Defining Pointers

Introduction

When writing a program, you declare the necessary variables that you will need in order to accomplish your work. When declaring these variables, you are asking the compiler to reserve a set amount of space in the computer's memory for the variable or object that you want to use. After declaring such a variable, the compiler uses the variable's name to refer to that memory space. Indeed, the computer refers to that space using an address. Everything you declare has an address in the computer memory.

As you declare various variables, the compiler has to make sure it keeps track of these items. To locate a variable that has been declared, the compiler uses an address that can be identified by its binary representation. Since the binary representation can be long and confusing, the computer displays this address using the hexadecimal system.

You can find out the address where a particular variable is stored using the & operator followed by the name of the variable. For example, after declaring an integer as

int NumberOfPages;

you can find the address where the NumberOfPages variable is located by using:

cout << &NumberOfPages;

This program would give you the address of the declared variable:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    int Value;

    cout << "Value lives at " << &Value;

    cout << "\n\NPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

After executing the program, you could get:

```
Value lives at 0063FE00

Press any key to continue...
```

As compilers are different, the program above on Microsoft Visual C++ produces (notice the 0x):

```
Value lives at 0x0012FF74

Press any key to continue
```

Definition

As you can see from the execution of the program above, the address of a variable is very difficult to read and interpret. Fortunately, we do not need to know that address and we do not need to know what it means or where the variable is located. C++ provides an alternative to this problem.

Instead of referring to a variable's address directly, you are allowed to declare another variable, and then use that new variable to refer to the address of the variable you are interested in.

A pointer is a variable whose value points to another variable's address.

Just like any variable in C++, you must declare (and sometimes initialize) a pointer variable before using it. To declare a pointer variable, use a data type, followed by an asterisk (*), followed by the name of the pointer, and a semi-colon. Here is the syntax:

DataType * PointerName;

The data type should be one of those we have learned already. This means it could be an int, a char, a double, etc. The data type of the pointer must be the same type of the variable it points to. Therefore, if you are declaring a pointer that will point to an integer variable, the pointer data type must be an integer.

The asterisk (*) lets the compiler know that the variable that follows is a pointer. There are three ways you can type the asterisk. These are

DataType* PointerName;
DataType * PointerName;
DataType *PointerName;

By default, it does not matter how you append the *, the compiler will know that the variable that follows is a pointer, as long as the word or group of words on the left side of the asterisk represents a valid data type. You should be careful when declaring various pointers. If you declare a few of them on the same line, like this:

DataType* Pointer1, Pointer2;

Only the first variable is a pointer, the second is a regular variable. If you want to declare different pointer variables, you can use:

DataType *Pointer1, *Pointer2;

Or

DataType * Pointer1;
DataType * Pointer2;

Initializing a Pointer

One of the reasons you are using a pointer is to find an alternative to knowing the address of a variable (later, we will know the real reason for using pointers). A variable should be initialized before being used. This allows the compiler to put something into the memory space allocated for that variable.

To use a pointer P effectively, for its intended purpose, you need to tell the compiler that: pointer P will be used to point to the address of variable V. You do this by initializing the pointer. A pointer is initialized (almost) like any other variable, using the assignment operator (=).

Imagine you declare a variable like this:

int Variable;

There are two main ways you can initialize a pointer. When declaring a pointer like this:

int* Pointer;

initialize it by following the assignment operator with & operator and the name of the variable, like this

int* Pointer = &Variable;

You can also initialize a pointer on a different line, after declaring it. This time, you should not use the asterisk on the Pointer, but the compiler should know that the pointer will point to a variable's address. Therefore, the name of the variable will still use the & operator. This is done as follows:

int Value = 12;
int *Pointer;

Pointer = &Value;

Here is an example:

int Value = 12;int *Pointer;// Do something, if necessary Pointer = &Value;// Now we can do something with the pointer.

Just like a regular variable has an address, a pointer, as a variable, has an address too. To find out the address of a pointer variable, type the ampersand on its left, without the *. The following program displays the addresses of both a variable and its pointer:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    int Value = 12;
    int *Pointer = &Value;

    cout << "Value lives at: " << &Value << "\n";
    cout << "Pointer lives at: " << &Pointer << "\n";

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

An example of running the program would produce:

```
Value lives at: 1245064
Pointer lives at: 1245060


Press any key to continue...
```

Once you have declared and initialized a pointer variable, it holds the same value as the variable it points to. For example, if a variable declared as int PrinterType is assigned an intial value of 4, its pointer declared and initialized as int* Type = &PrinterType would hold the value 4 that was given to PrinterType. To display the value of a variable such as PrinterType, you could write:

cout << PrinterType;

To display the value that a pointer holds, use its name preceded by the *. The following program illustrates that:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    int Value = 12;
    int *Pointer = &Value;

    cout << "Value = " << Value << "\n";
    cout << "Pointer = " << *Pointer << "\n";

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

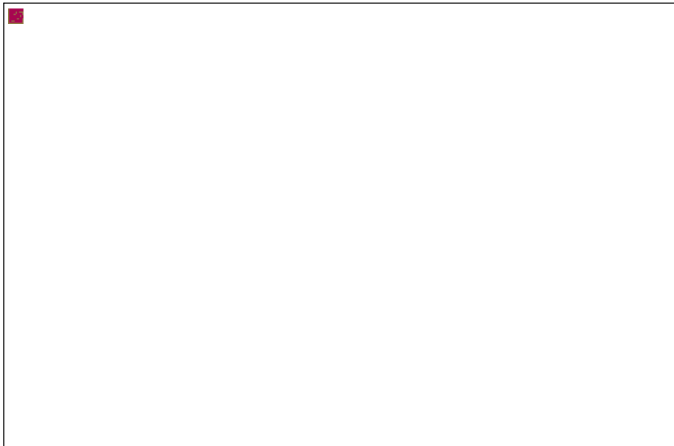
The program would produce:

```
Value = 12
Pointer = 12


Press any key to continue...
```

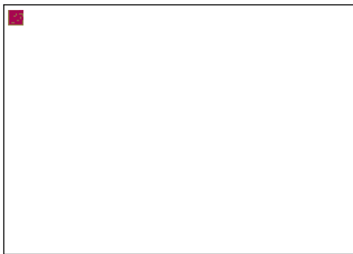
If you want to display the value held by a pointer, make sure you know what variable the pointer is pointing to. In other words, make sure you initialize it. If you attempt to display the value of an un-initialized pointer, the C++ Builder compiler would throw an error and would stop. As compilers are different, unlike Bcb, MSVC initializes a pointer to 0 (actually NULL) and would display 0 for an uninitialized pointer. For the following program, Bcb would not completely execute the program (it would stop on the line that tries to display the value of the pointer), while MSVC would display 0 for the pointer:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    int Value = 12;
    int *Pointer;

    cout << "Value = " << Value << "\n";
    cout << "Pointer = " << *Pointer << "\n";
    Pointer = &Value;

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

In C++ Builder, the program would stop as follows:



And it would display the following operating system error (this is from WinXP):



