Chapter 6
Error Checking & Reporting

Introduction:

Welcome back. After adding simple data types to the compiler, I am sure you are ready to keep slogging through the rest of the BASIC language features. Sorry to disappoint you, but we need detour a little to take care of something that needs a little (well, a lot!) of attention. That subject is error checking. Our compiler KoolB lacks quite a bit when it comes to this subject. As of now, all KoolB does is give cryptic errors and remarks about invalid thises and thats. Our goal is to remedy that problem and make KoolB a compiler that gives understandable errors reports to help the user pinpoint the problem and then correct it. In this manner, KoolB will find errors in the user's BASIC program and try to give suggestions on how to fix them. Before we delve into programming, we need to review how we plan to add error checking and reporting.

When it comes to error checking and reporting, KoolB has several options that we should take a look at:

1. Just stop or crash at an error.
2. Stop at the first error and give a thorough report.
3. Try to find as many errors as possible and give a short report on all them.

The easiest, of course, would be to just stop or crash at an error, but the user would be extremely unhappy programmer if KoolB were to do that. The second option would be a bit harder, and is one taken by many of the popular compiler such as Turbo Pascal and our own favorite, Rapid-Q. The third option is probably the hardest and most comprehensive; many of the really hard-core compilers use this method such as C++ compilers (strangely, not assemblers usually).

So what will KoolB do? Well, we can rule out the first one (that is pretty obvious as we wouldn't spend a whole chapter on error checking and reporting just to have Koolb crash), as it is plainly not a good option if we want happy users. The third is good, but it requires a lot of extra hassle, and often it doesn't work as well as it claims to work. Those of you who use C++ know what I am talking about: if you accidentally forget something vital like an ending brace (or curly bracket) on a function, C++ thinks the rest of the program is inside that function. Obviously, it is going to give a whole lot of errors that don't even relate to the missing ending curly bracket. I once got over 300 errors for doing something silly like that. In addition to being not exactly always right, it has a habit of being extremely cryptic since the report of each error is usually a one-line. "Parse error at end of input" or "Parse error before line 3970". What does that mean!?! Not much to the average programmer. To take the best of both worlds here, we are going to use the second option. It isn't really easy, but it isn't really hard either. It gives simple, (hopefully) meaningful reports at the first error it comes to – then it just exits. The user can correct the mistake and recompile to find the next mistake, until the

BASIC source code doesn't have any more errors and compiles completely. If you don't agree with me, change it; you have the source code to the compiler.

Before we can start programming the error module, we need to create a format for our errors. After looking at other compiler's error reports, I sort of combined the features I liked and came up with this format (sample error report):

```
Error in file "test.bas" on line 1:
  Cause: "EGGPLANT" is an unknown data type, please choose a valid data type.

  Line of code in error:
    Dim Message As EggPlant
                   ^^^^^^^^--Error!
  Extra Info:
    Please reference the Chapter 3, Section 6 of the documentation for a list
of valid data types.
```

That is the basic structure of our error report. You can see that it basically has four sections:

1. Location — This tells the user what file the error occurred in and on which line the error occurred.
2. Cause — Gives a brief description of the problem KoolB encountered.
3. Line of code — Prints the line of code where the error occurred and places little hats (^) under the word or place nearest to the error.
4. Extra Info — This is optional; it just gives some extra info that might help the user correct the error.

Now that we know what we are dealing with, let's start programming it.

The Modifying the Compiler Object:

In order to incorporate the Errors object into the Compiler object, I need you to use the search function of Dev-C++ to find and replace our old puny errors with the new error functions in our Errors object. To help you with this, I am providing a table below. The idea is to find and replace the text on the left with the text on the right-hand side of the table:

`cout << "Error - Invalid Command" << endl;`	`Error.BadStatement(Read);`
`cout << "Error - Expected End-Of-Line" << endl;`	`Error.EndOfLine(Read);`
`cout << "Error - Invalid name" << endl;`	`Error.BadName(Read);`
`cout << "Error - Invalid Type" << endl;`	`Error.NoType(Read);`
`cout << "Error - Expected Name" << endl;`	`Error.ExpectedNameAfterDIM(Read);`
`cout << "Error - Invalid name" << endl;`	`Error.BadName(Read);`
`cout << "Error - Already reserved" << endl;`	`Error.AlreadyReserved(Read);`
`cout << "Error - Expected 'As'" << endl;`	`Error.ExpectedAs(Read);`
`cout << "Error - Invalid Type" << endl;`	`Error.BadType(Read);`

