

# CHAPTER 2

## Starting to Program

In this chapter you will learn

- The first C++ program
- Object diagrams
- Sequential execution
- C++ syntax
- The Turtle class

## The First Program

- Our first C++ program doesn't do very much, but it illustrates a few key aspects of a C++ program.

```
// Program One: A simple program using an OKBox.
```

```
#include "GUIObj.h"

void main ( )
{
    OKBox myBox;
    myBox.Display("Hello, world!");
}
```

- Running this program will display



- We will explain the program line by line:

### Line 1 is a comment.

Line1 → 

```
// Program One: A simple program using an OKBox.  
  
#include "GUIObj.h"  
  
void main ( )  
{  
    OKBox  myBox;  
    myBox.Display("Hello, world!");  
}
```

- Comments do not affect the way the program works, but they can make it easier for others to understand what your program does and how it does it. More important, comments can help you to understand your own program weeks (or even hours) after you write it.
- When you have more than one line of comments, each line must be preceded by double slashes (//).
- Another way to put a comment in the program is to precede the comment with /\* and terminate it with \*/. A comment can extend beyond one line if you surround it with /\* and \*/.

```
/* This is a multiline  
   comment. Your comment can go beyond  
   one line. */
```

## Line 2 is an include instruction

Line2 →

```
// Program One: A simple program using an OKBox.  
#include "GUIObj.h"  
  
void main ( )  
{  
    OKBox myBox;  
    myBox.Display("Hello, world!");  
}
```

- Line 2 informs the compiler that the program uses things defined in a *header file*, or more simply a *header*, named `GUIObj.h`. This header file contains predefined program components, one of which is an object.
- As a convention, we use a suffix `.h` for a header file name. In this sample program, we use an `OKBox` object defined in this header file.
- In this book we will use several headers supplied by the C++ compiler as well as `GUIObj.h` and `Turtle.h`, the headers we have written specifically for this book. For the earlier part of this book, you will only use the predefined objects, but later, you will be able to define your own objects.

## Line 3 is a function declaration

```
Line3 → // Program One: A simple program using an OKBox.
#include "GUIObj.h"
void main ( )
{
    OKBox myBox;
    myBox.Display("Hello, world!");
}
```

- Every C++ program must have one function named `main`. We call this function the *main function*. The function serves as the main controller of a program and is also called the *main program*.
- Inside the parentheses [ ( ) ], we list input values to a function. Since there is no input to this main function, there is nothing inside the parentheses. Normally, we do not provide any input to the main function.
- The output, or result of a C++ function, is called a *return value*. A function in mathematics must return a value, but a C++ function does not have to return a value.
- A C++ function that does not return a value is called a *void function* and designated as such by the word `void` in front of the function name. The main function of this sample program is a void function as are the main functions of all other sample programs in this book.

- Other than this main function, which must be named `main`, we can name functions anyway we want (as long as the name doesn't violate the rule given later). We call these names *identifiers*.

### Line 4 is the beginning of a function body

Line4 →

```
// Program One: A simple program using an OKBox.

#include "GUIObj.h"

void main ( )
{
    OKBox myBox;
    myBox.Display("Hello, world!");
}
```

- Line 4 signals the beginning of a function body. Since this is the main function, it signals the beginning of the program. A function body consists of two components—declarations and commands. In this program we have only one declaration (line 5) and one command (line 6). A function body is terminated by the right brace [ ] (line 7).

## Line 5 is declaration

Line5 →

```
// Program One: A simple program using an OKBox.
#include "GUIObj.h"

void main ( )
{
    OKBox myBox;
    myBox.Display("Hello, world!");
}
```

- Line 5 is a declaration that declares the identifier `myBox` as the name of an `OKBox` object. We normally say, “`myBox` is an `OKBox` object,” for short. If you need to use more than one object, say three, then you need three unique identifiers declared as follows:

```
OKBox myBox, yourBox, herBox;
```

- Every declaration and command in C++ must be terminated with a semicolon [ ; ].

## Line 6 is a command

```

// Program One: A simple program using an OKBox.

#include "GUIObj.h"

void main ( )
{
    OKBox myBox;
myBox.Display("Hello, world!");
}

```

Line6 →

- Line 6 is a command that “commands” the `OKBox` object `myBox` to display the message `Hello, world!`. We can think this command as sending of a message `Display` to the `myBox` object. Receiving this message, the `myBox` object carries out the requested task.
- For an object to be able to respond to a message, a corresponding function must be defined for the object. This function tells the object exactly what to do to carry out the task. Many functions are normally defined for an object, and this collection of functions defines the “behavior” of an object. If no corresponding function is available to the received message, an error would result.
- An input value to a function is called an *argument*. If a function has two or more arguments, they are separated by commas. The general format for calling an object’s function is

```
object_identifier.function_name ( argument_list ) ;
```

- The left and right parentheses must be present even if there is no argument. Because of the period between the object identifier and its function name, this style of calling an object's function is called *dot notation*.