Therefore, before displaying the value of a pointer, know what value that pointer is holding. Otherwise, wait until you have initialized it before displaying its value:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    int Value = 12;
    int *Pointer;

    cout << "Value = " << Value << "\n";
    Pointer = &Value;
    cout << "Pointer = " << *Pointer << "\n";

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

You do not have to always initialize either the pointer or the variable it is pointint to. You can first declare both the variable and a pointer, then initialize them later on, when needed:


```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    int NumberOfPages = 452;
    int *Pages;

    cout << "This is my book collection\n";
    cout << "It helps me to keep track of books that "
         << "belong to me and those that I borrow\n\n";

    Pages = &NumberOfPages;

    cout << "Number of pages: " << NumberOfPages << "\n";
    cout << "The same as: " << *Pages << "\n";

    cout << "\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

Once you have declared a variable and assign it to a pointer, during the course of your program, the value of a variable is likely to change but as long as the pointer points to the same variable, it would hold the same value as the variable it points to:

```
This is my book collection
It helps me to keep track of books that belong to me and those that I borrow

Number of pages: 452
The same as: 452

Press any key to continue
```

The fact that a variable and its pointers hold the same valud is illustrated in the following:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    int NumberOfPages = 452;
    int *Pages;

    cout << "This is my book collection\n";
    cout << "It helps me to keep track of books that "
         << "belong to me and those that I borrow\n\n";

    Pages = &NumberOfPages;

    cout << "First Book\n";
    cout << "Number of pages: " << NumberOfPages << "\n";
    cout << "The same as: " << *Pages << "\n\n";

    NumberOfPages = 740;

    cout << "Second Book\n";
    cout << "Number of pages: " << NumberOfPages << "\n";
```

```
    cout << "The same as: " << *Pages << "\n";

    cout << "\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

This would produce:

```
This is my book collection
It helps me to keep track of books that belong to me and those that I borrow

First Book
Number of pages: 452
The same as: 452

Second Book
Number of pages: 740
The same as: 740

Press any key to continue
```

Since the variable and its pointer are holding the same value, you can safely change the value of the pointer and this would reflect on the value of the variable itself:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    int NumberOfPages;
    int *Pages;

    cout << "This is my book collection\n";
    cout << "It helps me to keep track of books that "
         << "belong to me and those that I borrow\n\n";

    Pages = &NumberOfPages;
    NumberOfPages = 248;

    cout << "First Book\n";
    cout << "Number of pages: " << NumberOfPages << "\n";
    cout << "The same as: " << *Pages << "\n\n";

    *Pages = 588;

    cout << "Second Book\n";
    cout << "Number of pages: " << NumberOfPages << "\n";
    cout << "The same as: " << *Pages << "\n";

    cout << "\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

This would produce:

```
This is my book collection
It helps me to keep track of books that belong to me and those that I borrow

First Book
Number of pages: 248
```



```
The same as: 248

Second Book
Number of pages: 588
The same as: 588

Press any key to continue
```

Initializing Pointers

1. Start Borland C++ Builder and create a new project based on the Console Wizard:
2. Create a C++ file that uses VCL and Console Application:
3. Change the content of the file as follows:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    float Length = 24.75, Width = 18.05;

    cout << "Rectangle Properties";
    cout << "\nLength = " << Length;
    cout << "\nWidth = " << Width;

    cout << "\n\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

4. To execute and test the program, press F9:

```
Rectangle Properties
Length = 24.75
Width = 18.05

Press any key to continue
```

5. Return to Bcb.
6. To declare a pointer that would point to the Length variable, change the variables section as follows:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    float Length = 24.75, Width = 18.05;
    float *RectLength = &Length;

    cout << "Rectangle Properties";
    cout << "\nLength = " << Length;
    cout << "\nLength = " << *RectLength;
    cout << "\nWidth  = " << Width;
```

```
    cout << "\n\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

7. Press F9 to test the program:

```
Rectangle Properties
Length = 24.75
Length = 24.75
Width  = 18.05

Press any key to continue
```

8. Return to Bcb

Requesting Pointers Values

You can request the value of a pointer variable the same way you request the value of a regular variable as long as you remember that it is a pointer. This ability allows a variable and its pointer to exchange information back and forth.

To request a value for a pointer using the cin operator, precede the name of the pointer with an asterisk:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    double* Wage;

    cout << "What is the employee's salary? ";
    cin >> *Wage;

    cout << "\nEmployee's Salary: $" << *Wage;

    cout << "\n\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

An example of running the program is (in MSVC, this program would produce a warning because the pointer has not been initialized):

```
What is the employee's salary? 10.82

Employee's Salary: $10.82

Press any key to continue
```

If the pointer had been initialized as pointing to another variable, whether the variable or the pointer got its data from the user, the assigned value would belong to either variable. Using this ability, you can get the value using the variable or its pointer; the result would be the same. And, of course, the value of the variable or the pointer can change any time (remember, the pointer is a "variable"). The following example requests the salary using the Salary regular variable, then it requests another value using the pointer:

```
//---------------------------------------------------------------------------
```



```
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    double Salary;
    double* Wage = &Salary;

    cout << "What is the employee's salary? ";
    cin >> Salary;
    cout << "\nEmployee's Salary: $" << Salary;
    cout << "\nEmployee's Wage:   $" << *Wage << "\n\n";

    cout << "What is the employee's wage? ";
    cin >> *Wage;
    cout << "\nEmployee's Salary: $" << Salary;
    cout << "\nEmployee's Wage:   $" << *Wage;

    cout << "\n\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

Here is an example of running the program:

```
What is the employee's salary? 15.62

Employee's Salary: $15.62
Employee's Wage:   $15.62

What is the employee's wage? 20.84

Employee's Salary: $20.84
Employee's Wage:   $20.84

Press any key to continue
```

Requesting Pointer Values

1. Continuing with our example, change the content of the file as follows:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    float Length, Width;
    float* RectLength;
    float* RectWidth;

    cout << "Enter rectangle's length: ";
    cin >> Length;
    cout << "Enter rectangle's width:  ";
    cin >> Width;

    RectLength = &Length;
    RectWidth = &Width;

    cout << "\nRectangle Properties";
    cout << "\nLength = " << *RectLength;
```

```
    cout << "\nWidth  = " << *RectWidth;

    cout << "\n\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

2. To test the program, press F9. Here is an example:

```
Enter rectangle's length: 52.08
Enter rectangle's width:  44.64

Rectangle Properties
Length = 52.08
Width  = 44.64

Press any key to continue
```

3. Return to Bcb.
4. To request variable values from the user using pointers, change the file as follows:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
int main(int argc, char* argv[])
{
    float Length, Width;
    float* RectLength;
    float* RectWidth;

    cout << "Enter rectangle's length: ";
    cin >> Length;
    cout << "Enter rectangle's width:  ";
    cin >> Width;

    RectLength = &Length;
    RectWidth = &Width;

    cout << "\nRectangle Properties";
    cout << "\nLength = " << *RectLength;
    cout << "\nWidth  = " << *RectWidth;

    cout << "\n\nNew rectangle's length: ";
    cin >> *RectLength;
    cout << "New rectangle's width:  ";
    cin >> *RectWidth;

    cout << "\nRectangle Properties";
    cout << "\nLength = " << Length;
    cout << "\nWidth  = " << *RectWidth;

    cout << "\n\nPress any key to continue";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

5. Press F9 to test the program:

```
Enter rectangle's length: 44.22
Enter rectangle's width:  38.24

Rectangle Properties
```



```
Length = 44.22
Width  = 38.24

New rectangle's length: 25.15
New rectangle's width:  20.85

Rectangle Properties
Length = 25.15
Width  = 20.85

Press any key to continue
```

6. Return to Bcb

Operations on Pointers

Added just a few rules, a pointer is a variable like any other: you can get its value from the user and you can apply any of the algebraic operations we have learned; you can increment its value; you can apply its concept on a function, etc.

