Chapter 5
Compiling Simple Data Types

Introduction:

Welcome back to the Compiler Tutorial Part II! To recap what we have done, the last four chapters have been building up to this point. Up until now, we have just been creating a skeleton for KoolB by teaching her how to read, writing, and create programs. Today, am glad to tell you, we will be moving into another arena, past the tedious and non-rewarding "boiler plate" code. Today we graduate and use the skills we have learned and the code we have programmed in the previous chapters to teach KoolB to compile real BASIC programs and adding features of the BASIC language. So congratulations on your persistence with this tutorial and on learning how to build compilers.

In this chapter, we will be doing quite a bit. Our ultimate goal for the chapter is to be able to compile statements that create simple data types. For example, by the time we finish this chapter, our compiler KoolB should be able to compile the following code into a program:

```
Index&          'A "&" after a variable name means to create an integer
Message$        'A "&" after a variable name means to create a string
PI#             'A "&" after a variable name means to create a double

DIM P AS String
DIM Rate204 as Double
DIM RateSymbol as Integer
DIM Rate188 as DOUBLE
dim x& as integer,dd& as integer
```

I know you are thinking that that really isn't that much of a program, and you are right. But a compiler writer has to start somewhere and data types seem to be a good place to start. As you can see here, we will be dealing with the two normal ways of creating data: through the suffix symbol right after the name of the data and through the traditional DIM statement. We will be teaching KoolB how to recognize both and generate the correct assembly language for each way. Astute readers might notice that we only use three data types: Integer, Double, and String. So where are the others such as Byte, Long, Variant, etc? Well, when I first started this tutorial I had one goal in mind: teach others how to build a compiler. To achieve that goal, I cannot do *everything*, instead, I decided that if I teach you how to manage Integers, Doubles, and Strings, you should be able to figure out how to do the rest. That saves me time (and you lots of reading) and allows you to have some hands-on exercises as you will have to add the code for Byte, Long, etc. So how are we going to achieve our goal? To deal with everything in a timely manner, let's list out some of our tasks that we must complete before the end of Chapter 5:

- Teach KoolB how to compile statements creating simple data types (i.e. numbers and strings) such as: "Dim Password As String". For those wondering, more complex data types would be arrays, User Defined Types, and objects.
- Buff up the Assembly object so we can generate the correct assembly language code for the creation of simple data types. Unfortunately for you, this means learning more about assembly language (but don't worry, it shouldn't be too hard). We will also be re-writing the functions we created last time to add more features.
- Create a Database object to keep track of what simple data types we have, what type they are, what their name is, and other vitally important information.

This doesn't seem to be a lot, but believe me, it is. This chapter will be foundational, as it will explain many of the concepts that we will use frequently in subsequent chapters. It will also require both of us working together. I will be trying to thoroughly and clearly explain everything that you need to know; you will need to concentrate to understand the concepts I present and ask questions about what you don't understand. For those of you who know my style from previous chapters, I am going to spring a (hopefully pleasant) surprise. I plan to completely revamp my way of explaining concepts. As I review the last couple of chapters, I noticed that they were a bit rushed and disjointed with small code segments strewn here and there. I also notice, as I look back, that I could have used more examples and more graphics that help explain concepts visually. So I pledge to do better in this chapter, as it will be the foundation that the rest of the chapters build upon. My approach will be to give diagrams of the objects we plan on creating and explain the purpose of a piece of code before I give you the code, give you the code, and finally discuss how it accomplishes what its purpose. So without further discussion, let us dive into the meat of this chapter.

Keeping Track of Our Data:

As a user starts creating data in his or her BASIC program, we need to keep track of it. To illustrate the need for keeping track of data, what happens if the user does this:

```
Dim Message As String
Dim Message As Integer
```

That could be a problem! Or even worse, what if the user did this:

```
Dim Dim As String
Dim = "What happens here!"
```

Or even something as simple as:

```
Dim Message As String
Message = 3.14
```

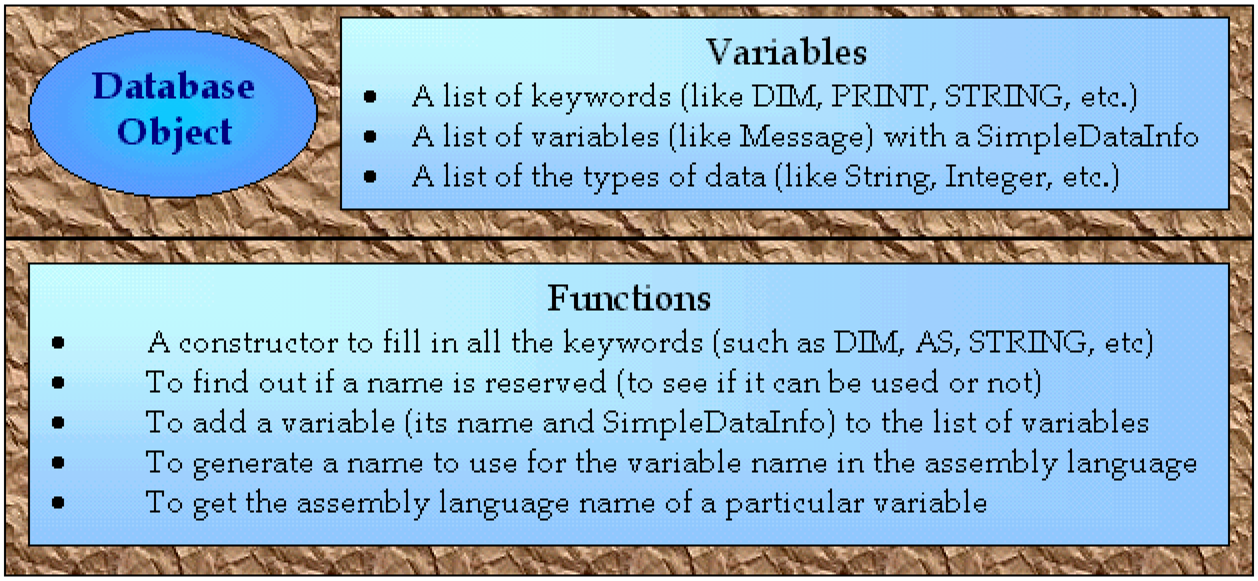
That is just one reason why we need to keep track of our data, but exactly what do we need to keep track of, and how do we do it? First, we need to keep track of the name the user assigns each variable (assuming the name the user uses isn't something like "DIM" or "This is an invalid Data Name"). Second, we need to keep track of the name that we will use for the variable in the generated assembly language. Why would the data's name in assembly language be any different than its name in the BASIC source code? That is a good question and easily answered. What if the user names a string the exact same name as a keyword in assembly language (for example 'MOV' or 'EAX') – that would pose a problem! A few other issues will come up as well, so having a separate name for the variable to use in the assembly language is a good idea. Third, we need to keep track of what type the data is! We cannot allow a user to assign a string to a number, now can we? To prevent this, KoolB has to be able to find out what type of data a particular variable is. So to help us keep track of all this information, we need to do several things. First, start up Dev-C++ and open the KoolB project. Then add a file named Database.h to the project. At the top of the Database file, we need a UDT (User Defined Type) that can hold all the information associated with each simple data type:

```
struct SimpleDataInfo{
  string Name;
  int    Type;
};
```

What does this do? This creates a struct (which stands for structure) named SimpleDataInfo that stores the name and type of each data. For instance, take the following example:

Dim Message As String

Here we have a String named Message. If we had an array of SimpleDataInfo's, we could store "Message_String" for the Name, and 'String' for the Type. Then whenever we encountered the variable Message, we could check its Type and Name. Think of this as an ID card that keeps track of all the information associated with each variable. Not bad, huh? Now let's create an object specifically designed to keep track of all the variables and data we use. Let's call this object 'Database' and put it in the Database.h. Before we delve into the code, let's outline what this object is supposed to do.


Database Object
Variables
A list of keywords (like DIM, PRINT, STRING, etc.)
A list of variables (like Message) with a SimpleDataInfo
A list of the types of data (like String, Integer, etc.)
Functions
A constructor to fill in all the keywords (such as DIM, AS, STRING, etc)
To find out if a name is reserved (to see if it can be used or not)
To add a variable (its name and SimpleDataInfo) to the list of variables
To generate a name to use for the variable name in the assembly language
To get the assembly language name of a particular variable

I am sorry that this is split over two pages; normally, this would all in one diagram. But itshould give you an idea of what the Database is supposed to do. Now let's create the skeleton for it:

```
class Database{
 private:
  vector<string>              KeyWord;
  map<string, SimpleDataInfo> SimpleDataTypes;

 public:
  Database(){
    KeyWord.push_back("DIM");
    KeyWord.push_back("REM");
    KeyWord.push_back("AS");
    KeyWord.push_back("INTEGER");
    KeyWord.push_back("DOUBLE");
    KeyWord.push_back("STRING");
  }
  bool IsAlreadyReserved(string Name);
  void AddSimpleData(string Name, SimpleDataInfo Info);
  string StripJunkOff(string Name);
  string Asm(string Name);

  enum SimpleDataTypes{Integer, Double, String};

};
```

Although a lot of this might be strange looking, you should be able to pick out what some of it does already by just looking at it and remembering what our Database object is supposed to do. As normal, we will run through the code and see what it does. First, let's take a look at our variables, which are a bit different than what we have normally been dealing with:

```
vector<string>              KeyWord;
```

Now, I am sure you are wondering what the heck 'vector <string>' means! Earlier we discussed strings and that C++ didn't really have a string data type, yet we have been making great use of strings throughout our compiler. How have we been doing that? By using a great little piece of code called the Standard Template Library (or STL for short). This nifty library allows us to really use neat data types such as strings.