### Line 7 is the beginning of a function body

```

// Program One: A simple program using an OKBox.

#include "GUIObj.h"

void main ( )
{
    OKBox myBox;
    myBox.Display("Hello, world!");
}

```

Line7 →

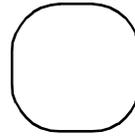
## Object Diagrams

- It is easier to understand a program when we can see a graphical representation of the program's objects and their interactions.

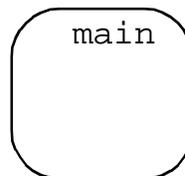


**Object Diagram for Program One**

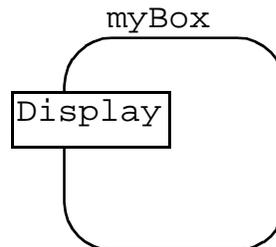
1. In an object diagram, objects are shown using an object icon, which is a rectangle with rounded corners.



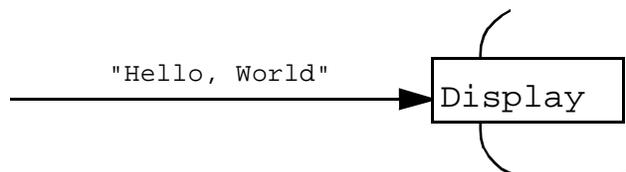
2. The program itself is represented as an object. The name is written inside the rectangle because this name is not used by any other objects. It is mainly for our own reference.



3. An object's function is listed on the edge of the object. (`Display` is a function of `myBox`.) The name of an object is written outside of the box to signify that the name is visible to other objects and is used by them.



4. An arrow from one object to a function in another object represents a function call. Arguments in a function call are written along the arrow.



## Sequential Execution

- Statements in a program are executed in the order they appear, from top to bottom. We can expand our first program to illustrate this sequential order of execution.

```
//Program Two: Another simple program using an OKBox.
```

```
#include "GUIObj.h"
```

```
void main ( )
```

```
{
```

```
    OKBox  myBox;
```

```
    myBox.Display("Hello, World!");
```

```
    myBox.Display("Welcome to Object Land");
```

```
    myBox.Display("May you have fun and learn a lot!");
```

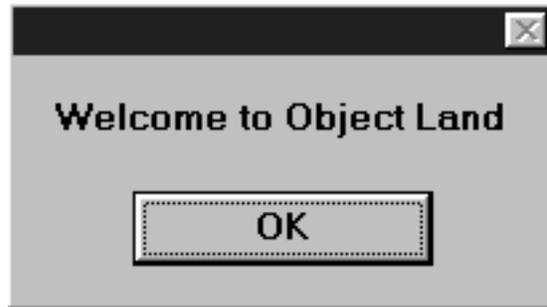
```
    myBox.Display("Goodbye");
```

```
}
```

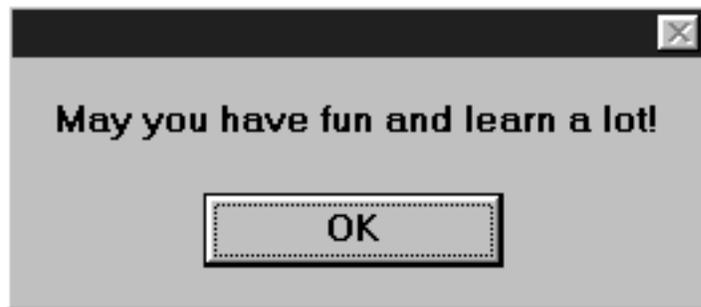
- We have four command statements, making four function calls. When this program is executed, it initially displays the same message box as the first program (by the execution of the first statement `myBox.Display("Hello, world!")`).



- When the user clicks the OK button, another message box appears, because clicking the button completes the execution of the first statement and allows the second one to be executed. The execution of the second statement causes `myBox` to display the second message.



- Clicking the OK button on this box causes the third message to appear.



- Clicking the OK button on this box causes the fourth statement to be executed and produces the following message.