A variable is a value that is supposed to change some time to time. Since a pointer is a variable (whose value points to another variable), the value of a pointer is affected by the variable it points to. You can use this indirection to change the value of a pointer when changing the pointed variable.

When using a pointer to get a value from the user, don't forget the * operator, otherwise, the result would be unpredictable. Once you have gotten a value and stored it in a variable, it is available for any regular operation. In the same way, you can request a value from the user and store it in a pointer. Here is an example:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    int Boys;
    int Girls;
    int *ptrBoys;
    int *ptrGirls;
    ptrBoys = &Boys;
    ptrGirls = &Girls;

    cout << "Number of male students: ";
    cin >> *ptrBoys;
    cout << "Number of female students: ";
    cin >> *ptrGirls;

    cout << "\nNumber of students:";
    cout << "\nBoys: " << Boys << "\nThat is: " << *ptrBoys;
    cout << "\nGirls: " << Girls << "\nThat is: " << *ptrGirls;

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

You can perform algebraic operations on pointers. If you declare two pointers to integers as:

```
Integer *Value1;
Integer *Value2;
```

To perform the addition operation, you could write:

```
*Value1 + *Value2;
```

The result would be an integer. Sometimes an operation and the asterisk of the pointer would seem confusing as in *u - *v or in *Value1 * *Value2. To make your expressions easier to read, you should include the pointer name and its asterisk between parentheses, like this: (*Value1) * (*Value2). In the following program, the user supplies values for variables and the program performs operations on pointers:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    int Boys;
    int Girls;
    int Total;
    int *ptrBoys;
    int *ptrGirls;
    int *ptrTotal;

    ptrBoys = &Boys;
    ptrGirls = &Girls;
    ptrTotal = &Total;

    cout << "Number of male students: ";
    cin >> *ptrBoys;
    cout << "Number of female students: ";
    cin >> *ptrGirls;
    cout << "\nNumber of students:";
    cout << "\nBoys: " << Boys << "\nThat is: " << *ptrBoys;
    cout << "\nGirls: " << Girls << "\nThat is: " << *ptrGirls;

    Total = Boys + Girls;
    *ptrTotal = *ptrBoys + *ptrGirls;

    cout << "\n\nTotal number of students: " << Total;
    cout << "\nThere are " << *ptrTotal << " students";

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

This would produce:

```
Number of male students: 24
Number of female students: 32
Number of students:Boys: 24
That is: 24Girls: 32
That is: 32
Total number of students: 56
There are 56 students

Press any key to continue...
```

Why Use Pointers?

Every time you declare a variable, the compiler puts it somewhere, which you can now refer to as an address. Once you know that address, you can use it.

Like a reference, when you pass an argument to a function using the argument as a pointer, the argument is passed using its address. This allows the calling function

to dig into the address of the variable (the argument) and use the value directly. This transaction, like that of passing an argument by reference, allows the calling function to alter the real value of the argument. Using this ability, pointers allow a function to return many values; as opposed to a regular argument passing where the data, although changed inside of the calling function, will regain its previous value once the called function is exited. Therefore, passing arguments as pointers allows a function to return many values, even if the function is declared as void. Other valuable reasons for using pointers involve arrays. Therefore, we will find out other reasons when studying arrays.

Pointers and Functions

We know that a function uses arguments in order to carry its assignment. When necessary, a function also declares its own variable(s) to get the desired return value. Like other variables, pointers can be provided to a function, with just a few rules.

Introduction

When declaring a function that takes a pointer as an argument, use the asterisk for the argument or for each argument. When calling the function, use the references to the variables. The function will perform its assignment on the referenced variable(s). After the function has performed its assignment, the changed value(s) of the argument(s) will be preserved and given to the calling function. Here is an example of what we have learned so far. The following program uses a function that calculates the area of a trapezoid:

An example of running the program is:

Here is an example of running the program:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
void __fastcall Dimensions(double R, double r);
double __fastcall Area(double R, double r);
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    double Radius, radius;

    cout << "Enter the dimensions of the ellipse\n";
    Dimensions(Radius, radius);

    cout << "\nCharacteristics of the ellipse";
    cout << setiosflags(ios::fixed) << setprecision(2);
    cout << "\nLong Radius: " << Radius;
    cout << "\nShort radius: " << radius;
    cout << "\nEllpise Area: " << Area(Radius, radius);

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
void __fastcall Dimensions(double R, double r)
{
    cout << "Long Radius: "; cin >> R;
    cout << "Short Radius: "; cin >> r;
```

```
}
//---------------------------------------------------------------------------
double __fastcall Area(double R, double r)
{
    const double PI = 3.14159;

    return R * r * PI;
}
//---------------------------------------------------------------------------
```

Passing Arguments as Pointers

As you can see, the Dimensions() function could not return the right values; this is because the arguments were passed by value. C++ pointers provide a valuable alternative to argument passing. If we pass the declared dimensions as references, the Dimensions() function would return the new values of the variables.

To pass arguments by reference using pointers, when declaring the function and when defining it, declare each desired argument as a pointer. When calling the function, call each passed argument (if the argument was passed as a pointer) as a reference. Meanwhile, you do not need to modify the declaration of the variables:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
void __fastcall Dimensions(double *R, double *r);
double __fastcall Area(const double R, const double r);
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    double Radius, radius;

    cout << "Enter the dimensions of the ellipse\n";
    Dimensions(&Radius, &radius);

    cout << "\nCharacteristics of the ellipse";
    cout << setiosflags(ios::fixed) << setprecision(2);
    cout << "\nLong Radius: " << Radius;
    cout << "\nShort radius: " << radius;
    cout << "\nEllpise Area: " << Area(Radius, radius);

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
void __fastcall Dimensions(double* R, double* r)
{
    cout << "Long Radius: "; cin >> *R;
    cout << "Short Radius: "; cin >> *r;
}
//---------------------------------------------------------------------------
double __fastcall Area(const double R, const double r)
{
    const double PI = 3.14159;
    return R * r * PI;
}
//---------------------------------------------------------------------------
```

After executing this program, we would get:

```
Enter the dimensions of the trapezoid
Lower base: 35.12
```



```
Upper base: 44.25
Height: 28.36

Characteristics of the trapezoid
Lower Base: 35.12
Upper Base: 44.25
Height: 28.36
Area: 1125.47

Press any key to continue...
```

This time, the Dimensions() function performs the request, which modifies the values of the variables, and returns the new values. This ability of pointers allows a function to return more than one value.