The STL also has another data type named a 'vector.' What is a vector? A vector is an array that automatically expands when you add stuff to it. Normally, an array has a fixed number of elements (called a static array), but now, we can put in as many elements into it as we want (a dynamic array). So what 'stuff' do we want to put into it? Look further down the line at the name of our vector (or dynamic array): KeyWord. Glance back to the variables we need for our Database object. Well what do you know – the first variable we need is a list of keywords; how much you want to bet that this vector is our list of keywords? Based on that information, we can safely say that we want to put strings into our new vector (like "DIM" and such). So to tell Mr. C++ that we want our vector named KeyWord to contain a list of strings, we put '<string>' right after the word 'vector.' Now if we wanted a list of numbers, we could put '<double>' after the word 'vector.' This vector named KeyWord is our dynamic array to contain a list of BASIC keywords.

```
map<string, SimpleDataInfo> SimpleDataTypes;
```

Now hold onto your hats because now we are going to meet one of vector's buddies: map. Just like vector, map is a data type that is part of the STL, but instead of being just a dynamic array, map has some other tricks up his sleeve. To access an element or the stuff in an array we usually use brackets (I know, I know, Mrs. Rapid-Q uses parenthesis, but Mr. C++ has to do things differently) and numbers like 'MyArray[4]' gets the fifth element in the array (Mr. C++ starts counting at zero, so that is not the fourth element). Maps use the brackets to get an element of the array, however, maps don't use numbers! You are probably thinking, "Hold on here, if they don't use numbers, what *do* they use?" The answer isn't clear-cut because they will use whatever you tell them to use! If you want to use numbers, you can, or you can use anything else.

To decide how to set up our map (and what data type to use for indexing), let's consider what we need a map for. If you glance back up to our Database object diagram, you will see that the second variable we need is "A list of variables (like Message) with a SimpleDataInfo." The two keys to understanding this is 'list' and 'with.' Up above, we just needed a list of keywords, so we used a vector, which is dynamic array (one that automatically resizes itself). Here, we need a list of names for each variable and a list of SimpleDataInfo for each variable. Instead of having two vectors, the STL map allows us to say 'MyArray["Message"]' to get the SimpleDataInfo for the variable named "Message." If you are a bit confused, don't worry too much as we will be doing lots of with this concept later on.

Now back to our original piece of code. We have decided that to get an element of the map, we will use strings instead of numbers. Also, we want to fill our map with

SimpleDataInfo structures (one for each variable name). To put this into C++ code, we put '<string, SimpleDataInfo>' after the word 'map.' The first part of tells Mr. C++ that we want to index our array with the string data type (not numbers). Then after the comma, we tell him that we want to have an array of SimpleDataInfo structures. Finally, we want to name this strange array SimpleDataTypes.

What does this do for us? We have an array named SimpleDataTypes were each variable's name is linked to a structure called SimpleDataInfo that holds valuable information about the variable. Later, we will see how this works.

```
enum SimpleDataTypes{Integer, Double, String};
```

The last variable we need for our Database object is a list of all the types of simple data. For now, KoolB will just handle three types of simple data: Integers (which are positive & negative numbers that don't have any fractional value), Doubles (which are positive & negative numbers that can have fractional values), and Strings (which is just a simple array of characters). I am leaving the rest of the simple data types to you as an exercise.

Now let's briefly look at the functions:

The constructor for our Database object adds all the keywords that we have so far to the vector KeyWord. The actual concept is pretty clear, but one thing you will learn about the STL is that it likes to use lots of confusing methods.

```
Database(){
  KeyWord.push_back("DIM");
  KeyWord.push_back("AS");
  KeyWord.push_back("INTEGER");
  KeyWord.push_back("DOUBLE");
  KeyWord.push_back("STRING");
}
```

Normally, you would expect to see `KeyWord[0] = "DIM"` and so forth, but the STL makes us use a function called `push_back`. This puts the string at the very back of the array. In actuality, we don't really care where the keyword string goes, so this is fine with us. I guess it would be faster and more efficient to somehow alphabetize them, but that is just too much work. You will notice that we are not going to put all the BASIC keywords in at once; instead we will be adding to this function as the chapters progress.

As the user goes about creating data types, Koolb needs to make sure that the user names all the variables with valid names. Up above, we saw some of the things a user could do wrong, such as creating a String named 'DIM.' To prevent that, we create a function that checks to see if each data type has a valid name. IsAlreadyReserved, the function that does this, it takes one parameter: a string called Name. IsAlreadyReserved searches through both the vector KeyWord and the map SimpleDataTypes to see if Name appears in either one. If Name does not appear in either one, IsAlreadReserved returns false, indicating that the Name can safely be used. Otherwise, the function returns true to tell KoolB that the name is already in use and cannot be used again. So let's look closer at this function by examining the code:

```
bool Database::IsAlreadyReserved(string Name){
  if (find(KeyWord.begin(), KeyWord.end(), Name) != KeyWord.end()){
    return true;
  }
  if (SimpleDataTypes.find(Name) != SimpleDataTypes.end()){
    return true;
  }
  return false;
}
```

Here, we can see that isAlreadyReserved searches through KeyWords (from the beginning to the end) for Name. It also searches through SimpleDataTypes for Name. If the result of either search is not the end of the array, IsAlreadyReserved knows that Name is already taken and returns true. Else, the function returns false, which tells KoolB to go ahead and use the name.

Now that KoolB has approved the Name of the data for use, she needs to store the data type's Name and Info for reference later on. AddSimpleData does exactly this. It takes the Name of the data type and the information that KoolB has filled out in a SimpleDataInfo structure and adds it to the SimpleDataTypes map. Once it is safely stored there, KoolB can access the information later. You will also notice that by putting the Name of the data type in SimpleDataTypes, KoolB is adding the Name to the list of reserved words. This prevents the user from creating two variables with the same name. Now for the code:

```
void Database::AddSimpleData(string Name, SimpleDataInfo Info){
  SimpleDataTypes[Name] = Info;
  return;
}
```

Wow! That is a **short** function (we could use more of those, couldn't we)! All this function does is link the variables information to its name in the SimpleDataTypes array. Later, when we want to get the variable's information, we can access it through 'SimpleDataTypes[Name].' And that is precisely what the next function does.

Now that we have validated the name of the data and stored it's information away, we need to generate a name for it in assembly language. Why do we need to do this? Let's think it through. If the user created a string named Message$ or an integer named Index& or a double named PI#, what would happen if we used these names in the assembly language code? The assembler would choke! In assembly language, data types can only have names that contain letters of the alphabet, digits, and the underscore (_). Speaking of what characters can be in identifiers, I don't think we allowed $, &, or # to be in *our* identifiers. Let's correct that quickly before we proceed. First, open up the file that contains the Reading object and replace GetIdentifier with this:

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

The only difference here is that we are now allowing the charcters &, #, and $ to tag along at the end of an identifier. I know that Rapid-Q allows you to have these in the middle of an identifier (such as Na$me), but that really complicates things. Now that we allow these extra characters in our identifiers, we have to generate an assembly language name from the BASIC names. My idea is to replace the extra characters with a certain number of underscores. For example, Index& would become Index_, PI# would become PI__, and Message$ would become Message___. This wouldn't pose a problem with variables that have an underscore at the end of their name because (as I discussed earlier), underscores can only appear in the middle of a variable name (to recap, I do this to avoid the confusion of the word A_ at the end of the line. An underscore all by itself means to continue the current instruction on the next line. Would A_ mean that or is A_ an identifier all by itself? To avoid this, I just say that underscores cannot appear at the end of a variable name). This method appears to work, but feel free to replace it with your own.

