PART V

DEALING WITH NUMBERS

In Part V, you will further develop your skills in handling numeric data. Thanks to the special software designed for this book, you will be able to learn numeric manipulation by producing graphics and animations. You will learn how to program relatively complex numeric expressions before moving on to manipulating Screen Objects and animations. A limited knowledge of geometry will come in handy.

SKILL

THIRTEEN

MANIPULATING NUMERIC EXPRESSIONS

- Identifying the numeric types used in C++
- Writing and evaluating expressions
- Using math functions

In SKILL thirteen, we will deepen your knowledge of using numbers. You will learn how to determine what types of variables to use for your data, how to manipulate values in variables by using expressions, and how to use the math function library. Many applications require extensive numeric manipulation. In SKILLs fourteen and fifteen, you will be provided with interesting and relevant opportunities to practice your newfound skills in manipulating numeric expressions.

Deepening the Discussion of Numbers

Very often, you will need to manipulate numbers with your computer. In fact, when computers were first put into use, all their work related to numeric computation. Computers were used to compute trajectories and integrals, to solve equations, and so on. Computers represent information in numeric form, but numbers can represent colors, shapes, sounds, and nearly anything else you can imagine.

Since many applications require extensive use of numeric data, we will spend some time in this skill to explore numeric manipulation. To make the subject more interesting, our applications in SKILLs fourteen and fifteen will involve graphics and animations.

Identifying Numeric Types

Numeric types and expressions were introduced briefly in SKILL five. In the following sections, we shall deepen the discussion.

You don’t usually manipulate numbers that are represented by an arbitrary number of digits in a computer. It is much simpler for the computer to set apart some space in its memory for

numbers of a fixed size, according to the *type* of number with which you want to work. This process is similar to how some hand calculators work. If the calculator has an eight-digit display, you cannot operate with numbers that are longer than eight digits.

Storing Integer Numbers

As you know, computers can operate with integers and floating point numbers. To store integers, there are the types `int` and `long`. Here is a summary of how they work:

- We have already used the type `int` in previous skills. To store an integer number (of type `int`), the computer sets aside a space in which you can store an integer number.
    - Check your particular compiler for the range of numbers you can store. Most compilers use 16 bits to represent `int`s, which results in a range of numbers from –32,768 to +32,767. Other compilers may be able to accommodate larger ranges of numbers.
- The type `long` stands for *long int*, and tells the compiler that you want to deal with an integer that has a wider range. In that case, twice the space will be used to store this kind of number, and then you may deal with a wider range of numbers.
    - Check your particular compiler for the range of numbers you can store. Most compilers use 32 bits to represent `long` integers, which results in a range of numbers from –2,147,483,648 to +2,147,483,647. In any event, `long` integers take up twice the space in memory, and can accommodate a much larger range of numbers than an `int`.

Don't try to memorize these numbers—just remember that you can go to approximately two billion each way.

Unsigned integers

If you are dealing with a variable that takes only positive values (for example, the number of students in a class), you can stretch the range of an `int` (or `long` integer) by prefixing the declaration with the keyword `unsigned`. For example:

```
unsigned int number_of_students;
```

Since you don’t need to accommodate the negative values, the computer will be able to accommodate a positive number that is twice as large as the largest number that you could accommodate previously. If your compiler has an `int` range from –32,768 to +32,767, the unsigned `int` can range from 0 to +65,535.

You donít have to use unsigned integers if you don’t need the extra values. If you are dealing with the number of students in a class, it is likely that 32,767 integers will be enough.

Short integers

If you really want to be savvy about memory, you can declare variables that are expected to accommodate very small integer values by prefixing the declaration with the keyword `short`. For example:

```
short int color;
```

Check your compiler for the range. Most compilers use 8 bits, which will accommodate integers from –128 to +127. You can combine `unsigned` and `short`.

Storing Floating Point Numbers

Some problems arise in which integer numbers are not appropriate. Often, you may need to use a decimal point and/or an exponent. In these cases, you have to resort to floating point numbers.