The idea here is to replace all our error reporting code with calls to our Errors object so it can do a full analysis and error report. As you can see, the function names are similar to the problem, so you shouldn't have too much trouble. We also pass the Reading object that the Compiling object is using to each function of the Errors object to build our error report. Speaking of the Reading object, the Errors object is going to need to be able to access the internals of the Reading object to get things like the line on which the error occurred. Unfortunately, we said that all this data was private, meaning only the Reading object could access it. This is a good idea since we don't want people modifying these values, but there is no harm in adding some functions to the Reading object that allow anybody read-only access to these values (By read-only, I mean that nobody but the Reading object can actually modify them, only look at them). To do this, swap to Dev-C++ and to the Reading object. In the public section, after all the rest of the functions, declare these following functions:

```
   long GetBookMark();
   long GetCurrentLine();
   string GetBookName();
   string GetBook();
```

These functions will just get the data from the Reading object and then return it. Now move down to the bottom of the file and create the code for these functions:

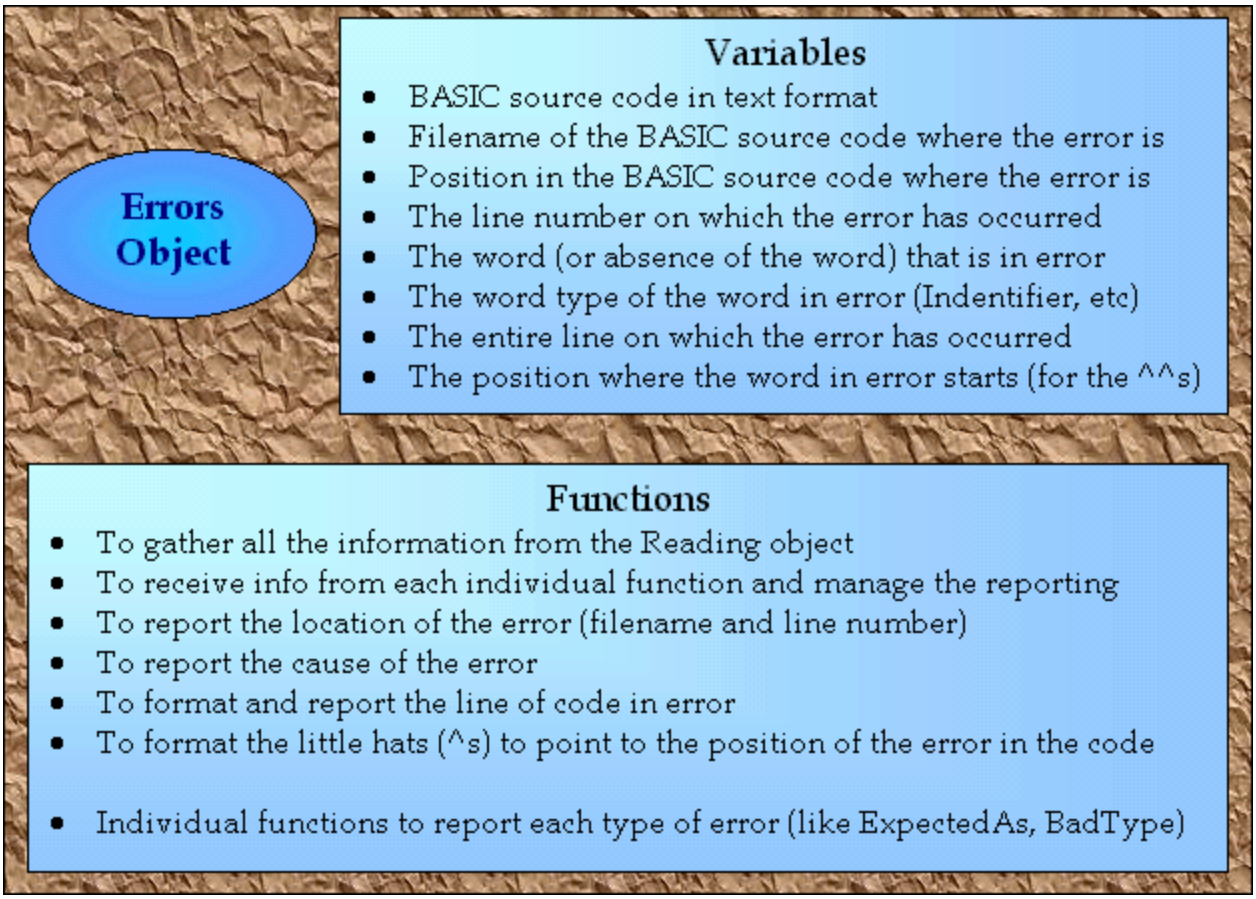
```
long Reading::GetBookMark(){
  return BookMark;
}

long Reading::GetCurrentLine(){
  return CurrentLine;
}

string Reading::GetBookName(){
  return BookName;
}

string Reading::GetBook(){
  return Book;
}
```

That is pretty straightforward, isn't it? These functions just return the data that the Errors object will need. Now everything is read for us to create the Errors object.