This is the approach I take in the function StripJunkOff, which takes one parameter – a string called Name. It checks the last character of Name to see if it is &, #, or $. If it finds one of those symbols, it replaces it with underscores and then returns the new assembly language name for the variable. Here is the code:

```
string Database::StripJunkOff(string Name){
  if (Name[Name.length() - 1] == '&'){
    return Name.substr(0, Name.length() - 1) + "_";
  }
  if (Name[Name.length() - 1] == '#'){
    return Name.substr(0, Name.length() - 1) + "__";
  }
  if (Name[Name.length() - 1] == '$'){
    return Name.substr(0, Name.length() - 1) + "___";
  }
  return Name;
}
```

All this function does it check the last character of the string (which is the length minus one) to see if it is a &, #, or $. If it is, it replaces & with one underscore, # with two underscores, and $ with three underscores. It does this by using the substr function, which returns only part of the string; here we want the whole string except for the last character (in effect, that strips off the symbol at the end). Then we add the underscores to the stripped strings and returns our generated assembly name. If you add more data types, you will be dealing with four, five, etc underscores for each data type. If the name does not contain any of these characters (for instance 'Message'), StripJunkOff just returns Name without modifying it. KoolB can then store the assembly name in the SimpleDataInfo structure.

So now that we have generated the assembly name from a variable name, how do we get it again? We will need that assembly name later on in the program when we generate more assembly language, so how do we access it without re-generating it every time? This turns out be pretty easy. Our SimpleDataInfo structure holds two items: a string containing the name and also the type of data. Since the map SimpleDataTypes holds both the name of the variable and linked SimpleDataInfo structure, we could store the generated assembly name of the variable SimpleDataInfo name and the real variable's name in the SimpleDataType map. So to retrieve the generated assembly name from the variable's real name (or the name the user gave the variable), would look up the SimpleDataInfo linked with the real name and then return the Name item of the SimpleDataInfo structure. This is what the function Asm does. It's only parameter is the real name of the variable. It returns the generated assembly name. Let's examine the code:

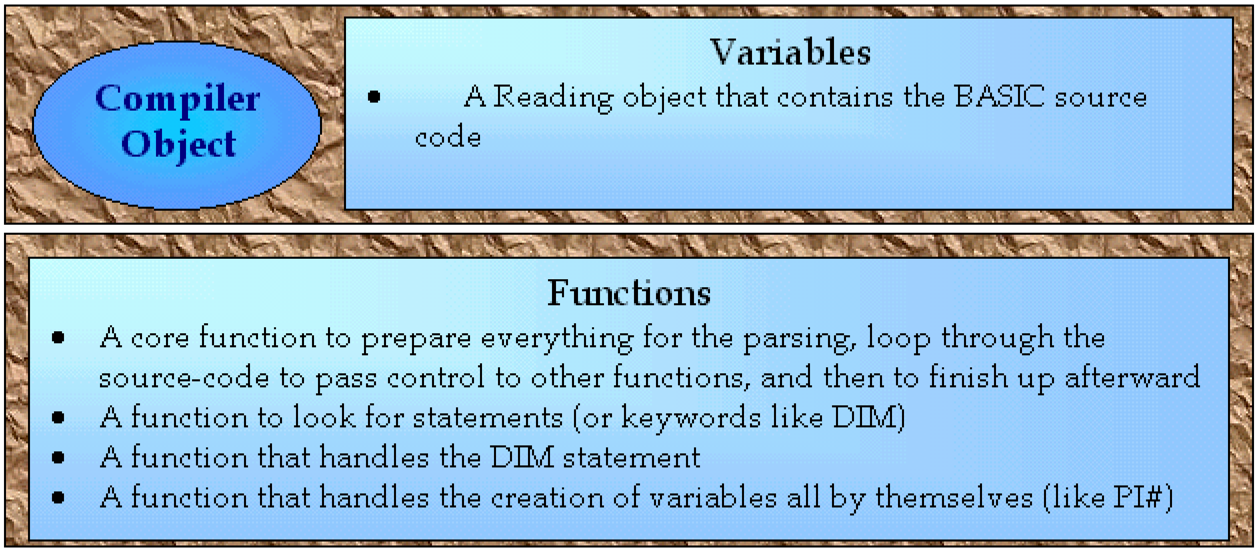
```
string Database::Asm(string Name){
  return SimpleDataTypes[Name].Name;
}
```

This is pretty simple here. We get the SimpleDataInfo structure out of the SimpleDataTypes map by using the real name as the index. We then get the generated assembly Name element out of the SimpleDataInfo structure that is linked to the real name. Finally, we return the generated assembly name.

I am sorry to say it, but we have finished the Database section of the compiler. Now we can move onto bigger and better things: the compiler part.

Compiling (Parsing) the BASIC source-code:

These next two sections are the heart of compilers the next two objects we will talk about are the translator objects. The first one needs to be created and the second one needs revising. Create another file named Compiler.h in our ever-increasing KoolB project. In it we will create a Compiler object, which will look through the BASIC source code and tells the Assembly object what assembly language to generate. This looking through the BASIC source-code is called parsing in compiler textbooks. But before we get into all the code, here is visual of the Compiler object and what it does and how it works:



Compiler Object

Variables

- A Reading object that contains the BASIC source code

Functions

- A core function to prepare everything for the parsing, loop through the source-code to pass control to other functions, and then to finish up afterward
- A function to look for statements (or keywords like DIM)
- A function that handles the DIM statement
- A function that handles the creation of variables all by themselves (like PI#)

You can see from the diagram that this object a rather strange variable: another object! The Compiler object basically will have its own Reading object to compile. It also doesn't have a lot of functions, which is nice, but wait until we have covered a couple more chapters. If functions are pounds, then this object will be gain quite a bit of weight throughout the rest of the tutorial. Until then, we have some work to do. First, let's create the structure of the Compiler object:

```
class Compiler{
 private:
  Reading Read;

 public:
  void Compile(Reading SourceFile);
    void Statement();
      void DIM();
      void NotDIM();

};
```

There isn't much to describe here as we have already covered most of this. However, you might wonder at the strange way I have organized the functions. The reason I indent the functions is so you can tell how the control flows. By the code above, you can see the top function is Compile, which passes control to Statement, which passes control to either Dim or NotDIM. We will see how this works in more detail.

First, we have the function that starts everything rolling. It takes one parameter and that is the Reading object that contains the BASIC source code to compile. Then, it enters the main loop where it loops through every line of BASIC. It analyzes the beginning of each line and decides what function to call. For now, it only calls Statement. Later, we will add more stuff for it to do. Once it has dealt with all the functions, it returns. Now for the code:

```
void Compiler::Compile(Reading SourceFile){
  Read = SourceFile;
  Read.GetNextWord();
  while (Read.WordType() != Read.None){
    switch(Read.WordType()){
      case Read.Identifier:
        Statement();
        break;
      case Read.EndOfLine:
        //Don't do anything if end of line
        break;
      default:
        cout << "Error - Invalid Command" << endl;
        break;
    }
    if (Read.WordType() != Read.EndOfLine && Read.WordType() != Read.None){
      cout << "Error - Expected End-Of-Line" << endl;
    }
    Read.GetNextWord();
  }
  return;
}
```

First, we store the Reading object we got through the parameter (SourceFile) to the Compiler object's internal Reading object Read, so now we have a carbon copy of the Reading object we have to compile. Then we use Read, the Compiler Reading object, to get the first word of its BASIC source code. While the current word is not the end of the file (Read.None), we loop through the BASIC source code. Inside the while loop, we check to see if the current word is an identifier; if it is, we call Statement to handle that line. If we just have an empty line, we don't do anything. If the type of word is something else, we give an error. After dealing with a statement or an empty line, we check to see if the next word is an end of the line or the end of the file. If it is not, we know that something is wrong, so we give an error. The reason we do this is because in BASIC, the user can only put one statement per line. The function Statement should deal with an entire statement – so if there are any words still on the line, the don't belong to the first statement; thus we give an error. If the current line is totally OK, we get the next word (which will be the first word on the next line) and start the loop all over again. When we come to the end of the file, return.

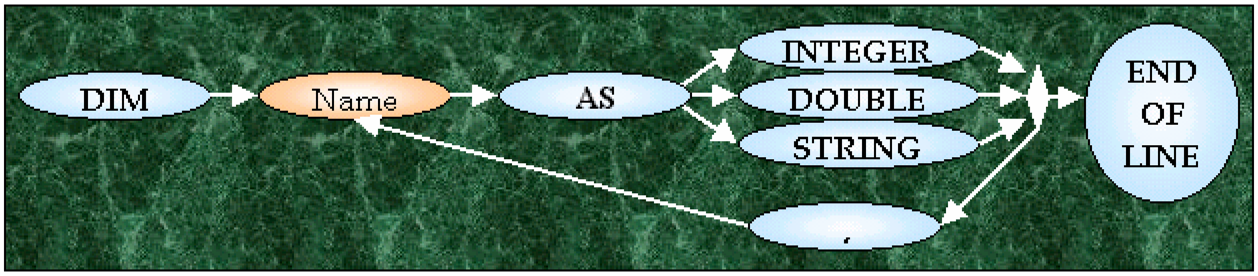
You will notice that our errors are not exactly the best. Right now, they are just placeholders for when we replace them with better error checking routines in Chapter 6. In Chapter 6, we will create an object to deal with errors in a fast and more user-friendly way.