- The standard type in which to store floating point numbers is `float`. The type `float` takes up a little more space in the computer memory than `int` does, and, since it is a bit more complicated, more time is consumed to perform operations with floating point numbers than with integers. Again, there are some limitations. The regular floating point numbers may range from $3.4 \lozenge 10^{-38}$ to $3.4 \lozenge 10^{38}$, may be either positive or negative, and may hold up to seven digits of precision. If you need more than that, do not despair….

Check your compiler for the actual range. It may vary from one compiler to another.

WHAT DO YOU MEAN—SEVEN DIGITS?

If you try to store the number 1.0000001, the computer will not store the complete number. Only the seven most significant digits (1.000000) will be stored. Anything beyond that is not guaranteed, because only seven digits were stored. However, it has nothing to do with the position of the decimal point; the same thing would happen with the number 234.56789, which would end up being 234.5678.

- Another type that is useful for dealing with large floating point numbers is the type `double`. It works just like the type `float`, but it is able to store numbers that have more significant digits and, depending on the compiler, a wider range of exponents.
- If you have to manipulate numbers with even more digits, you can prefix `double` with the keyword `long`. Again, check your compiler for the actual range.

Using Constants

Since you can refer to integer and floating point numbers, you may also need to use integer and floating point constants.

Constants are values that cannot change during the program execution. You will learn three ways to manipulate constant values in your programs:

- *Literal* constants literally write the value.
- *Labeled* constants are exactly like variables to which you give a name and assign a value, but you instruct the compiler to prevent any change to their value.
- *Enumerated* constants are a set of labeled constants to which you can assign a specific type and which use a list of names to denote the value.

Integer constants

You write an integer constant pretty much like you write an integer number in everyday math. Observe the following items:

- You can precede the number with a + sign or a – sign.
- You cannot use commas or decimal points.
- You cannot leave spaces between the digits.
- You should avoid preceding the number with irrelevant zeros, which may cause confusion.

Be aware of leading zeros! Leading zeros are used in C++ to identify integer constants that are expressed in octal notation. A value expressed in this notation is different from everyday decimal notation. For example, a constant expressed as *010* in C++ will be assumed to be in octal notation, resulting in the value *8* in decimal notation.

Floating point constants

You can write a floating point constant in C++ in two forms. The first form is similar to our everyday notation:

- You can precede the number with a + sign or a – sign.
- You can include one (and only one) decimal point.
- You cannot use commas or spaces between the digits.

Here are some examples of floating point constants:

35.
73.12
0.001
+54.3
–0.2

The second form is exponential notation, which is similar to how we represent large numbers in physics. This notation consists of two parts: the mantissa, which is represented by a floating point number, as in the examples above, and the exponent, which consists of the letter *E* followed by an integer number that represents the exponent.

For example, the following number:

$6.02 \lozenge 10^{23}$

can be written in C++ as 6.02 E23 or 6.02E+23. Both parts—the mantissa and the exponent—can have a sign.

Labeled constants

It is possible to use an identifier to denote a constant value. If you need to use the value *3.1416* many times in your program, you may be better off declaring as follows:

```
float pi=3.1416;
```

and using `pi` instead of `3.1416` throughout your program. This idea is pretty cool! The only slight problem you may have is if this value of `pi` is erroneously altered in the program. You can avoid having a value altered by including the keyword `const` in the variable declaration. If you declare as follows:

```
const float pi=3.1416;
```
the compiler will not allow changes to this value. Any statement that attempts to alter the value of `pi` will cause an error.

Another convenience of `const` is that sometimes you may want to use a different value for a particular quantity. For example, you may decide to use the more precise value *3.14159*, instead of using *3.1416*. It is much simpler to change just the declaration than to look through the entire program and change the values!

`const` lets you use a variable whose value is fixed during program execution.