Creating the Errors Object – Part 1:

The Errors object is divided into two general parts: the core part and then the individual error reports part. The first part does the heavy work of gathering information, formatting the information, and reporting the error. The second part is composed of the individual functions like BadName, ExpectedAs, and ExpectedNameAfterDIM. These supply the core functions with the data needed to

format and report the error. Think of it as a computer: the computer does all the hard work, but it can only do it if you supply the information. As normal, let's look briefly at the overall structure of the Errors object:

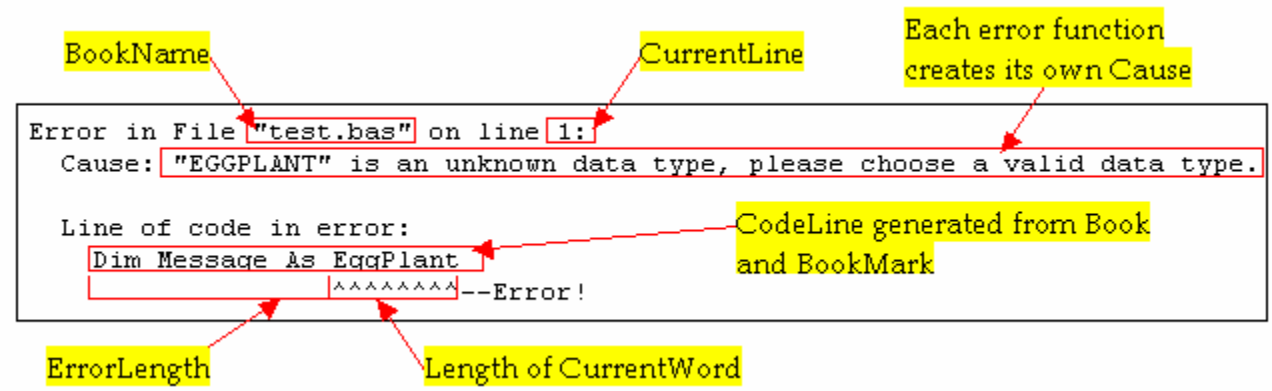


This is quite a large object! It has many functions and whole slew of data that it needs. Now let's look at the source code for the object:

```
class Errors{
 private:
   string Book;
   string BookName;
   long   BookMark;
   long   CurrentLine;
   string CurrentWord;
   int    TypeOfWord;
   string CodeLine;
   long   ErrorLength;

 public:
  void Prepare(Reading SourceFile);
  void PrintError(string Cause);
  void PrintLocation();
  void PrintCause(string Cause);
  void PrintCode();
```

```
  void PrintErrorPosition();

  void EndOfLine(Reading SourceFile);
  void BadStatement(Reading SourceFile);
  void BadName(Reading SourceFile);
  void ExpectedNameAfterDIM(Reading SourceFile);
  void BadType(Reading SourceFile);
  void NoType(Reading SourceFile);
  void ExpectedAs(Reading SourceFile);
  void AlreadyReserved(Reading SourceFile);

};
```

Whew! That is a lot of data and functions! I don't expect you to know what all that means at one glance, so let's break it down and analyze it. First let's look at the variables and how the relate to the error message we saw earlier. Some of them relate directly and others are only used in to generate other data. Anyhow, I think it would be beneficial for you to see how they inter-relate (Note: I don't show the extra info here):



That should give you an idea of the internal structure of the Errors object. Now let's get down to the details by looking at the functions. First of all, we don't need a constructor for the object. Second, we have the two groups of functions we mentioned earlier. Let's look at the first group.

The first function in the first group is Prepare, which takes one parameter: a Reading object. The sole purpose of Prepare is to gather information from the Reading object and store it into the object's private section. For instance, Prepare will call the Reading object's function GetBook to get the BASIC source code and store that in Error's private string named Book. Essentially, it is the gofer for the object; it copies data from one object to another. This allows any other function in the Errors object to access a copy of the data in the Reading object. That should be enough of an introduction, so let's look at the code:

```
void Errors::Prepare(Reading SourceFile){
  CurrentWord = SourceFile.Word();
  TypeOfWord  = SourceFile.WordType();
  Book        = SourceFile.GetBook();
  BookName    = SourceFile.GetBookName();
  BookMark    = SourceFile.GetBookMark();
  CurrentLine = SourceFile.GetCurrentLine();
```

```
  return;
}
```

This is pretty straightforward. As I said earlier, it just copies all the data in the Reading object to the Errors object.