Considering that the BASIC file actually has something in it other than empty lines, Compile will call the function Statement to deal with the lines. Statement just checks to see if the first word on the line is a keyword (like DIM). If it is, it calls the DIM function to handle the DIM statement. Otherwise, it calls NotDIM to handle the creation of an undeclared data type. Here is the code for Statement:

```
void Compiler::Statement(){
  if (Read.Word() == "DIM"){
    DIM();
    return;
  }
  NotDIM();
  return;
}
```

Not too hard, is it? If the current word is "DIM," we call the DIM function to take care the rest of the line and then return to let Compile check what to do for the next line. If the current word is not "DIM," we call NotDIM to handle the creation of an undeclared variable and then return.

Before we look at the DIM function, let's review exactly how the DIM statement works with a little diagram:



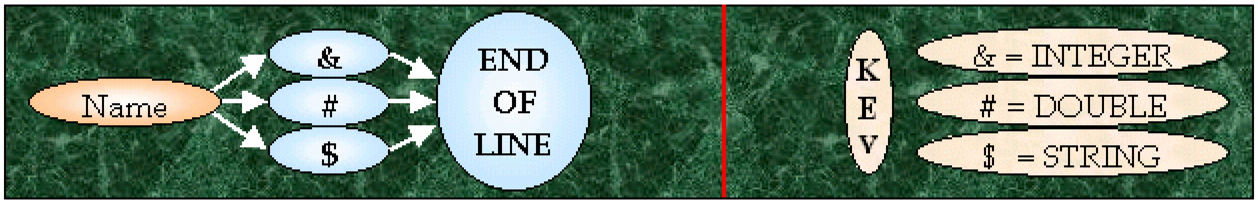
Here we can visually see how the DIM statement works. Just follow the arrows until you get to the end of the line token (or word). You will notice that after AS, we branch out to three possibilities. Only one path will be taken, so either the variable will be an Integer, Double, or String, not all three! The paths then converge right after that to a strange diamond looking thing. I chose that to symbolize two alternate paths. The first one is to end the DIM statement and have an end of the line word next. The second path is to have a comma and loop back to Name to allow the user to create multiple variables in the same instruction (or on the same line).

So how are we going to parse (or translate) this into assembly language? By using our trusty function DIM. So how is DIM going to handle this? Let's think it over first and then see. First, DIM is going to need a string called Name to keep track of the name of the variable. Since the name of the data type comes right after the "DIM" keyword, DIM should store the next word in Name. Then it needs to check to see if the name the user gave the variable is a good name, so DIM uses some of the functions we created earlier in the Database object such as IsAlreadyReserved. If the name is OK, the next word DIM should look for is "AS." Once it finds that, it knows that it is looking for one of three words that tell it what type of data the variable is – either Integer, Double, or String. Once it has the name the variable and the type of data, it can create the variable. Instead of DIM having to do it, we tell the assembly language object (Asm) to create the variable. Then it looks at the next word; is it a comma? If it is, we loops back to the beginning and gets the name of the next variable and its type of data. When DIM finally finds that the next words isn't a comma, it returns and lets Compile verify that the DIM statement is finished and the next word is indeed an end of the line token. Now let's see how Mr. C++ would say this:

```
void Compiler::DIM(){
  string Name;
  do{
    Read.GetNextWord();
    Name = Read.Word();
    if (Read.WordType() != Read.Identifier){
      if (Read.WordType() == Read.None ||
          Read.WordType() == Read.EndOfLine){
        cout << "Error - Expected Name" << endl;
      }
      cout << "Error - Invalid name" << endl;
    }
    if (Data.IsAlreadyReserved(Name)){
      cout << "Error - Already reserved" << endl;
    }
    Read.GetNextWord();
    if (Read.Word() != "AS"){
      cout << "Error - Expected 'As'" << endl;
    }
    Read.GetNextWord();
    if (Read.Word() == "INTEGER" || Read.Word() == "DOUBLE"
                             || Read.Word() == "STRING"){
       if (Read.Word() == "INTEGER"){
         Asm.CreateInteger(Name, Data.StripJunkOff(Name) + "_Integer");
       }
       if (Read.Word() == "DOUBLE"){
         Asm.CreateDouble(Name, Data.StripJunkOff(Name) + "_Double");
       }
       if (Read.Word() == "STRING"){
         Asm.CreateString(Name, Data.StripJunkOff(Name) + "_String");
       }
       Read.GetNextWord();
    }
    else{
      cout << "Error - Invalid Type" << endl;
    }
  }while(Read.Word() == ",");
  return;
}
```

Whew! Unlike some of our other functions, this one is pretty long. First, we have the string called Name. Second, we have a do loop. We have seen those before, so I won't dwell on it too long. Right inside the do loop, we get the next word and store it in Name. First we check to make sure that the user actually put a name, and then we verify that it is an identifier (what if the user accidentally used a string for the variable's name!); finally we check to make sure that it is a valid name. I know you are just shaking your head at my terrible error message, but remember that we will be fixing that next chapter, so hang in there. After we have made sure that we have a good, solid name for our variable, we make sure that the next word is "AS." Then we get the next word and verify that it is a valid type of data (for now, that means it has to be "INTEGER", "DOUBLE", or "STRING"). If it isn't, we give an error message. If it is, we then do another check for each one. You will notice that we basically do the same thing for each data type. We tell Asm (our assembly language object) to generate an integer, double, or string depending on what the user has typed. The two parameters we pass to help Asm create our variable are the variable's real name (Name) and then it's assembly name. How do we get the assembly name? Well, first we strip off the symbols at the end (if there are any) and then put the type of data at the end of the name. For instance, if the user was creating an integer named Index&, the generated assembly name would be "Index__Integer" (one underscore for the '&' and one when we append the "_Integer" to the end of the name). After telling Asm to create the variable, we get the next word and come to the end of the do while loop. If that word that we just got is a comma, we loop back to the beginning. Otherwise we return from the function back to Statement, which in turn, returns back to Compile. Compile will then check to see if the word that we had gotten (the one that wasn't a comma) is the end of the line or the end of the file. If it isn't, it gives an error that extra words are at the end of the line; if it is, it checks the next line to see what to do.

That covers what happens if the line is a DIM statement, but what if the line isn't a DIM statement? Since the only statements KoolB can handle right now are declared data types (with the DIM statement) and undeclared data types, we will assume that if the line isn't a DIM statement, it must be an undeclared variable. Before we examine the structure of the NotDIM function, let's see how the user would create undeclared variables:



This diagram of is much more simple than the DIM statement, but it is a bit more cryptic. Basically, any unknown identifier by itself with &, #, or $ after it will be created (& as an integer, # as a double, and $ as a string). However, there is one crucial difference. When you declare variables with the DIM statement, you cannot declare two variables with the same name; here it is different. We don't know if the user is creating this variable or the user is using the variable. For instance, take the expression "Var# = 3.14 + 5 ^ 3.14". Is Var# being created or used? The answer could be either one…or both! So basically, NotDIM needs to check to see if the variable Var# has been created. If it hasn't been, NotDIM will create it. If Var# has already been created, NotDIM knows it its job is done and therefore returns. Without further ado, let's see how NotDIM works by looking at the code:

```
void Compiler::NotDIM(){
  string Name = Read.Word();
  char LastChar;
  if (Read.WordType() != Read.Identifier){
    cout << "Error - Invalid name" << endl;
  }
  if (Data.IsAlreadyReserved(Name)){
    return;
  }
  LastChar = Name[Name.length() - 1];
  if (LastChar == '&' || LastChar == '#' || LastChar == '$'){
    if (LastChar == '&'){
      Asm.CreateInteger(Name, Data.StripJunkOff(Name) + "_Integer");
    }
    if (LastChar == '#'){
      Asm.CreateDouble(Name, Data.StripJunkOff(Name) + "_Double");
    }
    if (LastChar == '$'){
      Asm.CreateString(Name, Data.StripJunkOff(Name) + "_String");
    }
  }
  else{
    cout << "Error - Invalid Type" << endl;
  }
  Read.GetNextWord();
  return;
}
```

For parsing such a seemingly small amount code (such as a single line that consists of a single word like “Message$”), NotDIM sure is rather large. However, I am sure you can see some similarities to NotDIM’s cousin, DIM. We have the same old string to hold the variable’s name and the same old routines to check to see if the name is a valid name. However, we don’t have a loop because you can only create multiple variables on a line with the DIM statement. In addition, we have a strange char named LastChar. The data type char holds one character, so it is ideally supported for what we plan to do with it. You also might notice that instead of printing out an error if the variable’s name is already reserved, we just return. That follows from what we said that sometimes we would only want to use a variable that has already been created. The next part gets the last character of the name and stores it in LastChar. We then compare LastChar to the valid symbols that it can be: ‘&’, ‘#’, or ‘$.’ If it isn’t one of those symbols, we give an error. Otherwise, we then examine FirstChar again and see which type of data the variable is. Then we tell Asm to create either an integer, double, or string. Then we get the next word and return.