Enumerated constants

Enumerated constants are especially useful when a set of integers is used as a code. Take the example of the directions north, east, south, and west, which may be assigned the codes *0, 1, 2,* and *3*. To avoid dealing with the numbers in the program, we could declare as follows:

```
int N=0,E=1,S=2,W=3;
```
Or, better still:

```
const int N=0,E=1,S=2,W=3;
```
In either case, we could use the identifiers *N*, *E*, *S*, and *W* instead of the codes in the program. For example:

```
if(direction==N) ...
```
A simpler way to achieve the same result, with a few extra benefits, is to use the `enum` constant declaration:

```
enum  type  { identifiers list };
```
Or, in our case:

```
enum directions {N,E,S,W};
```
This declares the identifiers that are contained in the list to be constants of type `int`, and automatically initializes them to 0, 1, 2, and 3. Unless otherwise specified, the first identifier is initialized to 0, and each other identifier is initialized to the next integer.

The *type* is optional. In our example, `directions` could be omitted. The advantage of using the type is that you may declare a variable in the program that has this new type. For example:

```
directions  comingfrom,goingto;
```
allows you to assign any direction to these new variables. Therefore, if your program has assignments of the following form:

```
comingfrom=N;
goingto=S;
```
these statements will be considered correct. However, if you try to assign anything different (including an integer), the compiler will issue a warning. In the following example:

```
comingfrom=8;
goingto=2;
```
the compiler may suspect that you are doing something wrong.

Finally, it is possible to overrule the default sequence by specifying which value should be assigned to each identifier. For example:

```
enum colors (red=1,green,pink=5,yellow);
```
assigns value 1 to red, value 2 to green (previous value plus 1), value 5 to pink (as specified), and value 6 to yellow.

Reviewing Expressions

The rules for performing arithmetic operations and assigning values are reviewed here with some examples. For the examples in this section, let's assume that we have declared as follows:

```
int i, j, k;
float x,y,z;
```

Mixing Types, Variables, and Constants

You can combine variables and constants of either type in expressions. For example:

```
float x=0.5,y;
int i=5;
y=x-i+1;
```

takes the value of `x`, subtracts from it the value of `i`, and adds 1. The result will be stored as the new value of `y`.

So far, this is very close to what you would expect. In fact, the way we represent arithmetic expressions in C++ (as well as in most other computer languages) bears a close similarity to the way we do it in math. There are a few things that you must know before we examine any rules:

- In math, you can omit the multiplication operator: 3xy means 3 times x times y.
  - In C++, we cannot do this. There is no way for a compiler to tell whether you have an object *xy* or *x* times *y*. Therefore, we must explicitly use the multiplication operator. Which symbol should we use? We cannot use x, because it could signify an object called *x*. Therefore, the multiplication operator is represented by *.
- In math, you can write values above and below a bar to indicate divisions.
  - In a program, this is inconvenient, because we type from a keyboard. Instead, we use the slash symbol (/) to denote division. However, dividend and divisor are supposed to be in the same line. For example, x/y means that you should take the value of x and divide it by the value of y.
- In math, you don't usually differentiate between an integer and a floating point number when you compute an expression.
  - In most programming languages, integers and floating point numbers are treated quite differently. The result of an arithmetic operation that involves two integers is always an integer! Does this make any difference? Of course it does! If you divide 3./2. (notice that both are floating point numbers), the result is 1.5. If you divide 3/2 (notice that both are now integers of type `int`), the result is 1, because only the integer part is considered. Similar problems may happen in other operations.

For example:

```
int i,j;
i=30000;
j=i+i;
```

results in a value of 60,000, which most likely exceeds the range of an `int`.

Since you are now aware of these differences, we can actually move on to our rules.

Using Arithmetic Operators

The following arithmetic operators can be used in expressions:

+ addition

– subtraction
* multiplication
/ division
% remainder of division

In addition, the assignment operator (=) can be used to assign the result of an expression to the variable on its left side. Here are some examples:

```
x=3*a+1; // Multiply a by 3, add 1, and store result in x
y=a+1*3; // Multiply 1 by 3, add to a, and store result in y
```

The second example illustrates that in expressions in C++, just like in expressions in math, multiplication takes priority over addition. Therefore, multiplication is done first. You may remember that an expression such as x+3a means that you should multiply *a* by 3, then add the result to *x*.

The rules for precedence are as follows:

- Higher priority operators are *, /, and %.
- Lower priority operators are + and –.
- Higher priority operations are executed first.
- If more than one operation has the same priority, execute from left to right.