If the last function, Prepare, was our gofer function, our next function, PrintError, is the manager. It receives information from each one of the functions in the second group and then calls the core functions to format the information into an error report. The one parameter it takes is the cause of the error, which functions from the second group like ExpectedAs will obviously fill in. Once it has the cause of the error, it calls functions to print the location of the error, the cause of the error, the line of code in error, and finally the position of the error on the line (marked by the ^s). Here is the code for PrintError:

```
void Errors::PrintError(string Cause){
  PrintLocation();
  PrintCause(Cause);
  PrintCode();
  PrintErrorPosition();
  return;
}
```

Now you know why I call it the manager function: it doesn't do any of the work itself! It just tells other functions what to do.

The first function in PrintError is PrintLocation. This function reports which file the error occurred in and on what line. It is actually pretty straightforward except one thing. Say that KoolB is parsing the DIM statement and the user accidentally forgets to put the data type. So the user types in something to this effect:

```
Dim Message As
```

Instead of getting "STRING", "DOUBLE", or "INTEGER", KoolB gets an end-of-line word and will automatically adds one to the variable that keeps track of the line. This means that the variable that keeps track of the current line number would be too high by one (since KoolB adds one to CurrentLine every time KoolB encounters an end-of-line character). This problem will plague us for the rest of this chapter, but fortunately, the fix is rather simple. If the current word is an end-of-line word, we just subtract one from the CurrentLine. That brings CurrentLine back down to where it was before KoolB encountered the end of the line. Then, KoolB just prints out the file and the line number. Even the code is pretty simple:

```
void Errors::PrintLocation(){
  if (CurrentWord == "\n"){
    CurrentLine--;
  }
  cout << endl;
  cout << "Error in File \"" << BookName << "\" on line "
```

```
       << CurrentLine << ":" << endl;
  return;
}
```

First, we see if the current word is the end of the line. If it is, we subtract one from CurrentLine (the double minuses do this – it is called decrementing) to make CurrentLine point to the correct line. Then we print out a newline (endl) to give some space between the compiler's header (the small blurb printed out about the compiler) and the error. Then we proceed to print a line that tells the user where the error occurred.

For the next part of the error, we want to print out the cause of the error. This is just a brief description of why KoolB is having trouble compiling the BASIC source code. The function that does the work to print out the cause is PrintCause. It's single parameter, Cause, is a string that contains the reason for the error. PrintCause would be a relatively easy function except for one thing (there is always something, isn't there?). Oftentimes, KoolB will print out the CurrentWord in quotation marks to show the user what s/he put and then tell the user what KoolB expected. For instance, if you look at the example error report near the beginning of the chapter, you will see that it prints out the CurrentWord (EGGPLANT) in quotation marks. However, what happens if the CurrentWord happens to be an end-of-line word? Or worse yet, what happens if CurrentWord is the end of the file? Then you might get something like this:

```
Error in File "test.bas" on line 1:
  Cause: "
" is an unknown data type, please choose a valid data type.

  Line of code in error:
    Dim Message As
                  ^--Error!
```

The problem here is that the CurrentWord is an end-of-line word. So when it gets printed, it does exactly that: ends the current line and starts another line. However, this isn't very friendly behavior. Wouldn't it be so much nicer if we replaced that with:

```
Error in File "test.bas" on line 1:
  Cause: End-Of-Line is an unknown data type, please choose a valid data
type.

  Line of code in error:
    Dim Message As
                  ^--Error!
```

That would be easier to understand, wouldn't it? So how do we do this? We could search for "\n" (\n is the end of line character) and then replace it with "End-of-Line;" that would work, wouldn't it? That complicates our otherwise easy function, but life isn't always fair, now is it? So let's look at how to do this:

```
void Errors::PrintCause(string Cause){
  long Position;
  Position = Cause.find("\"\"", 0);
  if (Position != string::npos){
    Cause.replace(Position, 2, "End-Of-File");
  }
  Position = Cause.find("\"\n\"", 0);
  if (Position != string::npos){
    Cause.replace(Position, 3, "End-Of-Line");
  }
  cout << "  Cause: " + Cause << endl;
  cout << endl;
  return;
}
```

