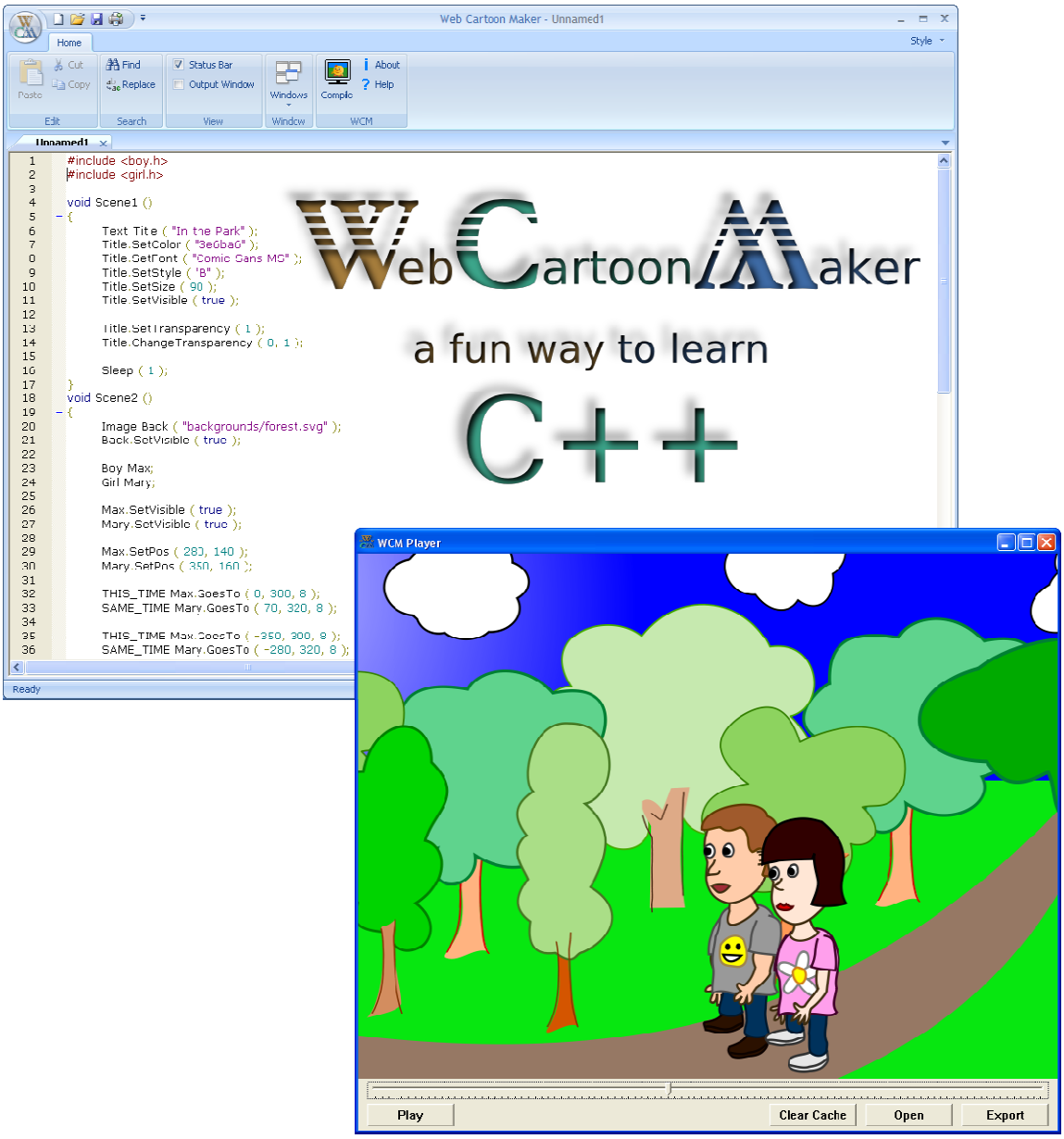


http://www.webcartoonmaker.com

Copyright Information

A lot of material for this book was contributed by our friend James E. Schroeder, Ph.D. (EECS, UC Berkeley, 1974). His help with this book and Web Cartoon Maker project is invaluable.

Support Web Cartoon Maker

Web Cartoon Maker is free! We do not ask you to pay for anything! But there are several ways you may help the project to live and be able to pay for hosting, compiler and other software:

1. Are you planning to buy something on amazon.com? Do this using our link: http://www.amazon.com/?tag=webcarmak-20 and we'll receive a percentage of your total purchase. This will help us and will not cost you anything!
2. Can you draw or compose music? You can donate your graphics and music to Web Cartoon Maker project! Just contact us at the link below: http://www.webcartoonmaker.com/?art=discuss.
3. Write a good review for Web Cartoon Maker somewhere on the web
4. Become our friend on YouTube and subscribe to our channel: http://www.youtube.com/subscription_center?add_user=webcartoonmaker
5. Upload a cartoon to YouTube and let people know it was done in Web Cartoon Maker

Table of Contents

Introduction

About Web Cartoon Maker

Web Cartoon Maker is a free online tool for development of animated 2D cartoons. Unlike much other similar software, it does not pretend to be a so called WYSIWYG (or What You See Is What You Get) tool. It allows you to program your cartoons, using a programming language. While it may sound odd at first, it is more convenient and quick in many cases, if you know how to write a program. Web Cartoon Maker uses a C++ programming language as its engine. Pure C++ is not the easiest language, but Web Cartoon Maker's C++ (also called WCM C++) has some extensions to make the programming easier.

There is a tutorial available at Web Cartoon Maker's web site at http://www.webcartoonmaker.com/?art=help/help which can help you to start programming simple animated cartoons in no time, but you must know some basic C++ syntax rules to write simple WCM scripts. A somewhat more complete understanding of C++ is needed to better understand the examples and for more advanced programming, such as creating characters. Finally, understanding of C++ as implemented in WCM is a good first step towards understanding the language as a whole.

If you haven't looked at the tutorial and some of the sample programs on the web site, it is suggested that you do so now. This document is focused on some of the underlying features of WCM and is not intended as a tutorial for the first time user. Actually examining and running the tutorial scripts, as well as the code excerpts presented here, will aid in understanding the underlying concepts. The best way to learn a programming language is to write programs!

The material presented in this document ranges from vary basic material that must be understood to effectively develop WCM animations to fairly advanced material that can safely be ignored by those not wishing to understand the C++ language. In many cases, the level of the material is indicated as follows:

Very important for developing cartoons

More advanced material for those who wish to learn C++.

Advanced cartoon making capabilities due to WCM being based on C++

There are also notes scattered throughout the document. These notes may help to clarify the concepts involved or give some additional background concerning the development and peculiarities of the syntax being described. There are also some hints that cover various tricks to make your cartoons more interesting. While hopefully of interest to the reader, such notes in general cover material not necessarily needed for either developing cartoons or learning C++.

What is a Programming Language?

The programming language you will be learning is C++. This is actually not a pure C++ but its modification adopted by Web Cartoon Maker for development of animated cartoons. C++ is a high-level programming language. Other high-level programming languages you might have heard of are Java, C, C#, Basic and Fortran.

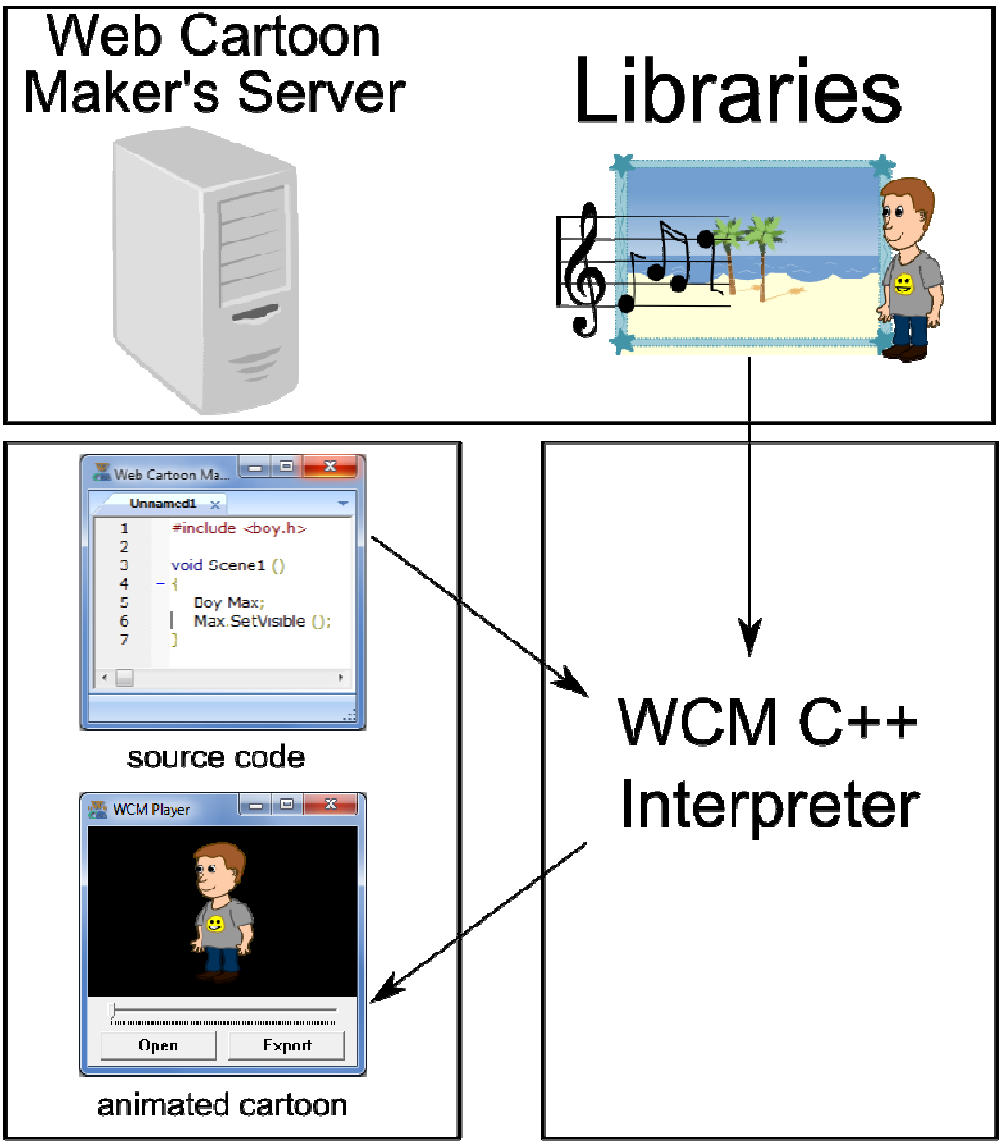
Note: Most of these languages also have several versions or dialects. Fortran was the first high level scientific computer language and was developed in the late 1950's, over half a century ago. It still exists – there is a Fortran 2008 – but has generally been replaced by "object oriented" languages, such as C++, C# and Java. Object Oriented Programming (OOP) will be discussed later in this document.

Fun Note: The most common first computer program shown for any computer language is a "Hello World" program. A WCM version of this classic program will be presented a few pages later. A collection of "Hello World" programs written in 441 different computer languages and 64 human languages is available at http://www.roesler-ac.de/wolfram/hello.htm

Programs written in a high-level language have to be compiled (translated) before they can run. Your programs (scripts) will be translated into a computer friendly (low level) language and run on your computer. The result you will receive is an animated cartoon in WCM format which can be played by WCM Player software and converted to other video formats to run on other video viewers.

Note that the WCM compiler differs from a general purpose compiler in that the output is a video file instead of an executable computer file!

In summary, the C++ compiler is going to translate and run your programs. It will compile your program to an intermediate format first, and then execute (interpret) it to produce animated cartoons. In the process, it will also use various library elements such as characters, sounds, backgrounds, etc. A fairly extensive library is available on the web site and will be accessed automatically by the software. You may also include other images and sounds from the many sources available on the web. For advanced users, you may also create your own characters, images and sound tracks and store them on your own computer. The WCM will use the file paths provided and incorporate these into the final video.



What is a Program – General C++ vs. WCM C++?

Advanced

A program is a sequence of instructions that specifies what should a computer do. Unlike a general program flow, however, your WCM program will specify how to build your animated cartoons rather than a general executable file. There are a few basic functions that appear in about every language, as well as syntax rules that are language specific. WCM follows the C++ rules and in general supports the same programming functions that a regular C++ program does.

A summary of the basic functions expected in C++, and the corresponding implementation (or lack of implementation) in WCM are summarized below. If you aren't interested in C++ as a regular programming language, you can either just skim this section or skip it altogether.

1. Input: Get data from the keyboard, or a file, or some other device. Since your programs will be converted into animated movies there are no interactive input commands supported. Since the main purpose of Web Cartoon Maker is to produce non-interactive animated cartoons this limitations is not very important. However, if you are interested in

C++ as a general programming language, you should study the available input commands carefully.

2. Output: Display data on screen or send data to file or other device. In our case we will display our data in generated animated cartoons.
3. Math: Perform basic mathematical operations like addition and multiplication. While in many cases we do not need them for programming animated cartoons these operations still may be useful if we want a realistic animation of physics (like ball trajectory or a jumping pattern). Random numbers are also sometimes useful.
4. String Manipulation: Combine, separate and otherwise manipulate groups of words and characters. Again, this is often not needed, but can be quite useful.
5. Testing: Check for certain conditions and execute the appropriate sequence of statements. Like math and string manipulation, this also is often not needed, but can be quite useful.
6. Object Manipulation: Many WCM C++ commands display, move and otherwise change the object characteristics that will be ultimately included in the final video output. In WCM C++, there is a built in library of objects and using these objects is the most commonly used capability, however in basic C++, there is no similar built in library unless some graphics package such as GPL (Graphics Programming Language) supported by the compiler.
7. Repetition: Perform some action repeatedly, usually with some variations. This is a very useful capability in WCM.

Believe it or not, that's pretty much all there is to it. Every program you've ever used, no matter how complicated, is made up of functions that look more or less like these. Thus, one way to describe programming is the process of breaking a large, complex task up into smaller and smaller subtasks until eventually the subtasks are simple enough to be performed with one of these simple functions. In our case of making non-interactive animated cartoons it is even simpler because we do not need input functions. In later sections efficient and effective ways to accomplish these basic tasks will be described in more detail.

What is Debugging?

Programming is a complex process, and since it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called bugs and the process of tracking them down and correcting them is called debugging.

Fun Note: The very early computers did not use transistors and integrated circuits but used vacuum tubes, which operated at very high voltages – or in some cases, even electromechanical relays. It was not unusual for a real bug to get electrocuted inside the machine and affect operation – thus hardware "bugs" were a real problem.

There are a few different kinds of errors that can occur in a program, and it is useful to distinguish between them in order to track them down more quickly.

1. Compile-time errors. The compiler can only translate a program if the program is syntactically correct; otherwise, the compilation fails and you will not be able to run your program. Syntax refers to the structure of your program and the rules about that structure. For example, in English, a sentence must begin with a capital letter and end with a period. This sentence contains a syntax error. So does this one For most readers, a few syntax errors are not a significant problem, which is why we can read the poetry of E. E. Cummings

without spewing error messages. Compilers are not so forgiving. If there is a single syntax error anywhere in your program, the compiler will print an error message and quit, and you will not be able to run your program. To make matters worse, there are more syntax rules in C++ than there are in basic English, and the error messages you get from the compiler are often not very helpful. During the first few weeks of your programming experience, you will probably spend a lot of time tracking down syntax errors. As you gain experience, though, you will make fewer errors and find them faster.

2. Run-time errors. The second type of error is a run-time error, so-called because the error does not appear until you run the program. For the simple sorts of programs we will be writing, run-time errors are rare, so it might be a little while before you encounter one. A good example of run-time error is a division by zero. At the moment Web Cartoon Maker does not have a debugger yet and in case of a runtime error you may receive a message like "unknown error".
3. The third type of error is the logical or semantic error. If there is a logical error in your program, it will compile and run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else. In our case events in animated cartoon may be different than we expected. Specifically, it will do what you told it to do. The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying logical errors can be tricky, since it requires you to work backwards by looking at the output of the program and trying to figure out what it is doing. In these cases it may be necessary to add code to your program for the specific purpose of aiding in locating the logical error.

One of the most important skills you should acquire from working with this book is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming. In some ways debugging is like detective work. You are confronted with clues and you have to infer the processes and events that lead to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one. As Sherlock Holmes pointed out: "When you have eliminated the impossible, whatever remains, however improbable, must be the truth." (from A. Conan Doyle's The Sign of Four).

For some people, programming and debugging are the same things. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should always start with a working program that does something, and make small modifications, debugging them as you go, so that you always have a working program. In our case you can start with a very simple animated cartoon and make small modifications to it one by one.

Formal and Natural Languages

Natural languages are the languages that people speak, like English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent

the chemical structure of molecules. And most importantly: Programming languages are formal languages that have been designed to express computations.

As I mentioned before, formal languages tend to have strict rules about syntax. For example, 3+3 = 6 is a syntactically correct mathematical statement, but 3 = +6$ is not. Also, H_2O is a syntactically correct chemical name, but $_2Zz$ is not.

Syntax rules come in two flavors, pertaining to tokens and structure. Tokens are the basic elements of the language, like words and numbers and chemical elements. One of the problems with 3=+6$ is that $ is not a legal token in mathematics (at least as far as I know). Similarly, $_2Zz$ is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement; that is, the way the tokens are arranged. The statement 3=+6$ is structurally illegal, because you can't have a plus sign immediately after an equals sign. Similarly, molecular formulas have to have subscripts after the element name, not before.

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is (although in a natural language you do this unconsciously). This process is called parsing.

For example, when you hear the sentence, "The other shoe fell," you understand that "the other shoe" is the subject and "fell" is the verb. Once you have parsed a sentence, you can figure out what it means, that is, the semantics of the sentence. Assuming that you know what a shoe is, and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common – tokens, structure, syntax and semantics – there are many differences.

1. Ambiguity: Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context
2. Redundancy: In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.
3. Literalness: Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

People who grow up speaking a natural language (everyone) often have a hard time adjusting to formal languages. In some ways the difference between formal and natural language is like the difference between poetry and prose, but more so:

1. Poetry: Words are used for their sounds as well as for their meaning, and the whole poem together creates an effect or emotional response. Ambiguity is not only common but often deliberate.
2. Prose: The literal meaning of words is more important and the structure contributes more meaning. Prose is more amenable to analysis than poetry, but still often ambiguous.
3. Programs: The meaning of a computer program is unambiguous and literal, and can be understood entirely by analysis of the tokens and structure.

Here are some suggestions for reading programs (and other formal languages). First, remember that formal languages are much denser than natural languages, so it takes longer to read them. Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to

right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure. Finally, remember that the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

For writing complex programs, it is often useful to first write a draft version of the program in a natural language and use this version of the program, called **pseudo code**, to develop the basic program flow. The natural language program can then be rewritten in the formal computer language, with careful attention paid to correct syntax. Examples of this approach will be given later in this document.

Our First Cartoon

WCM C++ Program Structure

You know that usually animated cartoons and other kinds of movies consist of scenes. Animated cartoons you are going to make with Web Cartoon Maker also consist of scenes. The typical WCM C++ program looks like this:

```
void Scene1 ()
{
      sentence1;
      sentence2;
      …
}

void Scene2 ()
{
      sentence1;
      sentence2;
      …
}

…
```

Scene1 and Scene2 (and there may be more of them like Scene3, Scene4 and so on) are **functions**. Please do not pay attention on the parentheses and word void yet - their meaning will be explained later.

Note: For those of you who don't want to wait for the full explanation – at a very simple level void and the parentheses just mean that Scene1 () is complete in itself, i.e. it doesn't need any input from other parts of the program and does not supply any output from the function.

These functions contain instructions or **sentences** inside a pair of squiggly-braces ({ and }) specifying how to build a particular scene of your cartoon. Some simple cartoons may have only one scene and require only one Scene1 function. The order of appearance of these functions in your program does not matter. For example Scene2 can be placed before Scene1. There could also be any number of sentences inside these functions. Almost all the sentences must end with a semi-colon (;) .

Please keep in mind that words sentence1, sentence2 and ellipsis symbols are not part of WCM C++ but used for illustration only, because we do not know any real sentences yet.

Note: Also be aware that the exact definition of a “function” and the very similar “method” are not standard between, or sometimes even within, the various high level languages. Usually, these differences are not important but merely confusing, but beware that occasionally the differences can be significant. While the usage is consistent within this document, it may not be consistent with other documents, especially those addressing other high level languages.

So what is a minimal program we can write using WCM C++? Can we just use one Scene1 function with no sentences in it? Sure! The program below can really be compiled using WCM C++ and will produce a one second long black screen cartoon:

```
void Scene1 ()
{
}
```

But why one second long? Actually the length of your animated cartoon is calculated automatically depending on statements used. We did not use any statements and the length of our cartoon should be zero. But zero length cartoons may cause troubles for some third party software and WCM changes the cartoon length to one second when it is supposed to be zero.

Hello World (as promised)

Ok, an empty cartoon is easy. But can we make a cartoon which does something. Sure! Here is an example of simple cartoon displaying text "Hello World" for one second:

```
void Scene1 ()
{
      // declare a text object
      Text MyText ( "Hello World" );
      // make it visible
      MyText.SetVisible ( true );
}
```

Let's look more closely on this example. The lines beginning with // are just comments. A **comment** is a bit of English text that you can put in the middle of a program, usually to explain what the program does. When the compiler sees a //, it ignores everything from there until the end of the line. The 2 lines beginning with // do nothing – but are very useful for understanding the operation or the program. Good programming technique makes liberal use of comments.

The following line: Text MyText ("Hello World"); is our first real statement. First word Text indicates that there will be a text **object** in our scene. What is an object? Imagine you making a real movie and you want to shoot a table, a chair, a car or something else in your movie. These are all objects! When you are making an animated cartoon, the line of text is also an object. You can do something with objects. You can move a table, sit on a chair or drive a car… You can even drive your car away from camera! You can also do a lot of things with a text object in an animated cartoon. You can make text visible and invisible, you can move, scale and rotate it and do other stuff. You will learn how to do this later.

The word MyText is a unique **name** for your object. For example if you shoot a movie you may have 2 chairs in front of your camera. And if you want an actor to sit on a chair you will need to specify on which one. You should say something like "Sit on the left chair please". MyText is something similar to "left chair". You may have many text objects in your scene and if you want to do something with one of them then you need a way to specify with which one. Word MyText (this could actually be almost any other word) will be used later to work with the text object.

The text inside parentheses indicates parameters of your text object. It this case this is just a text to display. As you will see later, this is a **string value** and it must be enclosed in double quotes by C++ standard.

The sentence, like most of the C++ sentences, is terminated with a semi-colon symbol.

Note: Forgetting semi-colons and non-matching braces or brackets are probably the two most common syntax errors made when first writing a program.

The next line MyText.SetVisible (true); is also a statement. You already know that MyText is the name of a text object. A dot after an object name in this case means that an instruction of what to do with an object will follow. Such instruction is called **method**. Different objects may have different methods for working with them. The method name in our case is SetVisible. It has a parameter true inside parentheses. As we will see later this is a boolean (or **bool**) variable which can be either true or false. If the parameter is true then we are going to make an object visible. If it is false then the object is going to be hidden. We use the method SetVisible with parameter true to make our text object visible. By default all objects are invisible, because most of the times you need to use only some of the objects in your scene while other objects are hidden.

Note: The poor dot or period (.), while not an operator (at least in C++) is one of the most overloaded characters there is. (Overloading in general will be discussed later.) It can be a sentence delimiter, the decimal point in a floating point number or, as used in the above paragraph, part of the "dot naming" convention. In general, that convention means the text to the right of the dot is somehow associated with the object to the left of the dot – either as a function operating on the object or as a sub-object of the object.

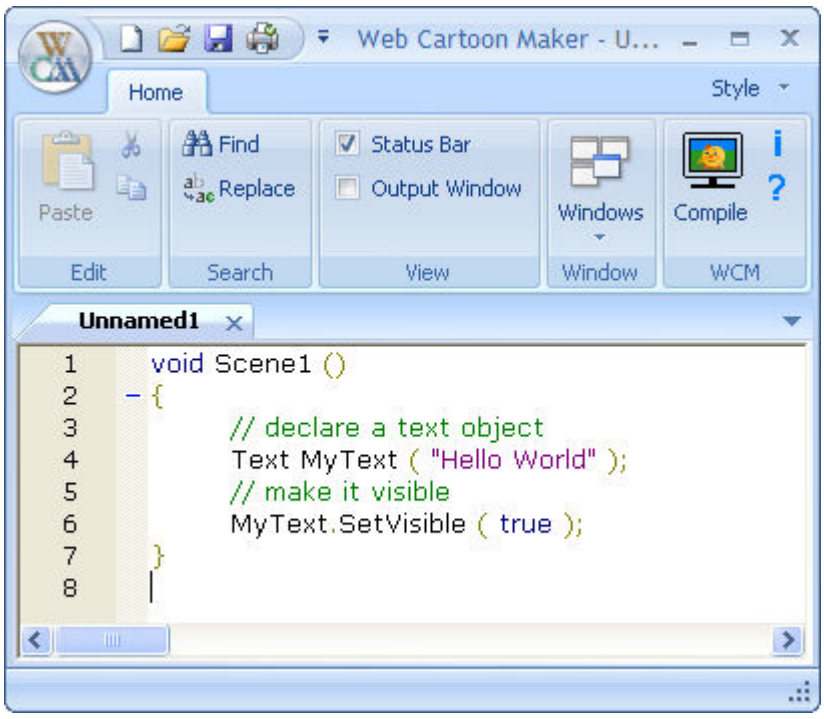
Compiling 

Let's compile our first cartoon. We are going to use the Desktop Edition of Web Cartoon Maker for this purpose.

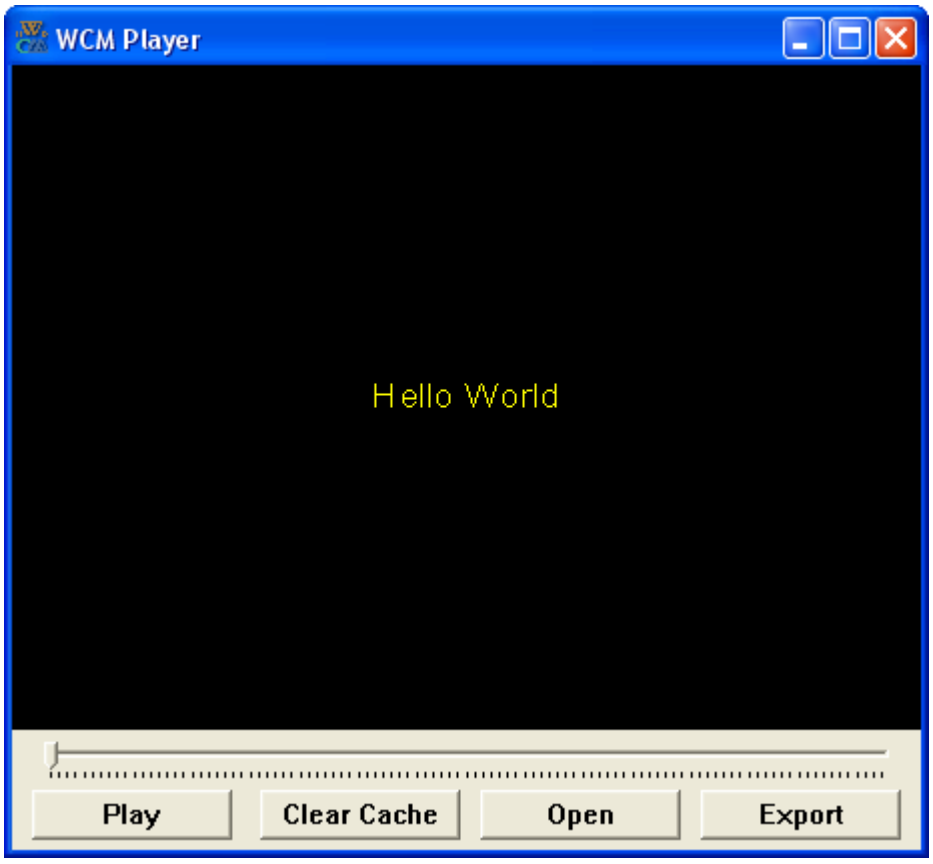
1. Start Web Cartoon Maker (Desktop Edition)
2. Type the following program

```
void Scene1 ()
{
      // declare a text object
      Text MyText ( "Hello World" );
      // make it visible
      MyText.SetVisible ( true );
}
```

3. Click "Compile" button on the toolbar



4. Wait until your program is compiled and movie is created. Status bar message should say "Ready" and "Download Movie" link should appear in the Output Window. WCM Player will start automatically
5. First frame of your cartoon saying "Hello World!" should be displayed



6. Click "Play" button in WCM Player. Your cartoon should play to the end.

7. Close WCM Player.
8. Try to play with your program by modifying the “Hello World” text.

There are three other buttons in the WCM Player window. These buttons control the loading and export of WCM cartoons and will be explained later.

Reference: At this point, you might also want to look at the “Hello World” tutorial and also run the “Text Objects in Web Cartoon Maker” sample script.

Variables, Types and Operators

More Output

As I mentioned in the last chapter, you can put as many statements as you want in Scene1. You can also have as many objects in a scene as you want. Let's try do show two text objects in a scene

```
void Scene1 ()
{
	Text MyText ( "Hello World" );
	MyText.SetVisible ( true );

	Text AnotherText ( "And Hello Again" );
	AnotherText.SetVisible ( true );
}
```

You can probably understand this program by yourself. We are going to have two text objects named MyText and AnotherText in this scene. And we are going to make them visible. But if you try to compile and run the program and preview the cartoon you will see that these 2 text objects are displayed at the same time (at the beginning of your cartoon) and at the same place (in the center of your animated cartoon) and this does not look good. But this is exactly what we asked the interpreter to do.

There is a special instruction (you will learn later that this is also a function) called SetTime for changing time in the scene. If we want our second text object to appear a couple of seconds later then we need to use it like this:

```
void Scene1 ()
{
	Text MyText ( "Hello World" );
	MyText.SetVisible ( true );

	SetTime ( 2.0 );

	Text AnotherText ( "And Hello Again" );
	AnotherText.SetVisible ( true );
}
```

As you can see it accepts a floating-point value as a parameter inside parentheses. This is a time in seconds. The statement SetTime (2.0); specifies that every instruction the interpreter sees after it will happen at 2 seconds after the beginning of a scene in your cartoon. If you want the second text object to appear two and half seconds after the beginning of a scene, then you should specify SetTime (2.5); instead.

You can try to compile and preview this cartoon but it is still does not perform what we wanted. You still see "Hello World" text for two seconds (because we used SetTime (2.0); and the interpreter automatically changed the length of our cartoon to 2 seconds) but then it stops. This is because we

did not specify for how long we want to see the second line of text. We should use SetTime again to change the length of our cartoon to be more than 2 seconds. It should be something like this:

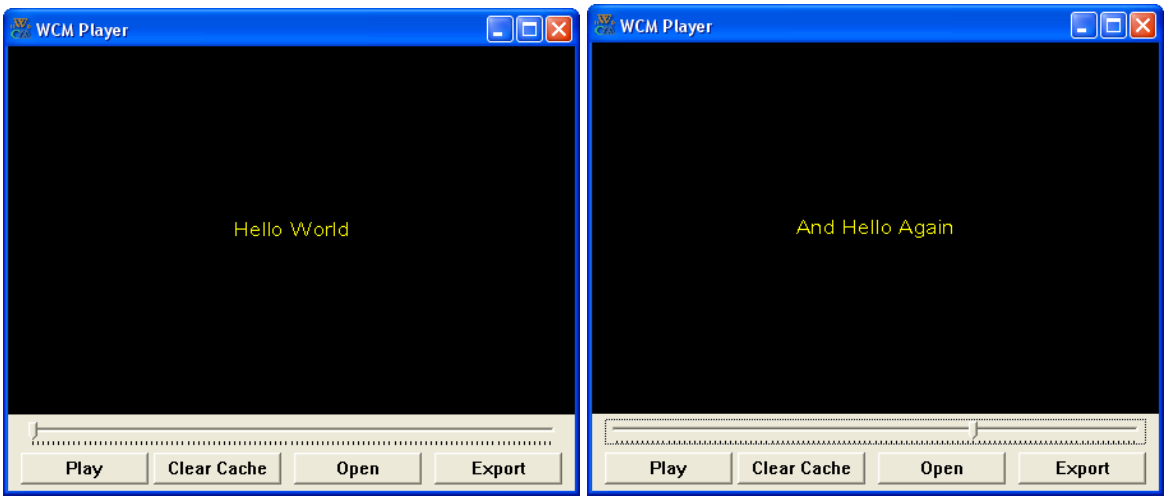
```
void Scene1 ()
{
	Text MyText ( "Hello World" );
	MyText.SetVisible ( true );

	SetTime ( 2.0 );

	Text AnotherText ( "And Hello Again" );
	AnotherText.SetVisible ( true );

	SetTime ( 3.0 );
}
```

This way the our cartoon will be 3 seconds long and second line of text will be displayed beginning from the 2nd second. You can try to compile it again.