Understanding Grandfather Asm:

Grandfather Asm is a strange programming language and I think it is wise to review some core foundational concepts before we jump into programming with it. Last chapter we sort of got used to how to use NASM and learned some of the assembly language concepts, but we didn’t touch on the core concepts. That is what this section aims to do.

Now I know many of you are thinking that assembly language is too hard and even more cryptic than C++ (as if that was possible!) and are generally balking at the idea of trying to learn it. "Can't we just copy & paste the assembly code?" I hear from the back of the room. Well, no, that defeats the purpose. Our goal is to create a compiler and to do so, we need to learn assembly language. It is hard, but if you have the right resources, it can be simplified greatly. There are also a lot of advanced concepts in assembly language that we won't be using, just as we don't use polymorphism in C++. Learning assembly language will also give you a firm idea of what happens behind the scenes in the computer when the user does this or that. So bear with me and let's see if we can make this at least somewhat amusing.

Since we *are* dealing with data in this chapter, let's see how you create data in assembly language. We covered this last chapter, but not in to much depth, so let's look closer. First, Grandfather Asm has a particular style for creating data types; just as Mrs. Rapid-Q does. Where as Mrs. Rapid-Q would do this:

```
Dim Message As String
Dim PI As Double
Dim NumberOfItems As Integer
Dim Int as Integer
```

Grandfather Asm would do this:

```
Message db "Hello",0
PI dq 0.0
NumberOfItems dd 0
Int dd 0
```

As you can see here, the over all structure is this:

```
<VariableName> <DataType> <Data>
```

KoolB will make the VariableName for us, so we don't have to worry about that. DataType can be one of the following:

db	- Holds 1 byte	- Creates a Byte
dw	- Holds 2 bytes	- Creates a Word
dd	- Holds 4 bytes	- Creates a doubleword or dword (Integer or Long)
dq	- Holds 8 bytes	- Creates a Quadword or qword for short (Double)

Finally <Data> is where we assign something to the data. Usually, we just put zero here to make sure that it is empty. The trick is to think of these variables as containers that hold something. You first tell Grandfather Asm what you want to call the container, how big you want the container to be, and then what you want to store in the container.

Once we have the data declared, we can do stuff with it. Usually, we move stuff around in memory and it finally winds up in a variable. For instance, say we wanted to move the number 1 into NumberOfItems. How would we do this? First we would use the assembly instruction MOV like so:

```
MOV dword[NumberOfItems],1
```

We start off by telling Grandfather Asm that we want to move something. Then we tell him the size of the destination (sometimes we will only want to fill part of the variable). When we want to put something into a container or take something out of a container, we have to put square brackets around it like [NumberOfItems]. Imagine these brackets as pinchers that hold the container while we put something in or take something out. Other times, we might want to use the container directly, like putting the lid on and shaking up the contents (just kidding!). Readers familiar with C++ might notice that the containers are really just pointers; however, I think the container analogy is easier to understand than addresses and pointers. So now that we have told Grandfather Asm what the destination and all about it, we now need to tell him what the source is. The source is the '1.'

To separate the destination from the source, we put a comma in-between. You will notice that a lot of instructions in assembly language follow this format with one instruction and then two operands. Now let's pretend that we want to copy the contents of NumberOfItems to our other integer named Int. How would we do that? Well, that is tricky because Grandfather Asm doesn't allow us to transfer contents directly between two containers. To get around this we will first store the contents of NumberOfItems in a temporary storage bin, and then transfer it safely inside Int. But were will we find such a temporary storage bin? Luckily, the makers of our CPU have thought of that and have provided some nice storage bins called registers. These registers are state of the art storage bins colored dark deep green with leather interiors and special loading and unloading equipment to boot! OK, I guess I better stop fantasizing. In reality, these registers are just grungy old storage bins and the worst thing is you never have enough of them because some other person is using them. But we use them any way. There are several general use registers called eax, ebx, ecx, edx, and edi. These are all double word registers (holding 4 bytes). So let's see how we would do this:

```
MOV eax,dword[NumberOfItems]
MOV dword[Int],eax
```

Like last time, we tell Grandfather Asm we want to MOV something. However, you will notice that we don't need to tell Grandfather Asm that our register eax is a dword or put the brackets around it. Why not? Because Grandfather Asm knows all the registers are just storage bins. Since we cannot use these storage bins for anything else except storing things in, he already knows what to do. With normal containers, we might want to do something with the container (like pass it off to a function or put it in the waste basket); with registers, we cannot do this. So what do we put into our register eax? We put 4 bytes (dword) of the contents (notice the [and]) of the container NumberOfItems into eax. The next statement, we move the stuff in the storage bin eax into the 4 bytes (dword) of the container (notice the [and]) named Int.

That is how you use integers with Grandfather Asm, but how do you use doubles and strings? Doubles, containers that hold 8 bytes, are a bit more complex so we will leave them for another chapter, especially since we won't be using them this chapter. Strings, on the other hand, are more complex than integers, but not as complex as doubles. Strings, as you will recall from previous chapters, are just an array of bytes ending with a byte with the value of zero. For those visual learners out there, here is a diagram that shows you the structure of the string:

I		a	m		a		s	r	i	n	g	!		O	,		y	e	s		I		a	m	\0

Here each little compartment is 1 byte and each byte holds one character. You know when the string ends because of the null-terminator at the end (a null terminator is \0 or a byte containing 0). There are two ways to create strings using Grandfather Asm. The first is pretty straightforward:

```
Message db “I am a string! O, yes I am”,0
```

This creates a fixed string, or one that cannot be resized. This is by far the easiest way and is great when you don't need to change the string during the program. Error messages would fit this category, they just sit there quietly until you need them. However, when you need to change or resize the string, you have to resort to dynamic allocation.

Dynamic allocation is telling your OS: "Hey there, I don't have enough storage space, could you lend me some from the RAM? Thanks." The OS then returns a chunk of memory for you to use. Then you can use this storage space to put stuff in, such as an array of characters, or a string. When you need to change the string, you can ask the computer to exchange your old piece of storage with a new piece containing the exact amount you need. When you are through using the piece of storage, you can hand it back to the computer to let the computer give it out to some other needy program. So how do we do this? Well, it depends on your operating system for one, but the process is not too difficult. We will look at how to do this more closely when we get to the part where we have to create strings.

There are two last things we have to know and that is calling functions and jumping around. We have already covered calling functions in the last chapter, but let's review. To call a Windows function, we use this format:

stdcall <FunctionName>,<Parameter1>,<Parameter2>,…

Now if we want to call a Linux function, we use this format:

```
ccall <FunctionName>,<Parameter1>,<Parameter2>,…
```

Pretty simple, huh? Now let's talk about jumping around; jumping is just the word we use for going from one spot in the program to another spot far away (jumping there). Why would we need to jump at all? For many reasons: perhaps something went wrong and we need to shut down the program and then exit, perhaps we have some code that needs to be run sometimes but not other times, or perhaps we just need to go somewhere else in the program. Jumping someplace is nearly exactly the same using GOTO in BASIC. First we need a spot to jump to (we use a label for this). So for simple jumps we can do this:

```
JMP ToLabel
;code in-between
ToLabel:
```

As you can see, we just tell Grandfather Asm to JMP to whatever label we want to go to. To make a label, we just put an unused name on a line by itself with a semi-colon after it. That is simple enough; here comes the tricky part: what is we want to jump somewhere on a condition (like if NumberOfItems equals 0, jump to Error)? How would we do that? That is where things start getting weird. First, we have to compare two values. For instance, we compare NumberOfItems and zero like this:

```
CMP dword[NumberOfItems],0
```

That sets a bunch of flags in the computer. Then we use one of these weird-looking jump instructions (in this example, dword[NumberOfItems] is the right side and 0 is the left side):