First we have create a number called Position. This will hold where in the string an end-of-line or end-of-file word is so we can replace it. We first search for the end-of-file, which is an empty string. Since we put words from the user's BASIC source code in quotes, we are looking for two quotation marks side by side with nothing between (""). The way we search a string is to call the find function with the string we want to find as the first parameter and the starting position in the string. Here, we are searching for two quotation marks starting at zero, the beginning of the string. If find finds a match, it returns the position in the string where the match starts; if it finds nothing, it returns a strange looking variable: "string::npos." After searching for an end-of-file position, we check the result (which we stored in Position). If Position is unequal to that strange variable above, we know we found a match. So we the empty quotes with "End-Of-Line." How do we do this? Use the replace function! The replace function replaces the original text in the string starting at the first parameter and for the next <second parameter> characters with the new text in the third parameter. So here if we find "", we replace it by passing replace Position (the position immediately before the quotes), followed by 2 (the length of the old text we want to replace: ""), and then the string to take the place of the old text "End-Of-File." We do the same thing again, except we search for the end of the line and replace it with "End-Of-Line." Since the Cause has now been formatted, we go ahead and print it out.

Unfortunately, the next function we will look at, PrintCode, is not easy to understand. PrintCode's job is to extract the line that the error occurred on from the BASIC source code. We start off with three variables: one to hold our current position in the BASIC source code, one to hold the beginning position of the line the error is on, and one to hold the ending position of the line the error is on. The idea is to look for an end-of-line word on either side of our current position in the BASIC source code. Once we have those two positions, we can then extract the sub-string out. However, several problems crop up. What happens if there isn't an end-of-line word on either side? What if this line is the last line of the file? Then there won't be any end-of-line character after our current position, only an end-of-file word. Or what if the current position *is* an end-

of-line word? Then what? To make things easier to understand, I am going to create a little checklist for our PrintCode function:

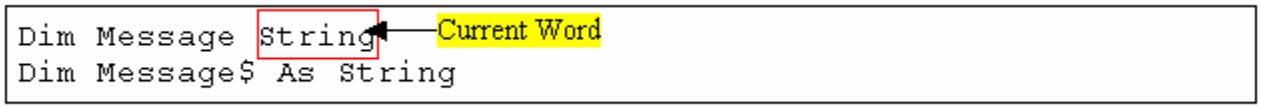
1. Create our three variables, Position (to hold current position), StartPos (to hold beginning position of the line), and EndPos (to hold the ending position of the line).
2. Store one less than BookMark in Position. We do this because BookMark is always one step ahead to keep track of the next word.
3. Make sure that Position isn't straddled on top of an end-of-line word. If it is, move Position to the place right before the end-of-line word by subtracting one.
4. Find the end-of-line word to the left of (before) Position. If there is not one, the beginning of the line starts at the beginning of the file. Put the start of the line in StartPos. If StartPos isn't zero (meaning the line starts at the beginning of the file), add one to StartPos to move StartPos right after the end-of-line word.
5. Find the end-of-line word to the right of (after) Position. If there is not one, the end of the line ends at the end of the file. We put that position in EndPos.
6. Then we get the sub-string starting at StartPos and ending at EndPos and store it in CodeFile.
7. Then we calculate the position of the ^s for later use
8. Finally, we print out our hard earned line of code.

Now let's look at the code, and then an example:

```
void Errors::PrintCode(){
  long Position = BookMark - 1;
  long StartPos;
  long EndPos;
  if (CurrentWord == "\n"){
    Position -= 1;
  }
  StartPos = Book.rfind("\n", Position);
  if (StartPos == string::npos){
     StartPos = 0;
  }
  else{
    StartPos += 1;
  }
  EndPos = Book.find("\n", Position);
  if (EndPos == string::npos){
    EndPos = Book.length();
  }
  CodeLine    = Book.substr(StartPos, EndPos - StartPos);
  ErrorLength = BookMark - StartPos - CurrentWord.length();
  if (CurrentWord == "\n"){
    ErrorLength++;
  }
  cout << "  Line of code in error:" << endl;
  cout << "    " + CodeLine << endl;
```

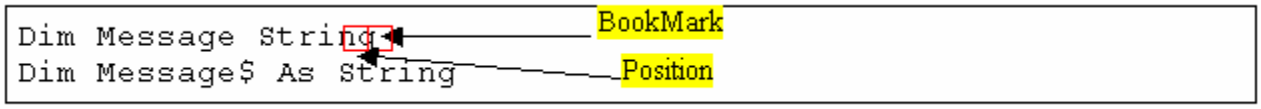
```
  return;
}
```

That is a pretty confusing function, if I do say so myself. Let's look at an example and apply the steps we learned earlier. Here is the entire BASIC source code that we will be compiling:

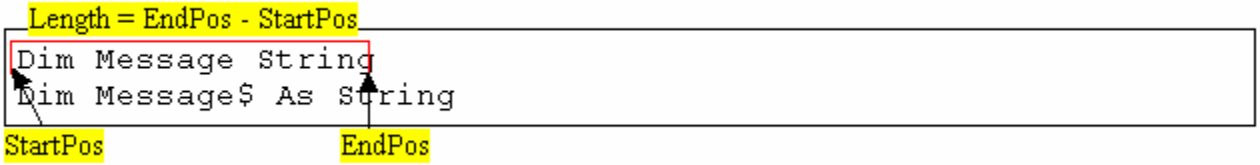


Correct, we are missing the word 'As' between the 'Message' and 'String' on the first line. In another words, the CurrentWord should be 'As', not 'String'. Let's see how we would use PrintCode to get the first line.