But it is still not perfect. The first line of text is still displayed though the entire cartoon. And the second line is displayed beginning from 2nd second. And they are displayed at the same place. We probably want to hide the first text object after 2nd second. You already know how to do this:

```
void Scene1 ()
{
	Text MyText ( "Hello World" );
	MyText.SetVisible ( true );

	SetTime ( 2.0 );

	MyText.SetVisible ( false );
	Text AnotherText ( "And Hello Again" );
	AnotherText.SetVisible ( true );

	SetTime ( 3.0 );
}
```

As it was explained before we used MyText.SetVisible (false); statement to hide the first text object. It is now perfect! Try to compile and preview it. The first text object is displayed for two seconds and then disappears. Then the second text object is displayed for one more second. The total cartoon length is 3 seconds:



I specially wanted to show you that you can have multiple objects in your cartoons by using 2 text objects MyText and AnotherText. But since you are using only one object at a time and they have all the same default parameters like font and size (you will learn how to change them later), you can use a special method called SetText to change the default text. We can simplify this program:

```
void Scene1 ()
{
	Text MyText ( "Hello World" );
	MyText.SetVisible ( true );

	SetTime ( 2.0 );
	MyText.SetText ( "And Hello Again" );

	SetTime ( 3.0 );
}
```

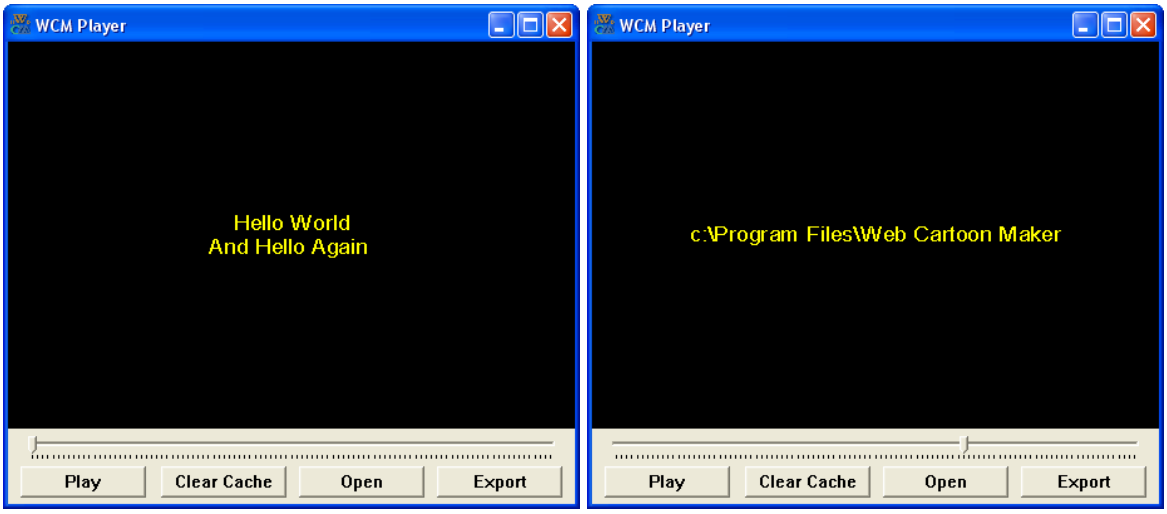
But what if you want to output 2 lines of text at the same time? You can define 2 text objects and make them visible at the same time. But there is another alternative. Actually strings can contain a special line break symbol. It is encoded with 2 symbols – back slash and letter n - "\n". For example if you want to show "And Hello Again" text below "Hello World", then you should use "\n" between them:

```
void Scene1 ()
{
	Text MyText ( "Hello World\nAnd Hello Again " );
	MyText.SetVisible ( true );
	SetTime ( 1.0 );
}
```

This will produce a one second long cartoon with 2 lines of static text. Actually symbol "\" is reserved in C++ to include special symbols like line breaks ("\n") or tabulate ("\t") and some other in the strings. If you want to include an actual backslash symbol in a string, then you must use two of them like "\\". The example below will show two lines of text for a second and then a line of text with backslash:

```
void Scene1 ()
{
	Text MyText ( "Hello World\nAnd Hello Again " );
	MyText.SetVisible ( true );

	SetTime ( 1.0 );
	MyText.SetText ( "c:\\Program Files\\Web Cartoon Maker" );

	SetTime ( 2.0 );
}
```

The output will looks like this:



Variables, Types and Operators

In the preceding section, we worked with both text and numbers to get the screen output desired, but little attention was paid to the exact syntax involved. In the following sections, the necessary syntax for the various variable types and the available operators that act on them will be described in more detail.

Values

A value is one of the fundamental things – like a string, a letter or a number – that a program manipulates. The only values we have manipulated so far are the string and the floating-point numbers. We used strings as parameters of our text objects and compiler identified them because they are enclosed in quotation marks. We used floating point values as parameters to SetTime function and compiler identified them because they only contain digits and a floating point sign (dot).

There are other kinds of values, including integers and characters (in general, we are not talking about cartoon characters yet! We are talking about letters or symbols which make up strings). An integer is a whole number like 1 or 17. A character value is a letter or digit or punctuation mark enclosed in single quotes, like 'a' or '5'.

Fun Note: A character is actually stored in memory as an integer value. For output, the compiled program uses the character's value plus the defined font, color, size and other information (italic, case, etc.) to display the character on the screen. There are fonts available that can display a normally alphanumeric character as a cartoon character. A quick Google search (try "free dingbat fonts") will yield dozens of them.

It is easy to confuse different types of values, like "5", '5', 5.0 and 5, but if you pay attention to the punctuation, it should be clear that the first is a string, the second is a character, the third is a floating-point and the forth is an integer. The reason this distinction is important should become clear soon.

Variables

One of the most powerful features of a programming language is the ability to manipulate variables. A **variable** is a named location that stores a value. Just as there are different types of values (string, floating-point, integer, character, etc.), there are different types of variables.

When you create a new variable, you have to declare what type it is. For example, the string type in C++ is called **string**. The following statement creates a new variable named fred that has type string:

```
string fred;
```

This kind of statement is called a declaration. The type of a variable determines what kind of values it can store. A string variable can contain strings, and it should come as no surprise that int variables can store integers. To create an integer variable, the syntax is:

```
int bob;
```

In C++, there are two floating-point types, called float and double. In this book we will use doubles exclusively. The syntax for a floating-point variable is:

```
double mike;
```

To declare a character variable you suppose to use keyword int:

```
char sam;
```

Where fred, bob, mike and sam are just arbitrary names you made up for the variable to identify it. In general, you will want to make up variable names that indicate what you plan to do with the variable. For example, if you saw these variable declarations:

```
string sFirstName;
char cFirstLetter;
int iHour, iMinute;
double dSeconds;
```

you could probably make a good guess at what values would be stored in them. In this example the first small letter (s, c, i or d) indicate a variable type and the remaining part says something about variable meaning (note: this is only a convention for this book, not a general feature of C++). This example also demonstrates the syntax for declaring multiple variables with the same type: iHour and iMinute are both integers (int type).

You have also seen values like true and false. These are actually boolean values and there is a special type of variables called bool for them. We will learn about this type later.

C++ is generally considered a “loosely typed” language, unlike languages such as Ada which has much stricter type syntax. However, even though general C++ is somewhat forgiving with regard to variable types, and the WCM C++ implementation even more so, if your intent is to progress to general purpose programming, it is very important that you understand the concept and implications of the “type” concept.

Assignment

Now that we have created some variables, we would like to store values in them. We do that with an assignment statement.

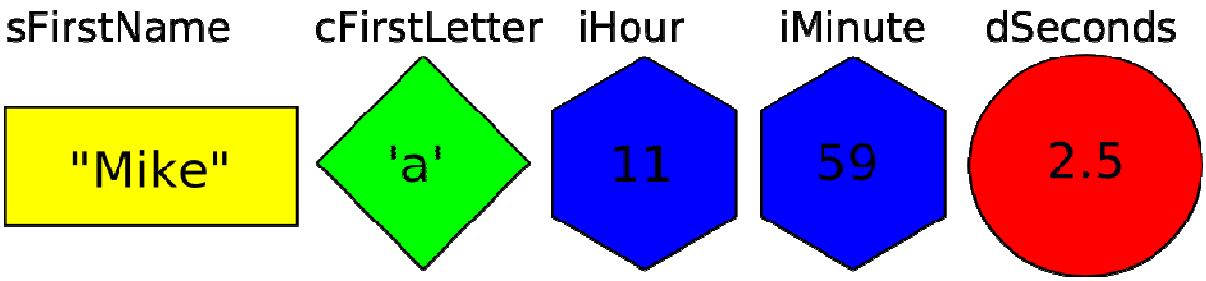
```
sFirstName = "Mike"; // assign string "Mike" to sFirstName
```

```
cFirstLetter = 'a';  // give cFirstLetter the value 'a'
iHour = 11;          // assign the value 11 to iHour
iMinute = 59;        // set iMinute to 59
dSeconds = 2.5;      // give dSeconds the value 2.5
```

This example shows four assignments, and the comments show several different ways people sometimes talk about assignment statements. The vocabulary can be confusing here, but the idea is straightforward:

- When you declare a variable, you create a named storage location.
- When you make an assignment to a variable, you give it a value

A common way to represent variables on paper is to draw a box with the name of the variable on the outside and the value of the variable on the inside. This kind of figure is called a state diagram because is shows what state each of the variables is in (you can think of it as the variable's "state of mind"). This diagram shows the effect of the five assignment statements:



Note: You can also combine these shapes with other flow symbols to actually diagram your program flow, however that will not be discussed in this book.

I sometimes use different shapes and colors to indicate different variable types. These shapes should help remind you that one of the rules in C++:

A variable has to have the same type as the value you assign it. For example, you cannot store a string in an int variable.

The following statement generates a compiler error:

```
int iHour;
iHour = "Hello."; // WRONG !!
```

There is one source of confusion is that some strings look like integers, but they are not. For example, the string "123", which is made up of the characters 1, 2 and 3, is not the same thing as the number 123. This assignment is illegal:

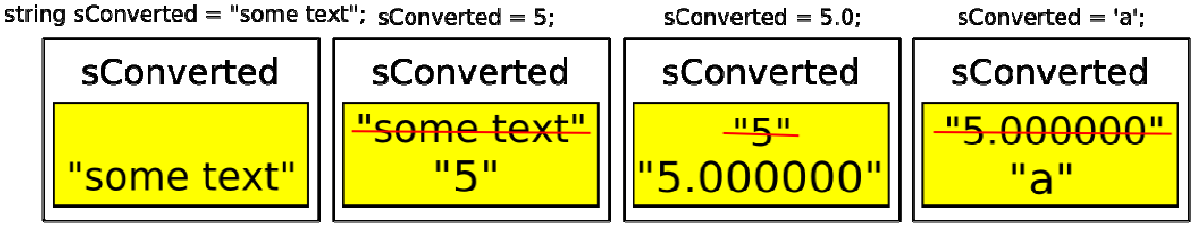
```
iMinute = "59"; // WRONG!
```

Sometimes values can be converted to another type automatically though. For example you can assign all these values to strings and they will be converted automatically. Please keep in mind that this is a WCM C++ extension and you cannot do the same thing in classic C++:

```
string sConverted = "some text";
sConverted = 5;   // sConverted will be "5" after this statement
sConverted = 5.0; // sConverted will be "5.000000" after this statement
sConverted = 'a'; // sConverted will be "a" after this statement
```

You can see from this example that it is possible to assign a value to variable immediately after a declaration: string sConverted = "some text";

You can also see that it is legal in C++ to make more than one assignment to the same variable. The effect of the second assignment is to replace the old value of the variable with a new value. This kind of multiple assignment is the reason I described variables as a container for values. When you assign a value to a variable, you change the contents of the container, as shown in the figure:



There are also other cases when values can be converted to another type automatically. For example integer values can be converted do double and vice versa:

```
double dVal = 5; // dVal will be 5.0 after this statement
int iVal = 5.72; // iVal will be 5 after this statement
```

When you assign a floating-point value to an integer variable the values are rounded down.

Another example is char which can be converted to int

```
int iNumber = 'a';
```

The above statement will assign 97 to iNumber. 97 is the number that is used internally by C++ to represent the letter 'a'. However, it is generally a good idea to treat characters as characters, and integers as integers, and only convert from one to the other if there is a good reason.

Automatic type conversion is an example of a common problem in designing a programming language, which is that there is a conflict between formalism, which is the requirement that formal languages should have simple rules with few exceptions, and convenience, which is the requirement that programming languages be easy to use in practice.

More often than not, convenience wins, which is usually good for expert programmers, who are spared from rigorous but unwieldy formalism, but bad for beginning programmers, who are often baffled by the complexity of the rules and the number of exceptions (fun note: it is also bad for programs, such as aircraft flight controls, that must be extremely reliable since the ambiguities may give problems at very inconvenient times. For such applications, more strictly typed languages are often used). In this book I have tried to simplify things by emphasizing the rules and omitting many of the exceptions.

Not So Fun Note: For an example of an "inconvenient time", see: http://www.youtube.com/watch?v=faB5bIdksi8. Fortunately the pilot walked away unharmed.

Outputting Variables

You can output the value of a string variable using the same commands we used to output string values by using a text object in our cartoon:

```
void Scene1 ()
```

```
{
	string sHello = "Hello";
	Text HelloText ( sText );
	HelloText.SetVisible ( true );
}
```

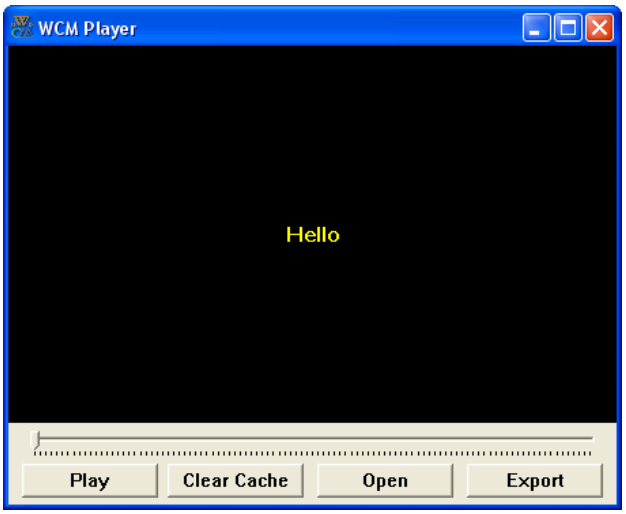
We just need to use a variable name instead of a string value.

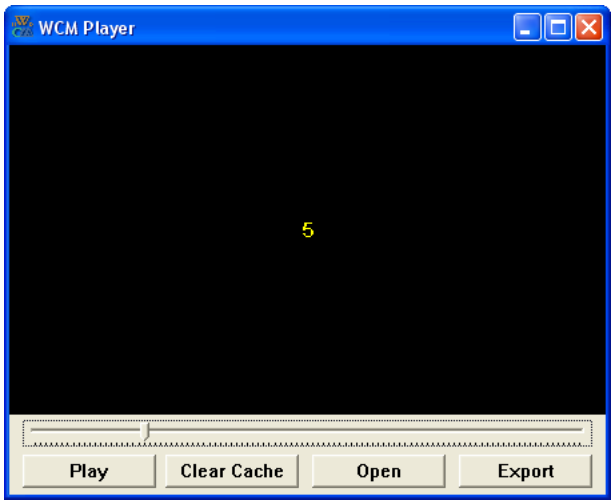
In classic C++ you would not be able to use other kinds of values and variables to create a similar text object, because a text object requires a string value as a parameter. But as I said before, values of other types (int, char and double) can be converted to strings automatically in WCM C++. This means that you can use these values and variables to create a text object or change the text:

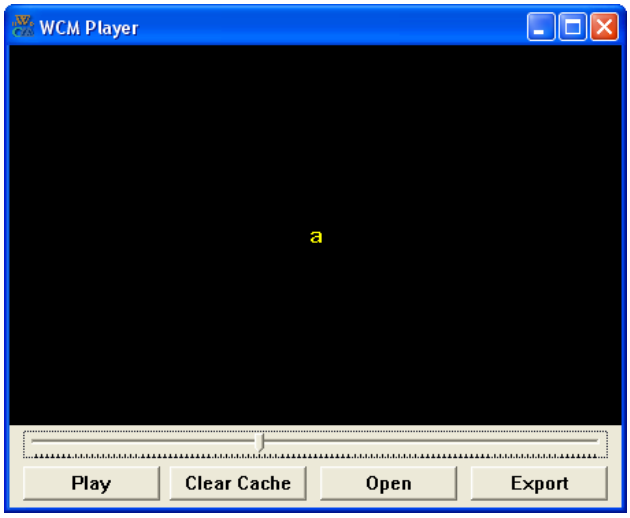
```
void Scene1 ()
{
	Text MyText ( "Hello" ); // create a text object from string value
	MyText.SetVisible ( true );

	SetTime ( 1.0 );
	MyText.SetText ( 5 ); // change the text using int value

	SetTime ( 2.0 );
	MyText.SetText ( 5.72 ); // change the text using double value

	SetTime ( 3.0 );
	MyText.SetText ( 'a' ); // change the text using char value

	string sVal = "Hello";
	int iVal = 5;
	double dVal = 5.72;
	char cVal = 'a';

	SetTime ( 4.0 );
	MyText.SetText ( sVal ); // change the text using string variable

	SetTime ( 5.0 );
	MyText.SetText ( iVal ); // change the text using int variable

	SetTime ( 6.0 );
	MyText.SetText ( dVal ); // change the text using double variable

	SetTime ( 7.0 );
	MyText.SetText ( cVal ); // change the text using char variable

	// wait one more second before the end of cartoon
	SetTime ( 8.0 );
}
```

As you can see from this example, you can create all the objects before using any of them and use only when you need them.

You can try to compile and preview the above program. You will see an 8 seconds long cartoon showing you the following text strings two times (one using values and one using variables):







Operators

Operators are special symbols that are used to represent simple computations like addition and multiplication. Most of the operators in C++ do exactly what you would expect them to do, because they are common mathematical symbols. For example, the operator for adding two integers is +. The following are all legal C++ expressions whose meaning is more or less obvious:

1+1 iHour – 1 iHour * 60 + iMinute iMinute / 60

Another less obvious but interesting operator is %. It is called **modulus operator**. The modulus operator works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second. You can use it the same way as other operators. For example 7 % 3 will yield 1.

Expressions can contain both variables names and integer values. In each case the name of the variable is replaced with its value before the computation is performed. Addition, subtraction and multiplication all do what you expect, but you might be surprised by division. For example, the following program:

```
void Scene1 ()
{
```

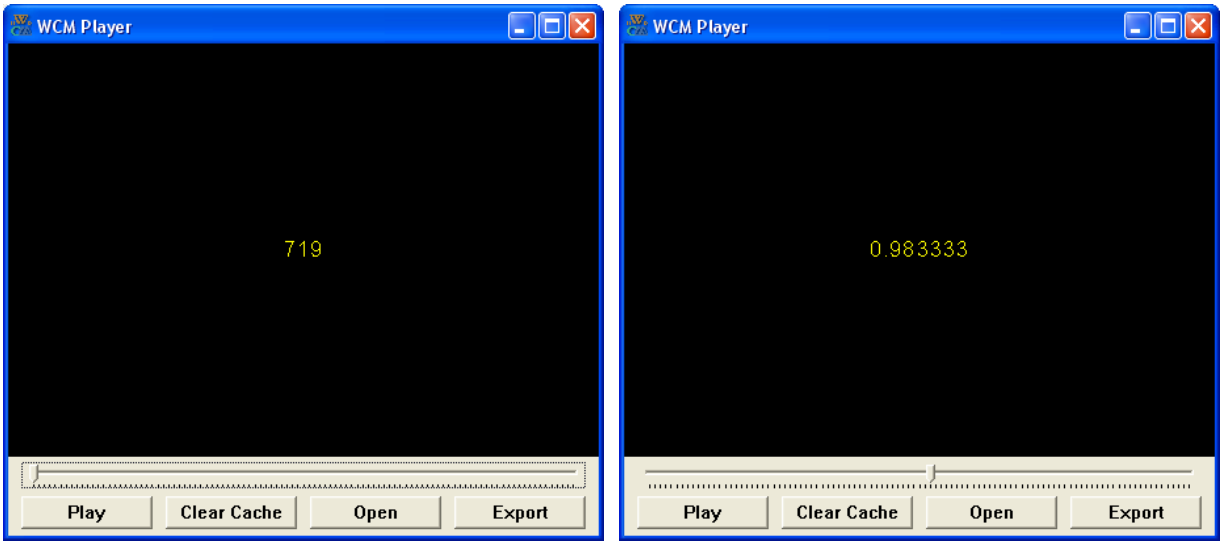
```
        int iHour, iMinute;
        iHour = 11;
        iMinute = 59;

        Text TotalMinutes ( iHour * 60 + iMinute );
        Text HourFraction ( iMinute / 60 );
        TotalMinutes.SetVisible ( true );

        SetTime ( 1.0 );
        TotalMinutes.SetVisible ( false );
        HourFraction.SetVisible ( true );

        SetTime ( 2.0 );
}
```

will show you two numbers in your cartoon: 719 and 0. Stop, but why zero???!!! It should be 0.983333, should not it? The reason for the discrepancy is that C++ is performing integer division. When both operands are integers the result is also integer and it is rounded down. It is here that the modulus (remainder) operator comes in. 59 % 60 is 59. You're basically back to first grade arithmetic (before you were introduced to the decimal points, i.e. "59 divided by 60 is 0 with a remainder of 59).

Note: Historically, integer arithmetic was used for thousands of years before the decimal point was introduced. Simple things like 3/2 = 1 ½ work fine – but for larger or much smaller numbers, things get messy.

If you want to perform a floating-point division then at least one operand must be a floating-point. For example you could use 60.0 or a double variable. The following program will work just fine:

```
void Scene1 ()
{
        int iHour, iMinute;
        iHour = 11;
        iMinute = 59;

        Text TotalMinutes ( iHour * 60 + iMinute );
        Text HourFraction ( iMinute / 60.0 );

        TotalMinutes.SetVisible ( true );
        SetTime ( 1.0 );
        TotalMinutes.SetVisible ( false );
        HourFraction.SetVisible ( true );
        SetTime ( 2.0 );
}
```



In C++ these mathematical operators work the same way for int and double values and variables. If both operands are int then the result is also int. If at least one operand is double then the result is double as well.

Order of Operations

When more than one operator appears in an expression the order of evaluation depends on the rules of precedence. A complete explanation of precedence can get complicated, but just to get you started:

1. Multiplication and division happen before addition and subtraction. So 2*3-1 yields 5, not 4, and 2/3-1 yields -1, not 1 (remember that in integer division 2/3 is 0).
2. If the operators have the same precedence they are evaluated from left to right. So in the expression iMinute*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had gone from right to left, the result would be 59*1 which is 59, which is wrong.
3. Any time you want to override the rules of precedence (or you are not sure what they are) you can use parentheses. Expressions in parentheses are evaluated first, so 2 * (3-1) is 4. You can also use parentheses to make an expression easier to read, as in (iMinute * 100) / 60, even though it doesn't change the result.

Note: Incorrect understanding (usually accidental) of operator precedence is another major cause of logic errors in a program. If there is a chance of misunderstanding, it is a good idea to use parentheses.

Operators for Characters

Just a reminder, we are still not talking about cartoon characters. We are talking about symbols or letters which make up strings, i.e., the char variable type.

Interestingly, the same mathematical operations that work on integers also work on characters. For example:

```
char cLetter;
cLetter = 'a';     // cLetter has 'a' value now
cLetter = 'a' + 1; // cLetter has 'b' value now
```

The last line gives a result that uses the fact that the character is stored as an integer even though it is of type char. The standard (ASCII) value for ‘a’ is 97 and for ‘b’ is 98. Similarly ‘a’ – 3 will give ‘^’.

Although it is syntactically legal to multiply characters, it is almost never useful to do it.

Operators for Strings

As I've said before WCM C++ has some extensions for more convenient work with strings. In pure C++ strings are just arrays of characters. In WCM C++ advanced users can also work with arrays of characters but they are converted to strings immediately after most of the operations. For example you can use operator + to concatenate (or merge) two strings together. Please keep in mind that the example below is impossible in pure C++:

```
void Scene1 ()
{
	string sFirstName = "Bill";
	string sLastName = "Gates";

	Text Name ( "His name is: " + sFirstName + " " + sLastName );
	Name.SetVisible ( true );

	// we do not necessary need to have the last statement
	// here, because if the cartoon length is zero it is
	// automatically extended to one second
	SetTime ( 1.0 );
}
```

It is also legal in WCM C++ to use operator + between strings and integers, doubles, and characters. But please keep in mind that while these operations are also enabled in classic C++, only in WCM C++ you can use them to concatenate strings. In classic C++ they mean different thing. Here is an example of how you can concatenate string with other values and variables:

```
void Scene1 ()
{
	int iHour = 11;
	int iMinute = 59;

	string sTime = iHour + ":" + iMinute + ':' + 2.5;

	Text Time ( sTime );
	Time.SetVisible ( true );

	SetTime ( 1.0 );
}
```

Please keep in mind that the order of operations does matter. In our example we first concatenate iHour and ":". As a result we receive a string and then concatenate it with iMinute and so on. But if we use it like this:

```
	string sTime = iHour + ':' + iMinute + ":" + 2.5;
```

then the first operation happens between iHour and ':'. And ':' is not the same thing as ":". This is a character. As a result of this operation we will receive another character as I mentioned in previous section of this book. Operator + can be used in WCM C++ for concatenation only when at least one operand is a strings. For example this code:

```
string sNumbers = 1 + 2 + "3";
```

will assign "33" to sNumbers, because 1 + 2 is an operation between integers and the result of this operation is 3. But the code below:

```
string sNumbers = 1 + ( 2 + "3" );
```

will assign "123" to sNumbers, because there is at least one string operand in every operation.

Functions

Math Functions

WCM C++ supports some of the classic C++ math functions. However it is recommended to use function from WCM C++'s own extended library.

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like sin(π/2) and log(1/x). First, you evaluate the expression in parentheses, which is called the argument of the function. For example, π /2 is approximately 1.571, and 1/x is 0.1 (if x happens to be 10).

Then you can evaluate the function itself, either by looking it up in a table or by performing various computations. The sin of 1.571 is 1, and the log of 0.1 is -1 (assuming that log indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like log(1/sin(π /2)). First we evaluate the argument of the innermost function, then evaluate the function, and so on.

WCM C++ provides a set of built-in functions that includes most of the mathematical operation. The math functions are invoked using a syntax that is similar to mathematical notation with exception that for your convenience all the angles must be specified it degrees (not radians as in mathematics and classic C++). Here are the examples of using mathematical functions available in WCM C++:

```
double dRes;

// assign dRes sine of 60 degrees which is 0.866025
dRes = Sin ( 60 );
// assign dRes cosine of 60 degrees which is 0.500000
dRes = Cos ( 60 );
// outputs tangent of 60 degrees which is 1.732051
dRes = Tan ( 60 );
// assign dRes arcsine of 0.866025 which is 59.999954 degrees
dRes = ASin ( 0.866025 );
// assign dRes arccosine of 0.500000 which is 60.000001 degrees
dRes = ACos ( 0.500000 );
// assign dRes arctangent of 1.732051 which is 60.000003 degrees
dRes = ATan ( 1.732051 );
// assign dRes 2 raised to the power 3 which is 8
dRes = Pow ( 2, 3 );
// assign dRes e or 2.7182 raised to the power 3 which is 20.085537
dRes = Exp ( 3 );
// assign dRes the natural logarithm of 2.7182 which is 0.999970
dRes = Log ( 2.7182 );
// assign dRes square root of 4 which is 2.000000
dRes = Sqrt ( 4 );
// assign dRes a random value between 0 and 1 (not including 1)
```

```
dRes = Rand ();
```

As you can see from this example, some functions may have multiple parameters and some might not have any. This will be explained later.

Notes and Cautions: These functions are also examples of "loose typing". Even though the functions are defined in the library to use a double argument and return a double result, this is only loosely enforced. This will generally be fine, but care should be used with this approach. Also keep in mind that even if you are using function Rand() then our compiled cartoon will not be random once it is created. But different compilations of the same cartoon will be.

String Functions

You already know that sometimes char, int and double values and variable can be automatically converted to strings. But sometimes you need to convert them manually. For example, if you want to receive a concatenated string out of 2 integers, you cannot just use operator +, because this will be an operation between integers. You must convert them to strings first. You should use one of these functions to convert them manually using functions IntToString, CharToString and DoubleToString:

```
string sRes;

// the following statement will assign sRes "1+2.500000=3.5";
sRes = IntToString ( 1 ) +
       CharToString ( '+' ) +
       DoubleToString ( 2.5 ) +
       "=3.5";
```

You can also see from this example that it is legal in C++ to carry over a sentence to the next line. You just need to terminate it with semicolon.

Sometimes you may also want to convert a string value to int or double. Please keep in mind that not every string can be converted. For example string "one" cannot be converted to int or double. While it may sound like a number, it is not. Only strings which look like a number inside double quotes, like "123" or "0.28", can be converted. You suppose to use functions StringToInt and StringToDouble to do this:

```
int iInt = StringToInt ( "123" );           // assign 123 to iInt
double dDouble = StringToDouble ( "12.3" ); // assign 12.3 to dDouble
```

If there is no conversion possible, then these functions return zero.

There are a couple of other functions to work with strings. These are StrLen, UpperCase and LowerCase. You can probably guess what they do by their name and example below:

```
// assign 4 to iLen because string "test" has 4 characters in it
int iLen = StrLen ( "test" );

// assign "TEST" to sUpperCased
string sUpperCased = UpperCase ( "test" );

// assign "test" to sLowerCased
string sLowerCased = LowedCase ( "TEST" );
```

System Time Functions 

There are several functions available in WCM C++ to retrieve the current system time. These functions have nothing to do with time in your cartoon. They were implemented mostly for illustration purposes. You will find functions to deal with time in your cartoon in the next section.

```
// retrieve time in seconds since 00:00:00 UTC, January 1, 1970
int iTime = GetUTCTime ()

// retrieve current UTC hour, you must pass
// a value received from GetUTCTime
int iHour = GetUTCHour ( iTime );

// retrieve current UTC minute, you must pass
// a value received from GetUTCTime
int iMinute = GetUTCMinute ( iTime );

// retrieve current UTC second, you must pass
// a value received from GetUTCTime
int iSecond = GetUTCSecond ( iTime );
```

It is important to realize that these functions are evaluated at the time the program is compiled. Once the cartoon is compiled, the time values are fixed to the time that the cartoon was compiled.

Cartoon Time Functions

You already know one of these functions: SetTime. It does not return anything but changes a current scene time. Here is the full list of functions available to change and retrieve a current scene time: SetTime, GetTime and Sleep. In our first examples it was clear what the current scene time was at every moment. But in more complex examples it might not be that clear. And you might want to retrieve a current scene time using GetTime. Function Sleep changes the current scene time by a given amount of seconds. For example if you use it as Sleep (1.5) it is equal to SetTime (GetTime () + 1.5). Here is an example below

```
void Scene1 ()
{
      SetTime ( 1.0 );
      Sleep ( 1.5 );
      Text TimeInScene ( "Scene Time is: " +  GetTime () + " seconds");
      TimeInScene.SetVisible ( true );
      Sleep ( 1.0 );
}
```

This example will display a black screen for 2.5 seconds (1.0 + 1.5) and then the following text: "Scene Time is: 2.500000 seconds";

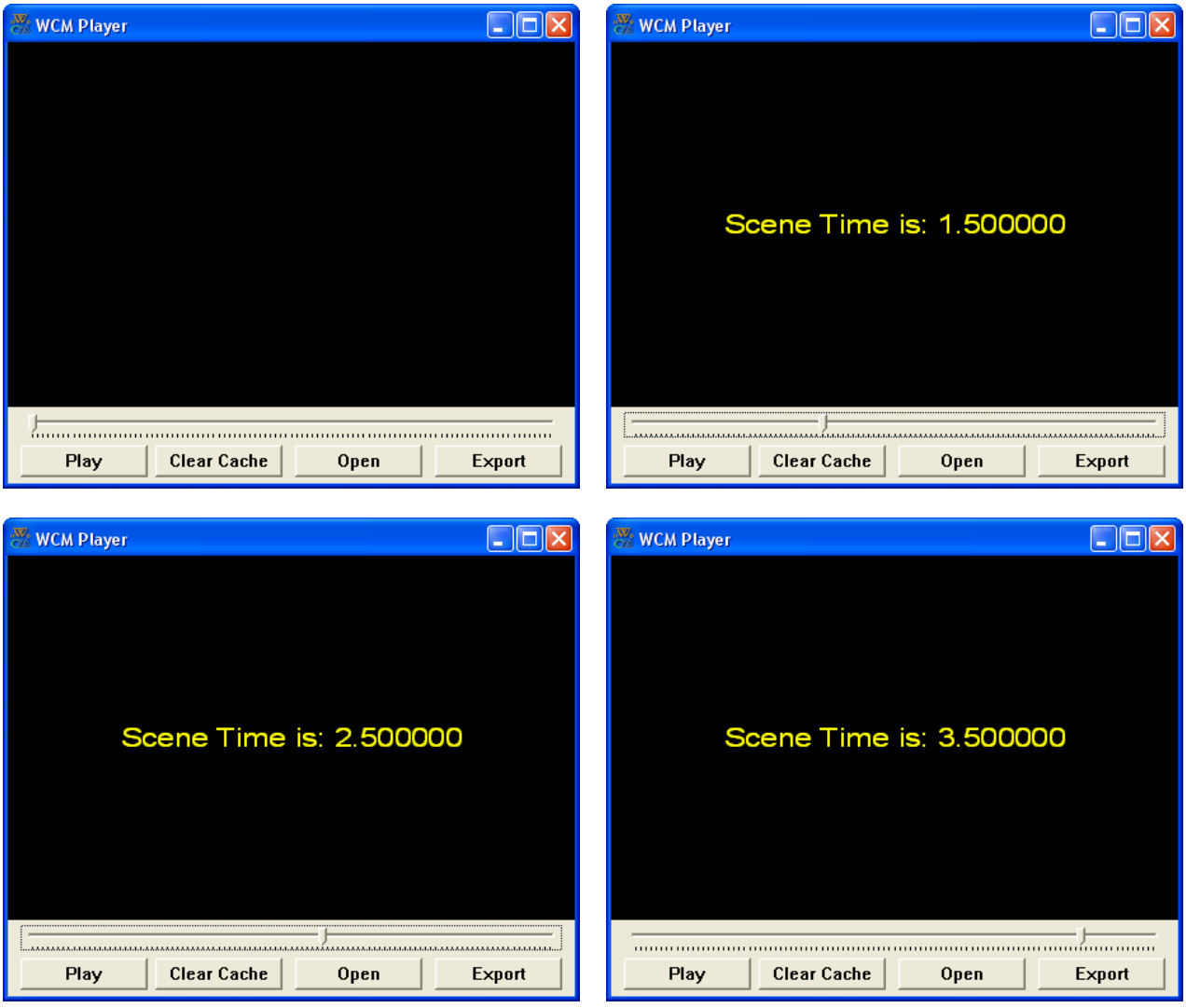
There is another function in WCM C++, which is not exactly a "cartoon time function". You probably noticed that it is not very convenient to create a text object every time you want to display a text. This is mostly because we work with simple programs and outputting test mostly for debugging purposes. When you are making a real cartoon, you will want to do different things with your text object, like rotate and move it, and you will find that having an object is a very convenient thing. But it is still not convenient for debugging.