- Clicking OK on this last message box terminates the program because it completes the execution of the final statement in the program.
- This expanded program shows how the statements of a program are executed one at a time. It also shows more about what an `OKBox`'s `Display` function actually does. Notice that when an `OKBox` displays text, it adjusts the size of the box to accommodate the length of the text. Also notice that the execution of a `Display` function isn't completed until you click the OK button.

## C++ Syntax

- The grammatical rules of the C++ language called *syntax rules* dictate the structural validity of C++ programs. These grammatical rules collectively enforce the valid syntax for C++ programs.
- To be precise and concise, we usually show syntax rules for programming languages as diagrams or in some stylized form. C++ syntax rules are normally stated in a stylized form called a

*Backus Naur Form*, or BNF for short, which we shall use in this book.

- Let's look at the BNF rules for the C++ identifiers.

① *identifier*  
*letter character-list<sub>opt</sub>*

② *character-list*  
*character*  
*character character-list*

③ *character*  
*digit*  
*letter*  
 —

④ *digit*  
 0 | 1 | ... | 9

⑤ *letter*  
 a | b | ... | z | A | B | ... | Z

- Each of the five rules are listed above defines a *syntax category* of the C++ language. A syntax category with the subscript *opt* denotes an optional element.

- **Rule 1:** An identifier is a letter followed by an optional character-list.
- **Rule 2:** A character list is a character OR a character followed by a character list.
- **Rule 3:** A character is either digit, letter, or a special underscore symbol (`_`).
- **Rule 4:** A digit is either 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. The vertical bar (`|`) is another way of showing alternatives.

A three-dot symbol (`...`) is used to denote a sequence of values without explicitly listing them.

- **Rule 5:** A letter is any one of the alphabetic characters.
- The five rules collectively state that an identifier is a sequence of letters, digits, or underscore characters with the first character being a letter.
- Here are some valid (top) and invalid (bottom) identifiers according to the above syntax rules.

MyBox    Box2    sam    x    Address    temperature    Bill\_Clinton

2Box    First.Name    ?okay?    Bill-Clinton    Sam\*Spade

- In some programming languages (e.g., Pascal and Ada), lowercase and uppercase letters are not distinguished. These languages consider the identifiers `HELLO`, `hello`, and `Hello` to be

the same, for example. C++, however, does distinguish between cases.

- Languages that distinguish between lowercase and uppercase letters are called *case-sensitive languages*, and those that do not are called *case-insensitive languages*. This type of information is not stated with the syntax rules.
- Another type of information that explicitly is the length of identifiers. The syntax rules place no limit on the number of characters we use for the identifiers. This freedom, of course, is not acceptable in practice. A compiler puts some limit on the number of characters that can be used for an identifier.
- The third piece of information that is not stated with the production rules for an identifier is the exclusion of certain words called reserved words. *Reserved words* have special meaning and purpose in C++, and they cannot be used as identifiers. Table below lists the C++ reserved words.

### C++ reserved words

asm	continue	float	new	short	try
auto	default	for	operator	signed	typedef
break	delete	friend	overload	sizeof	union
case	do	goto	private	static	unsigned
catch	double	if	protected	struct	virtual
char	else	inline	public	switch	void
class	enum	int	register	this	volatile
const	extern	long	return	template	while

- The BNF rules above are also known as *production rules* because they are rules for “producing” more syntax categories from a given syntax category. In other words, a production rule explains how to expand a syntax category into other syntax categories.
- We say a C++ program is syntactically correct if we can produce a given program by applying the production rules, starting from the top-most syntax category *program*. A syntax category is also called *nonterminal* because we can expand it further (i.e., we can produce more syntax categories from a given syntax category by applying some production rule to it). Reserved words, identifiers, and some other characters, which we have not yet learned, are called *terminals* because we cannot expand them any further (i.e., there are no further production rules that can be applied to them).
- Different font styles are used in the syntax rules. Syntax categories, or nonterminals, are shown in the italic font with all lowercase letters. Reserved words, characters, and special symbols, such as parentheses and braces, are shown in the bold fonts.
- Let’s look at a sample production. Keep in mind that we do not use these production rules to actually write a C++ program. These rules are for a compiler to verify whether a given program is syntactically correct. Here’s an example. We start productions from a single nonterminal *identifier*.

*identifier*

==> *letter character-list*

```

===>J character-list

===>J character character-list

===>J letter character-list

===>J o character-list

===>J o character

===>J o letter

===>J o e

```

(Note: The actual identifier generated here is Joe. The spaces between the letters in the example is to make the identifier more readable.)