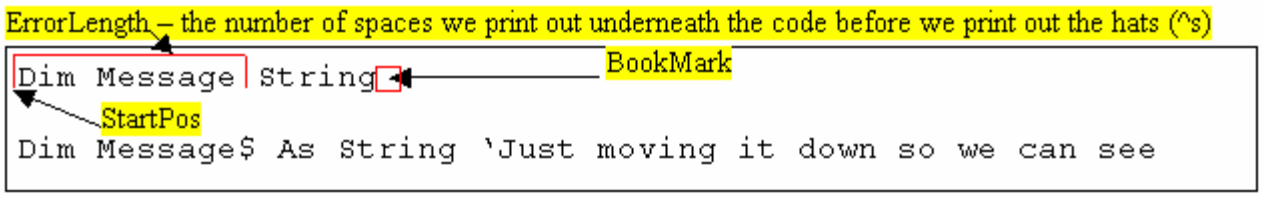
Step 1: Create our variables.
Step 2: Store BookMark – 1 in Position.



Step 3: Position isn't ontop an end-of-line word, so proceed
Step 4: Search for an end-of-line character before Position, which fails, of course. So we instead default StartPos to 0, the beginning of the file.
Step 5: Search for an end-of-line word after Position. We find one right after string, so EndPos is now 18.
Step 6: Now we extract the sub-string. We use the substr function and pass it two parameters, where to start extracting the string and how much of the string to extract. The first parameter is obviously StartPos, but to get the length of the sub-string, we need to subtract StartPos from EndPos.



Step 7: Figure out where the ^s should start (ErrorLength). Here, we would want them to line up the hats under 'String' so we can tell the user that we wanted to see 'As,' not 'String.' However, we need to know how many spaces to print before we start printing out the hats. This takes a bit of thinking, but we will manage.



First we start out with BookMark. Then we subtract StartPos from that to get the length from the beginning of the line to the BookMark. Since we want to put little hats (^s) underneath the current word, we subtract the length of the word to get the position starting just under the word. Now we have the number of spaces we need to position the ^s under the current word.

Step 8: Print out the line of code.

The last of the core Errors functions is PrintErrorPosition, which prints out the little hats underneath the error (usually that is the current word). We know how many spaces to print out (that is just what we calculated), but we have a slight problem. If there is a tab in the original line, we the hats will be messed up because one tab is not the size of one space. That complicates things because now we have to loop through the CodeLine and see if the current position is a tab. If it is, we print out a tab; if it is not, we print out a space. Once we have reached the beginning of the current word (where the error is), we stop printing out spaces and start printing out hats. This tells the user very plainly: "Here is the error!" We keep printing out hats until the current word ends (so the number of hats we print out is equal to the current word's length). However, we have another slight problem. What if the current word is an end-of-file word? Then the length of the current word would be zero! No hats at all would be printed. To get around this, we do a little bit of cheating. Instead of starting at zero (like normal), we start the counter at 1. Then we print hats while our counter is less than the length of the current word. This will short us one hat (because we started at 1, not 0). The solution then we print out "^--Error!" That *always* gives us at least one hat (even if the current word is an empty string and has a length of zero) and it also adds that one extra hat that we need to words that have non-zero lengths. These solutions are a bit out of the ordinary, but they seem to work, so why not? Let's take a closer look at how PrintErrorLength does this:

```
void Errors::PrintErrorPosition(){
  long Position = 0;
  cout << "    ";
  while (Position < ErrorLength){
    if (CodeLine[Position] == '\t'){
      cout << "\t";
    }
    else{
      cout << " ";
    }
    Position++;
  }
```