Thus, there is a special function which displays a text for debugging in WCM C++ called ShowText. It automatically creates a text object, makes it visible, waits for one second and then makes it invisible. The last three lines of our example above could be rewritten with just one

sentence using ShowText. Please keep in mind, that beside displaying text, this function also changes a current scene time by one second similar to Sleep (1). Here is another example:

```
void Scene1 ()
{
      SetTime ( 1.5 );
      ShowText ( "Scene Time is: " +  GetTime () );
      ShowText ( "Scene Time is: " +  GetTime () );
      ShowText ( "Scene Time is: " +  GetTime () );
}
```

This example will display a black screen for 1.5 seconds and then "Scene Time is: 1.500000 seconds" for one second, then "Scene Time is: 2.500000 seconds" for another second and "Scene Time is: 3.500000 seconds" for one more second:



Finally there is a pair of macros, THIS_TIME and SAME_TIME that are essentially equivalent to GetTime and SetTime but were defined aid in synchronizing multiple actions to a particular cartoon time "tick". The best way to understand these macros is to examine and run the "Synchronization of Actions" tutorial script. An example of their use will also be presented in a later section

Composition

Just as with mathematical functions, C++ functions can be composed, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function. You can also take the result of one function and pass it as an argument to another:

```
double dX = Cos ( pAngle + 180.0 / 4 );
double dY = Exp ( Log  ( 10.0 ) );
string sLen = IntToString ( StrLen ( "test" ) + 2 );
```

Adding New Functions

So far we have only been using the functions that are built into C++, but it is also possible to add new functions. Actually, we have already seen some function definitions: Scene1, Scene2 etc. These functions are special because they indicate where the code for every cartoon scene is located, but the syntax for them is the same as for any other function definition:

```
void NAME ( LIST OF PARAMETERS )
{
      STATEMENTS
}
```

You can make up any name you want for your function, except that you can't call it the same name as any other C++ keyword. The list of parameters specifies what information, if any, you have to provide in order to use (or call) the new function.

Scene1 doesn't take any parameters, as indicated by the empty parentheses () in its definition. The first couple of functions we are going to write also have no parameters, so the syntax looks like this:

```
void ShowTime ()
{
      ShowText ( "The System Time is" );

      int iTime = GetUTCTime ();
      string sHour = IntToString ( GetUTCHour ( iTime ) );
      string sMinute = IntToString ( GetUTCMinute ( iTime ) );

      ShowText ( sHour + ":" + sMinute );
}
```

This function is named ShowTime; it contains several statements. It displays a text "The System Time is" for one second and then system time for one more second. In our scene we can call this new function using syntax that is similar to the way we call the built-in C++ commands:

```
void Scene1 ()
{
      ShowTime ();
}
```

The program will output a system time to your cartoon. But what if we want to do this several times? We can call ShowTime several times from Scene1 then:

```
void Scene1 ()
{
      ShowTime ();
      ShowTime ();
      ShowTime ();
}
```

Or we can write another function which invokes ShowTime several times and call it from Scene1:

```
void ShowTimeThreeTimes ()
{
	ShowTime ();
	ShowTime ();
	ShowTime ();
}

void Scene1 ()
{
	ShowTimeThreeTimes ();
}
```

You should notice a few things about this program:

1. You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.
2. You can have one function call another function. In this case, Scene1 calls ShowTimeThreeTimes and ShowTimeThreeTimes calls ShowTime. Again, this is common and useful

So far, it may not be clear why it is worth the trouble to create all these new functions. Actually, there are a lot of reasons, but this example only demonstrates two:

1. Creating a new function gives you an opportunity to give a name to a group of statements. Functions can simplify a program by hiding a complex computation behind a single command, and by using English words in place of arcane code.
2. Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print a time nine consecutive times is to call ShowTimeThreeTimes three times. How would you print it 27 times?

Definition and Uses

Pulling together all the code fragments from the previous section, the whole program looks like this:

```
void ShowTime ()
{
	ShowText ( "The System Time is:" );

	int iTime = GetUTCTime ();

	// integer values will be converted
	// to strings automatically
	string sHour = GetUTCHour ( iTime );
	string sMinute = GetUTCMinute ( iTime );

	ShowText ( sHour + ":" + sMinute );
}

void ShowTimeThreeTimes ()
{
	ShowTime ();
	ShowTime ();
	ShowTime ();
```

```
}

void Scene1 ()
{
      ShowTimeThreeTimes ();
}
```

This program contains three function definitions: ShowTime, ShowTimeThreeTimes and Scene1.

Inside the definition of Scene1, there is a statement that uses or calls ShowTimeThreeTimes. Similarly, ShowTimeThreeTimes calls ShowTime three times. Notice that the definition of each function appears above the place where it is used.

This is necessary in C++; the definition of a function must appear before (above) the first use of the function. You could try running this program with the functions in a different order and see what error messages you get.

Programs with Multiple Functions

When you look at a program that contains several functions, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the order of execution of the program.

Execution always begins at the first statement of Scene1, regardless of where it is in the program (sometimes it is at the bottom). Statements are executed one at a time, in order, until you reach a function call. Function calls are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the called function, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one function can call another. Thus, while we are in the middle of main, we might have to go off and execute the statements in ShowTimeThreeTimes. But while we are executing ShowTimeThreeTimes, we get interrupted three times to go off and execute ShowTime.

Fortunately, C++ is adept at keeping track of where it is, so each time ShowTime completes, the program picks up where it left off in ShowTimeThreeTimes, and eventually gets back to Scene1.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

Parameters and Arguments

Some of the built-in functions we have used have parameters, which are values that you provide to let the function do its job. For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, Sin takes a double value as a parameter.

Some functions take more than one parameter, like Pow, which takes two doubles, the base and the exponent

Notice that in each of these cases we have to specify not only how many parameters there are, but also what type they are. So it shouldn't surprise you that when you write a function definition, the parameter list indicates the type of each parameter. For example

```
void ShowTextTwice ( string sMessage )
{
      ShowText ( sMessage );
      ShowText ( "" );
      ShowText ( sMessage );
      ShowText ( "" );
```

```
}
```

This function takes a single parameter, named sMessage, which has type string. Whatever that parameter is (and at this point we have no idea what it is), it gets displayed twice (it is displayed once for a second, then empty string is displayed for a second, then the text is displayed for one more second and empty string is displayed for one more second).

In order to call this function, we have to provide a string. For example, we might have a Scene function like this:

```
void Scene1 ()
{
      ShowTextTwice ( "WARNING" );
}
```

The string value you provide is called an argument, and we say that the argument is passed to the function. In this case the value "WARNING" is passed as an argument to ShowTextTwice where it will get displayed twice.

Alternatively, if we had a string variable, we could use it as an argument instead:

```
void Scene1 ()
{
      string sWarning = "WARNING";
      ShowTextTwice ( sWarning );
}
```

Notice something very important here: the name of the variable we pass as an argument (sWarning) has nothing to do with the name of the parameter (sMessage).

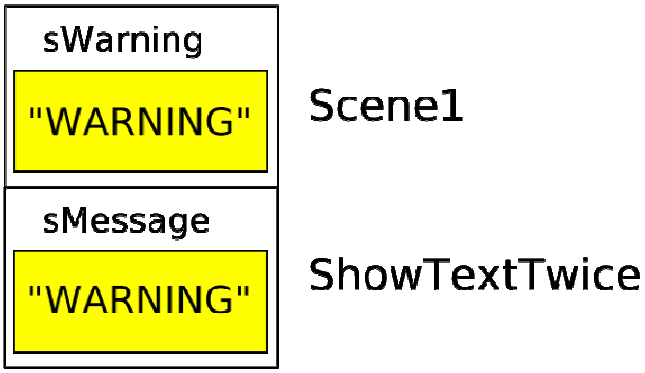
Let me say that again: The name of the variable we pass as an argument has nothing to do with the name of the parameter! They can be the same or they can be different, but it is important to realize that they are not the same thing, except that they happen to have the same value (in this case the string "WARNING").

The value you provide as an argument must have the same type as the parameter of the function you call. This rule is important, but it is sometimes confusing because C++ sometimes converts arguments from one type to another automatically. For example you can use an integer parameter instead of a string one because it can be converted to string automatically in WCM C++. But for now you should learn the general rule, and we will deal with exceptions later.

Function Parameters and Variables are Local

Function parameters and variables only exist inside their own functions. Within the confines of Scene1, there is no such thing as sMessage. If you try to use it, the compiler will complain. Similarly, inside ShowTextTwice there is no such thing as sWarning. Variables like this are said to be **local**. In order to keep track of parameters and local variables, it is useful to draw a stack diagram. Like state diagrams, stack diagrams show the value of each variable, but the variables are contained in larger boxes that indicate which function they belong to.

For example, the state diagram for ShowTextTwice looks like this:



Whenever a function is called, it creates a new instance of that function. Each instance of a function contains the parameters and local variables for that function. In the diagram an instance of a function is represented by a box with the name of the function on the outside and the variables and parameters inside.

In the example, Scene1 has one local variable, sWarning, and no parameters. ShowTextTwice has no local variables and one parameter, named sMessage.

Functions with Multiple Parameters

The syntax for declaring and invoking functions with multiple parameters is a common source of errors. First, remember that you have to declare the type of every parameter. For example:

```
void PrintTime ( int iHour, int iMinute )
{
      // these will be concatenated just fine
      // because there is at least one string
      // in every + operation
      ShowText ( iHour + ":" + iMinute );
}
```

It might be tempting to write (int iHour, iMinute), but that format is only legal for variable declarations, not for parameters.

Another common source of confusion is that you do not have to declare the types of arguments when calling the function. The following is wrong!

```
int iHour = 11;
int iMinute = 59;
PrintTime (int iHour, int iMinute); // WRONG!
```

In this case, the compiler can tell the type of hour and minute by looking at their declarations. It is unnecessary and illegal to include the type when you pass them as arguments. The correct syntax is:

```
int iHour = 11;
int iMinute = 59;
PrintTime ( iHour, iMinute);
```

Functions with Results

So far we have only discussed functions preceded by the key word void, which means the function does not return a value to the calling program. In WCM, this is the most common case, however in general there are many functions which return a value to the calling program.

You might have noticed by now that some of the functions we are using, like the math functions, yield results. Other functions, like ShowTextTwice, perform an action but don't return a value. That raises some questions:

1. What happens if you call a function and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
2. What happens if you use a function without a result as part of an expression, like ShowTextTwice () + 7?
3. Can we write functions that yield results, or are we stuck with things like ShowTextTwice?

The answer to the third question is "yes, you can write functions that return values," and we'll do it in a couple of chapters. I will leave it up to you to answer the other two questions by trying them out. Any time you have a question about what is legal or illegal in C++, a good way to find out is to ask the compiler.

Note: Though beyond the scope of this book, the first question can be important in general C++ programming. All C++ programs must have a main program which usually has the form:

```
void main ()
{
      …
}
```

However in some cases, the main program may receive or supply information to the computer's operating system (OS). Thus, if this feature is to be used, the programmer must not only understand C++ but have a basic understanding of the OS as well. Fortunately for the novice, this is a relatively rare occurrence but you should be aware the possibility exists.

Making Real Cartoons

Now we know enough about C++ to start making first simple but real cartoons with moveable text and image objects. There will be later sections that discuss additional features of C++, but the following paragraphs just focus on actually making cartoons. Are you ready?

Image Objects

You already have learned about text objects. Beside text you can use images in your cartoons. You can add an image object to a scene almost the same way as text one.

```
void Scene1 ()
{
      Image MyImg ( "http://www.webcartoonmaker.com/wcm_icon.png" );
      MyImg.SetVisible ( true );

      // We do not use SetTime. The lehgth of this cartoon
      // will be set to one second automatically instead of zero
}
```

MyImg is and unique name for the image object. And we will use it to make it visible or invisible, move it, rotate and do other cool stuff. You can see that it also requires a parameter. But instead of a string of text to display, you should supply a valid URL of image. If URL is not valid,

then no error will be reported but no image will be displayed either. Web Cartoon Maker supports images in SVG, EMF, PNG, JPG and GIF formats.

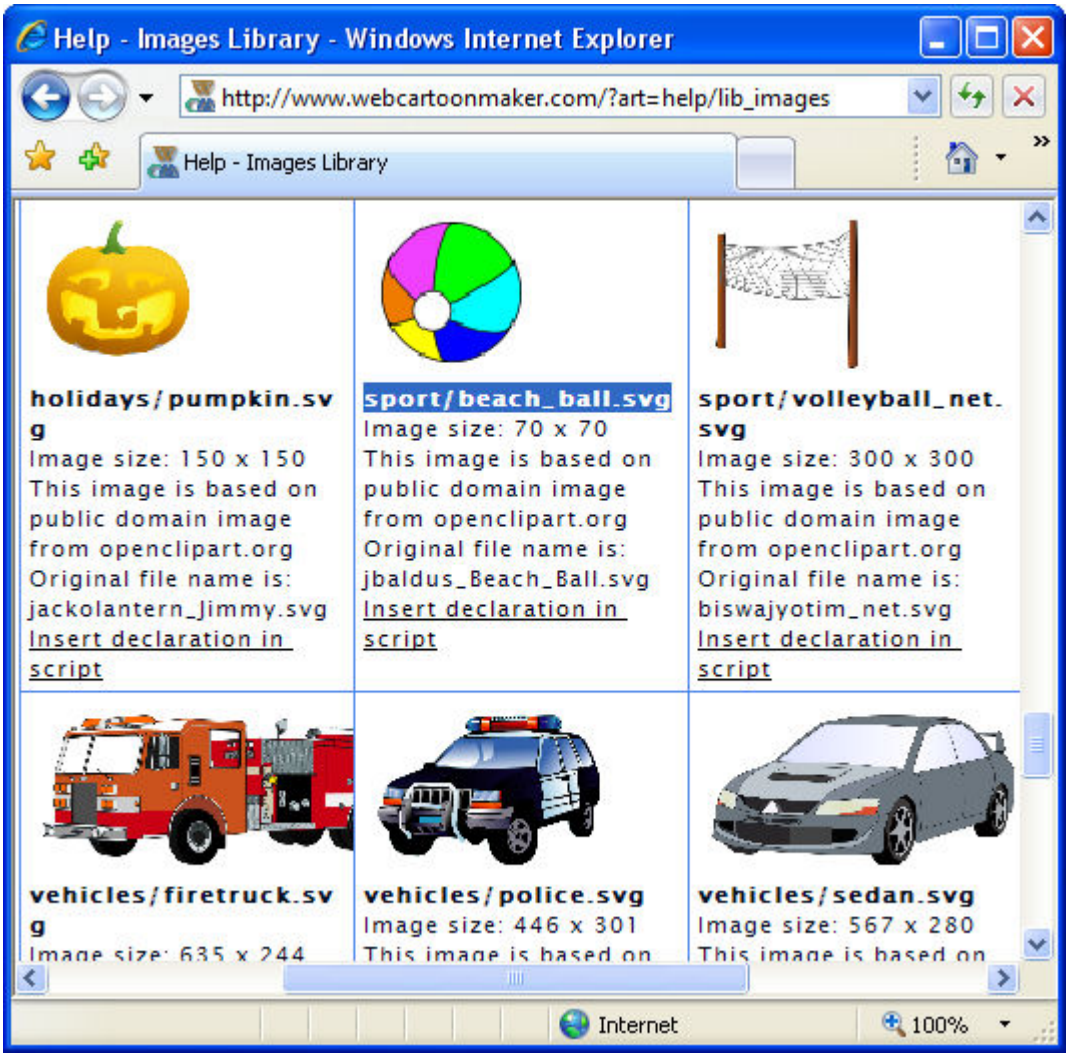
Note: If you haven't read the tutorial and scripting help files about images, this would be a good time to read them, as well as to run the "Image objects in Web Cartoon Maker" tutorial script. A brief discussion of vector (SVG, EMF) vs. raster (PNG, JPG, GIF) formats will be presented in a later section. There are also many good discussions of this subject on the web.

You can also use images from your hard drive. In this case you must indicate a full local path to an image file instead of URL. And remember, in C++ you need to use double slashes ("\\") instead of single ones ("\"), because single slashes are reserved for defining special symbols like "\n" or "\t":

```
void Scene1 ()
{
      Image LocalImg ( "c:\\myimage.png" );
      LocalImg.SetVisible ( true );
}
```

Again, this must be a valid full local path to an image. And an image must exist on your hard drive. Most likely there is no image at located at "c:\\myimage.png" on your hard drive. Please replace this string with a valid path to an image in SVG, EMF, PNG, JPG or GIF format if you want to compile this example.

Web Cartoon Maker has its own online library of images optimized for cartoons. You can browse the library in your web browser (Internet Explorer 7.0 or later is recommended) at http://www.webcartoonmaker.com/?art=help/lib_images:



You can see that all the pictures have a shorter path assigned to them like "sport/beach_ball.svg". You can use this short path instead of fully qualified URL if you want to use an image from online Web Cartoon Maker's library. For example if you want to display a ball you could write:

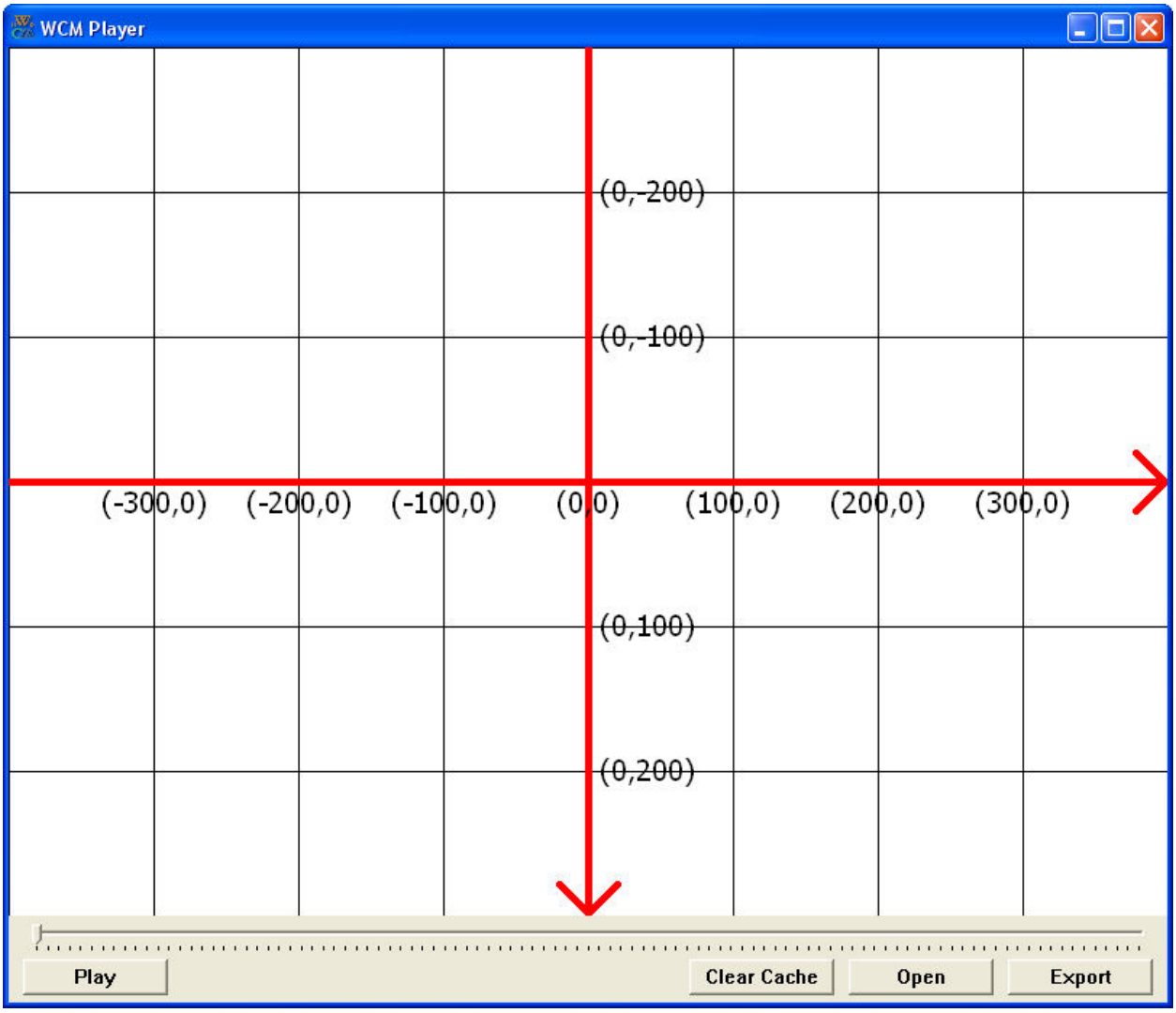
```
void Scene1 ()
{
	Image LibImg ( "sport/beach_ball.svg" );
	LocalImg.SetVisible ( true );
}
```

Coordinates

When we created our first cartoons we displayed text or images in the center of them. But we can place them anywhere we want to. We just need to learn about coordinates in Web Cartoon Maker. To help you with coordinates we have a special image in our library "wcm/coordinates.emf". Please try to compile the following program:

```
void Scene1 ()
{
	Image Coordinates ( "wcm/coordinates.emf" );
	Coordinates.SetVisible ( true );
}
```

You will see a static one second long cartoon displaying the coordinate's grid:



As you can see the default coordinates of the cartoon's center are (0,0). The coordinates of the right bottom corner are (400,300) and the coordinates of the left upper corner are (-400,-300). Every object can be placed anywhere in the scene by using these coordinates and SetPos method. The default coordinates for any object are (0,0). The origin of a text or an image object is located in its center and when you use SetPos method you specify the coordinates of its origin. If you specify coordinates outside of the visible grid, then an object may become partially or fully invisible. This is like driving a car away from the camera's view in a real movie.

Hint: When making a cartoon, don't confine your thinking to just the area currently seen by the camera. As the above paragraph indicates, it can be effective to move things in and out of the camera's view area and, as a later section will discuss, the camera can also be moved.

Let's use the coordinate's grid image as a background and add another ball image at the bottom:

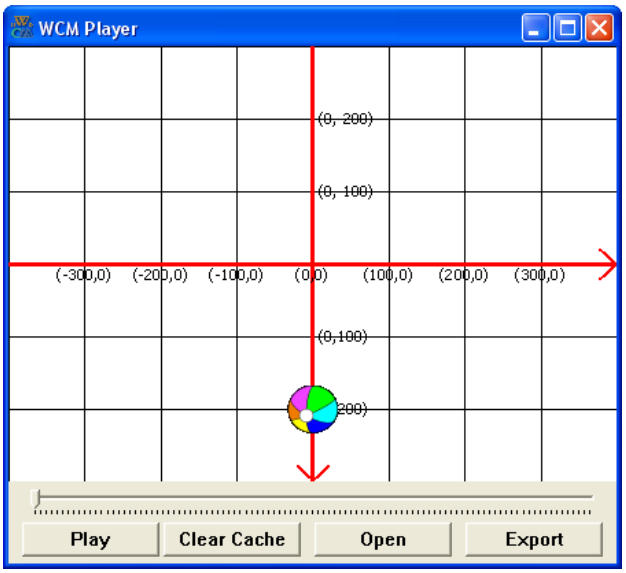
```
void Scene1 ()
{
      // setup background
      Image Coordinates ( "wcm/coordinates.emf" );
      Coordinates.SetVisible ( true );

      // add a ball image at (0,200)
      Image Ball ( "sport/beach_ball.svg" );
```

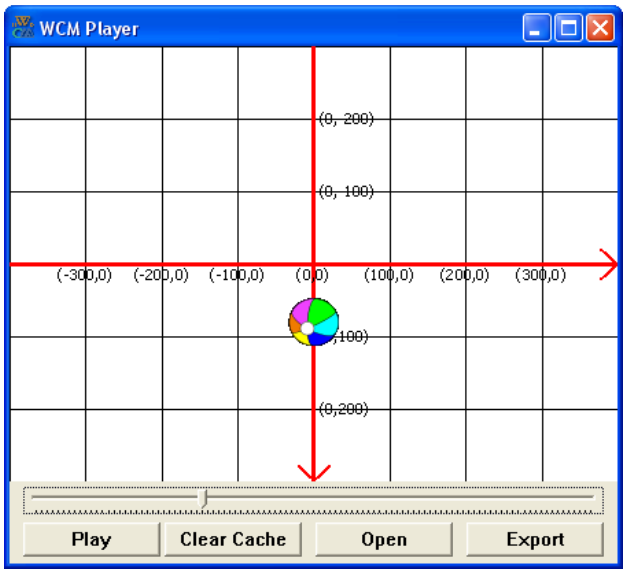
```
        Ball.SetVisible ( true );
        Ball.SetPos ( 0, 200 );
}
```

You can also change the ball coordinates. If you place another instruction like Ball.SetPos (0, -200); immediately after Ball.SetPos (0, 200); then you will not even see the ball at (0,200) because the second instruction is executed at the same time in your cartoon. But if you change the current cartoon time using SetTime or Sleep... You will get the ball moving!

```
void Scene1 ()
{
        // setup background
        Image Coordinates ( "wcm/coordinates.emf" );
        Coordinates.SetVisible ( true );

        // add a ball image at (0,200)
        Image Ball ( "sport/beach_ball.svg" );
        Ball.SetVisible ( true );
        Ball.SetPos ( 0, 200 );

        // move it to (0,-200)
        Sleep ( 2 );
        Ball.SetPos ( 0, -200 );
}
```

Here is what you should see after compilation of the program. You first real animated cartoon with a ball moving from bottom to the top:

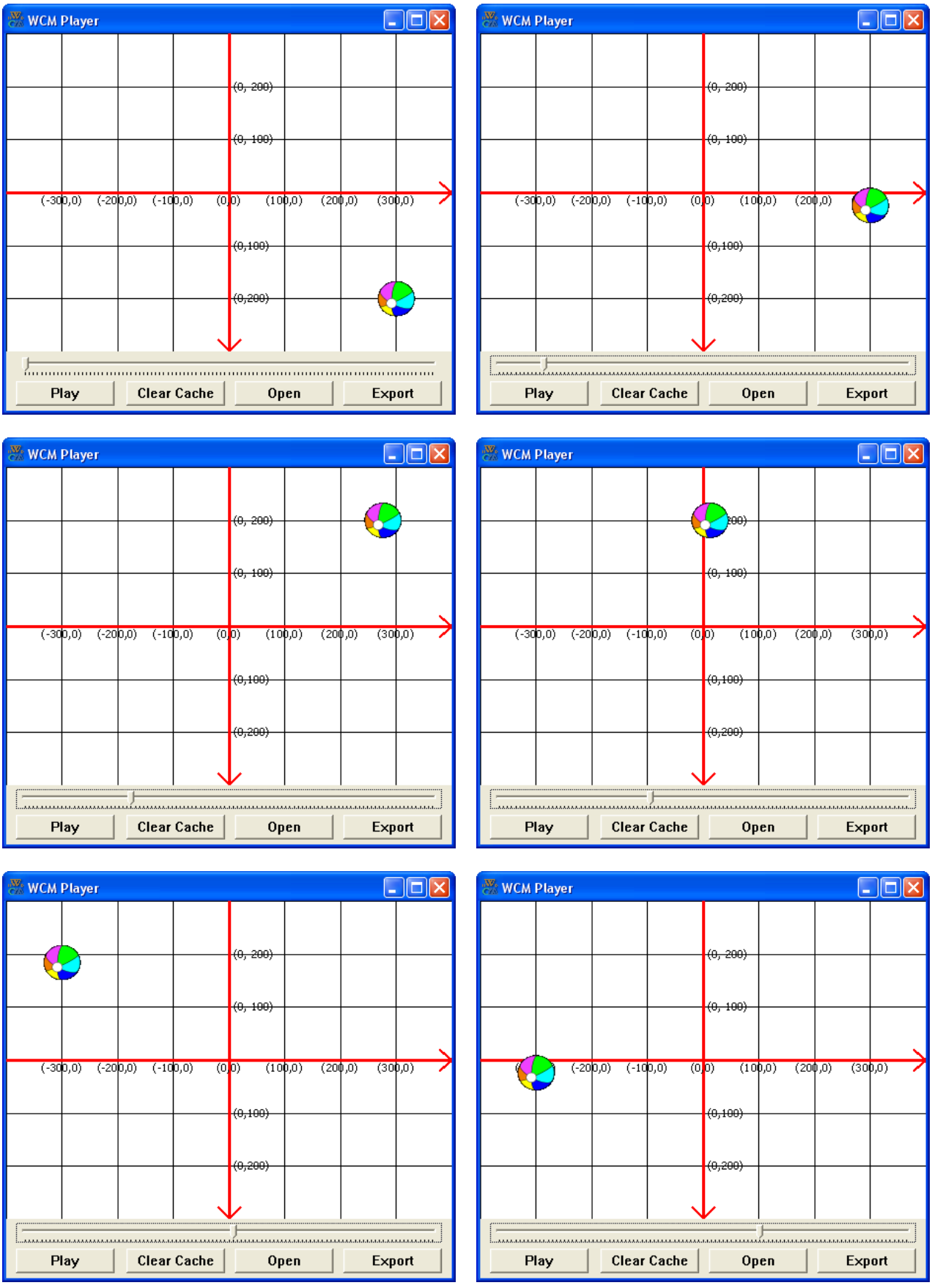




You probably noticed that the ball was not moved immediately after seeing second SetPos instruction. It was moved smoothly during 2 seconds Sleep command between 2 instances of SetPos. Using SetPos is just like adding a control point at a specific time (also often called a "time tick") in you cartoon and the position is changing smoothly from one control point to another. For example, if you want a ball to move through the cartoon edges, you can create a script like this one:

```
void Scene1 ()
{
	// setup background
	Image Coordinates ( "wcm/coordinates.emf" );
	Coordinates.SetVisible ( true );

	// add a ball image at (0,200)
	Image Ball ( "sport/beach_ball.svg" );
	Ball.SetVisible ( true );

	// add first control point
	SetTime ( 0.0 ); // start time, you can omit this one
	Ball.SetPos ( 300, 200 ); // right bottom corner

	// another control point
	SetTime ( 1.0 ); // one second after beginning
	Ball.SetPos ( 300, -200 ); // right upper corner

	// another control point
	SetTime ( 2.0 ); // two seconds after beginning
	Ball.SetPos ( -300, -200 ); // left upper corner

	// another control point
	SetTime ( 3.0 ); // three seconds after beginning
	Ball.SetPos ( -300, 200 ); // left bottom corner

	// another control point
	SetTime ( 4.0 ); // four seconds after beginning
	Ball.SetPos ( 300, 200 ); // right bottom corner again
}
```

If you compile this cartoon you will see something like this:


WCM Player
(0, 200)
(0, 100)
(-300,0)
(-200,0)
(-100,0)
(0,0)
(100,0)
(200,0)
(300,0)
(0,100)
(0,200)
Play
Clear Cache
Open
Export

Methods to Work with Image and Text Objects

"Set" Methods

You already know two methods to work with Image and Text objects. These are SetVisible and SetPos. You also know one method specific for text objects only. This is SetText. There are some more so called "set" methods available for text and image objects. All these methods set a control point for one or more object parameters at the current time in scene. If these parameters are numerical (int or double) they change smoothly between the control points, like a ball position in the above example. Methods are like functions. They also have parameters. They also can be called in your program. The only difference it that they are related to one particular object. The following "set" methods can be used in Web Cartoon Maker with all objects:

- SetVisible – this method accepts true or false as a parameter to make an object visible or invisible in the current scene after current time.
- SetPos – this method changes the object position on the screen. It accepts 2 double coordinates x and y and creates a control point at the current time in scene. Object's position between control points is changing smoothly in time. If you want to change coordinates immediately then you can create 2 control points at the same time by calling, for example, MyObject.SetPos (100, 100) and then MyObject.SetPos (-100, -100) immediately after first SetPos instruction. Here is an example with 2 scenes in it. In the first scene the coordinates are changing smoothly. In the second scene the coordinates are changing immediately. This is also our first example with 2 scenes. As you can notice, all the objects in Scen1 disappear when second scene is started. I'll show you some more examples with multiple scenes later.

```
void Scene1 ()
{
	Image Ball ( "sport/beach_ball.svg" );
	Ball.SetVisible ( true );
	Ball.SetPos ( 100, 100 );

	Sleep ( 1 );
	Ball.SetPos ( -100, -100 );
```

```
}

void Scene2 ()
{
	Image Ball ( "sport/beach_ball.svg" );
	Ball.SetVisible ( true );
	Ball.SetPos ( 100, 100 );

	Sleep ( 1 );
	Ball.SetPos ( 100, 100 );
	Ball.SetPos ( -100, -100 );

	Sleep ( 1 ); // to see the results of moving the object
}
```

- SetX and SetY. These 2 methods available for your convenience to set only individual coordinates of the objects. These methods are similar to SetPos but accept only one double parameter and update only individual coordinated
- SetAngle – this method changes an object's angle on the screen. It accepts one double parameter – angle in degrees. It creates a control point like SetPos and every other "set" method with numeric (int or double) parameters. Here is an example of rotating a ball:

```
void Scene1 ()
{
	Image Ball ( "sport/beach_ball.svg" );
	Ball.SetVisible ( true );
	Ball.SetAngle ( 0 );

	Sleep ( 1 );
	Ball.SetAngle ( 720 );
}
```

- SetScale – this method changes an object's size. It accepts a double parameter specifying the scale factor. Scale factor 1.0 means that the object is displayed with its own default size. Scale factor less than 1.0 means that the object is smaller than default size and scale factor greater than 1.0 means that the object is bigger its than default size. This method creates a control point like every "set" method with numeric (int or double) parameters. Here is an example of how you can make a ball to grow:

```
void Scene1 ()
{
	Image Ball ( "sport/beach_ball.svg" );
	Ball.SetVisible ( true );
	Ball.SetScale ( 0 );

	Sleep ( 1 );
	Ball.SetScale ( 2 );
}
```

- SetXScale and SetYScale. These two methods are similar to SetScale, but scale an object only horizontally or vertically. They also accept one double parameter and scale an object only in one direction.
- SetTransparency – this method can make an object semi-transparent. it accepts a double parameter specifying the transparency factor. Transparency factor 1.0 means that the object is fully transparent (invisible). Transparency factor less than 1.0 means that the object is partially transparent. Transparency factor 0.0 (default) means that the object is

not transparent. This method creates a control point like every "set" method with numeric (int or double) parameters. The example below will show you how you can make a ball to appear out of "thin air". Please keep in mind that transparency for some SVG files does not work too well. If this is a problem you might want to replace a ball image with a PNG image from your hard drive. Note the PNG image must support transparency and the area outside the ball must be transparent.

```
void Scene1 ()
{
      Image Ball ( "sport/beach_ball.svg" );
      Ball.SetVisible ( true );
      Ball.SetTransparency ( 1 );

      Sleep ( 1 );
      Ball.SetTransparency ( 0 );
}
```

- SetMirror – this method can flip your object horizontally. It accepts true or false as a parameter. Parameter true means that the object will be flipped (or mirrored) after current time in your cartoon scene. Parameter false means that it will not be flipped. Since this function does not accept a numeric parameter (int or double), it does not create a control point and the change happens immediately.