Pointers and Memory Management

By definition, the variables in your program are meant to "vary", that is, their values change regularly. When you declare a variable, such as

```
double Radius = 25.55;

cout << "Circle Radius: " << Radius;
```

the compiler reserves an appropriate amount of memory space for that particular variable. This is done when the program is compiling but before its execution. This method of providing memory space is called static allocation, the memory space is "allocated" to that variable. When the program executes, this static memory allocation does not change; but the memory space might be empty, especially if the variable is not initialized. This is important: the fact that a variable is not initialized means its memory space is empty, it is not equal to zero; it is simply empty. Nothing is occupying it.

If you decide to use a pointer to a declared variable, you can declare and initialize such a variable. This system of declaring and initializing a pointer also allows the compiler to statically allocate a memory space to the pointer. Remember, just like a regular variable, a pointer is also a variable and uses its own memory space. This pointer can also be assigned a value that would be shared with the pointed to variable:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    double Radius;
    double *ptrRad = &Radius;

    *ptrRad = 32.15;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

The new Operator

You can also ask the compiler to provide memory when the program is executing. This is called dynamic allocation. Dynamic allocation is performed by assigning the

new operator to the pointer variable. Here is the syntax:

PointerName = new DataType;

The keyword new is required. The data type can be any of those we are already familiar with, but it must be the same as the variable it is pointing to. This means that, if it is pointing to an integer variable, the data type must be an integer:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    double Radius;
    double *ptrRad = &Radius;

    *ptrRad = 32.15;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl;

    ptrRad = new double;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

Whenever you (decide to) dynamically allocate memory to a pointer variable, if the pointer had been previously initialized, the value that was stored in that pointer is lost (obsolete). Furthermore, if the pointer was pointing to an already declared variable, the relationship is broken: the pointer would not point to that variable anymore. In other words, both the pointer and the variable it was pointing to do not hold the same value anymore. The pointed to variable keeps its value because it is independent from the pointer (it is the pointer that is pointing to the variable and not vice-versa):

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    double Radius;
    double *ptrRad = &Radius;

    *ptrRad = 32.15;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl;

    ptrRad = new double;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

This would produce:

```
Circle Radius: 32.15
Pointer Radius: 32.15
Circle Radius: 32.15
```



```
Pointer Radius: 3.4666e-66

Press any key to continue...
```

Notice that the value of the pointer is now garbage. You have one of two options. You can completely dismiss the variable and reclaim the memory (which is the subject of the next section), or you can assign a new value to such a pointer. If you assign a new value to the pointer, since the pointer and the variable (it used to point to) are not related anymore, they would not hold the same value:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    double Radius;
    double *ptrRad = &Radius;

    *ptrRad = 32.15;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl << endl;

    ptrRad = new double;
    *ptrRad = 12.55;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

This would produce:

```
Circle Radius: 32.15
Pointer Radius: 32.15
Circle Radius: 32.15
Pointer Radius: 12.55

Press any key to continue...
```

Notice that the variable and pointer now have different values. You still have the option to repoint to the desired (and appropriate) variable. If you would like the pointer to point to a particular variable, you must re-initialize the pointer to point to the desired value. You can assign the desired value to the variable:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    double Radius;
    double *ptrRad = &Radius;

    *ptrRad = 32.15;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl << endl;

    ptrRad = new double;
    *ptrRad = 12.55;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl << endl;
```

```
    ptrRad = &Radius;
    Radius = 125.55;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

This would produce:

```
Circle Radius: 32.15
Pointer Radius: 32.15
Circle Radius: 32.15
Pointer Radius: 12.55
Circle Radius: 125.55
Pointer Radius: 125.55

Press any key to continue...
```

The delete Operator

After dynamically allocating memory, you can assign a new value to the pointer for any purpose. Once the memory is not in use anymore, you should reclaim it. This is done with the delete operator, like this:

delete PointerName;

The delete keyword is required. The PointerName is the name of the pointer that was dynamically allocated memory space with the new operator. You cannot use the delete operator if you did not previously use the new operator to allocate memory.

To use the delete operator, simply type delete followed by the name of the pointer whose memory you want to reclaim. This operation is referred to as deallocating the memory (you are deallocating the memory that was dynamically allocated):

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    double Radius;
    double *ptrRad = &Radius;

    . . .

    delete ptrRad;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

Remember something that happened previously (in my time, a few minutes ago). When you dynamically allocated memory to a pointer, it lost the value it used to hold. Even though you gave it some space, its value was garbage until it had been (re)initialized. After initializing it, you provided it with a value to "store" in its memory space. When you used the delete operator, you reclaimed "your" space. Unfortunately, the value you had initialized the pointer with is not gone (although

you cannot access it). More than once, the compiler will not be able to use that memory space because it still contains the value you had assigned to the pointer. This is called memory leak. The solution to definitely reclaim this space and know with certainty that it is available is to assign the value NULL to the pointer. This is a simple operation. If you do not do it, the program might still work but sometimes, you will get unreliable results.

To assign the value NULL to the pointer, perform this operation after performing the delete operation. Here is our complete program:

```
//---------------------------------------------------------------------------
#include <iostream.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
int main(int argc, char* argv[])
{
    double Radius;
    double *ptrRad = &Radius;

    *ptrRad = 32.15;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl << endl;

    ptrRad = new double;
    *ptrRad = 12.55;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl << endl;

    ptrRad = &Radius;
    Radius = 125.55;
    cout << "Circle Radius: " << Radius << endl;
    cout << "Pointer Radius: " << *ptrRad << endl;

    delete ptrRad;
    ptrRad = NULL;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}//---------------------------------------------------------------------------
```

Objects and Pointers

Like a regular data type, a class object can be declared as a pointer, it can be passed to a function as a reference or as a pointer and you can dynamically create an instance of an object.

Introduction

As a review, imagine you create a TTrapezoid object as follows:

```
//---------------------------------------------------------------------------
struct TEllipse
{
    double Radius;
    double radius;
};
//---------------------------------------------------------------------------
```

To use it in a function, you can declare an instance of that object and use the member access operator, the period ".", to call any of its variables. If the declare a TEllipse object as TEllipse Arcand, you can access the long radius member variable

using:

Arcand.Radius;

You can also pass any of its members using the same technique. For example, you can pass the PrinterType member variable of a TOrder to a ProcessOrder function as follows:

TOrder LastWeek;
ProcessOrder(LastWeek.PrinterType);

The trapezoid program we had ealier could be rewritten as follows:

```
//---------------------------------------------------------------------------
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
struct TEllipse
{
    double Radius;
    double radius;
};
//---------------------------------------------------------------------------
double __fastcall GetTheRadius(const string s);
double __fastcall Area(const double R, const double r);
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse Els;

    cout << "Enter the dimensions of the ellipse\n";
    Els.Radius = GetTheRadius("Long Radius: ");
    Els.radius = GetTheRadius("Short Radius: ");

    cout << "\nCharacteristics of the trapezoid";
    cout << setiosflags(ios::fixed) << setprecision(2);
    cout << "\nLong Radius: " << Els.Radius;
    cout << "\nShort Radius: " << Els.radius;
    cout << "\nArea: " << Area(Els.Radius, Els.radius);

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
double __fastcall GetTheRadius(const string s)
{
    double r;

    cout << s;
    cin >> r;
    return r;
}
//---------------------------------------------------------------------------
double __fastcall Area(const double R, const double r)
{
    const double PI = 3.14159;
    return R * r * PI;
}
//---------------------------------------------------------------------------
```