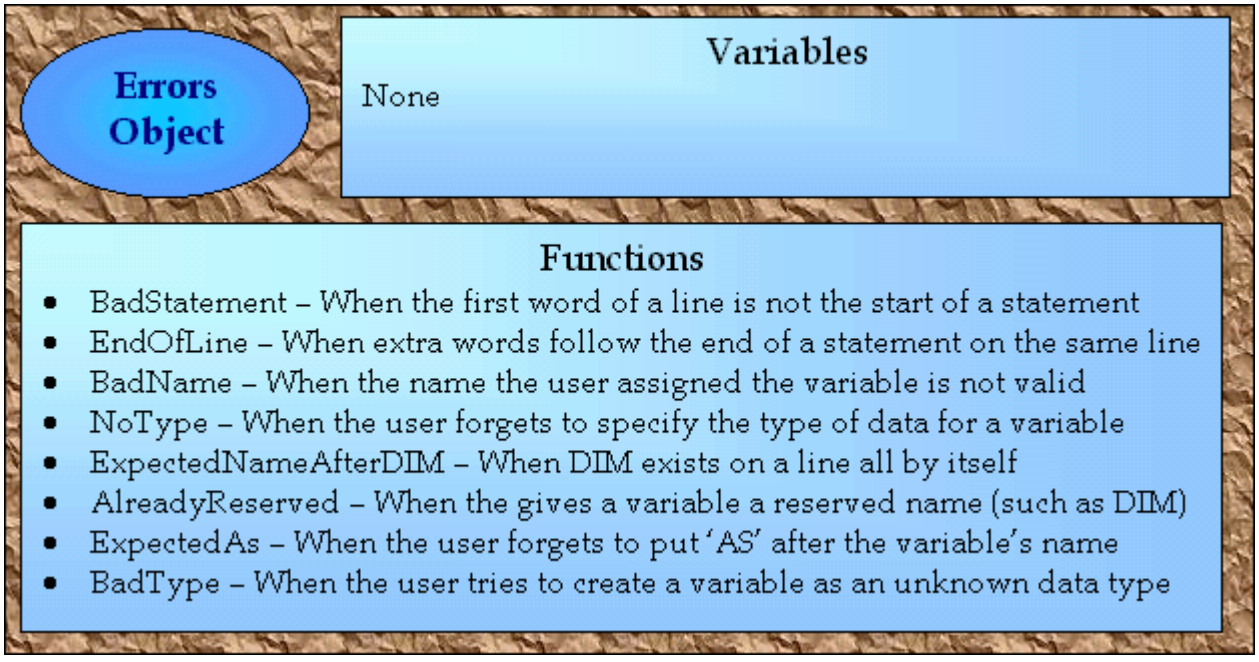
```
  Position = 1;
  while (Position < CurrentWord.length()){
    cout << "^";
    Position++;
  }
  cout << "^--Error!" << endl;
  return;
}
```

First we create our counter called Position and assign it zero. Then we print out four spaces because of the formatting of the error (we indent this part). While our Position is less than ErrorLength (the number of spaces we want), we either print out a single tab or a single space depending if the current position in our CodeLine is a tab or not. We then add one to Position and loop back around. When we have finished printing out all the spaces, we set Position to one. While Position is less than the length of the current word, we print out a single hat. We then add one to Position and loop back around. Finally, we print out another hat and then "^--Error!" to complete the error.

Hurrah! We have now finished our tour of the core Errors functions, so we can move onto the easier second part of the Errors object.

Creating the Errors Object – Part 2:

Part 2 of the creating the Errors object covers the individual error functions that the Compiler object calls such as ExpectedAs or EndOfLine. Just so you don't quit on me or anything, Part 2 will be much easier to understand than Part 1 – in fact, the functions might be so easy and so similar that you might actually get bored! To review the functions that the Compiler object needs, here is the familiar diagram of the Errors object (I have omitted the functions and variables we dealt with earlier):



This diagram should give you an overview of when the Compiler object calls each function. Now we need to examine the code for each. We will start off by thoroughly examining the first function and then briefly glancing at the rest (since the only thing different is the cause of the error).

Let's start off with the most generic: BadStatement. Basically, this just tells the user that KoolB cannot figure out the first word of the first line. Since KoolB only handles the creation of simple variables, this is just about everything. When KoolB encounters a number, for instance, as the first word of the line, it calls BadStatement because it doesn't know how to deal with numbers right now. Once BadStatement is called, it first calls Prepare to transfer all the data from the Reading object to the Errors object. Then it passes the cause of the error to our managerial function PrintError. After that, you can optionally print out additional info for the user. I don't make errors up very well, so I will leave that to you. You can some common solutions to the error, directions for finding the related sections in the docs, common reasons why this error occurs, or just leave it blank. It is all up to you. Finally, now that the error is formatted and printed out, we stop the program to allow the user to fix the problem and re-compile. Now that we know BadStatement's job, let's look at the code:

```
void Errors::BadStatement(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("Expected the beginning of a statement,"
             " but got \"" + CurrentWord + "\"");
  exit(1);
  return;
}
```

Now you know why I said this section would be easier! We just pass the Reading object to Prepare to get variables such as the BASIC source code and the current word. Then we call PrintError with the error (in this case, its "Expected the beginning of a statement, but got "123", where 123 is the current word). Those astute readers out there might notice something odd: between the first line (starting with "Expected…") and the second line (starting with " but…"), there is no plus sign. That is right, C++ allows you to do that. If you want to add two quoted string together, you can just put a space between them. That gives us the opportunity to split this otherwise long string over two lines. Pretty nice, huh? Finally, we exit from the program with abnormal termination (meaning the program encountered an error), and return from the function.

