;
      case Read.Symbol:
        cout << "Got Symbol:\t\t" << Read.Word() << endl;
        break;
      case Read.EndOfLine:
        cout << endl;
        break;
      default:
        cout << "Error!" << endl;
        break;
    }
    Sleep(150);
    Read.GetNextWord();
  }
  return;
```

Wow! This is something we haven't seen before, but actually it is pretty simple. The first part is:

```
  Read.GetNextWord();
```

This gets the next (or first) word from our book. Next we start a while loop:

```
while (Read.WordType() != Read.None){
```

We now want to loop through each word and print out it out along with what type it is. So we loop through the book until we reach the end. The end occurs when the Read.WordType equals Read.None. Until we get their, we loop through the body:

```
    switch(Read.WordType()){
```

Switch is a new keyword, but is similar to Mrs. Rapid-Q's select keyword. It takes a variable and then does something different depending on what that variable contains. The variable we want to examine is the type of the current word, or Read.WordType. Just like almost everything else in Mr. C++'s vocabulary, we must use curly quotations to surround the body. The first part of the body is this:

```
      case Read.Identifier:
        cout << "Got Identifier:\t\t" << Read.Word() << endl;
        break;
```

Case is just another way of saying if. So if Read.WordType equals Read.Identifer, we print out "Got Identifier", two tabs, the word, and then the end of a line. The break statement says we are done doing everything we need to do for the case Read.Identifier. The other ones are very similar, just varying in their comparisons, so we won't cover them. However, the last one is a bit different:

```
      default:
        cout << "Error!" << endl;
        break;
```

If none of the other options work, we default to printing out an error. We should never get to this, but you never know… Finally, right after the switch statement, we have:

```
    Sleep(150);
    Read.GetNextWord();
  }
```

In between the words, we need some sort of pause otherwise they will all scroll past in a fraction of a second. So we use Sleep to pause the program for a little bit. Mr. C++'s Sleep is exactly like Mrs. Rapid-Q's except Mr. C++ wants the amount of time to pause in milliseconds instead of seconds. After sleeping for about 3/20ths of a second, we get the next word and loop back around. Compile that and we are off.

As you are now one big knot of anticipation, bring up a console window and type in: "cd C:\Compiler\KoolB". First lets try throwing some curve balls at it by typing in "KoolB". You should get something like this:

```
Welcome to the KoolB Compiler by Brian C. Becke
Usage: KoolB <filename>
```

Pretty neat, huh? The same thing should happen if you type in “KoolB Too Many Parameters.” Now lets try something more useful: “KoolB KoolBmain.cpp” (or whatever you named your main C++ file). Wow, pretty impressive! If everything is working, you should get a long list of formatted words like this:

```
Got String:             Compile Time:\t
Got Symbol:             <
Got Symbol:             <
Got Symbol:             (
Got Identifier:         GETTICKCOUNT
Got Symbol:             (
Got Symbol:             )
Got Symbol:             -
Got Identifier:         STARTTIME
Got Symbol:             )
Got Symbol:             /
Got Number:             1000.0
Got Symbol:             <
Got Symbol:             <
Got Identifier:         ENDL
Got Symbol:             ;

Got Identifier:         RETURN
Got Symbol:             ;

Got Symbol:             }

Compile Time:   60.513
```

If you get tired of watching the entire file being read, hit Ctrl + C. Our compiler is slowly approaching a stage were it will actually be useful! I hear somebody back there say, “It’s about time!”

Exercises for the Reader:

Yes, you read right. I can’t put everything into the compiler, even if I wanted to, so you will have to fill in the gaps. Besides, it will give you practices and will give you something to do until I finish the next tutorial. Without further ado, were is a list of some improvements I thought up:

1. Add the ability to use comments with the REM keyword. Hint: In the SkipWhiteSpace function, try comparing the first letter, then the second letter, then the last letter to each character in the book and then treating the REM like ‘.
2. Add the ability to use underscores in a word, but remember that it cannot be at the end of the word otherwise the KoolB might get confused in trying to figure out if the underscore is part of the variable or if it is continuing the rest of the statement on the next line. Hint: Try adding the ability to have an underscore inside an Identifier in the middle of a variable like ‘a_a’, but not at the end of the variable like ‘a_’.

3. Add the ability to use floating-point scientific notation. For example, 1.0e10 or –1.0e-10, etc. Hint: Don't try to associate the first negative sign with the number, but if you have e-X, keep the negative; also have a ReachedE variable in the GetNumber variable to keep track of if you have encountered an 'e' or not.
4. If you feel you need a challenge, try adding the ability to use hex, octal, and binary numbers to the GetNumber function. Hint: use either prefixes or suffixes to help you determine what base each number is in; also try finding routines to convert them to base 10 so we can easily manipulate them.
5. If you are really ambitious, see if you can figure out and program KoolB to accept wildcards in file names. Like: "KoolB *.bas" to compile all basic programs in a folder (it's sort of nice as it relieves the need for BAT files). Hint: Use some Windows API functions like FindFirstFile and FindNextFile and a couple more while loops in the Compile function.

Conclusion:

Well, here we are—done with Chapter 3. I bet you thought we would never get this far, but we have. This chapter was pretty full with all the string manipulations we had to do to extract and identify words (or tokens) and all the loops we had to do. But you survived and are hopefully considering doing some homew…oops, I mean some exercises. After this exhausting chapter, I am sure you are anxiously awaiting the next one. ;-) Well, maybe not, but here is an overview of the next chapter. In Chapter 4, we will be teaching KoolB how to write. In addition, we will be meeting old grandfather Assembly (nicknamed Asm) and becoming acquainted with him. We will probably set up a Writing object so KoolB can write our Asm files. In addition, we will set up our assembler and set up our compiler to actually produce working programs (they won't do anything except for run and exit, though). Well, that is enough to whet your appetite. See you then!