Lets use some of these methods to compile an example. The example below will show you a moving and rotating ball changing its transparency and size. We will also use a landscape example

```
void Scene1 ()
{
      // setup background
      Image Back ( "backgrounds/landscape.svg" );
      Back.SetVisible ( true );

      // add a ball image at (0,200)
      Image Ball ( "sport/beach_ball.svg" );
      Ball.SetVisible ( true );

      Ball.SetPos ( 400, 0 );
      Ball.SetAngle ( 0 );
      Ball.SetTransparency ( 1 );
      Ball.SetScale ( 0 );

      Sleep ( 2 );

      Ball.SetPos ( 0, 100 );
      Ball.SetAngle ( -360 );
      Ball.SetTransparency ( 0 );
      Ball.SetScale ( 3 );
}
```

Here is what you should see:

"Get" Methods

Sometimes you may want to know what the current coordinates and other parameters of an object in the scene are. While it may sounds easy – just the latest coordinates or parameters we set, it is not always true. For example we can get back in time using SetTime function like this:

```
void Scene1 ()
{
	Image Ball ( "sport/beach_ball.svg" );
	Ball.SetVisible ( true );

	Ball.SetPos ( 0, 0 ); // initial position

	SetTime ( 2.0 );
	Ball.SetPos ( 100, 100 ); // target position

	SetTime ( 1.0 ); // we go back in time!
	// what is the ball position now???
```

```
}
```

There are several "get" methods available for this purpose. They return the current object's coordinates and other parameters. These are:

- GetVisible – returns true if an object is visible of false if it is not.

```
bool bVisible = Ball.GetVisible ();
```

- GetX – returns an x coordinate of an object as a double

```
double dXPos = Ball.GetX ();
```

- GetY – returns an y coordinate of an object as a double

```
double dYPos = Ball.GetY ();
```

- GetAngle – returns an angle of an object as a double

```
double dAngle = Ball.GetAngle ();
```

- GetXScale – returns a horizontal scale factor of an object as a double

```
double dXScale = Ball.GetXScale ();
```

- GetYScale – returns a vertical scale factor of an object as a double

```
double dYScale = Ball.GetYScale ();
```

- GetTransparency – returns a transparency scale factor of an object as a double

```
double dTransp = Ball.GetTransparency ();
```

- GetMirror – returns true if an object is mirrored of false if it is not.

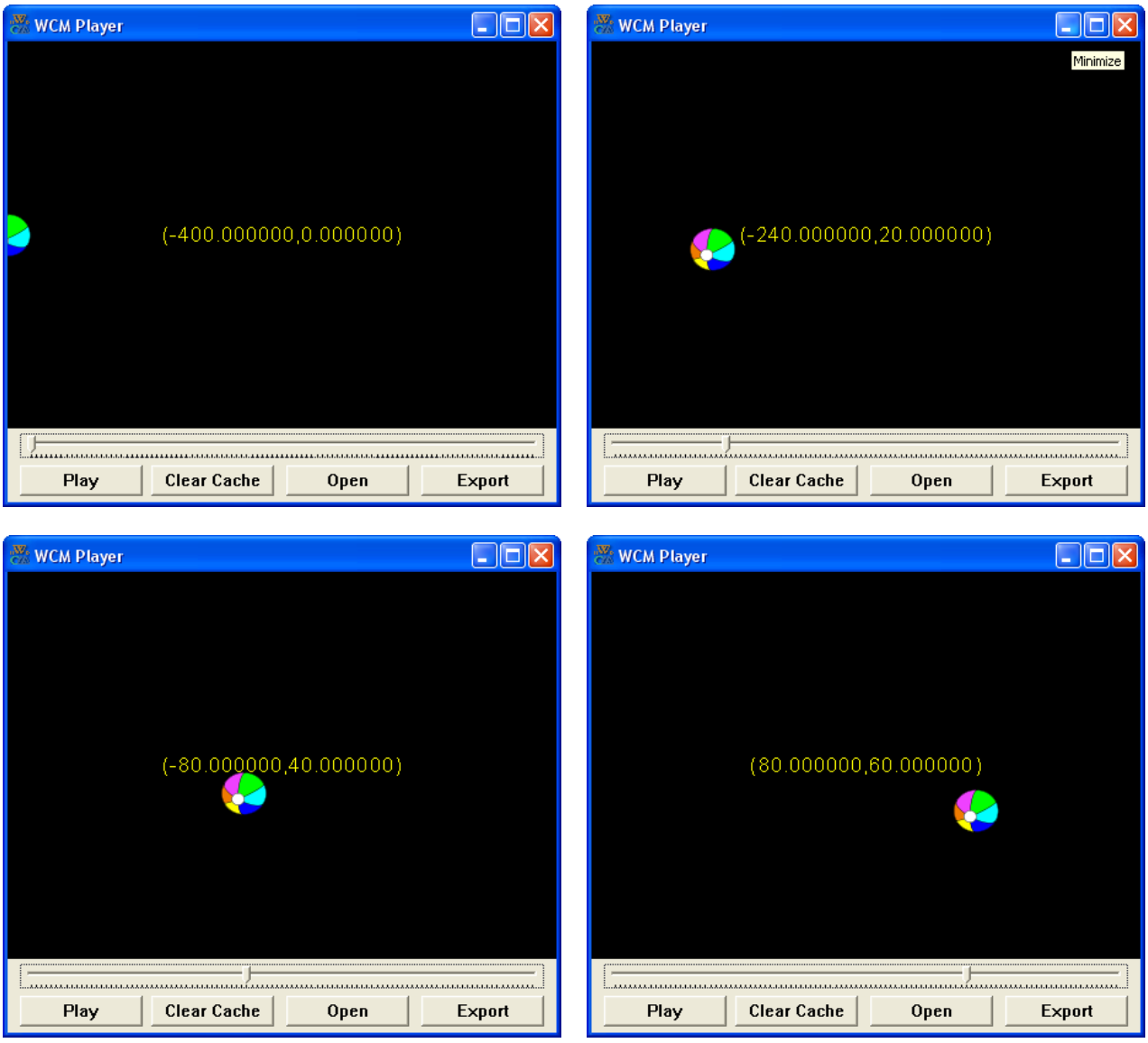
```
bool bMirror = Ball.GetMirror ();
```

In the example below you can see an example of ball moving and its coordinates displaying every second:

```
void Scene1 ()
{
	Image Ball ( "sport/beach_ball.svg" );
	Ball.SetVisible ( true );

	Ball.SetPos ( -400, 0 ); // initial position

	SetTime ( 5.0 );
	Ball.SetPos ( 400, 100 ); // target position

	SetTime (0);
	Text Coord ( "" ); // empty string, we'll change it later
	Coord.SetVisible ( true );
	Coord.SetText ( "(" + Ball.GetX () + "," + Ball.GetY () + ")" );

	SetTime (1);
	Coord.SetText ( "(" + Ball.GetX () + "," + Ball.GetY () + ")" );

	SetTime (2);
	Coord.SetText ( "(" + Ball.GetX () + "," + Ball.GetY () + ")" );

	SetTime (3);
	Coord.SetText ( "(" + Ball.GetX () + "," + Ball.GetY () + ")" );

	SetTime (4);
```

```
        Coord.SetText ( "(" + Ball.GetX () + "," + Ball.GetY () + ")" );

        SetTime (5);
        Coord.SetText ( "(" + Ball.GetX () + "," + Ball.GetY () + ")" );
}
```

You should be able to see the following cartoon when the above program is compiled:



"Change" Methods

As I said before "set" methods add some kind of control point in your animated cartoon. Sometimes it is very convenient, but sometimes it is not. For example what if we want to move a ball, then wait and then move it again. Let's look into following example:

```
void Scene1 ()
{
        Image Ball ( "sport/beach_ball.svg" );
        Ball.SetVisible ( true );
```

```
        Ball.SetPos ( 0, 0 );      // initial position
        Sleep ( 1 );
        Ball.SetPos ( 100, 100 ); // position after first move second
        Sleep ( 1 );
        Ball.SetPos ( 100, 100 ); // position before second move
        Sleep ( 1 );
        Ball.SetPos ( 200, 200 ); // position after second move
}
```

As you can see from this example, we need to have two control points with coordinates (100,100) to have our ball to stay still between the two moves. It is not very convenient. But all the "set" methods with numerical parameters (int or double) also have similar "change" methods. These "change" methods have the same parameters as "set" methods plus on more parameter – duration. For example there is a ChangePos method which accepts three double parameters. One – for new x coordinate, one - fro new y coordinate and one – for "duration". These functions actually add two control points. One – with the current object's parameter at the current time in cartoon, and one – with the target parameter after "duration" number of seconds passed. These functions also change the time in your cartoon by "duration" number of seconds. In the above example three lines can be replaced with just one:

```
void Scene1 ()
{
        Image Ball ( "sport/beach_ball.svg" );
        Ball.SetVisible ( true );

        Ball.SetPos ( 0, 0 ); // you can skip this, (0,0) is the default

        Ball.ChangePos ( 100, 100, 1 ); // first move
        Sleep ( 1 );                    // wait
        Ball.ChangePos ( 200, 200, 1 ); // second move
}
```

As I already said there are "change" methods equivalents for every "set" method with numerical parameters. These are:

- ChangePos
- ChangeX
- ChangeY
- ChangeAngle
- ChangeScale
- ChangeXScale
- ChangeYScale
- ChangeTransparency

Here is an example changing the ball's parameters one by one:

```
void Scene1 ()
{
        Image Ball ( "sport/beach_ball.svg" );
        Ball.SetVisible ( true );
```

```
        Ball.ChangePos ( 100, 100, 1 );
        Ball.ChangeAngle ( 360, 1 );
        Ball.ChangeScale ( 2, 1 );
        Ball.ChangeTransparency ( 2, 1 );
}
```

I recommend compiling the above example to see how it works. Using these "change" methods is probably the most convenient way of working with objects in Web Cartoon Maker.

Please also keep in mind that all the "set", "get" and "change" methods described in this chapter can work with either images or text. For example we can do exactly the same things with a text object:

```
void Scene1 ()
{
        Text MyText ( "Hello" );
        MyText.SetVisible ( true );

        MyText.ChangePos ( 100, 100, 1 );
        MyText.ChangeAngle ( 360, 1 );
        MyText.ChangeScale ( 2, 1 );
        MyText.ChangeTransparency ( 2, 1 );
}
```

Methods Unique to Text Objects

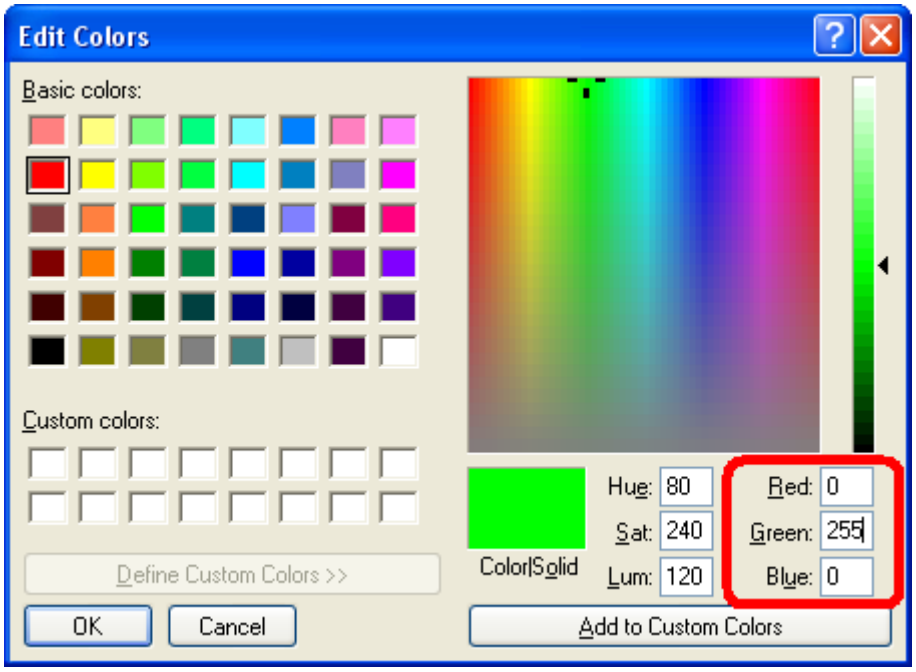
There are also some "set", "get" and "change" methods to work with text objects only. You already know one of them: SetText, which can change the default text associated with a text object at a current time in cartoon scene. Here is a full list of functions specifically designed to work with text objects

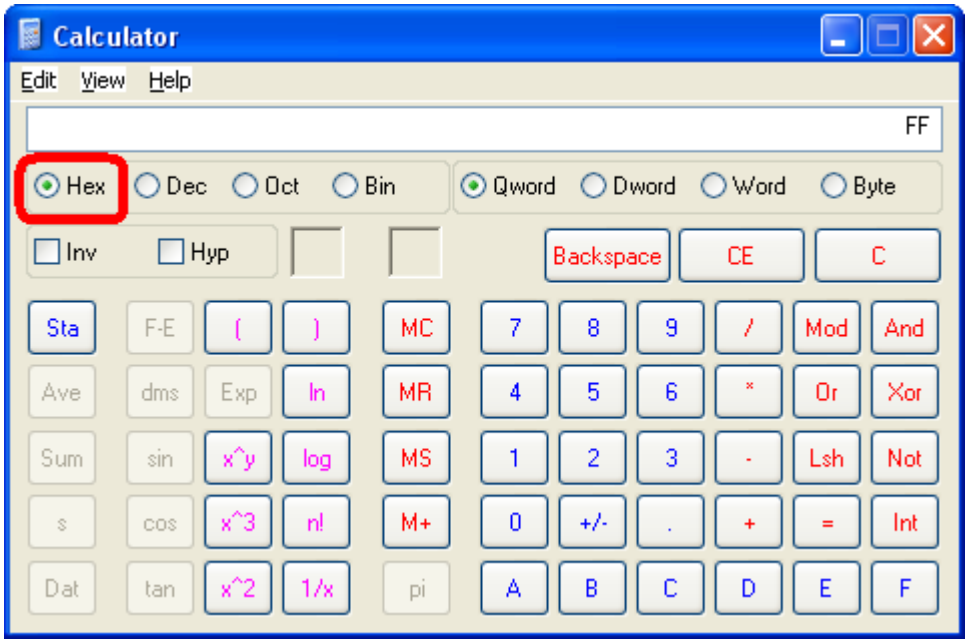
- SetText – accepts one string parameter. Changes the text associated with a text object. It is even possible to declare a text object without parameters, which is equal to a text object initiated with an empty string, and then use SetText to change it at a desired moment in your cartoon:

  ```
  void Scene1 ()
  {
        Text MyText;
        Sleep ( 5 );
        MyText.SetText ( "5 seconds passed" );
        Sleep ( 5 );
  }
  ```

- SetFont – accepts one string parameter – Windows font name. Changes the fonts associated with a text object. You can see the fonts available on your machine by clicking on the font icon in the Control Panel. You can also add or remove fonts there.
- SetStyle - accepts one string parameter. The string may contain some of the following letters: B, I, U or S, specifying the font style in use. These letters are responsible for bold, italic, underlined and stroked out styles respectively. For example if you want to make a text italic and stroked out, you should supply "IS".
- SetColor - accepts one string parameter. The string contains a hex encoded color in "RRGGBB" format. For example if you want to use a green color, you should supply the hex value "00FF00".

Note: You can find the RGB decimal values for any color by clicking on the color box in Paint and selecting "custom Color." The Scientific Calculator available in "Accessories" will do the decimal to hex conversion for you. Be sure to convert the red, green and blue numbers separately.





- SetSize – accepts one int parameter – windows font size. Changes the font size associated with a text object
- GetText – requires no parameters and returns a string – a text currently associated with a text object
- GetFont – requires no parameters and returns a string – a Windows font name currently associated with a text object

- GetStyle – requires no parameters and returns a string with encoded text style associated with a text object. Pease see SetStyle for encoding format.
- GetColor – requires no parameters and returns a string with encoded text color associated with a text object. Pease see SetColor for encoding format.
- GetSize – requires no parameters and returns an integer value – windows font size associated with a text object
- ChangeSize – as I already mentioned, all "set" methods with numerical parameters have a corresponding "change" method. ChangeSize accepts 2 parameters – int for font size and double for duration. It inserts two control points and changes the font size smoothly between them during specified number of seconds.

Methods Unique to Image Objects

There are 2 methods specific for Image objects available. These are

- SetImage – to change an image associated with an image object. It accepts one string parameter the same way as when you declare an image object. It could a valid URL of image, short path to an image file in WCM Library or fully qualified path to a local file (do not forget about double back slashes). It is even possible to declare an image object without parameters and assign an image using SetImage method later at a desired moment in your cartoon:

```
void Scene1 ()
{
	Image MyImage;
	Sleep ( 5 );
	MyImage.SetImage ( "vehicles/sedan.svg" );
	Sleep ( 5 );
}
```

- GetImage – returns an image currently associated with an image object

Summary

To summarize, the basic operations available for objects include Set, Get and Change commands for a variety of parameters associated with the objects placed in the scene. Much of a cartoon's script just consists of these commands used in various combinations. There are also commands unique to character objects, which are discussed in the next sections.

Character Objects

Including Characters in Scenes

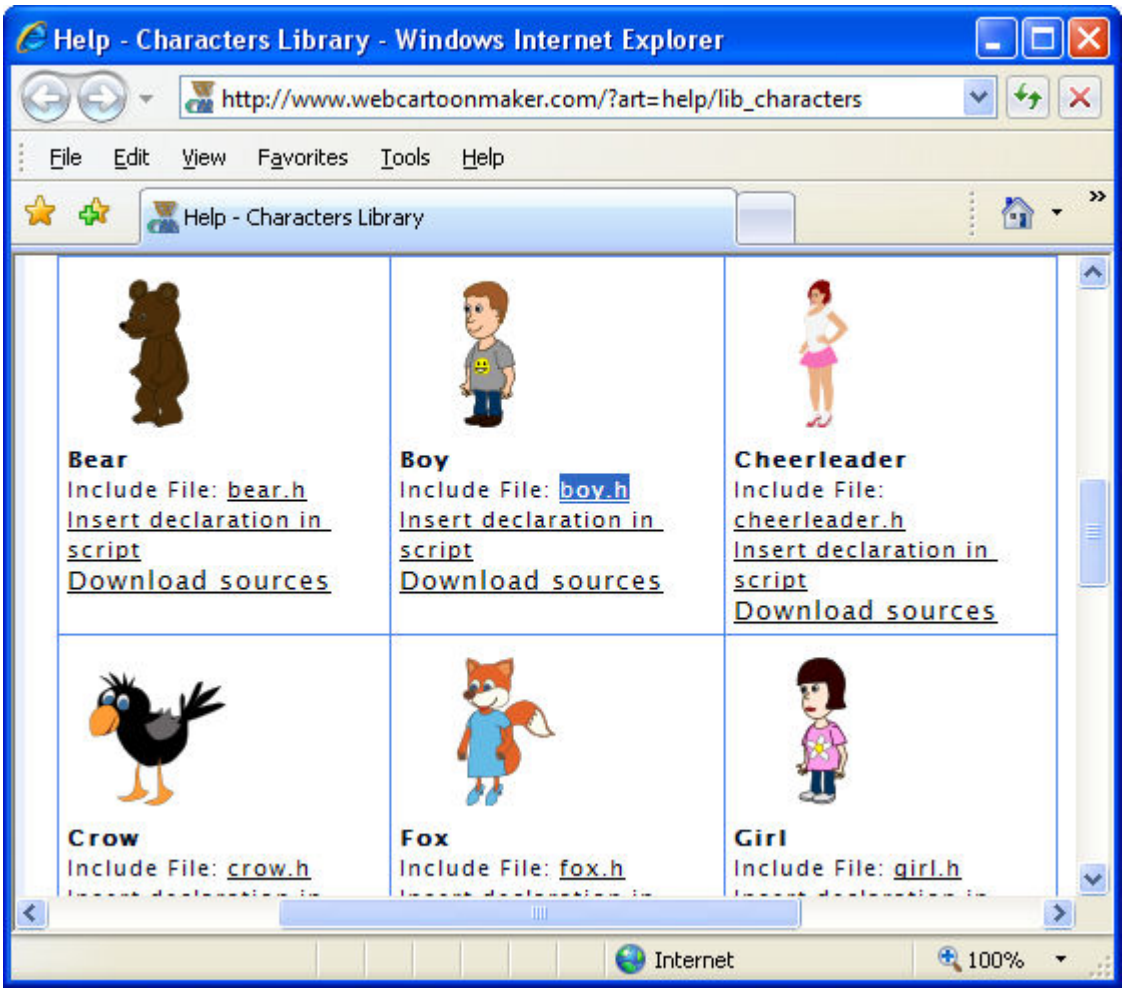
One of the most interesting features of Web Cartoon Maker is a support for characters. Any now we are not talking about symbols or letters which make up strings. We are talking about real cartoon character which can walk talk and move their parts.

Let's look into this example:

```
#include <boy.h>

void Scene1 ()
{
	Boy Max;
```

```
        Max.SetVisible ( true );
    }
```

I am sure you have a lot of questions looking on this simple example. First line of the program above contains a special directive for compiler to include a character file "boy.h". Every character from Web Cartoon Maker's library requires a special file to be included. These include instructions always begin with a special character #. They usually placed at the top of your program and have a file name embedded between symbols < and >. But how do you know what file should be included? Open your browser (preferably Internet Explorer 7.0 or later) and navigate to the following URL to see a list of all available characters: http://www.webcartoonmaker.com/?art=help/lib_characters



You will see an include file name for every character in the library. Just use the file name for an include instruction at the top of you program.

Looking further on the above example you will notice a line Boy Max;. It looks pretty similar to an image or text object declaration, does not it? The only difference is than image and text objects required a text parameter.

This is really an object declaration! This is a declaration of a character object . (more formally, an "instantiation of the class Boy which is a subclass of IhumanCharacterSideView. This will be discussed in more detail later. For now, just practice saying it and sounding impressive). A Boy character object declaration to be more specific. And Max – is an unique name for the object. You can have more than one Boy character object in your scene. You just need to name them differently (!!!). For example you can name a second object Mike.

You can use all the "set", "get" and "change" methods available for image objects with characters. As you can see we already used one of them – SetVisible.

Using a Character's Parts

But you can do more things with characters! All the characters contain parts – arms, legs, head, eyes, mouth and other. And you can move these parts, rotate and scale them! Actually the parts are almost like other objects. You can access a character part using a dot symbol after an object name. Take a look on example below:

```
#include <boy.h>

void Scene1 ()
{
      Boy Max;
      Max.SetVisible ( true );

      Max.RightArm.SetAngle ( 45 );
}
```

Note: Once again, the poor dot gets a workout. In this case, going from right to left, the "dot convention" uses the dot to separates the function from the part and then the part from the name of the character it is a part of. Although not true for any of the current characters in the WCM library, this process could go on indefinitely. It is certainly possible and even useful to have something like Character.RightArm.Hand.MiddleFinger.TopJoint.SetAngle(45);

You can probably guess just by looking on the above example that we change an angle of Max's right arm. The name of method we used to change an angle of the part is the same as we used to change an angle of image, text and character objects. But generally speaking, the method names available for parts are different from method names of other objects we know about.

Let's start by learning what kinds of parts are available. Generally speaking, different characters may have different parts. But most of the characters in the library are human-like. And they have the following parts:

- LeftLeg
- RightLeg
- Body
- LeftArm
- RightArm
- Head
- Mouth
- LeftEye
- RightEye

And you can use the following "set", "get" and "change" methods with parts:

- SetVisible – it works the same way as SetVisible for other objects. Parameter true means that a part will be visible. Parameter false means that it is going to be invisible. The only difference is that all parts are visible by default, while other objects are not
- SetShift – it is similar to other objects' SetPos method. It accepts two double parameters – two coordinates. But these coordinates are relative to a part's origin. The default shift coordinates are (0,0). And if you want, for example, move it to the left by five pixels and up by three, then you should supply (-5,-3) as parameters. Remember, all "set" methods with numerical values actually insert a control point in the cartoon at a current time and the values between these control points change smoothly.
- SetXShift and SetYShift – as with SetPos these functions are very similar to SetShift but accept only one double parameter and change only one coordinate.
- SetAngle – it works absolutely the same way as SetAngle for other kinds of objects
- SetScale – it works absolutely the same way as SetScale for other kinds of objects
- SetXScale and SetYScale – they work absolutely the same way as SetXScale and SetYScale for other kinds of objects

Let's see how you can move a character's parts

```
#include <boy.h>

void Scene1 ()
{
      Boy Max;
      Max.SetVisible ( true );
      Max.SetPos ( 0, 290 );

      Max.RighArm.ChangeAngle ( 360, 1 ); // rotate it clockwise
      Max.RighArm.ChangeAngle ( 0, 1 );   // rotate it counter clockwise

      Max.RighArm.ChangeShift ( -50, -50, 1 ); // move it left and up
      Max.RighArm.ChangeShift ( 0, 0, 1 );     // move it back

      Max.RighArm.ChangeScale ( 2, 1 ); // increase the size
      Max.RighArm.ChangeScale ( 1, 1 ); // decrease it back
}
```

Changing a Part's Decals

Every part has a picture associated with it. We call these pictures "Decals". Actually some of the parts can have multiple decals and you can choose which one you want to show. Every decal has its own name. The name of default decal is "DEFAULT". Most of the parts have only one "DEFAULT" decal. But eyes and mouth have more. LeftEye and RightEye also have decals "WINK1" and "WINK2". "WINK1" usually is a picture of a half closed eye. "WINK2" usually is a picture of a fully closed eye. Part Mouth usually has decals "SPEAK_M", "SPEAK_E", "SPEAK_A", "SPEAK_O" and "SPEAK_W". These are mouth pictures associated with speech. There are two more "set" and "get" methods you can use with characters. These are:

- SetDecal – accepts a string as a parameter. It changes the decal (or picture) associated with a part

- GetDecal – accepts no parameters and returns a string – name of the decal associated with a part

Take a look into the following example:

```
#include <boy.h>

void Scene1 ()
{
	Boy Max;
	Max.SetVisible ( true );
	Max.SetPos ( 0, 290 );

	Max.Mouth.SetDecal ( "SPEAK_M" );
	Sleep ( 1 );
	Max.Mouth.SetDecal ( "SPEAK_E" );
	Sleep ( 1 );
	Max.Mouth.SetDecal ( "SPEAK_A" );
	Sleep ( 1 );
	Max.Mouth.SetDecal ( "SPEAK_O" );
	Sleep ( 1 );
	Max.Mouth.SetDecal ( "SPEAK_W" );
	Sleep ( 1 );
	Max.Mouth.SetDecal ( "DEFAULT" );
	Sleep ( 1 );

	Max.LeftEye.SetDecal ( "WINK1" );
	Max.RightEye.SetDecal ( "WINK1" );
	Sleep ( 1 );
	Max.LeftEye.SetDecal ( "WINK2" );
	Max.RightEye.SetDecal ( "WINK2" );
	Sleep ( 1 );
	Max.LeftEye.SetDecal ( "WINK1" );
	Max.RightEye.SetDecal ( "WINK1" );
	Sleep ( 1 );
	Max.LeftEye.SetDecal ( "DEFAULT" );
	Max.RightEye.SetDecal ( "DEFAULT" );
	Sleep ( 1 );
}
```

Walking, Talking and Winking

As you can see, you can do many things with the characters. But sometimes it may be little difficult. There are several methods available with characters to simplify most common actions – walking, talking and winking

- GoesTo – it is very similar to ChangePos. It also accepts 3 double parameters – a target coordinates and duration. It also changes the time in your cartoon scene. But beside moving a character, it also moves its parts (legs and arms) to simulate a walking pattern. You can try the following example to see the difference:

```
#include <boy.h>

void Scene1 ()
{
	Boy Max;
	Max.SetVisible ( true );
	Max.SetPos ( 200, 290 );
```

```
        Max.ChangePos ( -200, 290, 2 );
}

void Scene2 ()
{
        Boy Max;
        Max.SetVisible ( true );
        Max.SetPos ( 200, 290 );

        Max.GoesTo ( -200, 290, 2 );
}
```

- Says – it is a one of the most interesting methods. It accepts one string parameter and optionally 2 more double parameters. String can specify an URL of WAV file name like "http://www.webcartoonmaker.com/lib/sounds/people/laugh.wav", or, like with images, fully qualified path to a WAV file on your computer, like "c:\\laugh.wav" (remember about double back slashes). Alternatively, it could be any English text, which will be converted to audio. There are two more double parameters, but these are optional and you can just omit them. First of them is an audio volume factor. When it is equal to 1 (default), the audio is playing with its original volume. If it is less than 1, then audio is playing quieter. If it is more than 1, then audio is playing louder. Another double parameter specifies how fast you want mouth decals to change during a speech. You suppose to indicate a number of decal changes per second. The default value is 6 which looks compilerpretty good in most cases. Please also keep in mind that this method does not change a current scene time. You must use function Sleep with approximate speech duration after using Says. This inconvenience is related to the fact that the compiler knows nothing about your audio files and does not generate speech. Everything is done in the WCM Player. Take a look into the example below:

```
#include <boy.h>

void Scene1 ()
{
        Boy Max;
        Max.SetVisible ( true );
        Max.SetPos ( 200, 290 );

        Max.Says (
        "http://www.webcartoonmaker.com/lib/sounds/wcm/hello.wav" );
        Sleep (1);

        Max.Says ( "c:\\hello.wav" ); // local file must exist
        Sleep (1);

        Max.Says ( "Hello" ); // use text to speech
        Sleep (1);

        Max.Says ( "Hello", 1.2, 10 ); // same but louder and faster
        Sleep (1);
}
```

- Winks, WinksLeft and WinksRight – you can use them with no parameters at all or supply one double parameter – winking duration in seconds. Default wink duration is 0.5 seconds. Obviously, these methods make a character to wink with two or just one eye. They also change a current time in scene:

```
#include <boy.h>

void Scene1 ()
{
	Boy Max;
	Max.SetVisible ( true );
	Max.SetPos ( 200, 290 );

	Max.Winks ();
	Max.WinksLeft ();
	Max.WinksRight ();
	Max.Winks ( 0.2 ); // winks faster
}
```

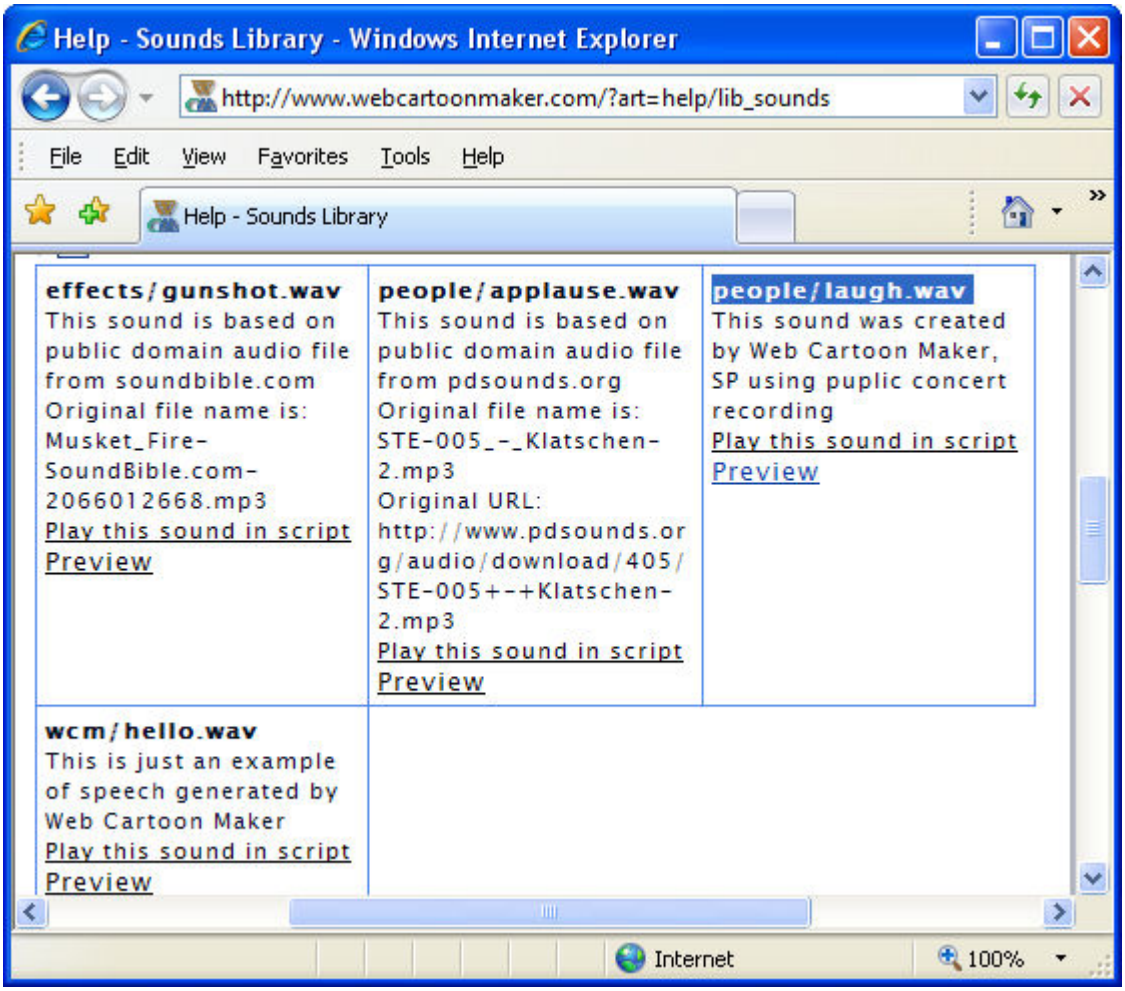
Summary

Five additional commands for characters were introduced – one to simulate walking, one for speech and three for having characters wink. Combined with the basic Set, Get and Change commands, these allow for quite a range of character movement and behavior. However, remember you can always develop your own functions to expand your options.