The rest of the functions for this second part of the Errors function differ only in the string that we pass to PrintError (the cause of the error). To avoid needless repetition, I will just display the rest of the function's code right here. Then we can look at them as a whole.

```
void Errors::EndOfLine(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("Expected the end of a statement (End-Of-Line),"
             " but got \"" + CurrentWord + "\"");
  exit(1);
```

```
  return;
}

void Errors::AlreadyReserved(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("\"" + CurrentWord + "\" is already reserved,"
             " please choose a different name.");
  exit(1);
  return;
}

void Errors::BadType(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("\"" + CurrentWord + "\" is an unknown data type,"
             " please choose a valid data type.");
  exit(1);
  return;
}

void Errors::NoType(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("\"" + CurrentWord + "\" has no data type identifier,"
             " please add a valid data type like $ to the end.");
  exit(1);
  return;
}

void Errors::BadName(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("\"" + CurrentWord + "\" is not a valid name, please "
             "choose use only letters and numbers.");
  exit(1);
  return;
}

void Errors::ExpectedNameAfterDIM(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("Expected the variable\'s name after \"DIM\", but "
             "got \"" + CurrentWord + "\"");
  exit(1);
  return;
}

void Errors::ExpectedAs(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("Expected the word \"AS\" after the data\'s name,"
             " but got \"" + CurrentWord + "\"");
  exit(1);
  return;
}
```

There you have it: all the functions for the second part of the Errors object except BadStatement since we already reviewed it. There really isn't much more to say except we will be adding a ***lot*** more as we progress. As a side note, if you feel any of these error messages is lacking, go ahead and change it – I would be happy if you made them

better. Ah, well, I guess I do have something else to say. I mentioned earlier that I recommended adding some little extra info to these error reports, yet I didn't show you how. Let me do so now. Let's take BadType for an example. Right now it looks like this:

```
void Errors::BadType(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("\"" + CurrentWord + "\" is an unknown data type,"
             " please choose a valid data type.");
  exit(1);
  return;
}
```

However, if we wanted to add some extra oomph and make this error report more useful, we could change it to something like this:

```
void Errors::BadType(Reading SourceFile){
  Prepare(SourceFile);
  PrintError("\"" + CurrentWord + "\" is an unknown data type,"
             " please choose a valid data type.");
  cout << endl;
  cout << "  Extra Information:" << endl;
  cout << "    Valid data types are: STRING, INTEGER, and DOUBLE." << endl;
  cout << "    For a list of components, see Chapter 5 of the Docs." << endl;
  exit(1);
  return;
}
```

That sort of puts a finishing touch on the error report, doesn't it? It gives the impression of thoughtfulness as it points the user in the right direction to solve the problem. Now that we have the Errors object completed, we need to #include it in our program. In Dev-C++, go to the main file and add these lines right before you #include the Compiler module:

```
#include "Errors.h"
  Errors Error;
```

There we go! Now we are finished with the code. Let's give what we have been working on a whirl.

Testing It Out:

Now we can do something that we won't be doing very often: deliberately making errors. That's right, in order to check to see if KoolB gives good error reports, we need to deliberately make some errors. So create a file named "Test.bas" in your KoolB folder. Then type in some errors like so:

```
Dim
123
Dim Message
Dim Message String
```

```
Dim Message As EggPlant
Short : Short
Short# Short&
```

Save the file off, grab a console, change the directory to C:\Compiler\KoolB, and then try to compile the file by typing in "KoolB Test.bas." Does KoolB catch the errors? Display them with the right formatting? Are the errors helpful? Hopefully the answer to these questions is yes. If not, you can modify them to your liking or until they meet your standards.

Conclusion:

I am happy to inform you that we have finished everything we needed to do for this chapter. We now have a compiler that not only compiles simple BASIC programs, but also gives good errors to boot. A pretty major accomplishment if I might say so. For the next chapter, we will depart from our detour and re-join the major road in adding more functionality to the KoolB compiler. My plan is for us to add the ability to create slightly more complex data types such as arrays and UDTs in Chapter 7. That should keep us occupied for a while! See you then!