You can explicitly specify the priority by using parentheses:

```
3*(x+1); // Add x to 1, then multiply by 3
3*x+1;   // Multiply x by 3, then add 1
```

If you need to, you can also nest parentheses in parentheses. In this case, the inner parentheses are solved first. For example:

```
3*(x/(y+1) + 4);
```

adds 1 to *y*, then divides *x* by this result. Then, it adds 4. Finally, this result is multiplied by 3. Notice that *x* is divided by *y+1* and not by *y+1+4*, as you might think if you were distracted.

Assigning Results

The assignment operator (=) can be used in an expression to take the result (shown on the right side of the =) and assign it to the variable on the left side. Remember that the meaning of the operator is not the same as its meaning in a mathematical equation! For example:

```
s=s+3;
```

is correct. But the following expression:

```
s+3=s;
```

is not acceptable in C++!

Generating results

Besides moving a value from one place to another, the assignment operator also generates a result that has the value of the value that was moved. In other words, the following statement:

```
s=3;
```

besides storing the value *3* in the variable `s`, will generate a result of 3. This is why the compiler considers the following statement to be correct in terms of syntax:

```
if(s=3) ...
```

Instead of comparing `s` to 3, this statement will move 3 to `s`, and generate 3 as a result. Since 3 is nonzero, it gives a *true* result inside the `if`.

Multiple assignments

Since each assignment generates a result, it is possible to write the following statement:

```
x=y=1;
```

because the assignment operation:

```
y=1;
```
generates a result of 1. This result will be used by the next assignment operator.

Multiple assignments are executed from right to left.

Conversions

If an assignment operation involves operands of different types, the result will be converted to assume the type of the variable on the left side of the assignment operator. For example, suppose that we have the following expressions:

```
int  m=1,j=2,k;
float x=0.5,y;
```

An expression such as the following one:

```
m=x;
```
will have a result of 0.5, which will be converted to `int` so it can be stored in `m`. The new value of `m` will then be zero. Now consider the following expression:

```
x=m=x;
```
which will result in a value of 0 for both `m` and `x`.

Finally, how about the following expression:

```
m=x=x;
```

What do you think will be the result?

Mixing Types

Expressions may contain objects of different types. When an operation involves operands of different types, the compiler tries to "promote" one of them. For example, if a `float` is to be added to an `int`, the `int` is promoted to `float`, and then the operation takes place.

PROMOTIONS AND DEMOTIONS

Although it may not look important to you, the compiler automatically converts one of the operands so the operation can take place. When an expression calls for an operation between an integer and a floating point number, the integer is first converted to the floating point equivalent, and then the operation takes place (we say that the `int` was "promoted" to `float`, in this case). Similarly, if a `float` result is to be assigned to an `int`, the compiler "demotes" the `float` result by truncating its decimal part and converting it to an integer.

You can control the conversions yourself by using typecasting. To convert a variable or an expression to another type, write the type into which you want the variable or the expression cast in parentheses immediately before the variable or the expression.

For example, `x=(float)m/(float)j;` guarantees that both `m` and `j` will be converted to `float` before they are divided.

Returning Values in Functions

Expressions can also use functions. Any function can generate a result of any type, and that result can be used as part of an expression. For a function to generate a result, you must do the following things:

- Specify the type of the result. For example, the type may be `int`, `float`, `long`, etc. `void` means that nothing will be returned as a result.
- Use a `return` statement to indicate the result you are returning.

The *return* Statement

The `return` statement simply consists of the keyword `return` followed by an expression. The value of the expression is returned as a result. After executing the `return`, the function ends—the next statement in the sequence (if there is one) will not be executed.

For example, suppose that you want to create a function to compute the position of a body in free fall. The formula in physics is as follows:

$h=\Omega gt^2$

in which *h* is the height, *g* is the gravity, and *t* is the time. Since *g* can be assumed to be constant near the earth's surface (32 feet per second squared or 9.81 meters per second squared), for any given time *t*, you can compute a value for *h*. In this case, you want *h* to be the result of this function. What is the type of this result? You probably want it to be a `float`, since the height may be measured in feet or meters, and it may have a fractional part.