Playing Sounds

There is a special function Play in Web Cartoon Maker for playing WAV files. It accepts one string parameter and two optional (double and int) parameters which you can omit if you want. The string parameter specifies a WAV file name. This could be an URL like "http://www.webcartoonmaker.com/lib/sounds/wcm/hello.wav" or path to a file on your hard drive like "c:\\myaudio.wav" (please remember about double back slashes). Instead of using fully qualifier URLs you can also use a short path to an audio file in Web Cartoon Maker's library. To browse the library you can go to the following URL:

http://www.webcartoonmaker.com/?art=help/lib_sounds



Please keep in mind that function Play does not change time in a cartoon scene and does not make your cartoon longer. If you want to wait until all the playing is done, you must use Sleep immediately after it. Here is an example:

```
void Scene1 ()
{
      // play them at the same time
      Play ( "people/applause.wav" );
      Play ( "people/laugh.wav" );

      Sleep ( 4 );

      // play audio files one after another
      Play ( "c:\\my_audio.wav" ); // file must exist on your hard drive
      Sleep ( 4 );                 // duration of your wav file
      Play (
      "http://www.webcartoonmaker.com/lib/sounds/effects/gunshot.wav" );
      Sleep ( 2 );
}
```

If you compile the above example, you will notice that "applause.wav" and "laugh.wav" sounds were mixed together.

As I said before there are two more optional parameters for function Play available. First of them is an audio volume factor. Similarly to the character's Says method, when it is equal to 1 (default),

the audio is playing with its original volume. If it is less than 1, then audio is playing quieter. If it is more than 1, then audio is playing louder.

Another parameter specifies how you want your sounds to be mixed together. You probably noticed from the above example that "applause.wav" and "laugh.wav" sounds were mixed together. This is the default behavior. But it is also possible to replace a first sound with a second one or make first sound quieter. You suppose to use one of the following constants:

- MIXMODE_MIX – just mixes the sounds together. This is the default behavior
- MIXMODE_REPLACE – replaces the existing sound
- MIXMODE_REPLACEFADE – smoothly replaces the existing sound
- MIXMODE_DAMP – makes the existing sound quieter and then mixes the sounds
- MIXMODE_DAMPFADE – smoothly makes the existing sound quieter and then mixes the sounds

You can try the example below to understand the constants. But you can just omit them if default mixing forks fine for you:

```
void Scene1 ()
{
	Play ( "people/applause.wav" );
	Play ( "people/laugh.wav", 1, MIXMODE_MIX );
	Sleep ( 4 );

	Play ( "people/applause.wav" );
	Play ( "people/laugh.wav", 1, MIXMODE_REPLACE );
	Sleep ( 4 );

	Play ( "people/applause.wav" );
	Play ( "people/laugh.wav", 1, MIXMODE_REPLACEFADE );
	Sleep ( 4 );

	Play ( "people/applause.wav" );
	Play ( "people/laugh.wav", 1, MIXMODE_DAMP );
	Sleep ( 4 );

	Play ( "people/applause.wav" );
	Play ( "people/laugh.wav", 1, MIXMODE_DAMPFADE );
	Sleep ( 4 );
}
```

A Complete Cartoon

We now know enough to make a simple but completed cartoon. Let's choose a simple story like this one:

Girl is walking with a boy through the park. She says: "If we get married, I want to share all your worries, troubles and lighten your burden". Boy replies: "It is very kind of you, darling, but I do not have any worries or troubles". Girl replies: "Well, that is because we aren't married yet".

Let's follow a plan when making cartoons:

1. Decide what kind of scenes you want to have in cartoons. It looks like there is only one scene in this simple story. But we may want to have a couple more for the beginning and ending titles. Lets have 3 scenes then

2. Decide what kind of background images are you going to use. We can use plain black screen for titles and "backgrounds/forest.svg" image fro Web Cartoon Maker's library as a park image. You can probably find a better background image on your hard drive
3. Decide what kind of music and other sounds you want to use. You can play some music from your hard drive. We are not going to play any music. But we'll play a laugh sound at the end similarly to some TV shows.
4. Decide which characters you are going to use. We'll use Boy and Girl characters.

This plan can now be expressed in pseudo code:

1. Beginning title appears
2. Forest background and main characters (boy and girl) appear
3. Boy and girl have conversation while walking through the park
4. Ending title appears

Note that we now have the basic cartoon flow completely independent of WCM. The script can be expanded, for example by including the titles text and the actual conversations, and various plot options examined without worrying about coding syntax, etc. In fact, at this level we could also go to Tales Animator or one of the many other animation tools available from very simple ones for animated GIF cartoons to professional level 3D animation tools such as 3D Max. But this book is about WCM and C++, so let's start coding by including characters' files:

```
#include <boy.h>
#include <girl.h>
```

Then create the beginning title

```
void Scene1 ()
{
	Text Title ( "In the Park" );
	Title.SetColor ( "3e6ba6" );
	Title.SetFont ( "Comic Sans MS" );
	Title.SetStyle ( "B" );
	Title.SetSize ( 90 );
	Title.SetVisible ( true );

	Title.SetTransparency ( 1 );
	Title.ChangeTransparency ( 0, 1 );

	Sleep ( 1 );
}
```

Then create the main scene. Let's just make the characters walking first:

```
void Scene2 ()
{
	Image Back ( "backgrounds/forest.svg" );
	Back.SetVisible ( true );

	Boy Max;
	Girl Mary;

	Max.SetVisible ( true );
	Mary.SetVisible ( true );

	// let's remember the beginning time for later
```

```
        double dStartTime = GetTime ();

        Max.SetPos ( 280, 140 );
        Mary.SetPos ( 350, 160 );

        Max.GoesTo ( 0, 300, 8 );

        SetTime ( dStartTime ); // happens at the same time
        Mary.GoesTo ( 70, 320, 8 );
}
```

We used functions SetTime and GetTime to synchronize the Boy's and Girl's walking actions. There is an easier way to synchronize them. There are two special macros defined in WCM C++ THIS_TIME and SAME_TIME for this. We'll learn more about macros later. But here is how to use them. If some of the actions happen at the same time you can use word THIS_TIME before the first action and SAME_TIME before other actions happening at the same time. We can rewrite the above example using these new macros:

```
void Scene2 ()
{
        Image Back ( "backgrounds/forest.svg" );
        Back.SetVisible ( true );

        Boy Max;
        Girl Mary;

        Max.SetVisible ( true );
        Mary.SetVisible ( true );

        Max.SetPos ( 280, 140 );
        Mary.SetPos ( 350, 160 );

        THIS_TIME Max.GoesTo ( 0, 300, 8 );
        SAME_TIME Mary.GoesTo ( 70, 320, 8 );
}
```

Let's add some more walking:

```
void Scene2 ()
{
        Image Back ( "backgrounds/forest.svg" );
        Back.SetVisible ( true );

        Boy Max;
        Girl Mary;

        Max.SetVisible ( true );
        Mary.SetVisible ( true );

        Max.SetPos ( 280, 140 );
        Mary.SetPos ( 350, 160 );

        THIS_TIME Max.GoesTo ( 0, 300, 8 );
        SAME_TIME Mary.GoesTo ( 70, 320, 8 );

        THIS_TIME Max.GoesTo ( -350, 300, 8 );
        SAME_TIME Mary.GoesTo ( -280, 320, 8 );
}
```

Now let's add some talking, winking and laughing at the end

```
void Scene2 ()
{
	Image Back ( "backgrounds/forest.svg" );
	Back.SetVisible ( true );

	Boy Max;
	Girl Mary;

	Max.SetVisible ( true );
	Mary.SetVisible ( true );

	Max.SetPos ( 280, 140 );
	Mary.SetPos ( 350, 160 );

	THIS_TIME Max.GoesTo ( 0, 300, 8 );
	SAME_TIME Mary.GoesTo ( 70, 320, 8 );

	THIS_TIME Max.GoesTo ( -350, 300, 8 );
	SAME_TIME Mary.GoesTo ( -280, 320, 8 );

	SetTime ( 0.5 );
	Mary.Says ( "If we get married, I want to share all your worries,
	troubles and lighten your burden" );

	SetTime ( 5.5 );
	Max.Says ( " It is very kind of you, darling, but I do not have any
	worries or troubles" );

	SetTime ( 10.5 );
	Mary.Says ( "Well, that is because we aren't married yet" );

	SetTime ( 4 );
	Mary.Winks ();
	Max.Winks ();

	SetTime ( 8 );
	Mary.Winks ();
	Max.Winks ();

	SetTime ( 12 );
	Mary.Winks ();
	Max.Winks ();

	SetTime ( 13.5 );
	Play ( "people/laugh.wav" );
}
```

Now we are ready to add the ending title:

```
void Scene3 ()
{
	Text Title ( "The End" );
	Title.SetColor ( "3e6ba6" );
	Title.SetFont ( "Comic Sans MS" );
	Title.SetStyle ( "B" );
	Title.SetSize ( 90 );
	Title.SetVisible ( true );

	Title.SetTransparency ( 1 );
	Title.ChangeTransparency ( 0, 1 );
```

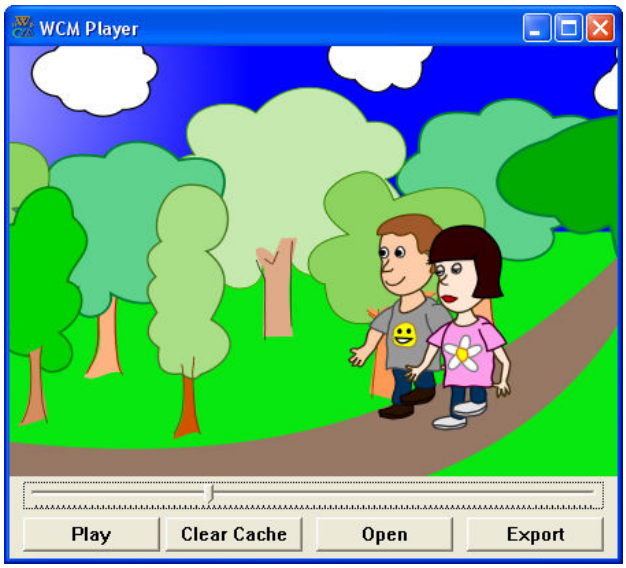
```
        Sleep ( 1 );
}
```

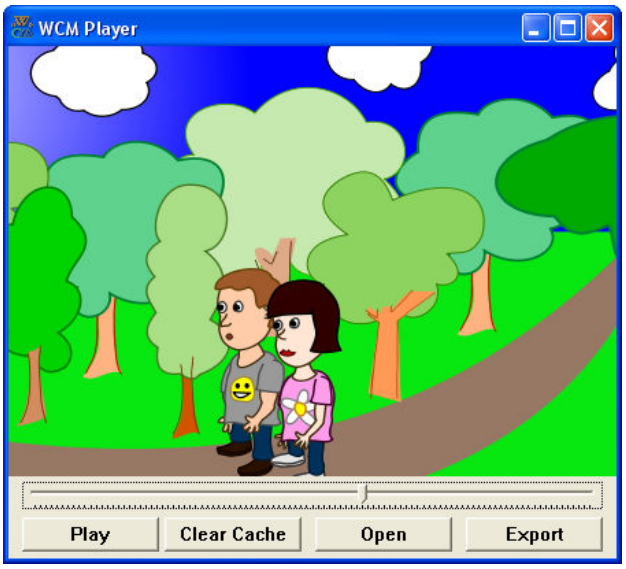
Now this looks like our first complete cartoon! Let's merge all the code together, compile and preview it!

```
#include <boy.h>
#include <girl.h>

void Scene1 ()
{
        Text Title ( "In the Park" );
        Title.SetColor ( "3e6ba6" );
        Title.SetFont ( "Comic Sans MS" );
        Title.SetStyle ( "B" );
        Title.SetSize ( 90 );
        Title.SetVisible ( true );

        Title.SetTransparency ( 1 );
        Title.ChangeTransparency ( 0, 1 );

        Sleep ( 1 );
}
void Scene2 ()
{
        Image Back ( "backgrounds/forest.svg" );
        Back.SetVisible ( true );

        Boy Max;
        Girl Mary;

        Max.SetVisible ( true );
        Mary.SetVisible ( true );

        Max.SetPos ( 280, 140 );
        Mary.SetPos ( 350, 160 );

        THIS_TIME Max.GoesTo ( 0, 300, 8 );
        SAME_TIME Mary.GoesTo ( 70, 320, 8 );

        THIS_TIME Max.GoesTo ( -350, 300, 8 );
        SAME_TIME Mary.GoesTo ( -280, 320, 8 );

        SetTime ( 0.5 );
        Mary.Says ( "If we get married, I want to share all your worries,
        troubles and lighten your burden" );

        SetTime ( 5.5 );
        Max.Says ( " It is very kind of you, darling, but I do not have any
        worries or troubles" );

        SetTime ( 10.5 );
        Mary.Says ( "Well, that is because we aren't married yet" );

        SetTime ( 4 );
        Mary.Winks ();
        Max.Winks ();

        SetTime ( 8 );
```

```
        Mary.Winks ();
        Max.Winks ();

        SetTime ( 12 );
        Mary.Winks ();
        Max.Winks ();

        SetTime ( 13.5 );
        Play ( "people/laugh.wav" );
}
void Scene3 ()
{
        Text Title ( "The End" );
        Title.SetColor ( "3e6ba6" );
        Title.SetFont ( "Comic Sans MS" );
        Title.SetStyle ( "B" );
        Title.SetSize ( 90 );
        Title.SetVisible ( true );

        Title.SetTransparency ( 1 );
        Title.ChangeTransparency ( 0, 1 );

        Sleep ( 1 );
}
```

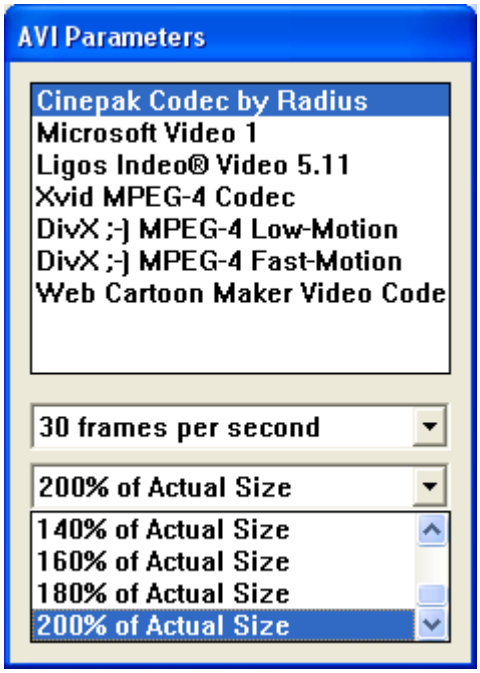
Here are some screenshots of our first complete cartoon:





Size of an Animated Cartoon

You already know that the default size of animated cartoons created by Web Cartoon Maker is 800x600 pixels and the coordinates of the right bottom corner are (400,300) and the coordinates of the left upper corner are (-400,-300). It is also possible to scale a carton during export (you should click "Export" in WCM player):



Since most of the graphics is in vector format, the quality of cartoons does not degrade during such scaling.

But there is another way to change a cartoon size right from your C++ program. You should use function SetMovieSize for this purpose. You can use it anywhere in your program. Since the size of cartoon is fixed it makes sense to call it only once in your first scene. The function accepts two int parameters – width and height.

Once you change the size of your animated movie, you also change the coordinates of left upper and right bottom corners. For example if you want to make a 640 x 360 pixels cartoon, the coordinates of right bottom corner will be (320,180) and the coordinates of left upper corner will be (-320,-180). In the example below the boy appears in the middle horizontally and at the bottom vertically:

```
void Scene1 ()
{
      SetMovieSize ( 640, 360 );

      Boy Max;
      Max.SetVisible ( true );
      Max.SetPos ( 0, 180 );

      Sleep (1); // you can skip this
}
```

Which resizing option is better? Both have advantages and disadvantages. Scaling during export may degrade raster format art (PNG, JPEG, GIF) but it also can be done at any time. Scaling inside the program itself should be done when you first start developing the program, since changing the size later in the program will affect the positioning of all the objects in the cartoon. On the other hand, your raster graphics will appear as they will look in the final cartoon. This may be particularly important if you are using photographs in your cartoon, since converting them to vector format is almost impossible.

Camera 

It is also possible to move 2D camera through a cartoon scene. You can use a camera when you want to show different pieces of your scene, when it is bigger than your movie size. This can be very effective as noted earlier.

There are several functions available in WCM C++ to move a camera. These are very similar to methods of some objects:

- SetCameraPos – it is very similar to an object's SetPos method. It accepts two double coordinates and changes the position of camera. When you change a camera position, it has the same effect as moving all the objects in current scene in an opposite direction. These so called "set" functions for working with camera also insert some kind of control point in your movie at a current time, and a camera position is changing smoothly between these control points. You can try to compile the example below to see an effect of moving camera. We'll use coordinates background again so you can see how they are changing on the screen:

```
void Scene1 ()
{
      // show coordinates
      Image Coordinates ( "wcm/coordinates.emf" );
      Coordinates.SetVisible ( true );

      // show a ball in the center
      Image Ball ( "sport/beach_ball.svg" );
      Ball.SetVisible ( true );

      // Place a camera at the center
      SetCameraPos ( 0, 0 );
```

```
        Sleep ( 1 );

        // move camera to the left and up
        SetCameraPos ( -100, -100 );
        Sleep ( 1 );

        // move camera to the right and up
        SetCameraPos ( 100, -100 );
        Sleep ( 1 );

        // move camera to the right and bottom
        SetCameraPos ( 100, 100 );
        Sleep ( 1 );

        // move camera to the left and bottom
        SetCameraPos ( -100, 100 );
        Sleep ( 1 );
}
```

- SetCameraX and SetCameraY – These are almost the same as SetCameraPos but accept only one double parameter and change only one camera coordinate.
- SetCameraAngle – this function accepts only one double parameter - camera angle in degrees. This is an equivalent of rotating all the objects in current scene in opposite direction relative to the screen center. You can try this example for better understanding:

```
void Scene1 ()
{
        // show coordinates
        Image Coordinates ( "wcm/coordinates.emf" );
        Coordinates.SetVisible ( true );

        // show a ball in the center
        Image Ball ( "sport/beach_ball.svg" );
        Ball.SetVisible ( true );

        SetCameraAngle ( 0 );
        Sleep ( 10 );
        SetCameraAngle ( 360 );
}
```

- SetCameraZoom – this function makes a camera too zoom in and out. Zooming is an equivalent of scaling your current view. It accepts one double parameter – a scale factor. Scale factor 1.0 means that there is no zooming happening. Scale factor less than 1.0 means that you are zooming out. Scale factor greater than 1.0 means that you are zooming in. Here is an example to try:

```
void Scene1 ()
{
        // show coordinates
        Image Coordinates ( "wcm/coordinates.emf" );
        Coordinates.SetVisible ( true );

        // show a ball in the center
        Image Ball ( "sport/beach_ball.svg" );
        Ball.SetVisible ( true );

        SetCameraZoom ( 1 );    // no zoom
        Sleep ( 3 );
```

```
        SetCameraZoom ( 0.8 ); // zoom out
        Sleep ( 6 );
        SetCameraZoom ( 1.2 ); // zoom in
}
```

As you probably noted these are all "set" functions similar to "set" methods. And all of them accept numeric parameters. This means that all of them insert a control point at a current time in scene and the actual value is calculated automatically between these control points. This also means that there are "change" equivalents exist. These "change" equivalents accept one additional double parameter – duration and insert two control points – at a current time and after the duration number of seconds. The "change" functions are:

- ChangeCameraPos
- ChangeCameraX and ChangeCameraY
- ChangeCameraAngle
- ChangeCameraZoom

You can also know current camera parameters using following "get" functions. All of them do not require any parameters and return a double value

- GetCameraX
- GetCameraY
- GetCameraAngle
- GetCameraZoom

In the following example a camera will move through the set of characters and then zoom in on one of the faces.

```
#include <boy.h>
#include <girl.h>
#include <man.h>
#include <woman.h>

void Scene1 ()
{
      Image Back ( "backgrounds/beach.svg" );
      Back.SetVisible ();

      Boy Max;
      Max.SetVisible ( true );
      Max.SetPos ( -300, 290 );

      Girl Mary;
      Mary.SetVisible ( true );
      Mary.SetPos ( -100, 290 );

      Man Mike;
      Mike.SetVisible ( true );
      Mike.SetPos ( 100, 290 );

      Woman Wendy;
      Wendy.SetVisible ( true );
      Wendy.SetPos ( 300, 290 );
```

```
        SetCameraPos ( -200, 0 );
        ChangeCameraPos ( 200, 0, 8 );
        ChangeCameraPos ( 100, 0, 2 );
        ChangeCameraZoom ( 2, 2 );
}
```

Once compiled you will see the following movie:

Conditionals

Objects as Function Parameters

You probably noticed that internal WCM objects look pretty much like variables. You'll get more familiar with them later after reading about classes. For now I just want to note one simple thing about these objects. You can use these objects as parameters to your functions. You just need to add a special symbol & before a parameter name. For example you can write a function which makes a Boy character to do some actions like this:

```
#include <boy.h>

void DoActions ( Boy &Who )
{
      Who.Says ( "Hello" );
      Sleep ( 0.5 );
      Who.Winks ();
}

void Scene1 ()
{
      Boy Max;
      Max.SetVisible ();
      DoActions ( Max );
}
```

Please note that you need to use & symbol, also called ampersand only in function declaration. You do not need to use this symbol in function calls or when using parameter in a function.

Conditional Execution

In order to make advanced cartoons, we almost always need the ability to check certain conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement:

```
if ( iX < 0 )
{
      Max.Wink ();
}
```

The expression in parentheses is called the condition. If it is true, then the statements in brackets get executed. If the condition is not true, nothing happens.

The condition can contain any of the comparison operators:

- x == y // x equals y
- x != y // x is not equal to y
- x > y // x is greater than y
- x < y // x is less than y
- x >= y // x is greater than or equal to y
- x <= y // x is less than or equal to y

Although these operations are probably familiar to you, the syntax C++ uses is a little different from mathematical symbols like =, 6= and ≤. A common error is to use a single = instead of a double ==. Remember that = is the assignment operator, and == is a comparison operator. Also, there is no such thing as =< or =>. The two sides of a condition operator have to be the same type. You can only compare ints to ints, doubles to doubles and so on. In classic C++ you cannot compare strings, but it is possible in WCM C++ because of its string extensions.

Alternative Execution

A second form of conditional execution is alternative execution, in which there are two possibilities, and the condition determines which one gets executed. The syntax looks like:

```
if ( iX < 0 )
```

```
{
	Max.Winks ();
}
else
{
	Sleep ( 0.5 );
}
```

If iX is negative, then character Max is going to wink. If iX is not negative, we just do nothing for half a second (the same time as default winking takes). Since the condition must be true or false, exactly one of the alternatives will be executed

Chained Conditionals

Sometimes you want to check for a number of related conditions and choose one of several actions. One way to do this is by chaining a series of ifs and elses:

```
if ( iX < 0 )
{
	Max.Winks ();
}
else if ( iX > 0 )
{
	Sleep ( 0.5 );
}
else
{
	Max.Says ( "Hello" );
	Sleep ( 0.5 );
}
```

These chains can be as long as you want, although they can be difficult to read if they get out of hand. One way to make them easier to read is to use standard indentation, as demonstrated in these examples. If you keep all the statements and squiggly-braces lined up, you are less likely to make syntax errors and you can find them more quickly if you do.

Nested Conditionals

In addition to chaining, you can also nest one conditional within another. We could have written the previous example as:

```
if ( iX < 0 )
{
	Max.Winks ();
}
else
{
	if ( iX > 0 )
	{
		Sleep ( 0.5 );
	}
	else
	{
		Max.Says ( "Hello" );
		Sleep ( 0.5 );
	}
}
```

There is now an outer conditional that contains two branches. The first branch contains a simple output statement, but the second branch contains another if statement, which has two branches of its own. Fortunately, those two branches are both output statements, although they could have been conditional statements as well.

Notice again that indentation helps make the structure apparent, but nevertheless, nested conditionals get difficult to read very quickly. In general, it is a good idea to avoid them when you can.

On the other hand, this kind of nested structure is common, and we will see it again, so you better get used to it.

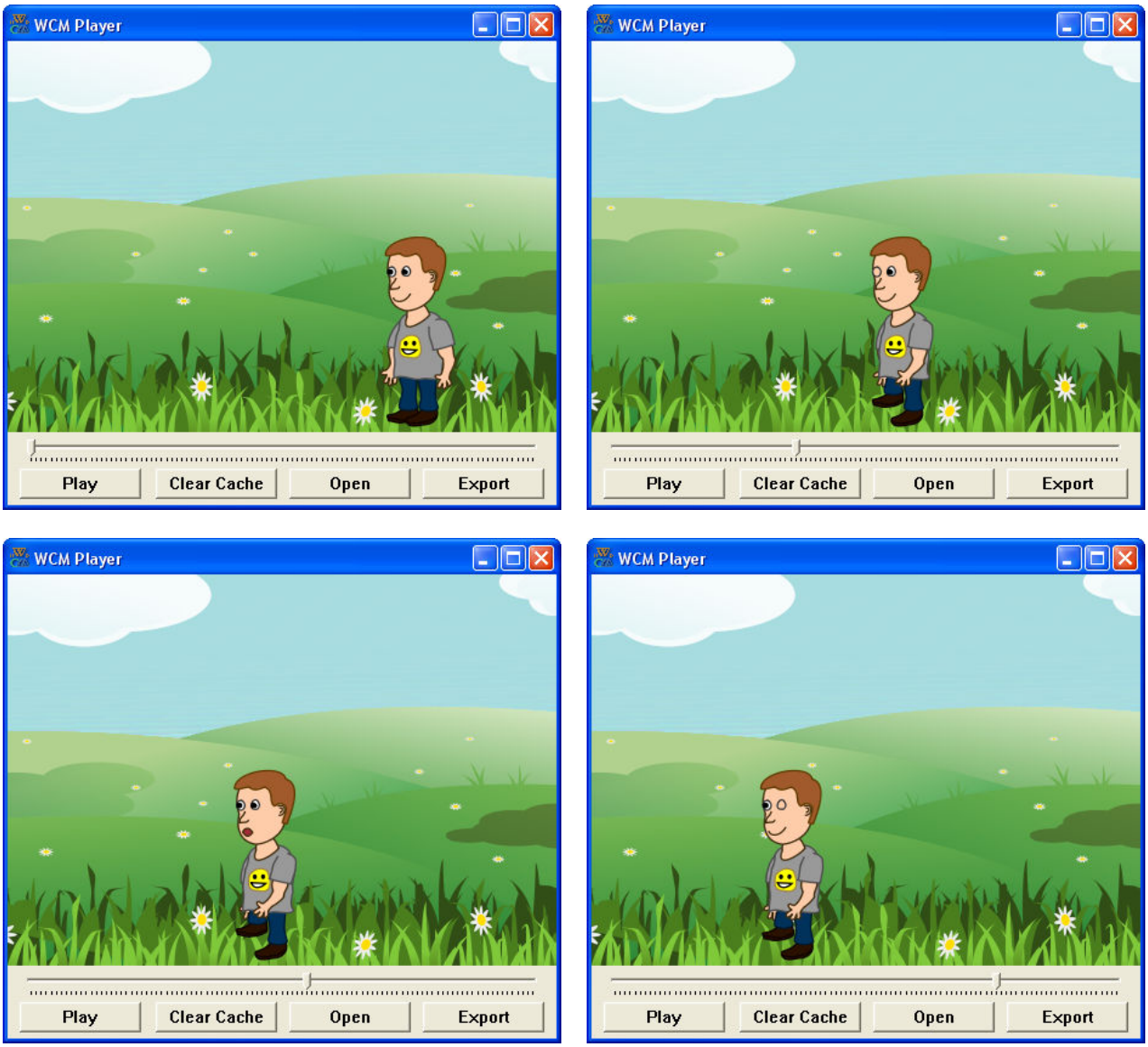
Working Example

This example may look little artificial. We'll be able to make it better after getting familiar with iterations and loops. Anyway, let's make a simple cartoon where a boy character is walking for 5 seconds and every half a second he will do some actions. He will wink with his right eye until reaching the screen center, and then he'll say "Switch" and start winking with his left eye:

```
#include <boy.h>

void DoWinking ( Boy &Who )
{
	if ( Who.GetX () < -0.1 )
	{
		Who.WinksLeft ();
	}
	else if ( Who.GetX () > 0.1 )
	{
		Who.WinksRight ();
	}
	else
	{
		Who.Says ( "Switch" );
		Sleep ( 0.5 );
	}
}

void Scene1 ()
{
	Image Back ( "backgrounds/landscape.svg" );
	Back.SetVisible ();

	Boy Max;
	Max.SetVisible ( true );
	Max.SetPos ( 200, 290 );
	Max.GoesTo ( -200, 290, 5 );

	SetTime ( 0 );

	DoWinking ( Max );
	DoWinking ( Max );
	DoWinking ( Max );
	DoWinking ( Max );
	DoWinking ( Max );
	DoWinking ( Max );
	DoWinking ( Max );
	DoWinking ( Max );
```

```
            DoWinking ( Max );
            DoWinking ( Max );
    }
```

Please note that we compare results of method GetX with 0.1 and -01, not with zero. This is because GetX returns a double value, not an int. During floating point operations precision may be lost a little bit and we do not want to miss the moment when character should say "Switch". You should see the following movie:



Fruitful Functions

Recursion

I've already mentioned that it is legal for one function to call another, and we have seen several examples of that. I neglected to mention that it is also legal for a function to call itself. It may not be

obvious why that is a good thing, and often it isn't, but it turns out to be one of the things a program can do. It is also one of the required subjects covered in the Advanced Placement tests for computer programming.

Note / Caution: There are a few well known pitfalls associated with recursive programming. The first, and most obvious, is that you can program a sequence of functional calls that never ends (until you shut off your computer). This is the most likely problem area with WCM. If you do use recursion, the recursive sequence should be very carefully thought out.

Second, many programs in general (though not WCM scripts) allow real time input from the keyboard, mouse or an external source. Should an interrupt occur during the recursive sequence, the results may be unpredictable but are seldom good. A similar situation occurs if many different program parts (called threads) are running at the same time (again not an issue with WCM). There are very stringent requirements for programming in this environment, which are well beyond the scope of this book, which should be carefully studied before attempting to program in a multi-threaded environment, especially using recursion.

For example, look at the following function:

```
void CountDown ( int iN )
{
	if ( iN == 0 )
	{
		ShowText ( "START!!!" );
	}
	else
	{
		ShowText ( iN );
		CountDown ( iN-1 );
	}
}
```

The name of the function is CountDown and it takes a single integer as a parameter. If the parameter is zero, it outputs the word "START!!!" Otherwise, it outputs the parameter and then calls a function named countdown – itself – passing iN-1 as an argument.

What happens if we call this function like this:

```
void Scene1 ()
{
	CountDown ( 3 );
}
```

- The execution of countdown begins with n=3, and since n is not zero, it outputs the value 3, and then calls itself...
  - The execution of countdown begins with n=2, and since n is not zero, it outputs the value 2, and then calls itself...
    - The execution of countdown begins with n=1, and since n is not zero, it outputs the value 1, and then calls itself...
      - The execution of countdown begins with n=0, and since n is zero, it outputs the word "START!!!" and then returns.
    - The countdown that got n=1 returns.
  - The countdown that got n=2 returns.
- The countdown that got n=3 returns.

And then you are back to Scene1 (what a trip). So the total output looks like this:

Recursion is a very powerful feature of C++ but it is not used very often in cartoons. We do not want to spend a lot of time in this book teaching you about recursion, but you are welcome to do your own experiments. If you do, be careful.

The Return Statement

The return statement allows you to terminate the execution of a function before you reach the end. One reason to use it is to avoid unnecessary conditional statements. For example we can rewrite DoWinking like this:

```
void DoWinking ( Boy &Who )
{
	if ( Who.GetX () < -0.1 )
	{
		Who.WinksLeft ();
		return;
	}
	if ( Who.GetX () > 0.1 )
```

```
        {
              Who.WinksRight ();
              return;
        }

        Who.Says ( "Switch" );
        Sleep ( 0.5 );
}
```

Return Values

Some of the built-in functions we have used, like the math functions, have produced results. That is, the effect of calling the function is to generate a new value, which we usually assign to a variable or use as part of an expression. For example:

```
double dE = Exp (1.0);
double dHeight = dRadius * Sin ( dAngle );
```

But so far all the functions we have written have been void functions; that is, functions that return no value. When you call a void function, it is typically on a line by itself, with no assignment:

```
ShowTime ();
PrintTime ( 11, 59 );
```

In this chapter, we are going to write functions that return things, which I will refer to as **fruitful** functions, for want of a better name. The first example is area, which takes a double as a parameter, and returns the area of a circle with the given radius:

```
double GetArea ( double dRadius )
{
      double dPi = 3.1415926;
      double dArea = dPi * dRadius * dRadius;
      return dArea;
}
```

The first thing you should notice is that the beginning of the function definition is different. Instead of void, which indicates a void function, we see double, which indicates that the return value from this function will have type double.

Also, notice that the last line is an alternate form of the return statement that includes a return value. This statement means, "return immediately from this function and use the following expression as a return value." The expression you provide can be arbitrarily complicated, so we could have written this function more concisely:

```
double GetArea ( double dRadius)
{
      return 3.1415926 * dRadius * dRadius;
}
```

On the other hand, temporary variables like dArea often make debugging easier. For example if you are unsure if your function is functioning properly you can add a ShowText (dArea); statement right before the return statement to verify its value in cartoon before it gets too complicated to find mistakes later.