## The Turtle Object

- In the early 1970s Seymour Papert at MIT built a programmable, mechanical “turtle” that could move around a drawing board, pushing a pen to draw all sorts of shapes.
- We use turtle objects to illustrate many new concepts in this book. The `Turtles.h` header file defines a `Turtle` object that behaves very much like Papert’s mechanical turtle. Here is a simple program that uses a `Turtle` object to draw a square.

```

// Program Square: A program that draws a square.

#include "Turtles.h"

```

```
void main ( )
{
    Turtle myTurtle;

    myTurtle.Init(260,180); //Start from location (260,180).

    myTurtle.Move(50); // Draw the bottom of the square.
    myTurtle.Turn(90); // Turn to draw the right side.

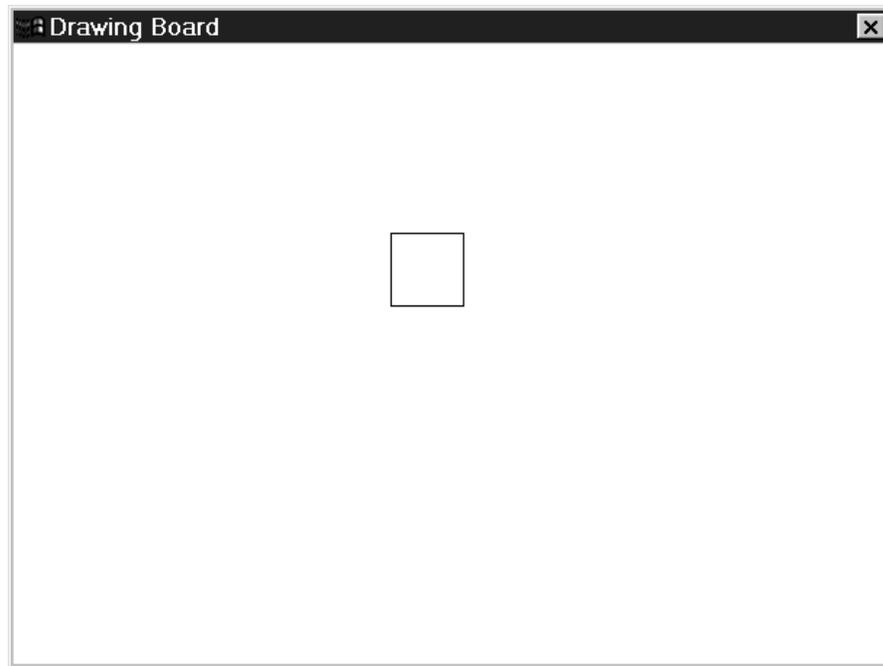
    myTurtle.Move(50); // Draw the right side.
    myTurtle.Turn(90); // Turn to draw the top.

    myTurtle.Move(50); // Draw the top.
    myTurtle.Turn(90); // Turn to draw the left side.

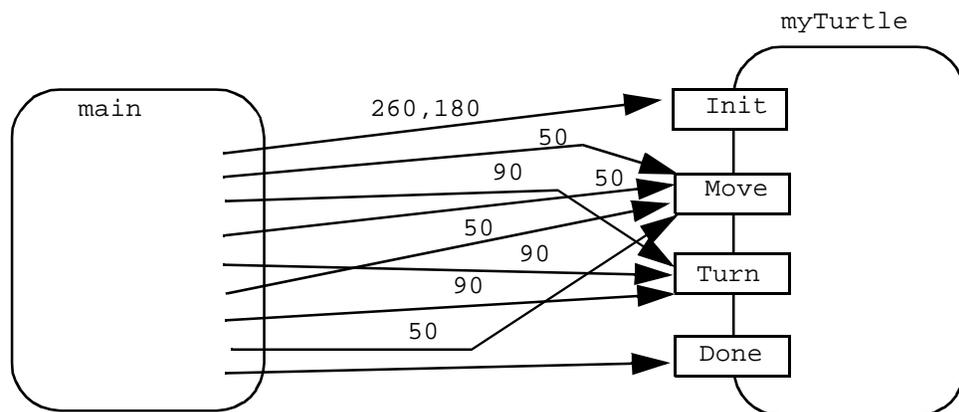
    myTurtle.Move(50); // Draw the left side.

    myTurtle.Done();
}
```

Here is what the above program displays when it is executed.



- This program makes nine calls to `myTurtle` functions. We can represent this program as an object diagram, but the diagram is somewhat cluttered.