If you call this function `height`, the actual code could be as follows:

```
float height (float t)
{
  return 0.5*g* t * t;
}
```

Instead of 1/2, I chose to use 0.5, which is the same. Beware! If you write 1/2, which implies the division of one `int` (1) by another (2), it will generate an `int` result equal to zero. This is not what we want! You can avoid this problem by making one or both of the operands a `float`. For example, 1./2 will work. However, every time the computer evaluates the formula, it will divide 1 by 2 again. It may be a little better to use 0.5.

C++ offers no operator for computing the square of a value. The simple solution is to recall that squaring *t* is the same as *t* times *t*. You may also use the function `sqr(t)`, as seen in the next section.

Using the *math.h* Library

Several of the mathematical functions that are used most are included in the header file `math.h`. Some of the most important functions are summarized below.

These functions take a double floating point argument (represented by *x*) and return a double floating point result. Since the conversion is automatic, you can supply arguments such as `float` and assign results to `float`, as well.

There are trigonometric functions to compute the sine, cosine, and tangent, as well as the arc sine, arc cosine, and arc tangent of a given argument. Be aware that the arcs must be specified in radians. Table 13.1 shows all the functions.

Table 13.1: Functions in `math.h`

FUNCTION	ITS PURPOSE
`sin(x)`	Returns the sine of *x*
`cos(x)`	Returns the cosine of *x*
`tan(x)`	Returns the tangent of *x*
`asin(x)`	Returns the arc sine of *x*
`acos(x)`	Returns the arc cosine of *x*
`atan(x)`	Returns the arc tangent of *x*
`exp(x)`	Returns the exponential of *x*
`log(x)`	Returns the natural (base e) logarithm of *x*
`log10(x)`	Returns the decimal logarithm of *x*
`sqrt(x)`	Returns the square root of *x*
`sqr(x)`	Returns the square of *x*
`fabs(x)`	Returns the absolute value of floating point *x*
`ceil(x)`	Returns the round-up value of *x*
`floor(x)`	Returns the round-down value of *x*
`rand()`	Returns a random integer (the range of the integer will vary with the compiler)

You may be interested in other mathematical functions that are included in `math.h`. However, we will not discuss them here.

An Example

You could use the program below to compute the square root of a value that is input from the keyboard:

```
#include "franca.h"
#include <math.h>
void mainprog()
{
   float value,root;
   value=ask("Enter a positive value:");
   if (value>=0)
   {
      root=sqrt(value);
      Cout<<root;
   }
   else
   {
      Cout<<"Sorry, negative value !";
   }
}
```

Are You Experienced?

Now you can…

Choose the appropriate type of variables to handle your data

Use expressions to compute results

Use functions from the `math.h` library in your programs

SKILL

FOURTEEN

WORKING WITH GRAPHICS

- Specifying coordinates to locate points on the computer screen
- Drawing and showing Screen Objects
- Moving Screen Objects
- Changing coordinate systems

In Skill 14, we will work with graphics, because they are an interesting application of numeric manipulation, and because they are a subject of growing importance in the computer field.

Novices usually cannot afford to learn how to use graphics because of the complexities involved. However, thanks to our special software, it is very easy to show and manipulate objects such as circles, squares, and boxes. A special class of objects—Screen Objects—will be used to help you improve your skills with graphics and numeric manipulation.

Objects of class `Circle`, `Square`, `Stage`, and others are not available in standard C++. They are only available with the special software developed for this book.

Dealing with Graphics

One of the most interesting applications of numeric computation is the manipulation of graphics on the computer screen. Graphics manipulation will serve two purposes:

- You will learn (and practice) how to deal with graphics, which is a very interesting thing in itself.
- You will learn (and practice) several aspects of numeric computation.

Locating Points on the Screen

To understand how graphics objects work, we must first understand how to locate points on the computer screen. Since the screen is a two-dimensional object, coordinates to locate a point on it must include two numbers, the *x* coordinate and the *y* coordinate.

This coordinate system is essentially the same one that was used in the short project in Skill 12 to place and locate the robot in a room.