In either case, the type of the expression in the return statement must match the return type of the function. In other words, when you declare that the return type is double, you are making a promise that this function will eventually produce a double. If you try to return with no expression, or an expression with the wrong type, the compiler will take you to task.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
double GetAbsoluteValue ( double dX )
{
	if ( dX < 0 )
	{
		return -dX;
	}
	else
	{
		return dX;
	}
}
```

Since these returns statements are in an alternative conditional, only one will be executed. Although it is legal to have more than one return statement in a function, you should keep in mind that as soon as one is executed, the function terminates without executing any subsequent statements.

Code that appears after a return statement, or any place else where it can never be executed, is called **dead code**. Since it can never be executed, good programming practice would be to delete it to avoid confusion.

If you put return statements inside a conditional, then you must guarantee that every possible path through the program hits a return statement. For example:

```
double GetAbsoluteValue ( double dX )
{
	if ( dX < 0 )
	{
		return -dX;
	}
	else if ( dX > 0 )
	{
		return dX;
	} // WRONG!!
}
```

This program is not correct because if dX happens to be 0, then neither condition will be true, and the function will end without hitting a return statement. As a result, the program may compile and run, but the return value when dX == 0 could be anything, and will probably be different in different environments.

Composition

As you should expect by now, once you define a new function, you can use it as part of an expression, and you can build new functions using existing functions. For example, what if someone gave you two points, the center of the circle and a point on the perimeter, and asked for the area of the circle?

Let's say the center point is stored in the variables dXC and dYC, and the perimeter point is in dXP and dYP. The first step is to find the radius of the circle, which is the distance between the two points. We can write a function for that:

```
double GetDistance ( double dX1, double dY1, double dX2, double dY2)
{
	double dX = dX2 - dX1;
	double dY = dY2 - dY1;
```

```
        double dSquared = dX*dX + dY*dY;
        double dResult = Sqrt ( dSquared );
        return dResult;
}
```

The second step is to find the area of a circle with that radius, and return it. Fortunately we already have a function for that. Wrapping all that up in a function we get:

```
double Fred ( double dXC, double dYC, double dXP, double dYP )
{
        double dRadius = GetDistance ( dXC, dYC, dXP, dYP );
        double dResult = GetArea ( dRadius );
        return dResult;
}
```

The name of this function is Fred, which may seem odd. I will explain why in the next section. The temporary variables in these two functions are useful for development and debugging, but once everything is working we can make it more concise by composing the function calls:

```
double GetArea ( double dRadius)
{
        return 3.1415926 * dRadius * dRadius;
}
double GetDistance ( double dX1, double dY1, double dX2, double dY2)
{
        double dX = dX2 - dX1;
        double dY = dY2 - dY1;
        return dSquared = Sqrt ( dX*dX + dY*dY );
}
double Fred ( double dXC, double dYC, double dXP, double dYP )
{
        return = GetArea ( GetDistance ( dXC, dYC, dXP, dYP ) );
}
```

Overloading

In the previous section you might have noticed that Fred and GetArea perform similar functions – finding the area of a circle – but take different parameters. For GetArea, we have to provide the radius; for Fred we provide two points.

If two functions do the same thing, it is natural to give them the same name. In other words, it would make more sense if Fred were called GetArea.

Having more than one function with the same name, which is called **overloading**, is legal in C++ as long as each version takes different parameters. So we can go ahead and rename Fred:

```
double GetArea ( double dXC, double dYC, double dXP, double dYP )
{
        return = GetArea ( GetDistance ( dXC, dYC, dXP, dYP ) );
}
```

This looks like a recursive function, but it is not. Actually, this version of area is calling the other version. When you call an overloaded function, C++ knows which version you want by looking at the arguments that you provide. If you write:

```
double dX = GetArea ( 3.0 );
```

C++ goes looking for a function named GetArea that takes a double as an argument, and so it uses the first version. If you write:

```
double dX = GetArea ( 1.0, 2.0, 4.0, 6.0 );
```

C++ uses the second version of GetArea.

Although overloading is a useful feature, it should be used with caution. You might get yourself nicely confused if you are trying to debug one version of a function while calling a different one.

Note: The above reminds me of one of the cardinal rules of debugging: make sure that the version of the program you are looking at is the version of the program that is running! Some time you may find yourself making one change after another in your program, and seeing the same thing every time you run it. This is a warning sign that for one reason or another you are not running the version of the program you think you are. To check, stick in an output statement (it doesn't matter what it says) and make sure the behavior of the program changes accordingly.

Note: Additionally, your program code should always contain a comment indicating the date, and possibly a number, defining the current revision. If several programmers are working on a program, a formal revision control system should be implemented – it is far more efficient than scrapping a lot of code because it was based on out of date versions.

Boolean Values

The types we have seen so far are pretty big. There are a lot of integers in the world, and even more floating-point numbers. By comparison, the set of characters is pretty small. Well, there is another type in C++ that is even smaller. It is called **boolean** , and the only values in it are true and false. We were already using these values without knowing that they are boolean.

Without thinking about it, we have been using boolean values for the last couple of chapters. The condition inside an if statement or a while statement is a boolean expression. Also, the result of a comparison operator is a Boolean value. For example:

```
if ( Max.dX < 0 )
{
      // do something
}
```

The operator < compares two doubles and produces a boolean value.

The values true and false are keywords in C++, and can be used anywhere a boolean expression is called for. For example:

```
if ( true )
{
      // always executed
}
if ( false )
{
      // never executed
}
```

Boolean Variables

As usual, for every type of value, there is a corresponding type of variable. In C++ the boolean type is called bool. Boolean variables work just like the other types:

```
bool bStore;
bStore  = true;
bool bTestResult = false;
```

The first line is a simple variable declaration; the second line is an assignment, and the third line is a combination of a declaration and as assignment, called an initialization.

As I mentioned, the result of a comparison operator is a boolean, so you can store it in a bool variable:

```
bool bEvenFlag = ( iN % 2 == 0 ); // true if iN is even
bool bPositiveFlag = ( dX > 0 ); // true if dX is positive
```

and then use it as part of a conditional statement later:

```
if ( bEvenFlag)
{
      ShowText ( "iN was even when I checked it" );
}
```

A variable used in this way is called a flag, since it flags the presence or absence of some condition.

Note: As with the char type, boolean variables are actually stored as integers, 0 for false and 1 for true. It is not recommended to use other than boolean operators on these variables, however, because the results may be unpredictable.

Logical (Boolean) Operators

There are three logical operators in C++: AND, OR and NOT, which are denoted by the symbols &&, || and !. The semantics (meaning) of these operators is similar to their meaning in English. For example dX > 0 && dX < 10 is true only if dX is greater than zero AND less than 10.

bEvenFlag || iN%3 == 0 is true if either of the conditions is true, that is, if bEvenFlag is true OR the number is divisible by 3.

Finally, the NOT operator has the effect of negating or inverting a bool expression, so !bEvenFlag is true if bEvenFlag is false; that is, if the number is odd.

Logical operators often provide a way to simplify nested conditional statements. For example, it is easy to write the following code:

```
if ( iX > 0 )
{
      if ( iX < 10 )
      {
            ShowText ( "iX is a positive single digit" );
      }
}
```

using a single conditional:

```
if ( iX > 0 && iX < 10 )
{
      ShowText ( "iX is a positive single digit" );
}
```

Converting Bools to Strings

Boolean values can be converted to strings automatically the save way as integers or doubles:

```
string sConverted = true; // sConverted will be set to "true"
```

This may not be very useful in cartoons but may help you with debugging. Anytime you want to check a boolean variable you can do this using ShowText which will convert a boolean value to string automatically:

```
ShowText (dX > 0 && dX < 10 );
```

Please keep in mind that this is also a WCM C++ extension and does not work in classic C++.

Bool Functions

Functions can return bool values just like any other type, which is often convenient for hiding complicated tests inside functions. For example:

```
bool IsSingleDigit ( int iX )
{
	if ( iX >= 0 && iX < 10)
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

The name of this function is IsSingleDigit. It is common to give boolean functions names that sound like yes/no questions. The return type is bool, which means that every return statement has to provide a bool expression.

The code itself is straightforward, although it is a bit longer than it needs to be. Remember that the expression x >= 0 && x < 10 has type bool, so there is nothing wrong with returning it directly and avoiding the if statement altogether:

```
bool IsSingleDigit ( int iX )
{
	return ( iX >= 0 && iX < 10);
}
```

You can call this function in the usual ways:

```
void Scene1 ()
{
	…
	ShowText ( IsSingleDigit ( 2 ) );
	bool bBigFlag = ! IsSingleDigit ( 17 );
	…
}
```

The first line outputs the value true because 2 is a single-digit number. The second line assigns the value true to bBigFlag because 17 is not a single-digit number.

The most common use of bool functions is inside conditional statements:

```
if ( IsSingleDigit ( iX ) )
{
	ShowText ( "iX is little" );
}
else
```

```
{
        ShowText ( "iX is big" );
}
```

Leap of Faith

Following the flow of execution is one way to read programs, but as you saw in the previous section, it can quickly become labyrinthine. An alternative is what I call the "leap of faith." When you come to a function call, instead of following the flow of execution, you assume that the function works correctly and returns the appropriate value.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call Cos or Exp, you don't examine the implementations of those functions. You just assume that they work, because the people who wrote the built-in libraries were good programmers.

Well, the same is true when you call one of your own functions. For example, we wrote a function called IsSingleDigit that determines whether a number is between 0 and 9. Once we have convinced ourselves that this function is correct – by testing and examination of the code – we can use the function without ever looking at the code again.

One major risk is that the "trusted" function may become inadvertently corrupted by it's interaction with other program elements. There is a concept called encapsulation, which addresses itself to minimizing this risk. It will be discussed briefly in later sections, but a full discussion is beyond the scope of this book. However, if you are planning on writing large general C++ programs, you should become very familiar with this concept.

Iteration or Loops

Iteration

One of the things computers are often used for is the automation of repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. This is also important for animated cartoons. For example you may want your cartoon character to wink 3 times in a row.

We have seen a program that uses recursion to perform repetition using CountDown function. This type of repetition is called iteration, and C++ provides several language features that make it easier to write iterative programs.

The two features we are going to look at are the while statement and the for statement.

The While Statement

Using a while statement, we can rewrite CountDown:

```
void CountDown ( int iN )
{
        while ( iN > 0 )
        {
                ShowText ( iN );
                iN = iN - 1;
        }

        ShowText ( "START!!!" );
}
```

You can almost read a while statement as if it were English. What this means is, “While iN is greater than zero, continue displaying the value of iN and then reducing the value of iN by 1. When you get to zero, output the word "START!!!"”.

More formally, the flow of execution for a while statement is as follows:

1. Evaluate the condition in parentheses, yielding true or false.
2. If the condition is false, exit the while statement and continue execution at the next statement.
3. If the condition is true, execute each of the statements between the squiggly-braces, and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed. The statements inside the loop are called the body of the loop.

The body of the loop should change the value of one or more variables so that, eventually, the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an infinite loop. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop.

In the case of CountDown, we can prove that the loop will terminate because we know that the value of n is finite, and we can see that the value of iN gets smaller each time through the loop (each iteration), so eventually we have to get to zero. In other cases it is not so easy to tell.

Increment and decrement operators

Incrementing and decrementing are such common operations that C++ provides special operators for them. The ++ operator adds one to the current value of an int, char or double, and -- subtracts one. Neither operator works on a string, and neither should be used on a bool variable.

Technically, it is legal to increment a variable and use it in an expression at the same time. For example, you might see something like:

```
ShowText ( iX++ );
```

Looking at this, it is not clear whether the increment will take effect before or after the value is displayed. Because expressions like this tend to be confusing, I would discourage you from using them. In fact, to discourage you even more, I’m not going to tell you what the result is. If you really want to know, you can try it.

We can rewrite CountDown using -- operator:

```
void CountDown ( int iN )
{
	while ( iN > 0 )
	{
		ShowText ( iN );
		iN--;
	}

	ShowText ( "START!!!" );
}
```

It is a common error to write something like

```
iN = iN++; // WRONG!!
```

Unfortunately, this is syntactically legal, so the compiler will not warn you. The effect of this statement is to leave the value of iN unchanged (which should give you a hint as to the answer to the earlier question). This is often a difficult bug to track down.

Remember, you can write iN = iN +1; or you can write iN++; but you shouldn't mix them.

For Loops

The while loop we were using above have three elements common for most of the loops.

1. Most of them start by initializing a variable. We used a parameter variable called iN in CountDown, which is obviously initialized during the function call.
2. Most of them have a test, or condition, that depends on that variable
3. Inside the loop most of them do something to that variable, like increment or decrement it.

This type of loop is so common that there is an alternate loop statement, called for, that expresses it more concisely. The general syntax looks like this:

```
for ( INITIALIZER; CONDITION; INCREMENTOR )
{
      BODY
}
```

This statement is exactly equivalent to

```
INITIALIZER;
while ( CONDITION )
{
      BODY
      INCREMENTOR
}
```

except that it is more concise and, since it puts all the loop-related statements in one place, it is easier to read. For example:

```
for ( int i = 0; i < 4; i++ )
{
      ShowText ( i );
}
```

is equivalent to

```
int i = 0;
while ( i < 4 )
{
      ShowText ( i );
      i++;
}
```

You may notice that I used an integer variable called i in the example above. It is common to use simple variable names like i, j, k and so on in for loops. It is also possible to have embedded loops like this:

```
for ( int i = 0; i < 4; i++ )
{
      for ( int j = 0; j < 3; j++ )
```

```
        {
                ShowText ( i * j );
        }
}
```

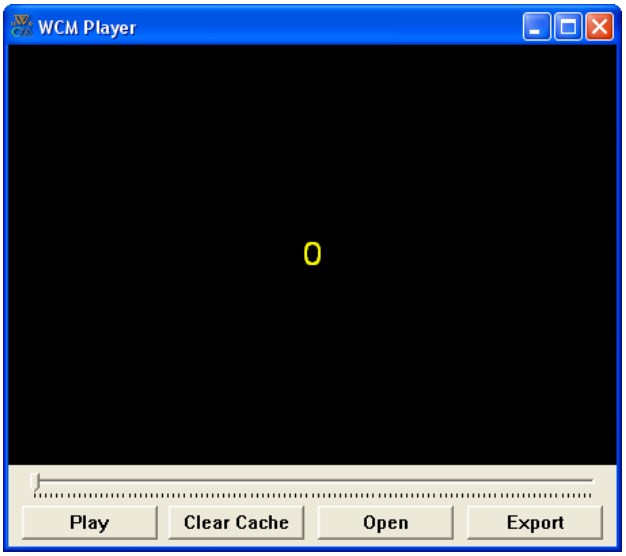
Historical Note: The i, j, k convention actually came from the first commercial high level scientific language, Fortran 2. (Fortran 1 was what we would now call "beta" software.) In Fortran 2 all integer variables, and only integers had to begin with i, j, or k.

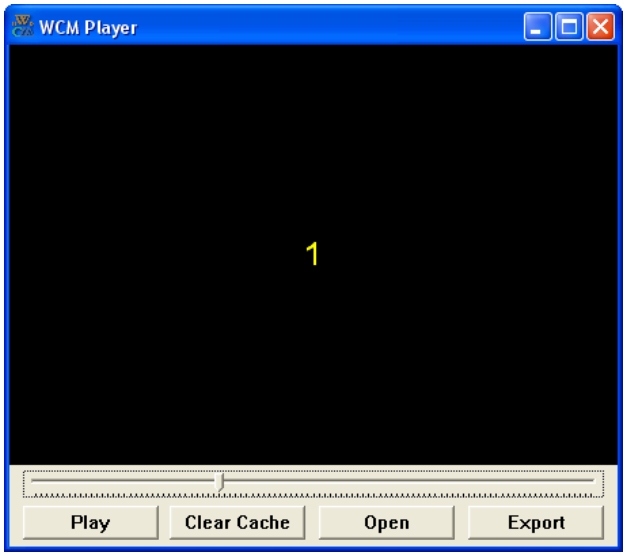
Break Statement

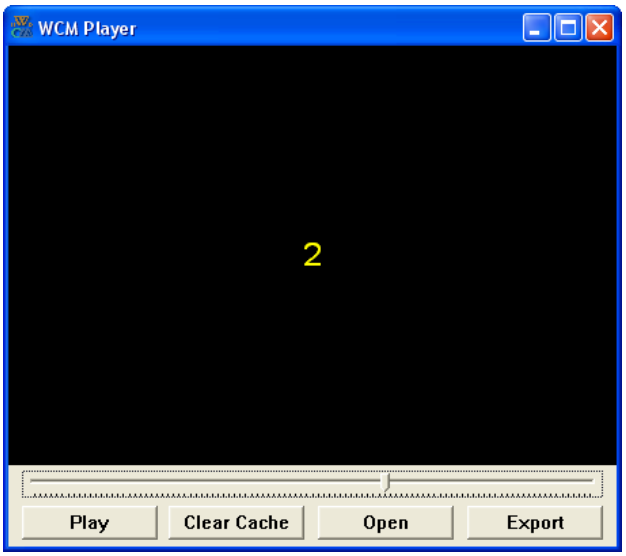
The break statement allows you to exit from a loop to a statement following it. In the example below the loop will be repeated only 3 times instead of 10:

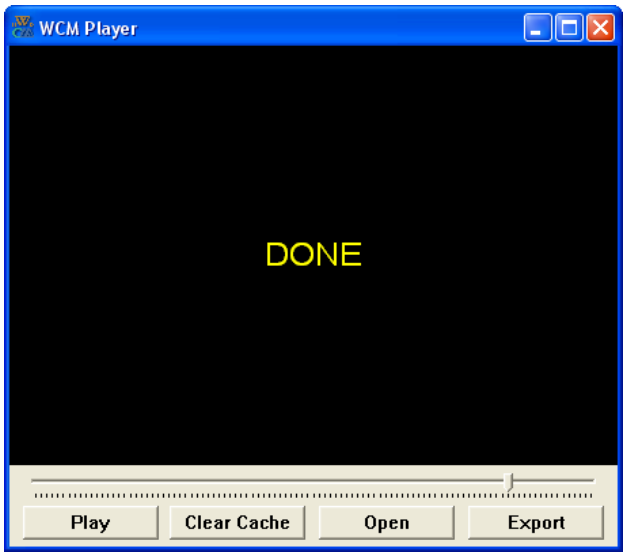
```
void Scene1 ()
{
        for ( int i = 0; i < 10; i++ )
        {
                if ( i == 3 )
                {
                        break;
                }

                ShowText ( i );
        }

        ShowText ( "DONE" );
}
```

And the result will be like this:







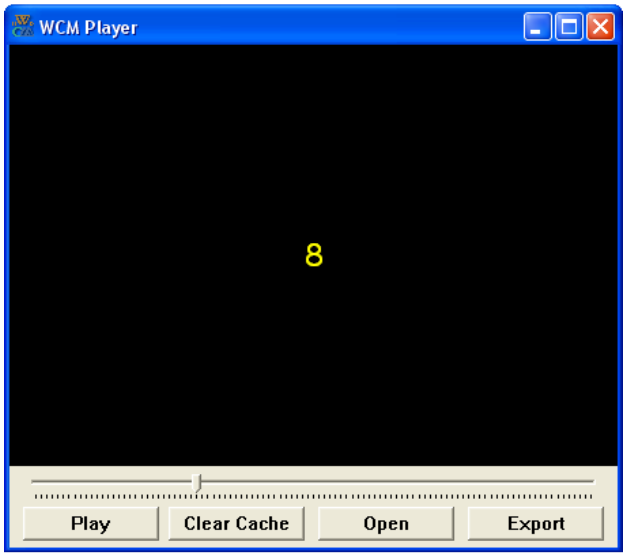


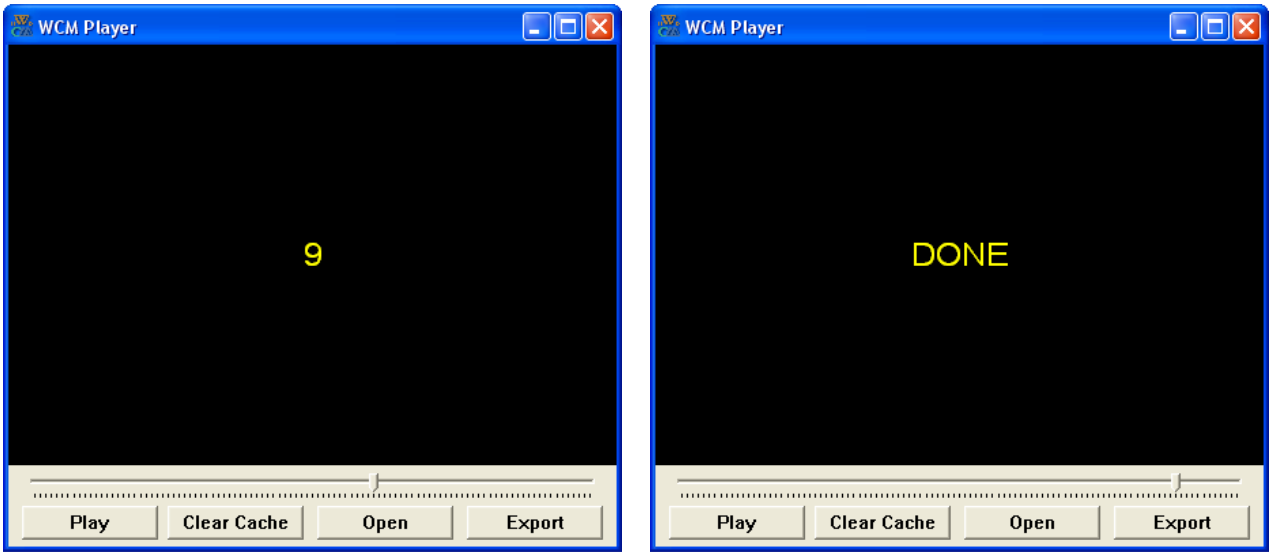
Continue Statement

The continue statement allows you to skip the remaining loop body and start a next iteration from the top of a loop. In the example below the first 7 iterations will be skipped:

```
void Scene1 ()
{
	for ( int i = 0; i < 10; i++ )
	{
		if ( i < 7 )
		{
			continue;
		}

		ShowText ( i );
	}

	ShowText ( "DONE" );
}
```

And the result will be like this:





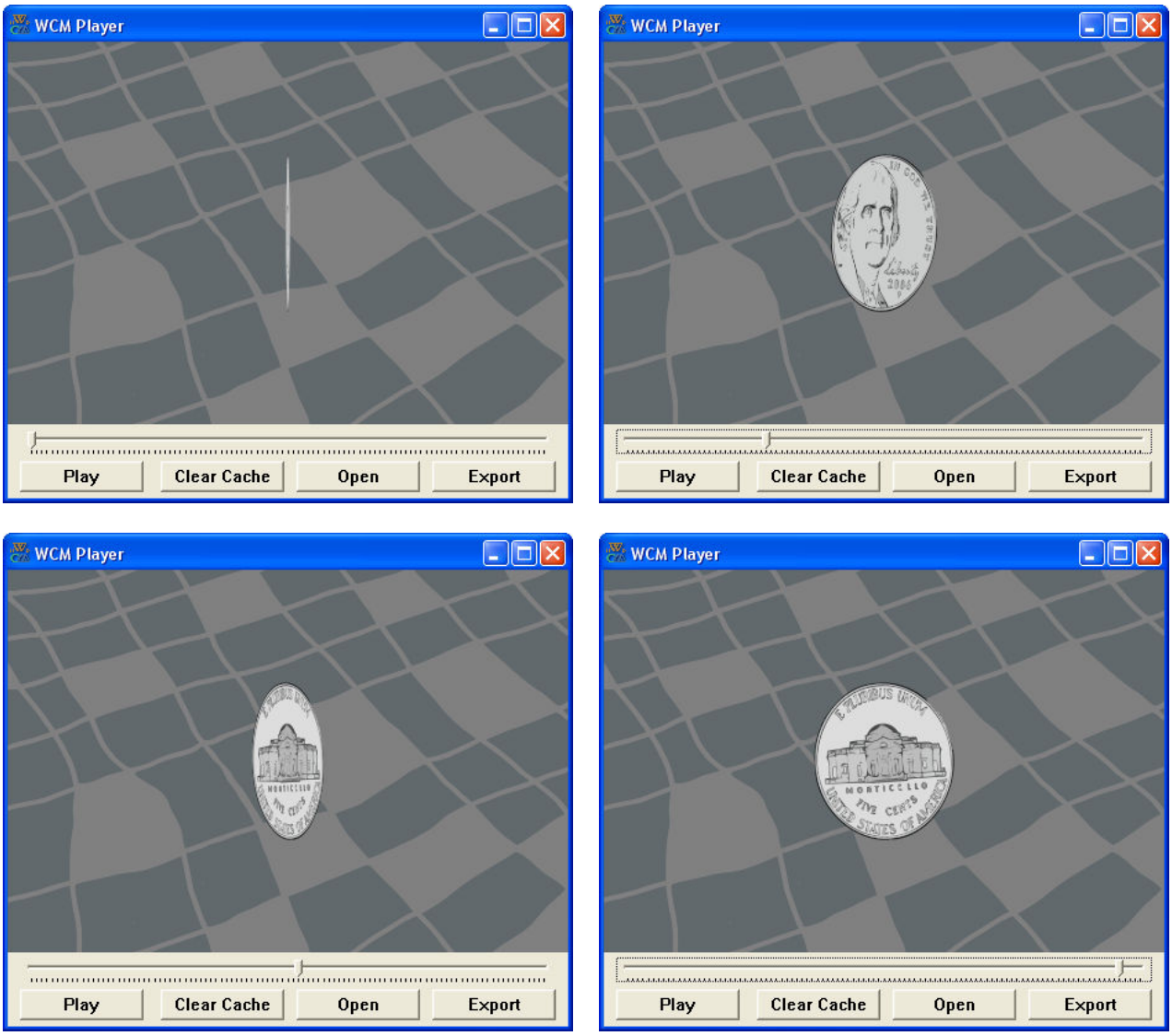
Cartoonish Example

All the above examples may look artificial and not very usable for cartoons. But actually loops are pretty usable in cartoons. Let's compile an example of rotating coin to demonstrate it:

```
void Scene1 ()
{
	// set some variable
	double dScale = 0.5; // scale factor of a coin
	double dQuarterTurn = 0.2; // time for 1/4th of coin rotation
	double dXPos = 0;  // coin position
	double dYPos = 0; // coin position


	// setup a background
	Image Back ( "backgrounds/flagstone_floor.svg" );
	Back.SetVisible ();

	Image CoinHead ( "wcm/nickel-coin-head.svg" ); // coin head image
	Image CoinTail ( "wcm/nickel-coin-tail.svg" ); // coin tail image

	// set coin images' position and scale factor
	CoinHead.SetPos ( dXPos, dYPos );
	CoinTail.SetPos ( dXPos, dYPos );
	CoinHead.SetScale ( dScale );
	CoinTail.SetScale ( dScale );

	for ( int i=0; i<5; i++ )
	{
		// head image is visible
		CoinHead.SetVisible ( true );
		CoinTail.SetVisible ( false );
		// simulate the first half of rotation cycle
		CoinHead.SetAngle ( 0 );
		CoinHead.SetXScale ( 0.02 * dScale );
		CoinHead.ChangeXScale ( 1 * dScale, dQuarterTurn );
		CoinHead.SetAngle ( 5 );
		CoinHead.ChangeXScale ( 0.02 * dScale, dQuarterTurn );
		CoinHead.SetAngle ( 0 );
```

```
                // tail image is visible
                CoinHead.SetVisible ( false );
                CoinTail.SetVisible ( true );
                // simulate the second half of rotation cycle
                CoinTail.SetAngle ( 0 );
                CoinTail.SetXScale ( 0.02 * dScale );
                CoinTail.ChangeXScale ( 1 * dScale, dQuarterTurn );
                CoinTail.SetAngle ( -5 );
                CoinTail.ChangeXScale ( 0.02 * dScale, dQuarterTurn );
                CoinTail.SetAngle ( 0 );
            }

    }
```

The result should look like this?



Do you think this is useful in cartoons?

Classes and Objects

Introduction

C++ is generally considered an object-oriented programming (OOP) language, which means that it provides features that support object-oriented programming. So far, though, we have not taken advantage of the C++ features that support object-oriented programming (except using internal WCM objects). For the most part they provide an alternate syntax for doing things we have already done, but in many cases this alternate syntax is more concise and more accurately conveys the structure of the program.

The basic features of an **object-oriented programming language** include:

1. Programs are made up of a collection of class definitions and function definitions, where most of the functions operate on specific kinds of classes (or, more specifically, instances of classes or objects).
2. Each class definition corresponds to some concept and/or type of object in the real world, and the functions that operate on that structure correspond to the ways real-world objects interact.
3. Each instance of a class (i.e., each object) has its own unique local data and capabilities that are not shared with other parts of the program.
4. Classes are hierarchical and sub-classes inherit the properties of the parent class.

The above features, especially inheritance and protected data, will be discussed in the following sections. They are the key to implementing something called polymorphism. Polymorphism is a rather abstract concept and is difficult to define precisely – but "you know it when you see it." Basically the concept is that each object is essentially complete in itself and can be used by other parts of the program without knowing the implementation details of the object itself. (You don't have to know the details of the Boy class or the IhumanCharacterSideView parent class to tell a boy named Max to move to a specific point, for example.)

Polymorphism, as well as classes and objects, are not really necessary for small programs or such things as complex numerical computations or even some data base manipulations. After all, it took over thirty years from the first high level languages before OOP concepts started to emerge. Even now, C++ supports the older procedural features of its predecessor, C. Java, and to some extent C#, are regarded as more "pure" OOP languages. Java in particular is currently the preferred language for academic use (C# is fairly tightly tied to the Windows operating system and is not particularly portable). However C++ remains the preferred language in industry largely because of the amount of legacy code available and to a lesser extent because it will also run well on UNIX based systems – in fact its predecessor, C, was actually developed for use in developing UNIX.

So if it isn't really necessary, why is OOP regarded as the preferred programming method (with very few exceptions) today. The answer is simple – over the last 50+ years, as computers have become more powerful and more memory has become available, program complexity has increased accordingly. Trying to keep track of all of the details of the entire program in one place becomes nearly impossible. More importantly, without OOP a change in some aspect of the computing environment, such as a new graphics card, would require massive reprogramming instead of modifications only to some members of the graphics interface class.

Classes vs. Objects

Most of the data types we have been working with represent a single value – an integer, a floating-point number, a boolean value, etc. With these familiar things, there is no confusion between a variable type (int or bool) and its current value (5 or true).

Internal WCM objects (Image, Text and characters) are different in the sense that they are made up of smaller pieces. They contain information about objects in a cartoon scene like their IDs, associated text strings or image URLs, parts and some other information.

We may even want to create our own objects like these. Depending on what we are doing, we may want to treat an object as a single entity, or we may want to access its parts (or instance variables). This flexibility is useful.

The key concept here is to make class definition for the desired type of object or group of objects. The class is analogous to a variable type – for example we can define the class "Boy" and then use several different boy objects (say Max, Joe and Sam) in a scene. The specific objects are called instances (or instantiations) of the class "Boy" and can have different positions, scale factors, etc. This concept will be explored in later sections but first the basic principles involved will be illustrated by developing a simple (and somewhat artificial) class "Point", which defines a class of objects in two dimensional space such as "Starting Point" and "End Point."

The "Point" Class and Objects

As a simple example of the class/object relationship, consider a point in an animated movie. At one level, a point is two numbers (screen coordinates) that we treat collectively as a single object. In mathematical notation, points are often written in parentheses, with a comma separating the coordinates. For example in WCM (0, 0) indicates the origin, and (x, y) indicates the point x units to the right and y units down from the origin.

A natural way to represent a point in C++ is with two doubles, as will be done here. (For WCM specifically, someone may think that integers might be a better choice since there are no fractional pixels, but this changes when using zoom). The question, then, is how to group these two values into a compound object, or **class**. The answer is a class definition:

```
class Point
{
      public:
            double dX, dY;
};
```

Note that class definitions appear outside of any function definition, usually at the beginning of the program (after the include statements).

This definition indicates that there are two elements in this class, named dX and dY. These elements are called instance variables, for reasons I will explain a little later. The word public with a colon symbol at the end (:) means that the elements below are accessible for everybody. By contrast, you cannot access an internal WCM object's ID or any other parameter directly. This means that these parameters are not public. But we will discuss the protected data later.

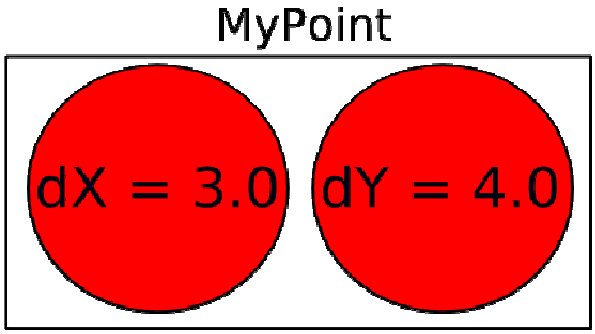
Note/Caution: it is a common error to leave off the semi-colon at the end of a class definition. It might seem odd to put a semi-colon after a squiggly-brace, but you'll get used to it.

Once you have defined the new class, you can create instances of type Point, called objects:

```
Point MyPoint;
MyPoint.dX = 3.0;
MyPoint.dY = 4.0;
```

The first line is a conventional variable declaration: MyPoint has type Point. The next two lines initialize the instance variables of the class. The "dot notation" used here is similar to the syntax for invoking a method on an internal WCM object, as in Max.SetPos (0, 290). One difference, however, is that function names are always followed by an argument list, even if it is empty.

The result of these assignments is shown in the following state diagram for the "MyPoint" object:



As usual, the name of the variable MyPoint appears outside the box and its value appears inside the box. In this case, that value is a compound object with two named **instance variables**.

Accessing instance variables

You can read the values of an instance variable using the same syntax we used to write them:

```
double dX = MyPoint.dX;
```

The expression MyPoint.dX means "go to the object named MyPoint and get the value of dX". In this case we assign that value to a local variable named dX. Notice that there is no conflict between the local variable named dX and the instance variable named dX. The purpose of dot notation is to identify which variable you are referring to unambiguously.