Here is an example of running the program:

```
Enter the dimensions of the trapezoid
Lower Base: 28.15
Upper Base: 22.55
Height: 24.25
```



```
Characteristics of the trapezoid
Lower Base: 28.15
Upper Base: 22.55
Height: 24.25
Area: 614.737

Press any key to continue...
```

Declaring an Object as a Pointer

An object can be declared as a pointer variable using the same * operator as we did for the regular variable. The syntax of declaring a pointer to an object is:

ObjectName * InstanceName;

The object name is one already created; it could in the program or one that shipped with the compiler. Once again, the asterisk lets the compiler know that the object is declared as a pointer. The instance name follows the same rules we have applied to other variables so far. For example, if you decide to declare the above TEllipse instance as a pointer, you could declare it as follows:

TEllipse* Trap;

Just like when initializing a regular variable, you can assign an instance of a regular object to an object’s pointer. Here is an example:

TEllipse Trapper;
TEllipse* Trap = &Trapper;

When an object has been declared as a pointer, use the pointer access operator “->” to access any of its members. For example, to access the Radius variable of the TEllipse object using the Trap pointer, you can type:

Trap->Radius;

The new version of our ellipse program declares a pointer to the object and uses the -> operator to access its members:

```
//---------------------------------------------------------------------------
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
struct TEllipse
{
    double Radius;
    double radius;
};
//---------------------------------------------------------------------------
double __fastcall GetTheRadius(const string s);
double __fastcall Area(const double R, const double r);
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse Els;
    TEllipse* Elisp = &Els;

    cout << "Enter the dimensions of the ellipse\n";
    Elisp->Radius = GetTheRadius("Long Radius: ");
    Elisp->radius = GetTheRadius("Short Radius: ");

    cout << "\nCharacteristics of the trapezoid";
    cout << setiosflags(ios::fixed) << setprecision(2);
```

```
    cout << "\nLong Radius: " << Elisp->Radius;
    cout << "\nShort Radius: " << Elisp->radius;
    cout << "\nArea: " << Area(Elisp->Radius, Elisp->radius);

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
double __fastcall GetTheRadius(const string s)
{
    double r;

    cout << s;
    cin >> r;
    return r;
}
//---------------------------------------------------------------------------
double __fastcall Area(const double R, const double r)
{
    const double PI = 3.14159;
    return R * r * PI;
}
//---------------------------------------------------------------------------
```

You can apply this same concept even if the object has methods. If the object has methods, define each regularly. When calling the method of an object, use the -> operator. The following version of our ellipse program features an object with a constructor, a destructor, and other methods:

```
//---------------------------------------------------------------------------
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
const double PI = 3.14159;
struct TEllipse
{
public:
    TEllipse(double R=0.00, double r=0.00);
    ~TEllipse(){}
    void __fastcall setRadius(const double R) { Radius = R; }
    double __fastcall getRadius() const { return Radius; }
    void __fastcall setradius(const double r) { radius = r; }
    double __fastcall getradius() const { return radius; }
    void __fastcall GetTheDimensions();
    double __fastcall Area() const;
private:
    double Radius;
    double radius;
};
//---------------------------------------------------------------------------
TEllipse::TEllipse(double R, double r)
    : Radius(R), radius(r)
{
}
//---------------------------------------------------------------------------
void __fastcall TEllipse::GetTheDimensions()
{
    cout << "Long Radius: "; cin >> Radius;
    cout << "Short Radius: "; cin >> radius;
}
//---------------------------------------------------------------------------
double __fastcall TEllipse::Area() const
{
    return Radius * radius * PI;
}
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
```



```
{
    TEllipse Els;
    TEllipse* Elisp = &Els;

    cout << "Enter the dimensions of the ellipse\n";

    Elisp->GetTheDimensions();

    cout << "\nCharacteristics of the trapezoid";
    cout << setiosflags(ios::fixed) << setprecision(2);
    cout << "\nLong Radius: " << Elisp->getRadius();
    cout << "\nShort Radius: " << Elisp->getradius();
    cout << "\nArea: " << Elisp->Area();

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

Passing an Object Pointer as an Argument

Using the same techniques of passing regular variables as pointers, you can pass an object as a pointer to a function. We have already learned how to pass a variable as reference and how to pass a pointer. To pass a pointer object to a function, when declaring the function, between the parentheses, type the * operator on the right side of the object name and type a name for the object.. Here is an example:

void Dimensions(TParallelogram* p);

When defining the function, use the pointer access operator (->) to access the desired member of the object:

```
//---------------------------------------------------------------------------
void Dimensions(TParallelogram* p)
{
    cout << "Base:   ";
    cin >> p->Base;
    cout << "Height: ";
    cin >> p->Height;
}
//---------------------------------------------------------------------------
```

When calling the object, call the argument as a reference. Here is an example:

Dimensions(&Para);

This feature of pointers (passing an argument as a pointer) allows a function declared as void to return values, which, as we learned, is one of the primary reasons for using pointers. The object can be as simple as a small structure:

```
//---------------------------------------------------------------------------
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
const double PI = 3.14159;
struct TEllipse
{
public:
    TEllipse(double R=0.00, double r=0.00);
    ~TEllipse(){}
    void __fastcall setRadius(const double R) { Radius = R; }
    double __fastcall getRadius() const { return Radius; }
    void __fastcall setradius(const double r) { radius = r; }
```

```
    double __fastcall getradius() const { return radius; }
    double __fastcall Area() const;
private:
    double Radius;
    double radius;
};
//---------------------------------------------------------------------------
TEllipse::TEllipse(double R, double r)
    : Radius(R), radius(r)
{
}
//---------------------------------------------------------------------------
double __fastcall TEllipse::Area() const
{
    return Radius * radius * PI;
}
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse Els;
    void __fastcall GetTheDimensions(TEllipse* Eps);

    cout << "Enter the dimensions of the ellipse\n";

    GetTheDimensions(&Els);

    cout << "\nCharacteristics of the trapezoid";
    cout << setiosflags(ios::fixed) << setprecision(2);
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
    cout << "\nArea: " << Els.Area();

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
void __fastcall GetTheDimensions(TEllipse* Elips)
{
    double R, r;

    cout << "Long Radius: "; cin >> R;
    cout << "Short Radius: "; cin >> r;