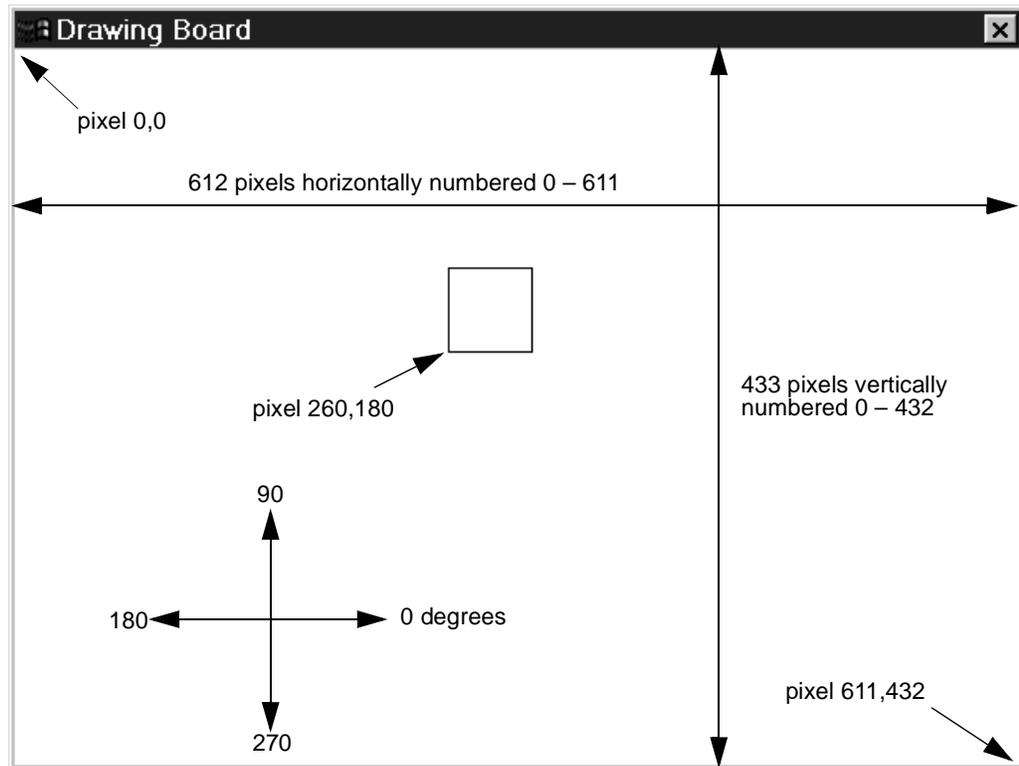


- For all but the smallest programs, the number of function calls is too many to be usefully depicted in an object diagram. For this

reason, individual function calls are not normally included as arrows in object diagrams. Instead of drawing an arrow for each function call, we draw only a single arrow from object A to B if object A makes a call to object B, regardless of the number of calls made. With this convention, the previous object diagram becomes



- The turtle's drawing board is a window with a drawing space consisting of 612 x 433 dots, or *pixels* (picture elements).



- The pixels are arranged in 433 rows and 612 columns so that each pixel has a column number (x coordinate) and a row number (y coordinate).
- The coordinates of the upper-left pixel are 0,0 and the coordinates of the lower-right pixel are 611,432. The `Turtle` draws in the drawing board by setting pixels to different colors. Initially all the pixels are white.
- When the turtle moves and its pen is down, it will set each pixel it moves across to the color of the pen. The pen is initially down, and its color is initially black. If we wish to start with a different color, then we have to set its color by using the `ChangePenColor` function.
- Here are the descriptions of the `Turtle` primary functions (a complete listing is provided in the Appendix).

<b>Init(x,y)</b>	Opens the drawing board window and initializes the turtle by facing it to the right (i.e., 0 degrees) at pixel x,y with its pen down and pen color BLACK.
<b>Move(distance)</b>	Moves the turtle <code>distance</code> pixels in the direction the turtle is currently heading. If the pen is down, the pixels it moves across are set to the current pen color. If the pen is up, then nothing is drawn.
<b>Turn(angle)</b>	Turns the turtle <code>angle</code> degrees counterclockwise.
<b>TurnTo(angle)</b>	Turns the turtle so that its heading becomes <code>angle</code> .
<b>GoToPos(x,y)</b>	Moves the turtle from its current position to pixel x,y. The turtle's heading does not change. If the pen is down, any pixels the turtle moves across changes to the current pen color.