You can use dot notation as part of any C++ expression, so the following are legal.

```
Max.SetPos ( MyPoint.dX + 5.0, MyPoint.dY * 1.1 );
double dDistance =
       Sqrt ( MyPoint.dX * MyPoint.dX + MyPoint.dY * MyPoint.dY );
```

In this sample code, the first line changes the position of a WCM character and second line calculates the distance of a point from cartoon's origin. Note that this code as written would not be part of a class definition. If it were, the distance variable would be written as MyPoint.dDistance.

Operations on Objects

Most of the operators we have been using on other types, like mathematical operators (+, -, etc.) and comparison operators (==, >, etc.), generally do not work on objects. It is possible to define the meaning of these operators for the new type (this is called operator overloading), but we won't cover that in this book.

On the other hand, the assignment operator does work for objects. It can be used in two ways: (1) to initialize the instance variables of an object or (2) to copy the instance variables from one object to another. An initialization looks like this:

```
Point MyPoint = { 3.0, 4.0 };
```

The values in squiggly braces get assigned to the instance variables of the classes one by one, in order. So in this case, dX gets the first value and dY gets the second.

Unfortunately, this syntax can be used only in an initialization, not in an assignment statement. So the following is illegal:

```
Point MyPoint;
MyPoint = { 3.0, 4.0 }; // WRONG !!
```

It is legal to assign one object to another. For example:

```
Boy Max;
Point MyPoint = { 3.0, 4.0 };
Point AnotherPoint = MyPoint;
Max.SetPos ( AnotherPoint.dX, AnotherPoint.dY );
```

sets the character's coordinates to (3.0, 4.0).

Note/Caution: While assigning one object to another works fine for simple classes and is legal syntax, it is not a good idea to do this with more complex ones. Sometimes it is really not clear what to expect from such assignment. Here is one basic rule - please do not assign internal WCM objects (Image, Text and characters) to each other. While the following will compile:

```
Image First ( "backgrounds/landscape.svg" );
Image Second ( "sport/beach_ball.svg" );
First = Second;
```

it is not quite clear what the result will do in your script, because internal WCM C++ objects of this kind are just interfaces to the actual objects in your cartoon. After the assignment on third line we will have two interface definitions referring to the same actual image "sport/beach_ball.svg" and there will be no interface to work with first actual image "backgrounds/landscape.svg" anymore. This is probably not what you want to accomplish with the assignment!

As another example, we can write a function GetDistance which takes two point objects as parameters:

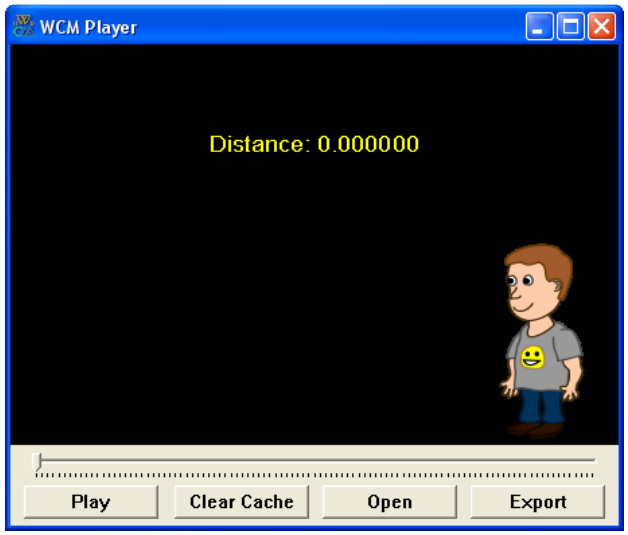
```
double GetDistance ( Point First, Point Second )
{
	double dDX = Second.dX - First.dX;
	double dDY = Second.dY - First.dY;
	return Sqrt ( dDX * dDX + dDY * dDY );
}
```

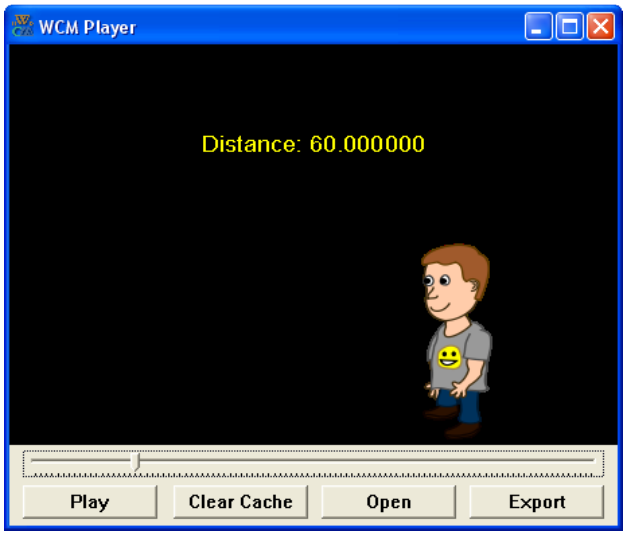
Let's compile a small complete example to demonstrate this. The example below will print the distance of a walking character from the starting point every second:

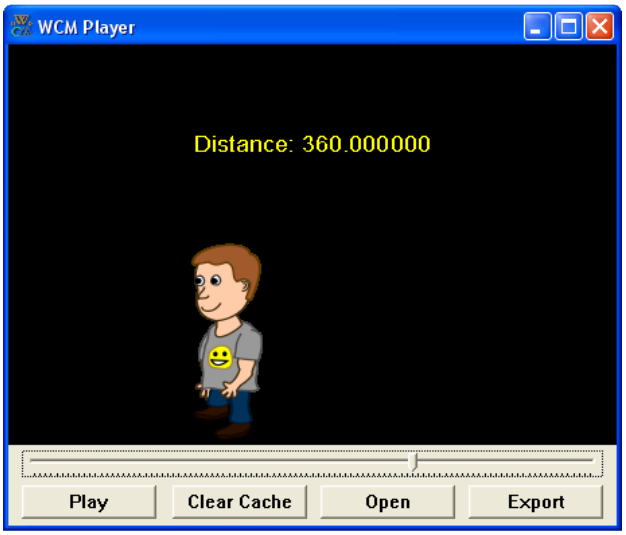
```
#include <boy.h>

class Point
{
	public:
		double dX, dY;
};

double GetDistance ( Point First, Point Second )
{
```

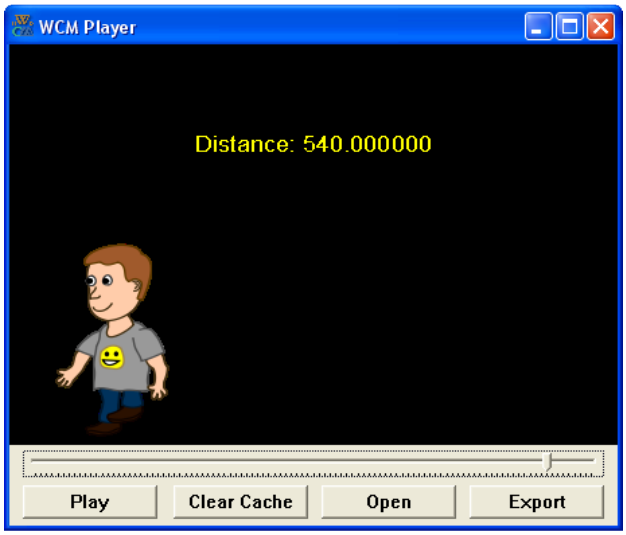
```
        double dDX = Second.dX - First.dX;
        double dDY = Second.dY - First.dY;
        return Sqrt ( dDX * dDX + dDY * dDY );
}

// Note that the classes and function are defined before Scene1

void Scene1 ()
{
        Point Start = { 300, 290 };
        Point End = { -300, 290 };
        Point Current;

        Boy Max;
        Max.SetVisible ();
        Max.SetPos ( Start.dX, Start.dY );
        Max.GoesTo ( End.dX, End.dY, 10 );

        SetTime ( 0 );

        Text Distance; // text will be assigned later
        Distance.SetVisible ();
        Distance.SetPos ( 0, -150 );

        for ( int i=0; i<10; i++ )
        {
                Current.dX = Max.GetX ();
                Current.dY = Max.GetY ();

                Distance.SetText ( "Distance: " +
                        GetDistance ( Start, Current ) );
                Sleep ( 1 );
        }
}
```

Here is what you should see after compilation:



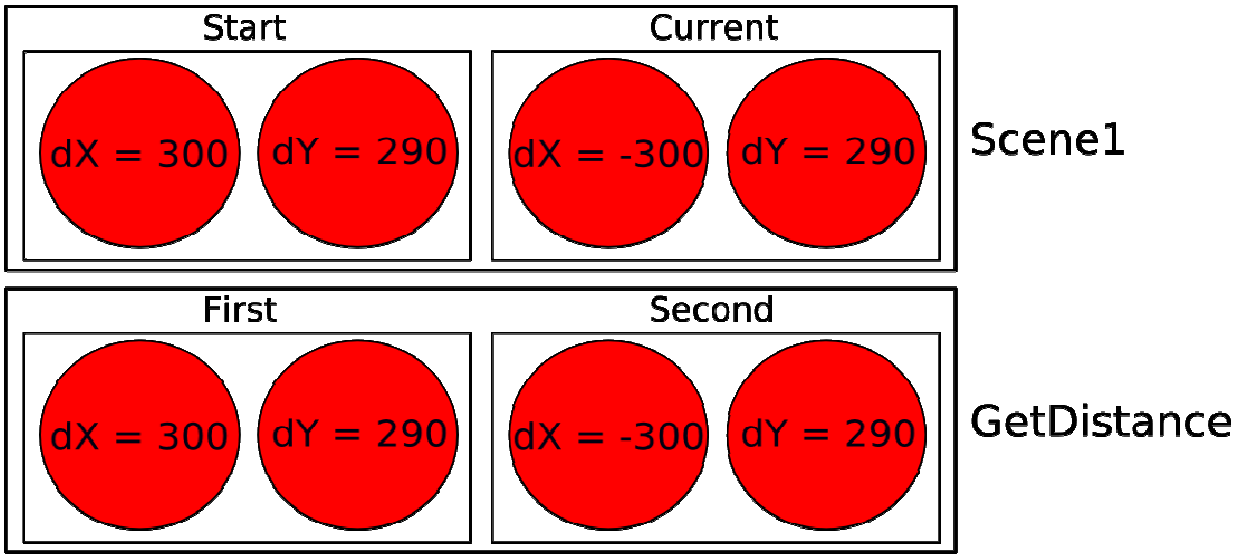






Call Pass by Value

When you pass an object as an argument, remember that the argument and the parameter are not the same variable. Instead, there are two variables (one in the caller and one in the callee) that have the same value, at least initially. For example we can draw a stack diagram for the last GetDistance call in our last example:



In this case, the variable values are the same. However if GetDistance happened to change one of the instance variables of First or Second, it would have no effect on Start and Current. Of course, there is no reason for GetDistance to modify its parameter, so this isolation between the two functions is appropriate.

This kind of parameter-passing is called “pass by value” because it is the value of the class (or other type) that gets passed to the function.

Call Pass by Reference

An alternative parameter-passing mechanism that is available in C++ is called "pass by reference." This mechanism makes it possible to pass an object to a procedure and modify it.

For example, you can reflect a point around the 45-degree line by swapping the two coordinates. The most obvious (but incorrect) way to write a Reflect function is something like this:

```
void Reflect ( Point P ) // WRONG !!
{
      double dTemp = P.dX;
      P.dX = P.dY;
      P.dY = dTemp;
}
```

But this won't work, because the changes we make in reflect will have no effect on the caller.

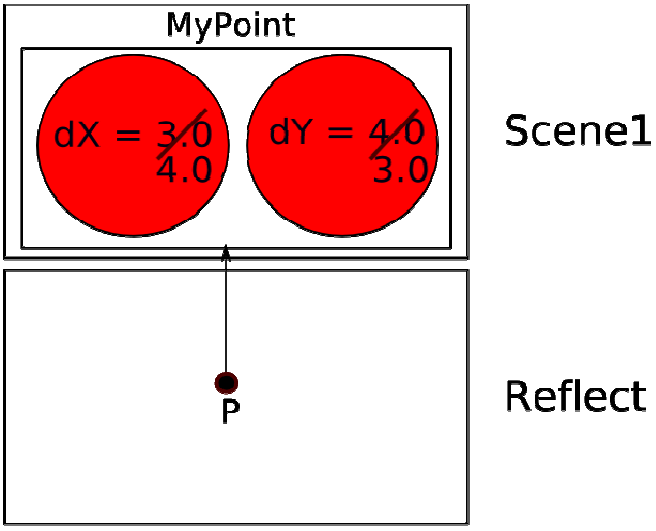
Instead, we have to specify that we want to pass the parameter by reference. We do that by adding the ampersand (&) operator to the parameter declaration:

```
void Reflect ( Point &P )
{
      double dTemp = P.dX;
      P.dX = P.dY;
      P.dY = dTemp;
}
```

Now we can call the function in the usual way:

```
Max.SetPos ( MyPoint.dX, MyPoint.dY );
Reflect ( MyPoint );
Max.GoesTo ( MyPoint.dX, MyPoint.dY, 5 );
```

Here's how we would draw a stack diagram for this program:



The parameter P is a reference to the object named MyPoint. The usual representation for a reference is a dot with an arrow that points to whatever the reference refers to.

The important thing to see in this diagram is that, unlike the "pass by value example," any changes that Reflect makes in P will also affect MyPoint.

Passing objects by reference is more versatile than passing by value, because the callee can modify the object. It is also faster, because the system does not have to copy the whole object. On the other hand, it is less safe, since it is harder to keep track of what gets modified where.

Nevertheless, in C++ programs, almost all objects are passed by reference almost all the time. In this book I will follow that convention since the chances for confusion in WCM are minimal compared to the advantages of this approach – but beware that this is not always the case. It is especially very convenient and safe to pass internal WCM objects (Image, Text and characters) by reference.

Classes as Return Types

You can write functions that return objects. For example, we can write GetReflectedCopy which takes a Point and returns a reflected Point:

```
Point GetReflectedCopy ( Point &From )
{
      Point Result;
      Result.dX = From.dY;
      Result.dY = From.dX;
      return Result;
}
```

To call this function, we have to pass a Point as an argument (notice that it is being passed by reference), and assign the return value to another Point variable:

```
Point Start = { 0, 200 };
Point End = GetReflectedCopy ( Start );
Max.SetPos ( Start.dX, Start.dY );
Max.GoesTo ( End.dX, End.dY, 5 );
```

Constant Parameters

In the GetReflectedCopy example we passed an argument by reference and we were not planning to change it. We passed it only for convenience and speed. If you are not planning to modify the value it is safer to use const keyword. You can rewrite the function like this:

```
Point GetReflectedCopy ( const Point &From )
{
      Point Result;
      Result.dX = From.dY;
      Result.dY = From.dX;
      return Result;
}
```

The const keyword does not change the behavior of GetReflectedCopy but does make it safer – remember there are risks associated with pass by reference. If by accident you try to change its value, the compiler will complain and will not allow you to do this.

It is much easier to catch the problem when compiler complains rather than let GetReflectedCopy to change its argument by mistake and then debug the problem later.

Passing Other Types by Reference

It's not just objects that can be passed by reference. All the other types we've seen can, too. For example, to swap two integers, we could write something like:

```
void Swap ( double &dX, double &dY )
{
      double dTemp = dX;
      dX = dY;
      dY = dTemp;
}
```

We would call this function in the usual way:

```
double dX = 0;
double dY = 100;
Max.SetPos ( dX, dY );
Swap ( dX, dY );
Max.GoesTo ( dX, dY, 5 );
```

When people start passing things like integers by reference, they often try to use an expression as a reference argument. For example:

```
double dX = 0;
double dY = 100;
Swap ( dX, dY + 1 ); // WRONG!!
```

This is not legal because the expression dY + 1 is not a variable – it does not occupy a location that the reference can refer to. It is a little tricky to figure out exactly what kinds of expressions can be passed by reference. For now a good rule of thumb is that reference arguments have to be variables.

Member Functions or Methods

As mentioned earlier, C++ is generally considered an **object-oriented programming (OOP) language**. One of the features of OOP is that it provides a way to define objects that behave very much like real world objects. For example, the class Point we defined before, obviously corresponds to the way people think about coordinates on the screen or in the movie, and the operations we defined (GetDistance, Reflect and GetReflectedCopy) correspond to the sorts of things people do with these coordinates.

So far, though, we have not taken advantage of the features C++ provides to support object-oriented programming (except using internal WCM objects). Strictly speaking, these features are not necessary. For the most part they provide an alternate syntax for doing things we have already done, but in many cases the alternate syntax is more concise and more accurately conveys the structure of the program.

For example, at first glance there is no obvious connection between class Point and functions GetDistance, Reflect and GetReflectedCopy. With some examination, it is apparent that these functions take at least one point object as an argument.

This observation is the motivation for **member functions** or so called **methods**. Member functions differ from the other functions we have written in two ways:

1. When we call the function, we invoke it on an object, rather than just call it. People sometimes describe this process as “performing an operation on an object,” or “sending a message to an object.”
2. The function is declared inside the class definition, in order to make the relationship between the structure and the function explicit.

Note: It was mentioned earlier that the names “function” and “method” are used differently in different languages and, in informal discussions, are essentially the same. “Method” and “Member Function” as used in this section have very specific C++ definitions.

In the next few sections, we will take some functions and transform them into member functions or C++ methods. One thing you should realize is that this transformation is purely mechanical; in other words, you can do it just by following a sequence of steps.

Anything that can be done with a member function can also be done with a nonmember function (sometimes called a free-standing function). But sometimes there is an advantage to one over the other. If you are comfortable converting from one form to another, you will be able to choose the best form for whatever you are doing.

Converting Functions to Methods

In previous section we defined a class Point and wrote a function named Reflect. To call this function we had to pass a Point object as a parameter. To make Reflect into a **member function** we should eliminate this parameter and place it inside the class definition.

As a result, inside the function we no longer have a parameter named P. Instead, we have a current object, which is the object this function is invoked on. And we can access all its instance variables by their names without using a dot symbol.

```
class Point
{
      public:

            double dX, dY;

            void Reflect ()
            {
                  double dTemp = dX;
                  dX = dY;
                  dY = dTemp;
            }
};
```

In order to invoke the new version of print, we have to invoke it on a Point object:

```
Max.SetPos ( MyPoint.dX, MyPoint.dY );
MyPoint.Reflect ();
Max.GoesTo ( MyPoint.dX, MyPoint.dY, 5 );
```

Similarly we can convert GetReflectedCopy to a method:

```
class Point
{
            …
```

```
        Point GetReflectedCopy ()
        {
                Point Result;
                Result.dX = dY;
                Result.dY = dX;
                return Result;
        }

        …
};
```

Remember, that GetReflectedCopy as a function used to have parameter From? And for safety we added a keyword const to indicate that we are not going to modify this parameter? It is also possible to indicate that we are not going to modify an object and its instance variables a member function or method was invoked on. The same keyword const can be placed after a member function declaration:

```
class Point
{
        …

        Point GetReflectedCopy () const
        {
                Point Result;
                Result.dX = dY;
                Result.dY = dX;
                return Result;
        }

        …
};
```

Again, keyword const does not change the behavior of a method. It just makes this method safer. If you try to change an instance variable (dX or dY) by an accident, the compiler will complain and you will be able to avoid problems later.

Constructors

When we declare a Point object, the instance variables dX and dY remain uninitialized until we specially assign values to them. It would probably make sense to set them to zero by default to specify an origin point of your cartoon. You can do this using a **constructor**. Constructor is a special member function (unique to OOP languages) which has the same name as its class and has no return type:

```
class Point
{
        …

        Point ()
        {
                dX = 0;
                dY = 0;
        }

        …
};
```

Constructor is called automatically when you declare an object like this:

```
void Scene1 ()
{
	…
	Point MyPoint;
	…
}
```

Constructors can also accept parameters. For example it is pretty common to assign dX and dY immediately after an object declaration. It is possible to do this using a constructor also:

```
class Point
{
		…

		Point ()
		{
			dX = 0;
			dY = 0;
		}

		Point ( double a_dX, double a_dY )
		{
			dX = a_dX;
			dY = a_dY;
		}


		…
};
void Scene1 ()
{
	…
	Point MyPoint; // initialized with zero values
	Point AnotherPoint ( 400, 300 ); // initialized with 300 and 400
	…
}
```

You can see two more things from the example above. First of all there could be several constructors in a class if they accept different parameters. You can also note that we used strange parameter names like a_dX and a_dY. This was done on purpose to distinguish between a method (constructor) parameters and instance variables dX and dY.

But what if we do not want a default constructor initializing our object with zeros? You can just delete it. But in this case you will no longer be able to declare an object without parameters:

```
class Point
{
		…

		Point ( double a_dX, double a_dY )
		{
			dX = a_dX;
			dY = a_dY;
		}


		…
```

```
};
void Scene1 ()
{
	…
	Point MyPoint; // WRONG!! There is no default constructor!
	Point AnotherPoint ( 400, 300 ); // OK
	…
}
```

It is a good practice to specify a default constructor with no parameters for most of the classes as well as constructors with parameters. If constructors are not specified, the C++ compiler (not the WCM compiler) will specify one, which can lead to problems beyond the scope of this book. Also beyond the scope of this book is the concept of a **destructor**, which is essentially the opposite of a **constructor** and is used in special circumstances to free up previously allocated memory space (garbage collection).

Initialize or Construct

Earlier we declared and initialized a Point object using squiggly-braces:

```
Point MyPoint = { 3.0, 4.0 };
```

Now, using constructors, we have a different way to declare and initialize:

```
Point MyPoint ( 3.0, 4.0 );
```

These two functions represent different programming styles, and different points in the history of C++. Maybe for that reason, the C++ compiler requires that you use one or the other, and not both in the same program.

If you define a constructor for a structure, then you have to use the constructor to initialize all new structures of that type. The alternate syntax using squiggly-braces is no longer allowed. Initializing using squiggly-braces is an older style of programming and is rarely used now. Please consider to use constructors for the most cases.

Protected Data

Sometimes you may want to hide implementation details from users or programmers that don't need to know them. More importantly, you may even want to hide or protect the implementation details from yourself to avoid accidental access to data.

For example let's try to create a class representing a callout balloon to display together with a character speech. It should contain 2 objects as members – an image object containing balloon picture and a text object containing a text to display. We'll use a callout balloon image from Web Cartoon Maker's online library:

```
class CalloutBalloon
{
	protected:

		Image BalloonImage;
		Text BalloonText;

	public:

		CalloutBalloon ( string sText )
```

```
        {
            BalloonImage.SetImage ( "wcm/callout.svg" );
            BalloonText.SetText ( sText );
            BalloonText.SetColor ( "000000" );

            SetPos ( 0, 0 );
        }

        void SetVisible ( bool bVisible )
        {
            BalloonImage.SetVisible ( bVisible );
            BalloonText.SetVisible ( bVisible );
        }

        void SetPos ( double dX, double dY )
        {
            BalloonImage.SetPos ( dX, dY );
            BalloonText.SetPos ( dX, dY - 20 ); // little above
        }
};
```

There are two sections of this definition, a protected part and a public part. The functions are public, which means that they can be invoked from other places. The instance variables are protected, which means that they can be read and written only by CalloutBalloon member functions. For example the following usage is wrong:

```
…
Scene1 ()
{
    CalloutBalloon MyBalloon ( "Hello" );

    // WRONG!! BalloonText is protected
    MyBalloon.BalloonText.SetPos ( 0,400 );
}
```

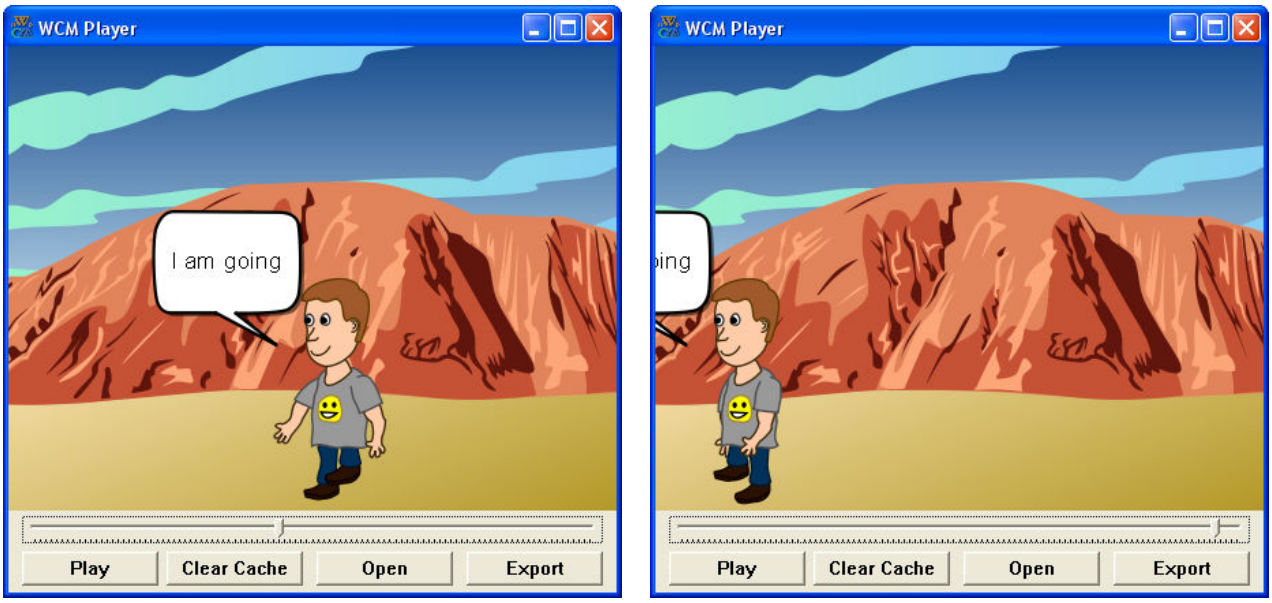
This is good because it makes you difficult to access and change position of BalloonImage or BalloonText individually and you are forced to use combined SetPos method changing position of both objects in synch.

Finally lets a compile an example of using the CalloutBalloon class:

```
#include <boy.h>

class CalloutBalloon
{
    protected:

        Image BalloonImage;
        Text BalloonText;

    public:

        CalloutBalloon ( string sText )
        {
            BalloonImage.SetImage ( "wcm/callout.svg" );
            BalloonText.SetText ( sText );
            BalloonText.SetColor ( "000000" );

            SetPos ( 0, 0 );
        }
```

```
        void SetVisible ( bool bVisible )
        {
              BalloonImage.SetVisible ( bVisible );
              BalloonText.SetVisible ( bVisible );
        }

        void SetPos ( double dX, double dY )
        {
              BalloonImage.SetPos ( dX, dY );
              BalloonText.SetPos ( dX, dY - 20 ); // little above
        }
};

void Scene1 ()
{
      Image Back ( "backgrounds/ayersrock.svg" );
      Back.SetVisible ();

      Boy Max;
      Max.SetVisible ( true );
      Max.SetPos ( 300,290 );
      Max.Says ( "I am going" );

      CalloutBalloon SpeechBalloon ( "I am going" );
      SpeechBalloon.SetVisible ( true );
      SpeechBalloon.SetPos ( 150, 0 );

      Max.GoesTo ( -300,290, 5 );

      SpeechBalloon.SetPos ( -450, 0 );
}
```

Try to compile the above cartoon to see the result:



Inheritance

One of the key features of C++ classes, and certainly one of the most useful for WCM, is **inheritance**. Inheritance allows creating classes which are derived from other classes, so that they automatically include its "parent's" member functions, character definitions and decals, plus its own, which may overwrite or supersede the definitions in the parent class. We will examine ways to do this in the next few sections, but first we will look at modifying the example presented in the previous section.

For example if we want to create an advanced callout balloon class based on implemented in previous section (to support scaling for example) we can do this by deriving a new class from CalloutBalloon:

```
…
// see the notes after this example for an explainatin of the ":" notation
class CalloutBalloonEx : public CalloutBalloon
{
      public:

            CalloutBalloonEx ( string sText ) : CalloutBalloon ( sText )
            {
            }

            void SetPos ( double dX, double dY )
            {
                  BalloonImage.SetPos ( dX, dY );
                  BalloonText.SetPos ( dX,
                                    dY - 20 * BalloonImage.GetYScale () );
            }

            void SetScale ( double dScale )
            {
                  BalloonImage.SetScale ( dScale );
                  BalloonText.SetScale ( dScale );
            }
```

```
};
```

Let's explain some things in the sample code:

1. 

Line class CalloutBalloonEx : public CalloutBalloon means that we declare a new sub-class CalloutBalloonEx which is derived from CalloutBalloon. Deriving means that all member functions and character definitions of CalloutBalloon will be accessible the same way in CalloutBalloonEx
2. Constructors are *not* inherited, however! We need to create a new constructor! Line CalloutBalloonEx (string sText) : CalloutBalloon (sText) declares a new constructor for class CalloutBalloonEx with one string parameter sText. Before we add any additional initialization, however, we want to call the constructor of inherited class CalloutBalloon by calling it with parameter sText. In this case, CalloutBalloonEx does not require any additional initialization so it has an empty body. But there are more complex examples possible where additional initialization is required.
3. It is possible to add new member functions to the new sib-class. For example, SetScale is a new member function.
4. It is also possible to overwrite the member functions of the parent class. SetPos was already declared in CalloutBalloon but here we define it to behave a little differently in CalloutBalloonEx in order to take into account the current scale factor.