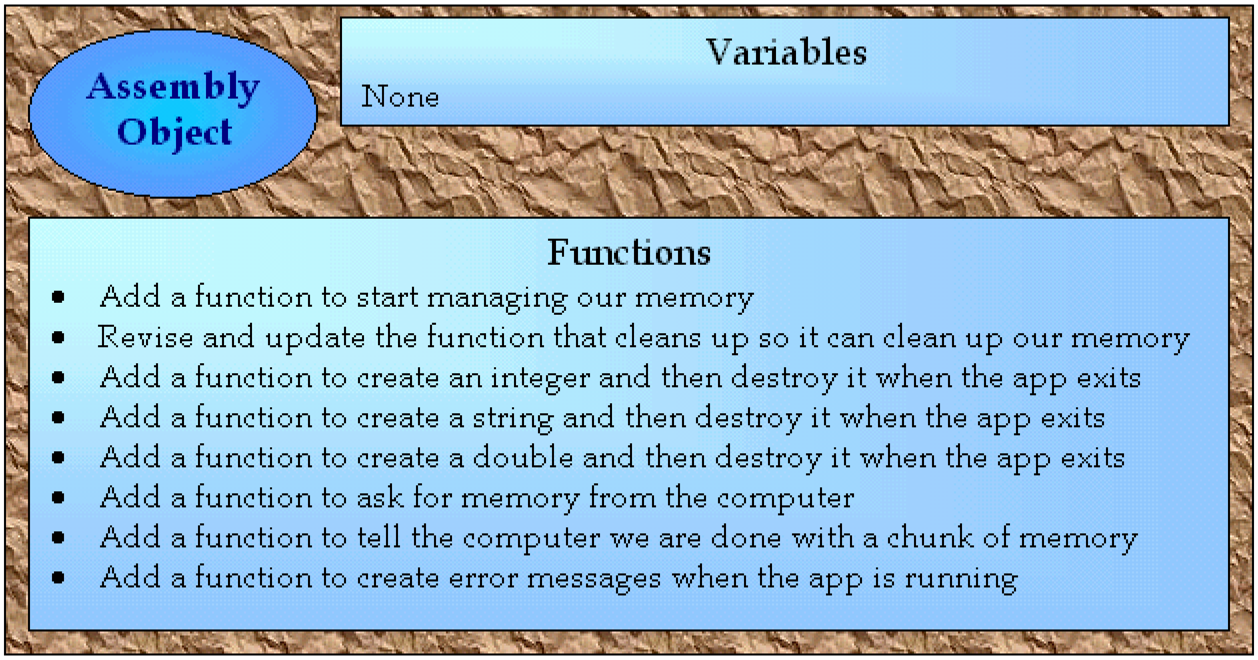
jl Label	Jump to Label if the right side is less than the left side - NumberOfItems <0
Jle Label	Jump to Label if the right side is less than or equal to the left side – NumberOfItems <= 0
je Label	Jump to Label if the right side is equal to the left side – NumberOfItems = 0
jne Label	Jump to Label if the right side is not equal to the left side – NumberOfItems <> 0 (or NumberOfItems != 0)
jge Label	Jump to Label if the right side is greater or equal to the left side – NumberOfItems >= 0
jg Label	Jump to Label if the right side is greater than the left side – NumberOfItems > 0

Here you just CMP two values and then use one of the conditional jumps you see above. If the condition is met, you jump to Label, otherwise you continue running on the next line.

That should get you started and somewhat familiar with Grandfather Asm. Throughout the rest of the chapter, we will be adding bits and pieces of information to your knowledge base of assembly language, so watch out!

Adding to the Assembly Object:

Although we have already created the Assembly object, we need to revisit it some of the functions to add more features. We will also need to add functions do create integers, doubles, and strings. We will also create some functions to manage the memory we need. Here is a diagram that includes all the additions we will be making to the Assembly Object this chapter:



Assembly Object

Variables

None

Functions

- Add a function to start managing our memory
- Revise and update the function that cleans up so it can clean up our memory
- Add a function to create an integer and then destroy it when the app exits
- Add a function to create a string and then destroy it when the app exits
- Add a function to create a double and then destroy it when the app exits
- Add a function to ask for memory from the computer
- Add a function to tell the computer we are done with a chunk of memory
- Add a function to create error messages when the app is running

That seems like a lot of work, and it is even harder because we have to generate assembly language for both Windows & Linux, making these functions even bigger. Let's get the biggest one out of the way first, shall we?

First, before we start requesting memory (allocating memory) from the OS, we need to create a private memory heap for our program. Why do we need a private pool of memory? Why not just use the system memory? Well, if we have our own heap we basically let Windows do all the hard work for us: managing the memory. All we have to do is tell Windows how much memory we want and then Windows will go ahead and get the memory for us. If our little private heap isn't big enough, Windows will automatically expand it for us. Pretty nifty, if you ask me (OK you didn't, but you got my opinion anyhow). However, consider this: The user has an old Windows 95 machine that has 16 MB of RAM; the user has 6 Internet Explorer windows, 2 Microsoft Works documents open, and an anti-virus, a download manager, and ICQ in the background. The system has about 2% resources left and is operating mainly on the system swap file (that is where the memory goes when the computer runs out of RAM). Then the user decides to run a KoolB app and when the KoolB app runs, it asks for memory. Windows says "Sorry old boy, but we are out of memory right now – try again later." Now what happens? The KoolB app would just crash! That is rather unfriendly behavior, so we need add the ability to give an error message telling the user that not enough memory exists to run the program and then gracefully exit.

Before we look at the code, we need to deal with the issue of Linux. If you will notice, I said earlier that our programs would create a private heap that Windows would manage. Where does that leave Linux, though? Without a private heap! We will have to ask Linux directly for system memory. It might be a bit slower, but it should work. In fact, you should be able to see direct similarities between the Windows & Linux versions (except for the private heap bit). Well, now that we have some background information, let's take a peep at this large function:

```
void Assembly::InitMemory(){
  Write.Line(Write.ToData, "NoMemory db \"Not enough memory!"
             " Please free up some memory and re-run this app. "
             " Thanks!\",0");
  Write.Line(Write.ToData, "Error db \"Error!\",0");
  PostError("Error_NoMemory", "Error", "NoMemory");
  if (OS == Windows){
    string LabelGotMemory = GetLabel();
    AddLibrary("HeapCreate");
    Write.Line(Write.ToData,     "HandleToHeap dd 0");
    Write.Line(Write.ToFireUp,   "stdcall HeapCreate,0,16384,0");
    Write.Line(Write.ToFireUp,   "MOV dword[HandleToHeap],EAX");
    Write.Line(Write.ToFireUp,   "CMP dword[HandleToHeap],0");
    Write.Line(Write.ToFireUp,   "JNE " + LabelGotMemory);
    Write.Line(Write.ToFireUp,   "JMP Error_NoMemory");
    PostLabel(Write.ToFireUp,    LabelGotMemory);
  }
  if (OS == Linux){
    return;
  }
  return;
}
```

That is a large function, but we can break it down. First we create two strings that we will use incase we cannot get any memory. Then we create an error by posting it to the assembly language (we will see exactly how that works later). First we pass it the name of the label it should create so we can jump to it later if we run out of memory, then we pass it container that holds the error we want to be displayed, and finally, we pass it the container that holds the error message. If we are compiling for Windows, you will notice that we do quite a bit to get the a private heap for our memory. First we get a temporary label named LabelGotMemory. Then we add HeapCreate to our list of Windows API list.

Now I am going to provide a running commentary on the assembly language by repeating the assembly language and putting a comment explaining it underneath:

```
    Write.Line(Write.ToData,     "HandleToHeap dd 0");
```

Creates a container to store the heap in. Think of the handle as the part of the container that you grab and hold onto. Whenever we need more memory, we can grab the handle to the container and tell Windows, "Put some more memory in here. Thanks!"

```
    Write.Line(Write.ToFireUp,   "stdcall HeapCreate,0,16384,0");
```

Call the Windows API function HeapCreate. The first parameter is any extra options we want. In this case, that is none, so we put zero. The second parameter is the size of the heap to begin with; I picked the arbitrary number 16384 bytes (16KB) for starters. You can modify this number if you want. The last parameter is largest possible size the heap can expand to as we need more memory. If we pass zero, that means Windows can expand it forever (or until the balloon pops – in English that means until Windows chokes).

```
    Write.Line(Write.ToFireUp,   "MOV dword[HandleToHeap],EAX");
```

Whenever we call a Windows API function, the result is always left in a certain storage bin: the register EAX. After telling Windows to go create us a heap of memory, we want to know the result. The result should be that Windows hands us a container with the memory heap in it. Our response is to 'handle with care' the container and store it in HandleToHeap. Can we store a container in a container? Sure why not? That is what a grocery cart is, isn't it? You usually don't load up your grocery cart with ice cream or frozen peas do you? No of course not, you put a box of frozen peas or a carton of ice cream in your grocery cart. Same thing here, we put the container that holds the memory in another container named HandleToHeap.

```
    Write.Line(Write.ToFireUp,   "CMP dword[HandleToHeap],0");
```

Now comes the tricky part. We need to make sure that Windows actually gave us the memory. How do we do that? Well, we just tried to put the container that holds the memory in HandleToHeap. If we succeeded and our HandleToHeap is not empty (meaning the container that holds the memory heap is in it), we have the memory. But if HandleToHeap is empty (the container that holds the memory heap isn't in it), we don't have any memory. So we compare the contents of the container HandleToHeap with zero (zero meaning empty).

```
    Write.Line(Write.ToFireUp,   "JNE " + LabelGotMemory);
```

We just compared HandleToHeap to 0 to see if Windows created a memory heap for us, so now we need to look at the results. If the two items (the contents of HandleToHeap and 0) are unequal, we jump to LabelGotMemory because that means HandleToHeap is not empty. JNE means "Jump if Not Equal."