    Elips->setRadius(R);
    Elips->setradius(r);
}
//---------------------------------------------------------------------------
```

If passing an object as a pointer, you can also access its methods using the same -> operator. The following version contains methods used to update the values of the member variables using member methods. Once again, since the object is passed as a pointer, the object methods can take care of updating their corresponding variables:

The Size of an Object

As a regular variable, an object such as TEllipse declared occupies a set amount of space. Like we learned with variables, this assignment of memory space is referred to as static allocation. Using the sizeof operator, you can find out how much space an object is occupying. Here is how we could get the size of the TTrapezoid object:

```
cout << "The size of the TEllipse object is "
     << sizeof(TEllipse) << " Bytes.";
```

In the same way you can find out the amount of space occupied by an instance of

an object:

```
//---------------------------------------------------------------------------
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
const double PI = 3.14159;
struct TEllipse
{
public:
    TEllipse(double R=0.00, double r=0.00);
    ~TEllipse(){}
    void __fastcall setDimensions(const double R, const double r);
    void __fastcall setRadius(const double R) { Radius = R; }
    void __fastcall setradius(const double r) { radius = r; }
    double __fastcall getRadius() const { return Radius; }
    double __fastcall getradius() const { return radius; }
    double __fastcall Area() const;
private:
    double Radius;
    double radius;
};
//---------------------------------------------------------------------------
    . . .
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse Els;
    TEllipse __fastcall GetTheDimensions();

    cout << "The size of the TEllipse object is "
         << sizeof(TEllipse) << " Bytes.\n";
    cout << "The size of the Els instance is "
         << sizeof(Els) << " Bytes.";

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
TEllipse __fastcall GetTheDimensions()
{
    double R, r;
    cout << "Long Radius: "; cin >> R;
    cout << "Short Radius: "; cin >> r;

    return TEllipse(R, r);
}
//---------------------------------------------------------------------------
```

When testing the program, I got (I am using Microsoft Windows 98SE):

```
The size of the TEllipse object is 16 Bytes.
The size of the Els instance is 16 Bytes.

Press any key to continue...
```

If the object is declared as a pointer, to get the size of a pointer to an object, use the sizeof operator. If you use just the name of the pointer, you will get the current size of the declared object. If you append the * operator, you will get the actual size of the object:

```
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse *Els;
```

```
    TEllipse __fastcall GetTheDimensions();

    cout << "The size of the TEllipse object is "
         << sizeof(TEllipse) << " Bytes.\n";
    cout << "The size of the Els instance is "
         << sizeof(*Els) << " Bytes.\n";
    cout << "The size of the Els instance is "
         << sizeof(Els) << " Bytes.\n";

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

This would produce:

```
The size of the TEllipse object is 16 Bytes.
The size of a pointer to TEllipse 16 Bytes.
The size of the Els instance is 4 Bytes.

Press any key to continue...
```

Dynamic Objects

The objects we have declared so far used static allocation to occupied their assigned space. For example, we learned that after declaring and initializing a pointer variable, the variable and its pointer hold the same value. This allows you to know that, when you call either one, you would get the same value. In the same way, after declaring and initializing an instance of an object, the object and its pointer hold the same values. This cal be illustrated the same way we did with regular variables. If you assign the right values to either the object or its pointer, you would see that both carry the same values:

```
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse Els;
    TEllipse *Elips = &Els;

    cout << "Characteristics of a pointer to a TEllipse";
    cout << setiosflags(ios::fixed) << setprecision(2);
    cout << "\nLong Radius: " << Elips->getRadius();
    cout << "\nShort Radius: " << Elips->getradius();
    cout << "\nArea: " << Elips->Area();

    cout << "\n\nCharacteristics of a TEllipse instance";
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
    cout << "\nArea: " << Els.Area();

    Elips->setDimensions(25.55, 20.15);

    cout << "\n\nCharacteristics of a pointer to a TEllipse";
    cout << "\nLong Radius: " << Elips->getRadius();
    cout << "\nShort Radius: " << Elips->getradius();
    cout << "\nArea: " << Elips->Area();

    cout << "\n\nCharacteristics of a TEllipse instance";
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
    cout << "\nArea: " << Els.Area();

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```


Although we assigned the new values using the pointer, we displayed the properties of the object using the regularly declared object. Notice that the values are valid.

As opposed to static allocation, you can ask the compiler to dynamically allocate memory to an object. This is done using the new operator. The syntax of dynamically allocating memory to an object is:

ObjectInstance = new ObjectName;

The ObjectInstance is the name you used when declaring the pointer to the object. From the declaration above, this would be Elips. The assignment (=) and the new operators are required to let the compiler know that you want to dynamically create an object. The ObjectName is the original name of the object, as if it were a regular data type. In our example, it would be TEllipse. For example, to dynamically create a pointer to a TEllipse object, you could use the following:

Elips = new TEllipse;

Just as done with variables, after dynamically allocating memory, the pointer looses the previous value it was holding. If you want to access values of the member variables of the object, you would see that the object is holding unspecified values. The following main() function of the program illustrates dynamically allocating memory to an object:

```
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse Els;
    TEllipse *Elips = &Els;

    Elips->setDimensions(25.55, 20.15);

    cout << "Characteristics of a pointer to a TEllipse";
    cout << "\nLong Radius: " << Elips->getRadius();
    cout << "\nShort Radius: " << Elips->getradius();
    cout << "\nArea: " << Elips->Area();

    cout << "\n\nCharacteristics of a TEllipse instance";
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
    cout << "\nArea: " << Els.Area();

    Elips = new TEllipse;

    cout << "\n\nCharacteristics of a pointer to a TEllipse";
    cout << "\nLong Radius: " << Elips->getRadius();
    cout << "\nShort Radius: " << Elips->getradius();
    cout << "\nArea: " << Elips->Area();

    cout << "\n\nCharacteristics of a TEllipse instance";
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
    cout << "\nArea: " << Els.Area();

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

If you want to use the object dynamically created, make sure you (re)initialize it. Once it has been reinintialized, you can again access either the pointer or the object instance it is pointing to:

```
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse Els;
    TEllipse *Elips = &Els;

    Elips->setDimensions(25.55, 20.15);

    cout << "Characteristics of a TEllipse instance";
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
    cout << "\nArea: " << Els.Area();

    Elips = new TEllipse;

    cout << "\n\nCharacteristics of a pointer to a TEllipse";
    cout << "\nLong Radius: " << Elips->getRadius();
    cout << "\nShort Radius: " << Elips->getradius();
    cout << "\nArea: " << Elips->Area();

    Elips = &Els;
    Elips->setDimensions(16.42, 12.28);

    cout << "\n\nCharacteristics of a TEllipse instance";
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
    cout << "\nArea: " << Els.Area();

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

Once you have finished using a dynamically created object, you should get rid of it and reclaim the memory space it was using. To delete a dynamic object, use the delete operator. The syntax is:

delete ObjectInstance;

The delete keyword is required. The ObjectInstance is the name used when declaring the pointer, the same name used when using the new operator. Once again, you must use the delete operator only if the memory had previously been allocated.

Although the (C++ Builder) compiler is smart enough and can take care of many issues behind the scenes, including cleaning after, the safest way to avoid side effects is to explicitly reclaim the memory space that was assigned to the dynamic object. Therefore, to avoid memory leaks, assign the NULL constant to the deleted instance of the object (when we start performing Windows programming, you will notice that some of our dynamically objects will not need their memory to be reclaimed; this is because C++ Builder has a smart compiler that can perform housekeeping for its objects). This is the new version of the main() function with the object dynamically deleted:

```
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    TEllipse Els;
    TEllipse *Elips = &Els;

    Elips->setDimensions(25.55, 20.15);

    cout << "Characteristics of a TEllipse instance";
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
```



```
    cout << "\nArea: " << Els.Area();

    Elips = new TEllipse;

    cout << "\n\nCharacteristics of a pointer to a TEllipse";
    cout << "\nLong Radius: " << Elips->getRadius();
    cout << "\nShort Radius: " << Elips->getradius();
    cout << "\nArea: " << Elips->Area();

    Elips = &Els;
    Elips->setDimensions(16.42, 12.28);

    cout << "\n\nCharacteristics of a TEllipse instance";
    cout << "\nLong Radius: " << Els.getRadius();
    cout << "\nShort Radius: " << Els.getradius();
    cout << "\nArea: " << Els.Area();

    delete Elips;
    Elips = NULL;

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//---------------------------------------------------------------------------
```

Self Returning an Object

The constructors are not the only member functions that can be declared with the name of the class. C++ allows you to manipulate the members of class without using an external function. This technique uses a member function that can return the parent class. To have a function that can refer to the object itself, in the body of the class, declare a function that holds the same name as the class, followed by a valid name of a function and the parentheses. Here is an example:

```
//---------------------------------------------------------------------------
const double PI = 3.14159;
struct TEllipse
{
public:
        TEllipse(double R=0.00, double r=0.00);
        ~TEllipse(){}
        void __fastcall setDimensions(const double R, const double r);
        void __fastcall setRadius(const double R) { Radius = R; }
        void __fastcall setradius(const double r) { radius = r; }
        double __fastcall getRadius() const { return Radius; }
        double __fastcall getradius() const { return radius; }
        double __fastcall Area() const;
        TEllipse __fastcall Additional();
private:
        double Radius;
        double radius;
};
//---------------------------------------------------------------------------
```

An example of using such a function would consist of changing the value of each member of the class. Since the function is declared as returning the value of the same variable, you can implement it as follows:

```
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::Additional()
{
        TEllipse E;

        // The constant double values used here were randomly chosen
        E.Radius = Radius + 12.52;
        E.radius = radius + 8.95;
```

```
        return T;
}
//---------------------------------------------------------------------------
```

Calling this self-returning function is equivalent to changing the values of the member variables, as illustrated when called in the main() function:

```
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
        TEllipse Els;

        cout << "An ellipse with default dimensions";
        cout << setiosflags(ios::fixed) << setprecision(2);
        cout << "\nLong Radius: " << Els.getRadius();
        cout << "\nShort Radius: " << Els.getradius();
        cout << "\nArea: " << Els.Area();

        TEllipse Plane = Els.Additional();

        cout << "\n\nAn ellipse as a self-returing object";
        cout << "\nLong Radius: " << Plane.getRadius();
        cout << "\nShort Radius: " << Plane.getradius();
        cout << "\nArea: " << Plane.Area();

        cout << "\n\nPress any key to continue...";
        getchar();
        return 0;
}
//---------------------------------------------------------------------------
```

You can use this ability to declare almost any type of function that returns the same class. A particular function can be used to modify the default values of the member variables. Another type of function can be used to convert the values of another class into those needed by the class. Here are example:

```
//---------------------------------------------------------------------------
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
const double PI = 3.14159;
//---------------------------------------------------------------------------
struct TRectangle
{
public:
        __fastcall TRectangle(double L = 0.00, double H = 0.00);
        __fastcall ~TRectangle() {}
        void __fastcall setDimensions(const double L, const double H);
        double __fastcall getLength() const { return Length; }
        double __fastcall getHeight() const { return Height; }
private:
        double Length;
        double Height;
};
//---------------------------------------------------------------------------
__fastcall TRectangle::TRectangle(double L, double H)
        : Length(L), Height(H)
{
}
//---------------------------------------------------------------------------
void __fastcall TRectangle::setDimensions(const double L, const double H)
{
        Length = L;
        Height = H;
}
//---------------------------------------------------------------------------
```



```
struct TEllipse
{
public:
        __fastcall TEllipse(double R=0.00, double r=0.00);
        __fastcall ~TEllipse(){}
        void __fastcall setDimensions(const double R, const double r);
        void __fastcall setRadius(const double R) { Radius = R; }
        void __fastcall setradius(const double r) { radius = r; }
        double __fastcall getRadius() const { return Radius; }
        double __fastcall getradius() const { return radius; }
        double __fastcall Area() const;
        TEllipse __fastcall Additional();
        TEllipse __fastcall AddAConstant(const double d) const;
        TEllipse __fastcall AddAnotherObject(const TEllipse& E) const;
        TEllipse __fastcall UseAnExternalObject(const TRectangle& P) const;
private:
        double Radius;
        double radius;
};
//---------------------------------------------------------------------------
__fastcall TEllipse::TEllipse(double R, double r)
        : Radius(R), radius(r)
{
}
//---------------------------------------------------------------------------
void __fastcall TEllipse::setDimensions(const double R, const double r)
{
        setRadius(R);
        setradius(r);
}
//---------------------------------------------------------------------------
double __fastcall TEllipse::Area() const
{
        return Radius * radius * PI;
}
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::Additional()
{
        TEllipse E;

        // The constant double values used here were randomly chosen
        E.Radius = Radius + 12.52;
        E.radius = radius + 8.95;

        return E;
}
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::AddAConstant(const double d) const
{
        TEllipse E;
        E.Radius = E.Radius + d;
        E.radius = E.radius + d;

        return E;
}
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::AddAnotherObject(const TEllipse& E) const
{
        TEllipse L;
        L.Radius += E.Radius;
        L.radius += E.radius;

        return E;
}
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::UseAnExternalObject(const TRectangle& R) const
{
        TEllipse E;
        E.Radius = R.getLength() / 2;

        return E;
```

```
}
//---------------------------------------------------------------------------
void __fastcall Characteristics(const TEllipse& Disk)
{
	cout << setiosflags(ios::fixed) << setprecision(2);
	cout << "\nLong Radius: " << Disk.getRadius();
	cout << "\nShort Radius: " << Disk.getradius();
	cout << "\nArea: " << Disk.Area();
}
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
	TEllipse Els;

	TEllipse Plane = Els.Additional();
	cout << "An ellipse as a self-returing object";
	Characteristics(Plane);

	const double Dim = 5.25;
	TEllipse Sous = Plane.AddAConstant(Dim);
	cout << "\n\nAfter adding a const to the ellipse";
	Characteristics(Sous);

	TEllipse Rounder(32.25, 28.64);
	TEllipse Cole = Els.AddAnotherObject(Rounder);
	cout << "\n\nAfter adding two objects";
	Characteristics(Cole);