On the computer screen, the point of origin is the upper-left corner—by definition, the point (0,0). The *x* coordinate increases from left to right (just like the convention used in geometry), but the *y* coordinate increases from top to bottom (unlike the convention used in geometry).

Coordinate Units

Only a finite number of points can be represented on the computer screen. Not only is the

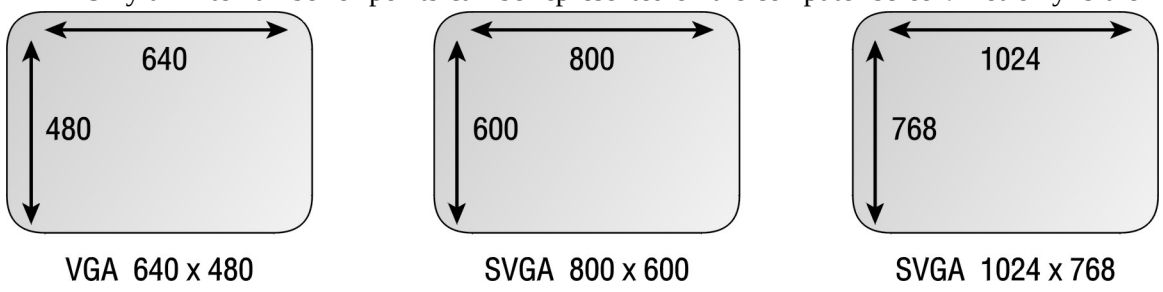


Fig 14.1 Common Screen Resolutions

size of the screen finite, but there is a minimum distance between points. The actual number of points may vary according to the computer display adapter that you are using. For example, if you are using a computer with a screen resolution of 1,024◊768, your screen can accommodate 1,024 points in a horizontal line and 768 points in a vertical line. Regardless of the size of your monitor's screen, your computer will be working with one given resolution. Figure 14.1 shows the most common resolutions available.

Since the number of points that can be represented in each line is finite, it is logical to use the actual number of points that can be drawn between two points as the distance between them. This is called the *pixel* coordinate system.

> 🕮 Pixel stands for *picture element*—a point that can be represented on the screen in a given color.

It is clear that it is possible to transform coordinates so you can work with inches or centimeters, instead of pixels. Changing coordinate systems will be studied later in this Skill.

In this pixel coordinate system, the coordinates of a given point are determined by the horizontal and vertical distances, measured in the number of points (pixels), from the point of origin in the upper-left corner of the screen to the given point.

Remember that, contrary to the usual geometric notation, the vertical coordinates increase from top to bottom (don't let this worry you—we will learn how to change the coordinate system later).

Drawing Screen Objects

To explore graphics applications, I have prepared a class of Screen Objects. This class is also included in the header file `franca.h`. Some of the interesting Screen Objects that we will use are as follows:

- Boxes (you used these in Skill 2)
- Squares
- Circles

If the object is a square or a circle, it is located on the screen by the coordinates of its center. If it is a box, it is located by the coordinates of its upper-left corner.

There are a few things that you can do with Screen Objects:

- You can place them anywhere on the screen. `place(x,y)` places an object's center at coordinates *(x,y)*. For example, `ball.place(20,20);`.

- You can show them on the screen (objects are not automatically shown). The `show()` function draws the object at its current location. For example, `ball.show();`.
- You can erase them from the screen. `erase()` erases the object by painting it white in its current location. For example, `ball.erase();`. Be sure not to erase the object before moving it.
- You can resize them. Screen Objects are created with a default size of 20. You may specify one or two arguments when resizing. If only one argument is specified—for example, `ball.resize(40)`—both dimensions (width and height) change to the same value. Otherwise, width and height will both be changed. For example, `ball.resize(40,30)` will transform the object into an ellipsis with a height of 40 and a width of 30. There are two functions to resize, `resize` and `absize`. The function `resize` is affected by any scales that you may have set up; `absize` is not affected by scales.
- You can change their color. By default, Screen Objects are created with a white fill color and a black contour color. You can specify one color to change the fill and the contour to the same color—for example, `ball.color(2)`—or you can specify two colors to change each to a different color—for example, `ball.color(2,4)`.