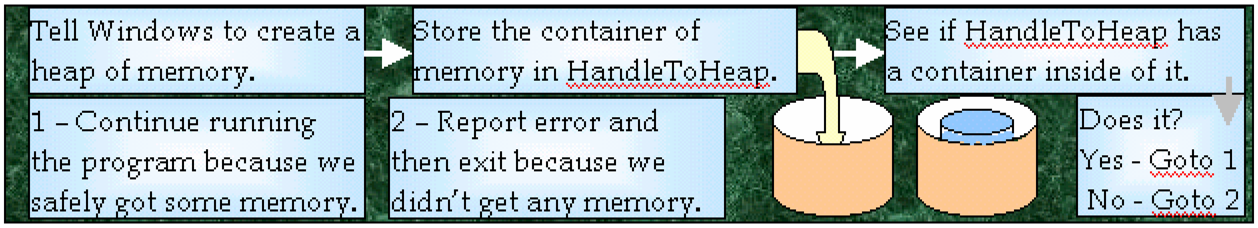
```
    Write.Line(Write.ToFireUp,   "JMP Error_NoMemory");
```

However, if HandleToHeap is empty (the contents of HandleToHeap equals zero), then we know that Windows didn't give us any memory. So what do we do? We jump (or GOTO) to Error_NoMemory (that is the error we posted earlier in the function). That will give the error to the user and then exit the program.

```
    PostLabel(Write.ToFireUp,    LabelGotMemory);
```

The last thing we do is to post LabelGotMemory. This is immediately after where we jump to the Error_NoMemory, so if HandleToHeap contains valid memory heap, we basically end up jumping over the line that gives us the error. The end result is this: if we have memory, we continue with the program; if we didn't get any memory, we report the error and then exit. This is a really good concept that we will be using again and again throughout the up-coming chapters.

For those graphic learners out there, let me see if I can draw the sequence of events that happened.


Tell Windows to create a heap of memory.
Store the container of memory in HandleToHeap.
See if HandleToHeap has a container inside of it.
Does it?
Yes - Goto 1
No - Goto 2
1 – Continue running the program because we safely got some memory.
2 – Report error and then exit because we didn't get any memory.

That about sums this up. You will also notice that for the Linux version, we don't have to initialize the memory by creating a heap, so we just return.

The counterpart function to this function is FinishUp, which we created last chapter, but needs to be modified slightly. FinishUp takes care of any last minute things right before the program exits. The two modifications we need to make are to give Windows back the container of memory so that we don't waste it when our program exits. The second thing we need to do is modify how we return our exit code. The exit code is the status of how our program exited. If our program exits normally, we return with an exit status of 0. If our program encounters an error, we return with an exit status of 1. To keep things easy, we will create an integer to keep the exit status in so we can modify it throughout the program. At the beginning, we will assign it zero. If an error occurs, we will change it to 1. At the end of the program we will just pass it to ExitProcess or exit (depending on whether we are compiling for Windows or Linux). So here is the code:

```
void Assembly::FinishUp(){
  Write.Line(Write.ToData, "ExitStatus dd 0");
  if (OS == Windows){
    AddLibrary("ExitProcess");
    AddLibrary("HeapDestroy");
    Write.Line(Write.ToFinishUp, "stdcall HeapDestroy,dword[HandleToHeap]");
    Write.Line(Write.ToFinishUp, "stdcall ExitProcess,dword[ExitStatus]");
  }
  if (OS == Linux){
    AddLibrary("exit");
    Write.Line(Write.ToFinishUp, "ccall exit,dword[ExitStatus]");
  }
}
```

FinishUp is bit shorter than the last function, which is good. First, we create a container to hold an integer size value for the exit code or status of our program. We initialize it to zero. Then if the OS is Windows we import the API functions ExitProcess and HeapDestroy. Then we pass the contents of HandleToHeap, or the container that holds the memory heap, to HeapDestroy. Basically this is throwing the container of memory heap back at Windows and saying: "Here, take your lousy pot of memory; we don't need it anymore." Then we terminate the program by calling ExitProcess with the contents of ExitStatus. If our OS is Linux, we import the standard C library function exit and call it with the contents of our container ExitStatus.

So how would we handle errors and changing the ExitStatus to 1? That is exactly what our next function, PostError, deals with. It takes three parameters:

1.	Label	A string containing the name of the place where we jump to in order to report the error
2.	Error	A string containing a brief description of the error (like "Error!" or "Error – Number 439")
3.	Message	A string containing a more in-depth explanation of the problem.

Basically, what PostError does is this: post the Label so that we can easily jump to the error if need be and then set the ExitStatus to 1. Then we somehow report the error through various means depending on which OS we are compiling for. Finally, we jump to Exit so that any clean up that needs to be done can be done. Let's examine the code:

```
void Assembly::PostError(string Label, string Error, string Message){
  PostLabel(Write.ToFunction,  Label);
  Write.Line(Write.ToFunction, "MOV dword[ExitStatus],1");
  if (OS == Windows){
    AddLibrary("MessageBoxA");
    Write.Line(Write.ToFunction, "stdcall MessageBoxA,0," + Message +
               "," + Error + ",0");
  }
  if (OS == Linux){
    AddLibrary("puts");
    Write.Line(Write.ToFunction, "ccall puts," + Error);
    Write.Line(Write.ToFunction, "ccall puts," + Message);
  }
  Write.Line(Write.ToFunction, "JMP Exit");
  return;
}
```

First we post the Label to the ToFunction section of assembly language file. The ToFunction section? We never created that before! And you are totally right, we didn't. But I decided to add it so we could put standalone functions there. These error reports are close to being functions (in assembly language, functions are just labels that you detour to for a while and then continue where you left off), so we might as well put them there. I won't bore you by showing you how I created the ToFunction section since we created a whole bunch last chapter; if you are really interested, go look at the Writing object. After posting the label, we store the number 1 in the container ExitStatus. Then, if the OS is Windows, we use a standard MessageBox function to display the error report. We pass zero to tell Windows that we don't need window owner; then we pass MessageBox the title and then the message part. Finally the last parameter is the icon we want to use; for an error message, I think the red X would be appropriate. To do that, we pass the number 16. If, on the other hand, KoolB is compiling for Linux, she uses the standard C library puts. Puts just prints an error out on the line of the console. We first print out the Error and then the Message. You might be wondering why we don't have to use that strange bracket mumbo-jumbo. The reason is that these functions actually want to have the container, not the contents of the container. Weird world, huh? Finally after reporting the error, we jump to exit to clean up.

That gets most of the really tedious stuff out of the way. Now we can get down to the heart of this chapter: creating data! The three data types we need to generate assembly language for are Integer, Double, and String. Let's start at the beginning with Integer.

CreateInteger takes two parameters: Name (the name the user gave the Integer) and AsmName (the name we use in assembly language). CreateInteger also has two jobs: to create the integer in assembly language and store the information about the integer in the Database object. To see how CreateInteger does this, let's look at the code:

```
void Assembly::CreateInteger(string Name, string AsmName){
  SimpleDataInfo Info;
  Info.AsmName = AsmName;
  Info.Type    = Data.Integer;
  Data.AddSimpleData(Name, Info);
  Write.Line(Write.ToData, AsmName + " dd 0");
  return;
}
```

First we create a SimpleDataInfo structure named Info. Then we fill Info.AsmName with the assembly language name and Info.Type with Data.Integer. Next, we store both Name and it's Info in Data by calling AddSimpleData. Finally, we create an integer by with the name of AsmName in the data section of the assembly language file.

CreateDouble is nearly the same:

```
void Assembly::CreateDouble(string Name, string AsmName){
  SimpleDataInfo Info;
  Info.AsmName = AsmName;
  Info.Type    = Data.Double;
  Data.AddSimpleData(Name, Info);
  Write.Line(Write.ToData, AsmName + " dq 0.0");
  return;
}
```

The only difference is that we store Data.Double in Info.Type; we also create a quadword (dq) instead of a doubleword (dd). We also assign the double zero by using decimals.