<b>PenUp( )</b>	Sets the turtle's pen position up. If it is already up, then this function does nothing.
<b>PenDown( )</b>	Sets the turtle's pen position down. If it is already down, then this function does nothing.
<b>ChangePenColor( clr )</b>	Changes the turtle's pen color to <code>clr</code> . Possible values for <code>clr</code> are BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, LIGHTGRAY, DARKGRAY, YELLOW, WHITE, DARKRED, DARKGREEN, DARKBLUE, DUSTYBLUE, PURPLE.
<b>Done( )</b>	This statement is required at the end of the program to make the drawing board window remain on screen.
<b>GetXY( x, y )</b>	Returns the current coordinates of the turtle in <code>x</code> and <code>y</code> .

## Sample Programs Using Turtles

- Here are some sample programs that use a `Turtle`.

// Program Square2: A program that draws the square backward.

```
#include "Turtles.h"

void main ( )
{
    Turtle myTurtle;

    myTurtle.Init(260,180);

    myTurtle.Move(-50);
    myTurtle.Turn(-90);
```

```

myTurtle.Move(-50);
myTurtle.Turn(-90);

myTurtle.Move(-50);
myTurtle.Turn(-90);

myTurtle.Move(-50);

myTurtle.Done();
}

```

- The next program also draws the same square but at a different position. The program uses the `GoToPos` function four times (the heading of `myTurtle` does not change).

```

//Program Square3: A program that draws the square with
//                the GoToPos functions.

#include "Turtles.h"

void main ( )
{
    Turtle myTurtle;

    myTurtle.Init(260,180);

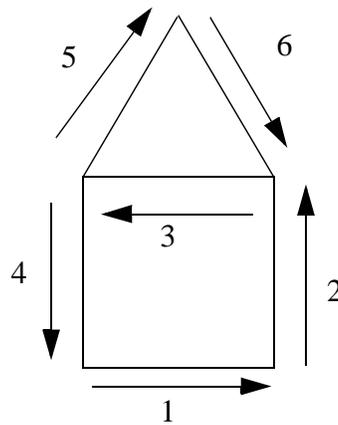
    myTurtle.GoToPos(310,180);
    myTurtle.GoToPos(310,230);
    myTurtle.GoToPos(260,230);
}

```

```
myTurtle.GoToPos(260,180);

myTurtle.Done();
}
```

- The final program uses the `PenUp`, `PenDown`, `TurnTo`, and `Turn` functions to draw a house. After drawing the bottom square, the `Turtle` draws a roof by moving into a position without drawing. The numbered arrows show the order which the `Turtle` draws the house.



//Program House: A program that draws a simple house.

```
#include "Turtles.h"

void main ( )
{
    Turtle t;

    t.Init(260,180);

    // Draw the bottom square
    t.Move(50); t.Turn(90);
```

```
t.Move(50); t.Turn(90);
t.Move(50); t.Turn(90);
t.Move(50);

//Reposition t for the roof
t.PenUp(); //don't draw while repositioning
t.GoToPos(260,130);
t.TurnTo(60);
t.PenDown();

//Now draw the roof
t.Move(50); t.Turn(-120);
t.Move(50);

t.Done();
}
```

## **Object Behavior**

- B. F. Skinner, a Harvard psychologist, was famous for his work in behaviorism. According to Skinner, humans are nothing but machines that exhibit well-defined responses to environmentally controlled stimuli. His work in operant conditioning showed that animals modify their behavior to adapt to new stimuli.
- Similar to a live turtle, our virtual turtle also exhibits well-defined behavior. Our virtual turtle, of course, is not capable of learning new behavior or adapting behavior on its own (at least not with our current state of computer science). In the case of virtual turtles and other “computer” objects, their behaviors are defined by the set of functions we attach to those objects.

- In addition to the set of functions, another aspect of objects also affects their behavior. For example, consider the legs of a turtle being tied. The turtle will not (cannot) respond to the stimuli in the same manner as when its legs were not tied. Just as live animals have different states, our computer objects also have different states. Our virtual turtle, for example, could be in the state of heading north with the pen down. Receiving a command in different states would result in a different response.
- A computer object has a set of attributes to keep track of its state. A turtle, for example, has attributes such as heading, pen color, pen status, and so forth. The set of values for these attributes determines the state of a turtle.
- Some functions modify the attributes, thus affecting the state of an object. For a `Turtle` object, functions such as `Turn` and `TurnTo` change the direction, `PenUp` and `PenDown` change the pen status, and `ChangePenColor` changes the pen color.