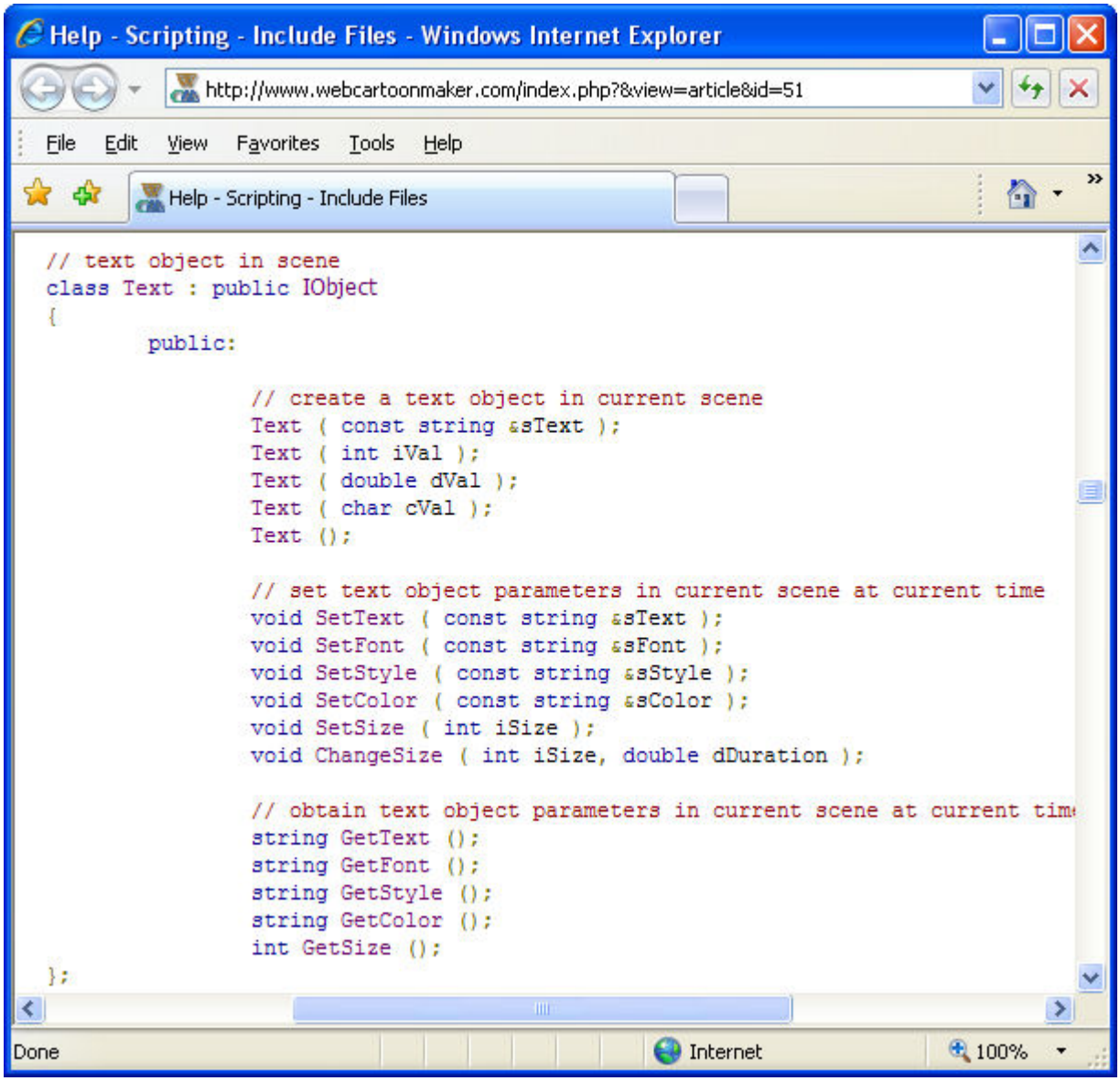
The new sub-class can be used in the same way as the parent class:

```
void Scene1 ()
{
	…

	Boy Max;
	Max.SetVisible ( true );
	Max.SetPos ( 300,290 );
	Max.Says ( "I am going" );

	CalloutBalloonEx SpeechBalloon ( "I am going" );
	SpeechBalloon.SetVisible ( true );
	SpeechBalloon.SetPos ( 150, 0 );
	SpeechBalloon.SetScale ( 0.7 );

	Max.GoesTo ( -300,290, 5 );

	SpeechBalloon.SetPos ( -450, 0 );
}
```

Custom Characters 

Introduction

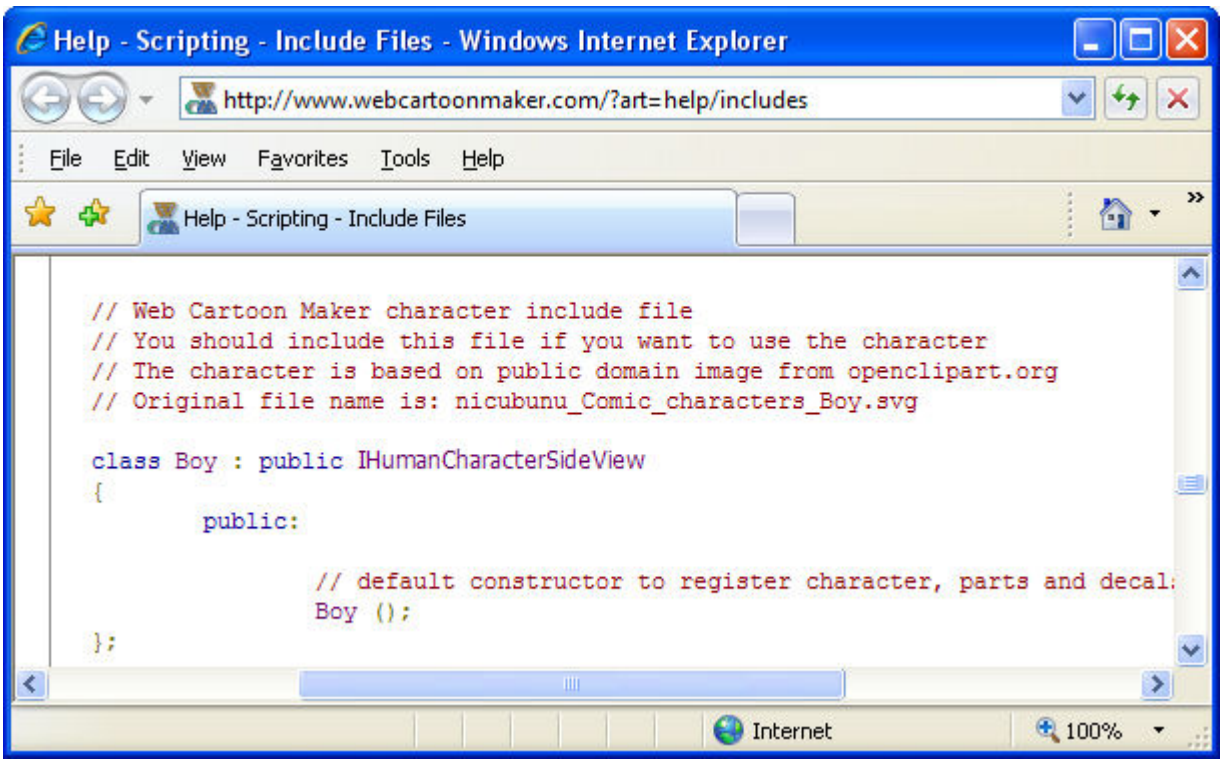
You probably already guessed that Image, Text and characters are classes. You can even find their declarations at http://www.webcartoonmaker.com/?art=help/includes:



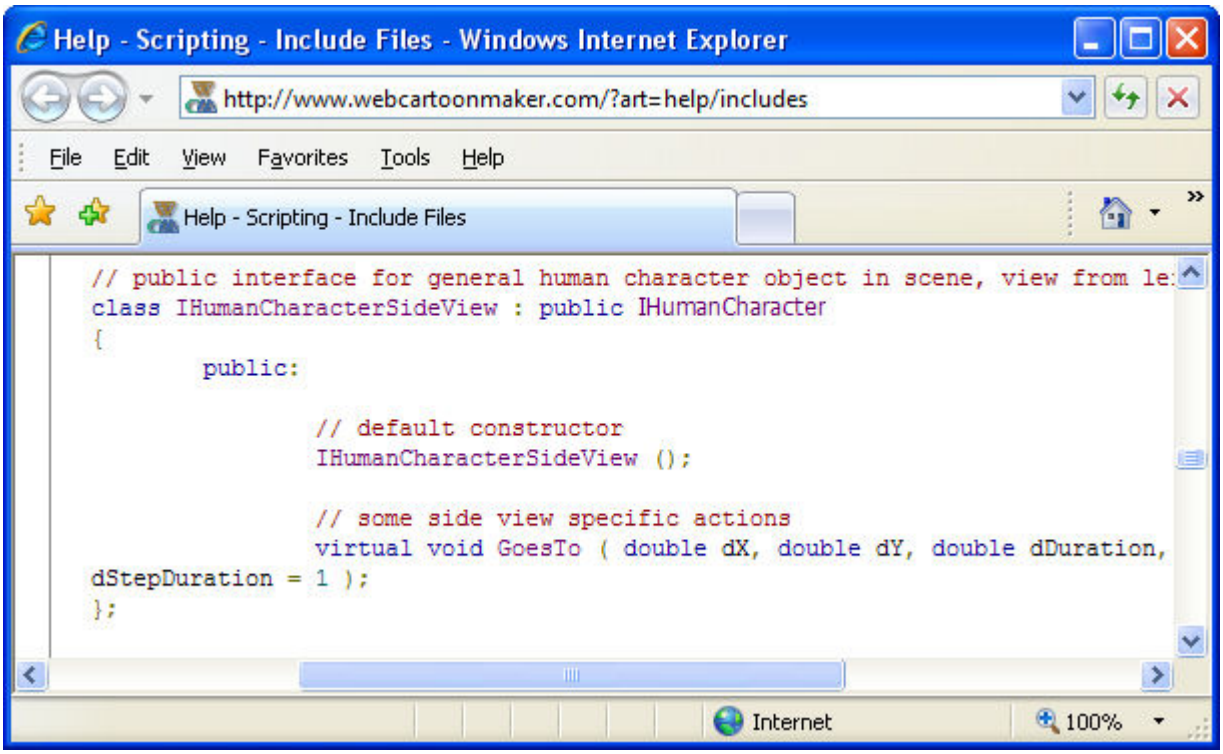
You will not find the actual code of these classes but only declarations of all members. This is because these classes are internal WCM C++ classes. But good news is that you can inherit them! For example we can find a declaration of character Boy and change his walking style!

The class hierarchy starts with the ICharacter class and incudes other classes class derived from ICharacter like ISpeakableCharacter, IHumanCharacter or IHumanCharacterSideView. You probably noted a first capital letter "I" in the names of these classes. We added these letters to indicate that these are "interface" (or abstract) classes. These classes are incomplete. They have some useful methods to register and work with characters; some of them have common parts declared but none of them have actual pictures assigned with parts. To make these classes useful you need to derive a new class, such as Boy from them.

Looking into character Boy declaration you will find out that it is derived from IHumanCharacterSideView:



IHumanCharacterSideView is, in turn, derived from the class IHumanCharacter:



IHumanCharacterSideView has a method GoesTo. (Please do not pay too much attention to word virtual, it just indicates that this method is being defined in an abstract class., a fact that is not particularly important for creating WCM cartoons.

Note: If you are interested in general C++ programming, you should study this concept as explained in a general C++ reference book or web site.

The method GoesTo takes 3 mandatory parameters and one optional parameter with default value of 1. dX and dY are the target coordinates. dDuration is the duration of walking in seconds. dStepDuration is the duration of one step in seconds. This method will be changed in the next section.

The full class hierarchy for Boy is:

IObject

|

ICharacter

|

ISpeakableCharacter

|

IHumanCharacter

|

IHumanCharacterSideView

|

Boy

Quite a list of "parents," each of which, through **inheritance,** provides additional functionality for the Boy class, which, unlike its parents, can actually be implemented, i.e. it can be used for actual WCM characters. We will illustrate the implementation of a new implementable class in a following section. For now, let's look at modifying the Boy class.

Changing Walking Style

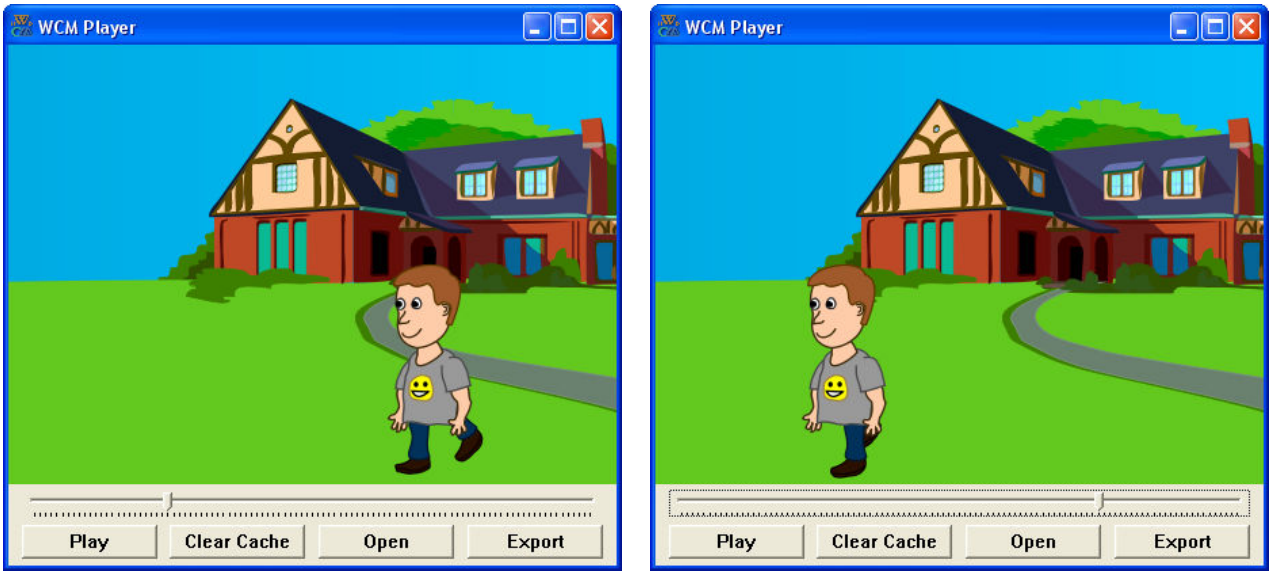
First we derive a new class from Boy and update the walking style:

```
#include <boy.h>

class BoyEx : public Boy
{
      public:

            BoyEx () : Boy ()
            {
            }
            void GoesTo ( double dX, double dY, double dDuration,
                          double dStepDuration = 1 )
            {
                  double dStartTime = GetTime (); // remember start time
                  SetPos ( GetX (), GetY () ); // initial control point

                  // lets split a step into 4 stages
                  for ( int i=0; i<dDuration * 4 / dStepDuration; i++ )
                  {
                        // set time for current stage
```

```
                        SetTime ( dStartTime + i * dStepDuration / 4 );

                        // if this is a first stage
                        if ( i%4 == 0 )
                        {
                              LeftLeg.SetShift ( 0, 0 );
                              LeftLeg.SetAngle ( 0 );
                              RightLeg.SetShift ( 0, 0 );
                              RightLeg.SetAngle ( 0 );
                              continue;
                        }

                        // if this is a second stage
                        if ( i%4 == 1 )
                        {
                              LeftLeg.SetShift ( 0, -30 );
                              LeftLeg.SetAngle ( -45 );
                              RightLeg.SetShift ( 0, 0 );
                              RightLeg.SetAngle ( 0 );
                              continue;
                        }

                        // if this is a third stage
                        if ( i%4 == 2 )
                        {
                              LeftLeg.SetShift ( 0, 0 );
                              LeftLeg.SetAngle ( 0 );
                              RightLeg.SetShift ( 0, 0 );
                              RightLeg.SetAngle ( 0 );
                              continue;
                        }

                        // if this is a fourth stage
                        if ( i%4 == 3 )
                        {
                              LeftLeg.SetShift ( 0, 0 );
                              LeftLeg.SetAngle ( 0 );
                              RightLeg.SetShift ( 0, -30 );
                              RightLeg.SetAngle ( -45 );
                        }
                  }

                  // ending control point
                  SetTime ( dStartTime + dDuration );
                  SetPos ( dX, dY );
                  LeftLeg.SetShift ( 0, 0 );
                  LeftLeg.SetAngle ( 0 );
                  RightLeg.SetShift ( 0, 0 );
                  RightLeg.SetAngle ( 0 );
            }
};

void Scene1 ()
{
      Image Back ( "backgrounds/house.svg" );
      Back.SetVisible ( true );

      BoyEx Max;
      Max.SetVisible ();
```

```
        Max.SetPos ( 300, 290 );
        Max.GoesTo ( -300, 290, 5 );
}
```

If you compile the above example you will see that walking style is changed and Boy bends his legs when walking and does not move his hands at all:



Although we have created a new sub-class for Boy, the Boy class is unchanged and can also be used to instantiate “boys” who will swing their arms when walking.

Customizing a Character's Decals

You already know how to change a decal image of a character's part using a SetDecal method. But you were able to change only between existing registered decals like "DEFAULT" or "WINK1". It is also possible to register a brand new decal or re-register an existing decal using a Part's RegisterDecal method. This method accepts two string parameters. The first one is a decal name (like "DEFAULT" or "WINK1" or any other name) and the second one is a path to an actual image. This MUST BE A FULLY QUALIFIED LOCAL PATH TO AN IMAGE ON YOUR HARD DRIVE like "c:\\myimage.svg" (it could also be a short path to an image in WCM character's library but you do not have direct access to these images). Please keep in mind that RegisterDecal can be used only once for every character's part. If you use it more than once then the latest instance will take precedence. When you register a part's decal you do this for the entire scene and it has effect from the beginning to the end of a cartoon scene. For example:

```
…
void Scene1 ()
{
        Boy Max;
        Max.SetVisible ();
        Max.Head.RegisterDecal ( "MY", "c:\\myimage.svg" );
        Max.Head.SetDecal ( "MY" );
        Sleep (1);
}
```

does exactly the same the same thing as

```
…
void Scene1 ()
```

```
{
	Boy Max;
	Max.SetVisible ();
	Max.Head.SetDecal ( "MY" );
	Max.Head.RegisterDecal ( "MY", "c:\\myimage.svg" );
	Sleep (1);
}
```

It may look like the second example is wrong because decal "MY" is unknown, but it works the same way because it does not matter if decal was registered before of after usage. For code readability reasons it is recommended to register decals before usage though.

You will discover some more "Register" methods soon. They all work the same way. They affect the entire cartoon scene and it does not matter when they were called. The latest call takes precedence. Also, since the scenes are function calls, the decal information is local to a given scene and given object, thus changing a decal in Scene1 will not affect Scene2 (much like stage actors never change costumes in the middle of a scene, but often do so between scenes) and will not affect another instances of the same class.

In the above example we registered a custom decal "MY", but it is possible to re-register existing decals, even the "DEFAULT" one:

```
…
void Scene1 ()
{
	Boy Max;
	Max.SetVisible ();
	Max.Head.RegisterDecal ( "DEFAULT", "c:\\myimage.svg" );
	Sleep (1);
}
```

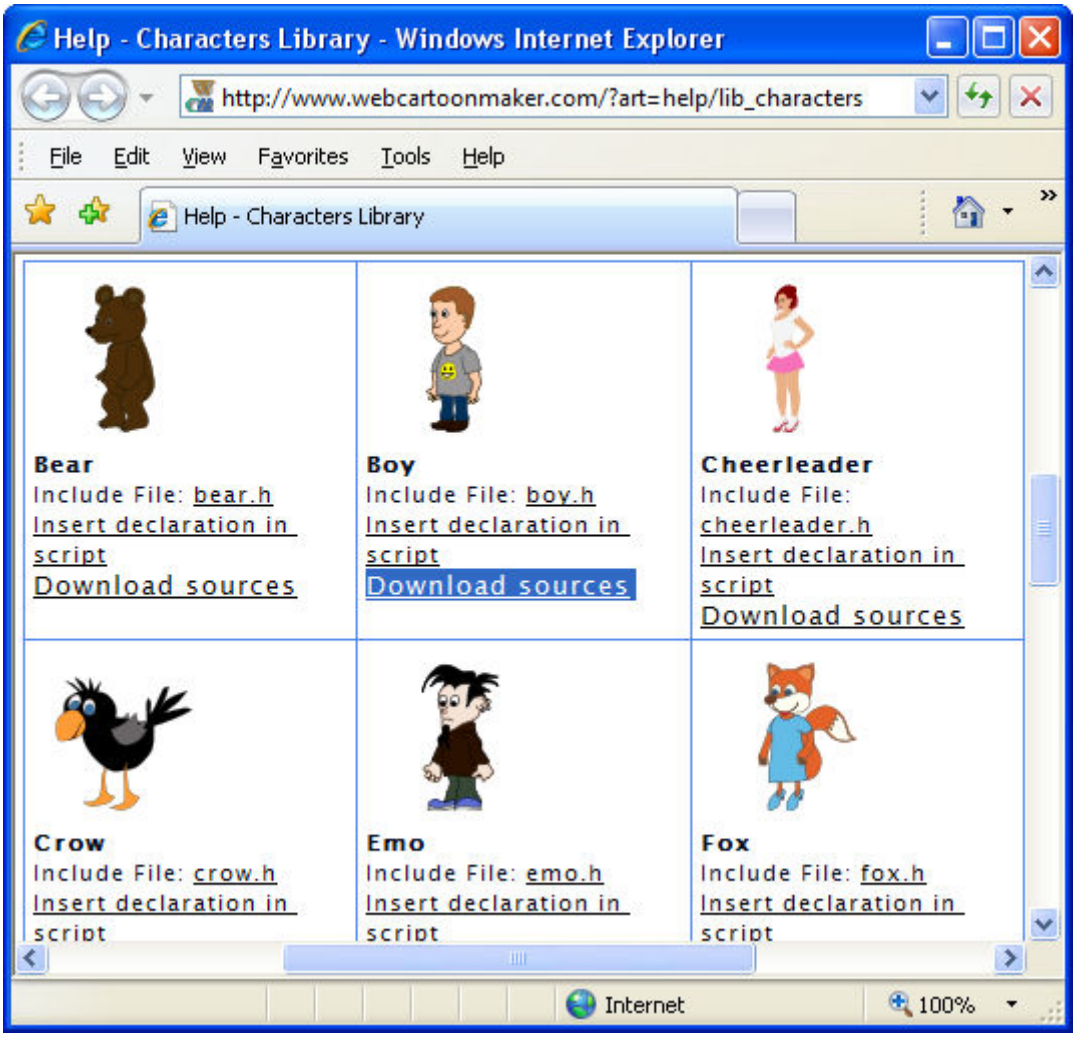
If we re-register the "DEFAULT" decal, then we even do not need to use SetDecal, because it is set to "DEFAULT" automatically.

There is even better place to register and re-register decals. You can derive a new character from existing one and register decals in constructor:

```
…
class BoyEx : public Boy
{
	public:

		BoyEx () : Boy ()
		{
			Head. RegisterDecal ( "DEFAULT", "c:\\myimage.svg" );
		}
};
…
```

Changing a Decal

You now know how to register a new decal. But how do you make a picture? What dimensions should it have? The easiest way to find out is to download a character's sources. You can go back to http://www.webcartoonmaker.com/?art=help/lib_characters and download the sources

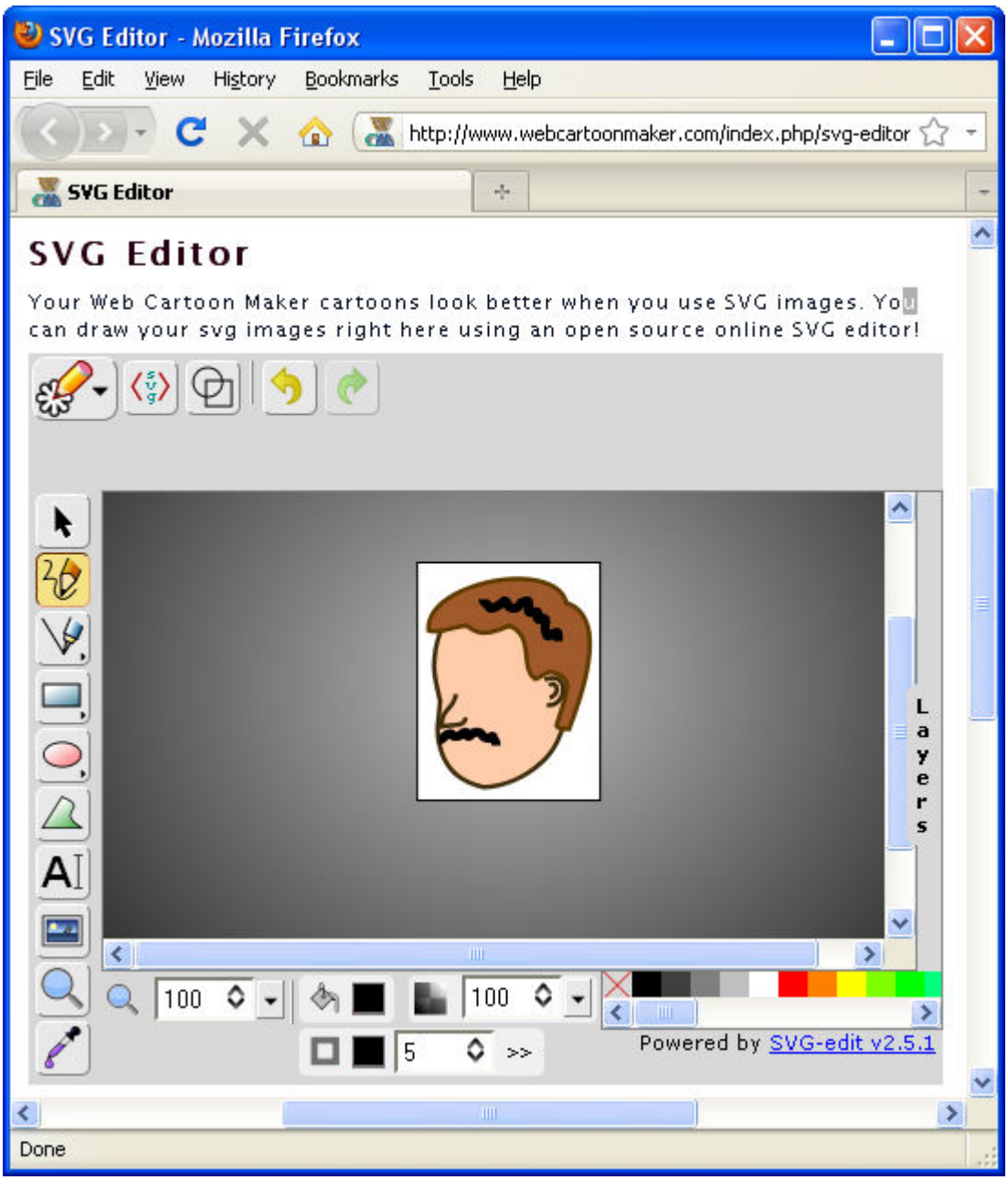


Once you download and unzip them you'll find several images and a header (.h) file in the archive. You'll need to find a decal picture. For example the default decal picture of a Boy character's head is in file "head.svg". Just edit this file any way you like, or even substitute a completely new picture, and save somewhere on your hard drive ("c:\\myimage.svg" in the above example).

Note: This would be a good time to look at the “Simple Character Modification” tutorial.

Do you know how to edit SVG files – or even what a .SVG file represents? Answering the second question first, “SVG” stands for Scalable Vector Graphics and is an open standard for pictures in vector (points and lines) format. Vector formats are much more scalable than raster (pixel) based formats such as .JPG, .PNG, .BMP and .GIF. The disadvantage of the vector format is that nearly all cameras and displays are pixel based, so ultimately the vector drawing must be converted back to raster format for display.

You may already have some vector graphics editing tools installed on your computer. The commercial standard tool is Adobe’s Illustrator. While an excellent tool, it is also both expensive and “expert friendly.” Fortunately, there is a great free and open source SVG editor called Inkscape which you may want to install (download from http://inkscape.org). Inkscape also has a good online user manual, which is referenced on their web site. Finally, you can also edit files online at http://www.webcartoonmaker.com/?art=svg. Please keep in mind that online editor will work in IE only with Chrome Frame plugin. But it works just fine in Firefox without any addons:



Notes/Cautions: While the online editor is quite convenient, the available user documentation (at http://code.google.com/p/svg-edit) is aimed more at the code developer than the first time user. Also the .SVG format can occasionally be a bit unstable, although the situation gets better every day. If you are using Illustrator, stay with the .AI format (a very stable but Adobe proprietary vector format) as long as possible.

Making a new character

For animations it is always good to allow the user community to create (and hopefully share) their own art and characters. Here, the advanced user will find the C++ based features of WCM extremely useful.

Note: This would also be a good time to review the "Simple Character Creation" tutorial, which is similar in content, but written from a somewhat different perspective, to the following paragraphs.

To make a completely new character in WCM, you must first define a new class derived from ICharacter or another class derived from ICharacter like ISpeakableCharacter, IHumanCharacter or IHumanCharacterSideView. As noted earlier these classes are incomplete, however they have many useful methods to register and work with characters. Some of them also have common parts declared but none of them have actual pictures assigned with parts. However to make these classes useful you need to derive a new class from them. In this example, we'll make a simple butterfly character by deriving a new class from ICharacter:

```
class Butterfly : public ICharacter
{
      public:

            Butterfly () : ICharacter ()
            {
            }

};
```

This class as currently written is still useless. It is basically just an outline and does not bring any new functionality to ICharacter. To make it a real character we need to do the following:

1. Declare all the desired parts (like Body, Head and so on). Some of the advanced interface classes already have some parts declared. For example ISpeakableCharacter has a Mouth part declared. IHumanCharacter and IHumanCharacterSideView have Mouth, RightArm, RightLeg, LeftLeg, Body, Head, RightEye, LeftEye and LeftArm declared. We are going to derive a new character from simplest ICharacter interface and will need to declare all the parts by ourselves.
2. Register character's dimensions and origin location using method Register in constructor.
3. Register all the parts in character's constructor specifying their default decal pictures, their coordinates relative to parent's origin (relative to character's origin if there is no parent) and part origin's coordinates using method RegisterPart.
4. If some parts have parents you must register parents using part's RegisterParent method
5. Register any additional decals using part's RegisterDecal method.

First of all let's find a couple of butterfly pictures with wings up and down. You can download these two images:

http://www.webcartoonmaker.com/lib/images/wcm/butterfly1.svg 

http://www.webcartoonmaker.com/lib/images/wcm/butterfly2.svg 

Download these images or create you own 300x300 pixels images and save them as "C:\\ButterflyCharacter\\butterfly1.svg" and "C:\\ButterflyCharacter\\butterfly2.svg"

These are two 300x300 pixels images. Let's use these dimensions as the dimension of our character. We'll use coordinates (175,145), relative to left upper corner, as the origin point. This is actually a middle of butterfly's body. Our character will have only one part – Body. It will also have 300x300 pixels dimensions and origin located at (175,145). The coordinates of Body's origin relative to character's origin will be at (0,0). (This is generally recommended for consistency

between characters.) We'll use the first picture as the "DEFAULT" decal and register the second image as an additional decal:

```
class Butterfly : public ICharacter
{
	public:

		Part Body;

		Butterfly () : ICharacter ()
		{
			// dimensions and the origin coordinates
			Register ( 300, 300, 175, 145 );

			// coordinates of the part (0,0)
			// and coordinates of the part's origin (175,145)
			RegisterPart ( Body,
			"C:\\ButterflyCharacter\\butterfly1.svg", 0, 0, 175,
			145 );

			// we need to register one additional decal
			Body.RegisterDecal ( "WINGS_DOWN",
			"C:\\ButterflyCharacter\\butterfly2.svg" );
		}
};
```

This is it! We have a new simple butterfly character created. We can move it using SetPos, we can change its body's decal and do other things.

We may also want our butterfly to fly. We can add an additional method to achieve this:

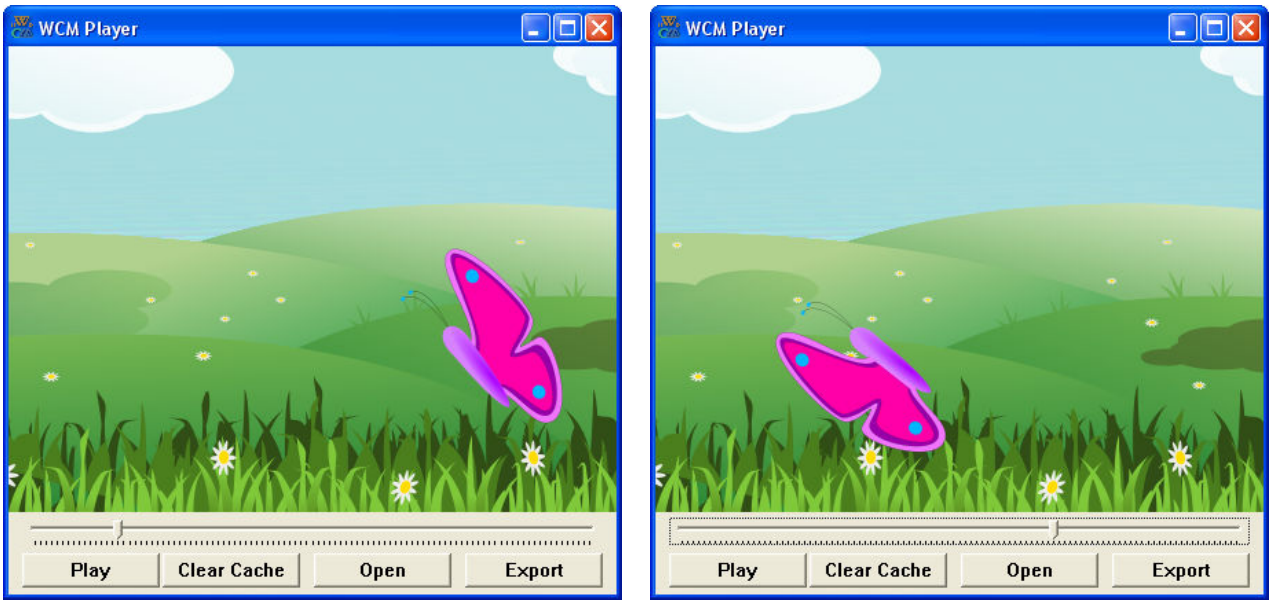
```
class Butterfly : public ICharacter
{
	public:
		…
		void FliesTo ( double dX, double dY, double dDuration,
				double dFlapDuration = 1 )
		{
			SetPos ( GetX (), GetY () ); // start control point
			double dStartTime = GetTime (); // remember time

			for ( int i=0; i<dDuration/dFlapDuration; i++ )
			{
				SetTime ( dStartTime + i * dFlapDuration );
				Body.SetDecal ( "WINGS_DOWN");

				SetTime ( dStartTime + i * dFlapDuration +
					dFlapDuration / 2 );
				Body.SetDecal ( "DEFAULT");
			}

			SetPos ( dX, dY ); // end control point
		}
		…
};
```

Here is a full example of using the character:

```
class Butterfly : public ICharacter
{
	public:
```

```
        Part Body;

        Butterfly () : ICharacter ()
        {
              // dimensions and the origin coordinates
              Register ( 300, 300, 175, 145 );

              // coordinates of the part (0,0)
              // and coordinates of the part's origin (175,145)
              RegisterPart ( Body,
              "C:\\ButterflyCharacter\\butterfly1.svg", 0, 0, 175,
              145 );

              // we need to register one additional decal
              Body.RegisterDecal ( "WINGS_DOWN",
              "C:\\ButterflyCharacter\\butterfly2.svg" );
        }

        void FliesTo ( double dX, double dY, double dDuration,
                       double dFlapDuration = 1 )
        {
              SetPos ( GetX (), GetY () ); // start control point
              double dStartTime = GetTime (); // remember time

              for ( int i=0; i<dDuration/dFlapDuration; i++ )
              {
                    SetTime ( dStartTime + i * dFlapDuration );
                    Body.SetDecal ( "WINGS_DOWN");

                    SetTime ( dStartTime + i * dFlapDuration +
                              dFlapDuration / 2 );
                    Body.SetDecal ( "DEFAULT");
              }

              SetPos ( dX, dY ); // end control point
        }

};

void Scene1 ()
{
      Image Back ( "backgrounds/landscape.svg" );
      Back.SetVisible ();
      Butterfly MyButterfly;
      MyButterfly.SetVisible ();
      MyButterfly.SetPos ( 300, 100 );
      MyButterfly.FliesTo ( -300, 100, 5 );
}
```

Please try to compile this movie to see a flying butterfly:



Making Advanced Characters

Actually you now know everything about making character and just need to practice on making custom characters by deriving you classes from more advanced interfaces like ISpeakableCharacter, IHumanCharacter and IHumanCharacterSideView. But there is even better and simpler way. Just follow the online tutorial (this tutorial, "CharMaker Utility," is different from the one mentioned earlier) at: http://www.webcartoonmaker.com/?art=help/tut_charmaker

We do not want to include this tutorial in this book for a couple of reasons. One of them is that utility used in tutorial is still in beta. It may be changed and improved. And another reason is to let you know that there is a great online tutorial available. You already know enough about C++ and Web Cartoon Maker to understand it very well!

Support Web Cartoon Maker

Web Cartoon Maker is free! We do not ask you to pay for anything! But there are several ways you may help the project to live and be able to pay for hosting, compiler and other software:

6. Are you planning to buy something on amazon.com? Do this using our link: http://www.amazon.com/?tag=webcarmak-20 and we'll receive a percentage of your total purchase. This will help us and will not cost you anything!
7. Can you draw or compose music? You can donate your graphics and music to Web Cartoon Maker project! Just contact us at the link below: http://www.webcartoonmaker.com/?art=discuss
8. Write a good review for Web Cartoon Maker somewhere on the web
9. Become our friend on YouTube and subscribe to our channel: http://www.youtube.com/subscription_center?add_user=webcartoonmaker
10. Upload a cartoon to YouTube and let people know it was done in Web Cartoon Maker

References

This book covered the aspects of C++ relevant to WCM. For information on general C++, there are a variety of good sources on the web:

1. "How to think like a computer scientist" by Allen B. Downey. The book which was referenced at the start of this document: http://www.greenteapress.com/thinkcpp
2. Another good C++ tutorial is at: http://www.cplusplus.com/doc/tutorial
3. A site with a lot of good tutorials from beginner to expert level and for C++ and other languages is at: http://www.dickbaldwin.com/toc.htm
4. There are also many books on C++, *some* of them quite good. A particularly readable book, which includes a useful CD-ROM, is "C++ Without Fear" (Prentice Hall) by Brian Overland: http://www.amazon.com/exec/obidos/ASIN/0321246950/webcarmak-20