CreateString is slightly different because we don't want to create a fixed string. Since the user will usually want to change the string, we will have to dynamically allocate the memory. So after we fill in the information and add it to the database, we will create a double container – one container to hold the container that holds the memory where the string will reside; it seems rather redundant, but you will see that it will come in useful sometimes. Once we have the first container, we pass Windows (or Linux) the container and say, "We want some memory, put it in this container." So the OS obliges and ladles up some memory in a Styrofoam cup and puts it in the container. The last thing CreateString has to do is give the memory back to the OS. Now how does it do that? Do we have to call CreateString at the end of the program? By no means. What we can do is to write to the FinishUp section of the assembly language, since that is the section that will run when the program is about to end. That takes care of the creation and clean-up of the string, so let's look at the code:

```
void Assembly::CreateString(string Name, string AsmName){
  SimpleDataInfo Info;
  Info.AsmName = AsmName;
  Info.Type    = Data.String;
  Data.AddSimpleData(Name, Info);
  Write.Line(Write.ToData,     AsmName + " dd 0");
  AllocMemory(Write.ToFireUp,  "1");
  Write.Line(Write.ToFireUp,   "MOV dword[" + AsmName + "],EAX");
  Write.Line(Write.ToFireUp,   "MOV byte[EAX],0");
  FreeMemory(Write.ToFinishUp, "dword[" + AsmName + "]");
  return;
}
```

First, we fill out the SimpleDataInfo structure and store it away. Then we create a 4 byte container (most containers are 4 bytes). Then we call AllocMemory to get some memory. First we pass where we want to write the allocation routines. Second, we pass a string containing how many bytes of memory to get. Windows or Linux will go off and get a container of memory; then it will leave that container in the storage bin register eax. So we copy the container that holds the memory into our string (AsmName) container. Since the pot that holds the memory is still in the register eax, we move zero into the first byte of the memory (Each byte contains one character like 'H' or '|' or '1'). This is called null-terminating the string. Null-terminating puts a zero right after the end of the string to tell anybody else where it ends. Since we are creating an empty string, we put the ending zero in the first byte of the string. Then we call FreeMemory to give the chunk of memory back to the OS. Basically, we are telling Windows to empty the container inside the container AsmName to free up the memory for other programs.

The only thing we have left to examine is the memory management functions AllocMemory and FreeMemory. AllocMemory first asks the OS for some memory. It takes two parameters: Section (the section of the assembly language file to put the code in) and Size (a string containing the number of bytes of memory to request). Once the OS returns the container with the memory in it, we make sure that there is memory in it (just like we did earlier). If the OS failed to allocate the memory, we will jump to the Error_NoMemory label we created earlier. So let's glance at the code:

```
void Assembly::AllocMemory(int Section, string Size){
  string MemoryOKLabel = GetLabel();
  if (OS == Windows){
    AddLibrary("HeapAlloc");
    Write.Line(Section, "stdcall HeapAlloc,dword[HandleToHeap],8," + Size);
  }
  if (OS == Linux){
    AddLibrary("malloc");
    Write.Line(Section, "ccall malloc," + Size);
  }
  Write.Line(Section, "CMP EAX,0");
  Write.Line(Section, "JNE " + MemoryOKLabel);
  Write.Line(Section, "JMP Error_NoMemory");
  PostLabel(Section,  MemoryOKLabel);
  return;
}
```

First we create our temporary MemoryOKLabel to allow us to skip over our jump to the error label. Then if the OS is Windows, we call HeapAlloc. The first parameter we pass is the container in HandleToHeap. The second parameter we pass is 8, which tells Windows to automatically fill the memory with zeros. Finally, we pass the size to the function to tell Windows how many bytes of memory to get. If the OS is Linux, on the other hand, we call malloc to get some system memory. The only parameter we pass to it is the number of bytes of memory we need. After we have the told the OS to get the memory, we test the return value to see if the OS was able to get some memory. We first compare EAX to 0 to see if the OS returned a container with the memory in it. If EAX is not equal to 0 (meaning EAX holds a container of memory), we jump to the label MemoryOKLabel and over the jump to the error label. If EAX is equal to zero, the jump fails and we continue to the next line, where we jump directly to the error label we created in the first function we examined. This will report the error to the user and then exit the program. Finally, we post the MemoryOKLabel so that the program has somewhere to jump to continue the rest of the program.

The last function is the function to give the OS back the memory we asked for earlier. It takes two parameters: Section (where in the file to put the assembly language) and Name, a string containing what container of memory to return to the OS.

```
void Assembly::FreeMemory(int Section, string Name){
  if (OS == Windows){
    AddLibrary("HeapFree");
    Write.Line(Section, "stdcall HeapFree,dword[HandleToHeap],0," + Name);
  }
  if (OS == Linux){
    AddLibrary("free");
    Write.Line(Section, "ccall free," + Name);
  }
}
```

If the OS is Windows, we call HeapFree with the container that holds the heap. The first parameter is the container that holds the heap (that container is inside HandleToHeap). Then we pass zero because we don't want to add any extra options. Finally, we pass the container that contains the memory. HeapFree will empty the memory inside the container back into the system memory and then destroy the container. If the OS is Linux, we do almost the same thing, except we call free instead of HeapFree and we only have to pass it the container that holds the memory.

Getting It to Work:

We have completed almost everything that we set out to, except to get it all to work. So move over to the main C++ file for KoolB and add these include files right below the rest:

```
#include <vector>
#include <map>
#include <algorithm>
```

The first two includes allow you to use those super arrays: vectors and maps. The third include gives us the ability to use algorithms on these arrays like find. Once we have that, we need to add our Database object to our program. To do so, add these lines after the #include after the Writing object:

```
#include "Database.h"
  Database Data;
```

This creates a Database object named Data for our program so that the Compiler object and the Assembly object can store stuff to recall at a later date. Finally, we add the following lines to the Compile function just after we examine the command line:

```
  Read.OpenBook(FileName);
  Asm.BuildSkeleton();
  Asm.InitMemory();
  CompileIt.Compile(Read);
  Asm.FinishUp();
  Write.BuildApp(FileName);
```

What we have here is basically the same old routine: open the BASIC source code, tell the Assembly object to build a skeleton for the app and intialize the memory. Then pass the Compiler object the Reading object so it can generate the right assembly language from the BASIC source code. Then we tell the Assembly object to clean up and generate the assembly language for the program to exit. Then we finally tell the Writing object to build the program by calling the assembler and the linker.

Testing It Out:

To test out the hard work that we accomplished in this chapter, compile KoolB and create a file to compile. Name the file Test.bas or something similar and then copy and paste the sample app I gave you earlier in the program. For your connivance, here it is again:

```
Index&          'A "&" after a variable name means to create an integer
Message$        'A "&" after a variable name means to create a string
PI#             'A "&" after a variable name means to create a double

DIM P AS String
DIM Rate204 as Double
DIM RateSymbol as Integer
DIM Rate188 as DOUBLE
dim x& as integer,dd& as integer
```

You can add some more BASIC code to it if you want (but remember that KoolB only supports the creation of data types). Now grab a console. Change the directory to C:\Compiler\KoolB and type in "KoolB Test.bas." KoolB should go away and "think" for a little bit as it compiles the program. It should come back with a successful report, indicating that it created a working Windows program: Test.bas.exe. To test it out, type in Test.bas.exe and press enter (alternatively, you can go to the KoolB directory and double-clik on the program Test.bas.exe). Either way, nothing spectacular should happen. The program should just start up, create some data, and then exit.

Now I know that this isn't very exciting, but sometimes we have to go through the boring basics before we can get to the exciting stuff. Near the end of this tutorial, we will hopefully have time to do some of the exciting stuff like creating the PRINT statement or allowing for CGI programs. Anyhow, what good would a program be without any data? Not much!

Before we continue with the next chapter, let's look at something for a minute. Go back and make some deliberate errors in Test.bas and then try to compile it. Whoa! You get some really scary error messages, don't you? Not only that, but you might get a whole slew of them. KoolB doesn't generate very user-friendly error messages, and this is something that we will have to work on.

Exercises:

I have hinted throughout this chapter that you will be creating the rest of the data types, and here is your chance to do so. Try adding the following data types:

```
Type       ID  Size  Range
--------- ---- ---- -------------
Byte        ?   1   0..255
Word       ??   2   0..65535
Short       %   2   -32768..32767
```

Integer	**&**	**4**	**-2147483648..2147483647**
Long	&	4	-2147483648..2147483647
Single	!	4	1.5 x 10^{45}..3.4 x 10^{38}
Double	**#**	**8**	**5.0 x 10^{324}..1.7 x 10^{308}**

We have done the ones in bold, but not the rest. If you want a challenge, go ahead and see if you can implement these.

Conclusion:

Well, that concludes the first chapter of Part II. How did you like it? Was it better than previous chapters? Or was it worse? Did you like my new style or hate it? Or didn't you notice any difference!? Let me know so I can try to write in the style that most suites you.

We have accomplished quite a bit here and I hope you feel so too. Our major accomplishment was that KoolB is no longer just a skeleton of a compiler – it has now climbed the ladder a rung and can now be considered a toy compiler. A toy compiler is a curiosity, a compiler that looks like it might have potential or some appeal, but really isn't too useful. KoolB will probably remain such until we get to Part III of this tutorial where it will claim it's rightful spot as a true compiler with easy to use BASIC language. The other accomplishment is that KoolB can compile the DIM statement, which by itself isn't too useful, but will be very useful when we add more functionality to the compiler.

As for the future, the next chapter will be a small detour – it will focus on making KoolB easier to use instead of adding support for more language features. Our goal will to notify the user that he or she made a mistake when creating the BASIC source code and then offer suggestions on how to the problem. Basically, we will be adding the ability for KoolB to check for errors and then report them in a user-friendly way. It will be a shorter chapter and only a small detour in adding new language features to KoolB. See you then!