	TRectangle Maker(18.95, 10.65);
	TEllipse Trap = Plane.UseAnExternalObject(Maker);
	Trap.setradius(5.55);
	cout << "\n\nAfter adding a rectangle to an ellipse";
	Characteristics(Trap);

	cout << "\n\nPress any key to continue...";
	getchar();
	return 0;
}
//---------------------------------------------------------------------------
```

The this Pointer

C++ proposes an alternative to returning an object from one of its member functions. Instead of explicitly declaring a variable when implementing a function that returns the same object, the compiler simply needs to know what object you want to return: the object that called the function or a newly declared one. If you want to return the same object, you can use a special pointer called this.

As its name implies, the this pointer is a self referencing object, which means it allows you to designate the object that is making the call as the same object you are referring to. Using the this pointer is a technique that allows you to perform any necessary operation on an object without the help of an external function and returning the same object.

Suppose you want to transform the Additional() member function of the TEllipse class so it would return the same variable that calls it. Because the function will return the same object, you do need to declare a local variable to hold the changed variable. Since the member variables of the object will be modified, the member function cannot be declared as a constant. It can be changed as follows:

TEllipse Additional();

When implementing this function, the values of the variables will certainly be modified to implement whatever behavior you want. To return the same object, the this object must be called as a pointer, with *this. Here is the new implementation of the function:


```
//---------------------------------------------------------------------------
TTrapezoid TTrapezoid::Additional()
{
	// The values assigned here were randomly chosen
	Base = Base + 12.52;
	base = base + 8.95;
	Height = Height + 16.44;

	return *this;
}
//---------------------------------------------------------------------------
```

Using the same logic, you can use the this pointer to return the object from any of its member methods that needs to reference the same object:

```
//---------------------------------------------------------------------------
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
//---------------------------------------------------------------------------
const double PI = 3.14159;
//---------------------------------------------------------------------------
struct TRectangle
{
public:
	__fastcall TRectangle(double L = 0.00, double H = 0.00);
	__fastcall ~TRectangle() {}
	void __fastcall setDimensions(const double L, const double H);
	double __fastcall getLength() const { return Length; }
	double __fastcall getHeight() const { return Height; }
private:
	double Length;
	double Height;
};
//---------------------------------------------------------------------------
__fastcall TRectangle::TRectangle(double L, double H)
: Length(L), Height(H)
{
}
//---------------------------------------------------------------------------
void __fastcall TRectangle::setDimensions(const double L, const double H)
{
	Length = L;
	Height = H;
}
//---------------------------------------------------------------------------
struct TEllipse
{
public:
	__fastcall TEllipse(double R=0.00, double r=0.00);
	__fastcall ~TEllipse(){}
	void __fastcall setDimensions(const double R, const double r);
	void __fastcall setRadius(const double R) { Radius = R; }
	void __fastcall setradius(const double r) { radius = r; }
	double __fastcall getRadius() const { return Radius; }
	double __fastcall getradius() const { return radius; }
	double __fastcall Area() const;
	TEllipse __fastcall Additional();
	TEllipse __fastcall AddAConstant(const double d);
	TEllipse __fastcall AddAnotherObject(const TEllipse& E);
	TEllipse __fastcall UseAnExternalObject(const TRectangle& P);
private:
	double Radius;
	double radius;
};
//---------------------------------------------------------------------------
__fastcall TEllipse::TEllipse(double R, double r)
: Radius(R), radius(r)
{
```

```
}
//---------------------------------------------------------------------------
void __fastcall TEllipse::setDimensions(const double R, const double r)
{
	setRadius(R);
	setradius(r);
}
//---------------------------------------------------------------------------
double __fastcall TEllipse::Area() const
{
	return Radius * radius * PI;
}
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::Additional()
{
	// The constant double values used here were randomly chosen
	Radius = Radius + 12.52;
	radius = radius + 8.95;

	return *this;
}
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::AddAConstant(const double d)
{
	Radius = Radius + d;
	radius = radius + d;

	return *this;
}
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::AddAnotherObject(const TEllipse& E)
{
	Radius += E.Radius;
	radius += E.radius;

	return *this;
}
//---------------------------------------------------------------------------
TEllipse __fastcall TEllipse::UseAnExternalObject(const TRectangle& R)
{
	Radius = R.getLength() / 2;

	return *this;
}
//---------------------------------------------------------------------------
void __fastcall Characteristics(const TEllipse& Disk)
{
	. . .
}
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
	. . .

	cout << "\n\nPress any key to continue...";
	getchar();
	return 0;
}
//---------------------------------------------------------------------------
```

Since the function that return the this pointer returns it as a pointer, you can reinforce the fact that the variable that is being returned is modified. Therefore, instead of returning the object as a regular variable, you should make the returned value a reference:

```
//---------------------------------------------------------------------------
#include <iomanip.h>
#pragma hdrstop
//---------------------------------------------------------------------------
#pragma argsused
```



```
//---------------------------------------------------------------------------
const double PI = 3.14159;
//---------------------------------------------------------------------------
struct TRectangle
{
	. . .
};
//---------------------------------------------------------------------------
__fastcall TRectangle::TRectangle(double L, double H)
	: Length(L), Height(H)
{
}
//---------------------------------------------------------------------------
void __fastcall TRectangle::setDimensions(const double L, const double H)
{
	Length = L;
	Height = H;
}
//---------------------------------------------------------------------------
struct TEllipse
{
public:
	. . .
	TEllipse& __fastcall Additional();
	TEllipse& __fastcall AddAConstant(const double d);
	TEllipse& __fastcall AddAnotherObject(const TEllipse& E);
	TEllipse& __fastcall UseAnExternalObject(const TRectangle& P);
private:
	double Radius;
	double radius;
};
//---------------------------------------------------------------------------
__fastcall TEllipse::TEllipse(double R, double r)
	: Radius(R), radius(r)
{
}
//---------------------------------------------------------------------------
void __fastcall TEllipse::setDimensions(const double R, const double r)
{
	setRadius(R);
	setradius(r);
}
//---------------------------------------------------------------------------
double __fastcall TEllipse::Area() const
{
	return Radius * radius * PI;
}
//---------------------------------------------------------------------------
TEllipse& __fastcall TEllipse::Additional()
{
	// The constant double values used here were randomly chosen
	Radius = Radius + 12.52;
	radius = radius + 8.95;

	return *this;
}
//---------------------------------------------------------------------------
TEllipse& __fastcall TEllipse::AddAConstant(const double d)
{
	Radius = Radius + d;
	radius = radius + d;

	return *this;
}
//---------------------------------------------------------------------------
TEllipse& __fastcall TEllipse::AddAnotherObject(const TEllipse& E)
{
	Radius += E.Radius;
	radius += E.radius;

	return *this;
```

```
}
//---------------------------------------------------------------------------
TEllipse& __fastcall TEllipse::UseAnExternalObject(const TRectangle& R)
{
        Radius = R.getLength() / 2;

        return *this;
}
//---------------------------------------------------------------------------
void __fastcall Characteristics(const TEllipse& Disk)
{
        . . .
}
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
        . . .

        cout << "\n\nPress any key to continue...";
        getchar();
        return 0;
}
//---------------------------------------------------------------------------
```

Previous Copyright © 2002 FunctionX Next