In addition, there are specific things you can do with a Box object:

- You can "say" something inside the box. For example, if `message` is declared to be a `Box`, `message.say("Here!");` displays the string `Here!` in the box. You may also use an integer or floating point number as an argument.
- You can provide a "label" for the box. Labels identify the kind of messages for which the box is being used. The labels are written above the message in the box. The statement `message.label("Your change:");` followed by `message.say(change);` displays a box with a label and a message.

For example, if the value of `change` is 12.45, the box above would look as follows:

Your change:
12.45

Boxes behave somewhat differently, because the box is automatically drawn when you want to say something. In other words, you don't have to show a box. Remember that box coordinates refer to the upper-left corner of the box, and not to the center.

It is also possible to change the point of origin and the scale. We will study these shortly.

Tables 14.1 and 14.2 summarize the orders you can give to Screen Objects.

Table 14.1: Orders You Can Give to a Screen Object

MESSAGE	ITS PURPOSE	ARGUMENTS
`place(float,float)`	Places object	*x,y* coordinates
`show()`	Shows object on screen	None
`erase()`	Erases object from screen	None
`resize(float)`	Resizes object	New size
`absize(float)`	Resizes without scaling	New size
`color(int,int)`	Changes object's color	Colors
`scale(float,float)`	Changes scales	New scales
`origin(float,float)`	Changes point of origin	*x,y* coordinates

Table 14.2: Orders You Can Give to a Box Object in Particular

MESSAGE	ITS PURPOSE	ARGUMENTS
`label("`*`Label`*`")`	Writes a label	A string of characters or an identifier representing a string of characters
`say("`*`Sentence`*`")`	Writes a sentence	A string of characters or an identifier representing a string of characters
`resize(`*`float`*`)`	Changes object's length	New length

Since boxes are Screen Objects, you can also `place()`, `show()`, and `erase()` them.

When you declare a `Circle`, it is assumed by default that the diameter is 20 pixels. If you want a circle with a different size, you can resize it.

```
Circle mycircle;
mycircle.resize(10);
```

creates a circle with a diameter of 10 pixels.

In a similar way, a `Square` is assumed to have a default side of 20 pixels (notice that this is the same as the circle's default diameter).

Adding Color

Screen Objects can be drawn using different colors. There may be two colors in an object, the fill color and the contour color. For example, the object may be a circle of red delimited by a black line. The colors are denoted by the following codes:

0	White
1	Red
2	Lime green
3	Blue
4	Light blue
5	Pink
6	Yellow
7	Black

It may be a good idea to use the following statement in your programs:

```
enum (white,red,green,blue,lightblue,pink,yellow,black);
```

If you use a number greater than seven, the remainder of the division by seven is used as the code.

Defaults

Here are some default assumptions when Screen Objects are created:

- Boxes are stacked vertically on the right side of the screen; other objects are positioned at (0,0).
- All objects are created with a white fill color and a black contour color.

Using Integer Values

Even though we are now dealing with coordinates in pixels, Screen Objects support coordinates and sizes with fractional parts—`float` values. If you use integer values, the compiler

automatically converts them to floating point values. It is important to learn how to use `float`, because we will use it later to make scale conversions.

The piece of program below creates a circle, places it at (50,50), and shows it on the screen.

```
#include "franca.h"
void mainprog()
{
   int x,y;
   x=50;
   y=50;
   Circle mycircle;
   mycircle.place(x,y);
   mycircle.show()
}
```

Remember that you have to declare the objects you want to use. You do this in the same way that you have already done it with other classes of objects.

As an improvement to this program, you can ask for the circle coordinates and show them in boxes. This is shown in the program `c5circ1.cpp`. If you execute this program and enter the coordinates (50,100), your computer screen should look like Figure 14.2.

```
#include "franca.h"
//                                  c5circ1.cpp

//  This program draws circles on the screen
//       at locations specified by the user.
void mainprog()
{
  Box coordx("X:"),coordy("Y:");       // Use boxes for
                                       //      x and y coordinates
  int x,y;                             // Declare coordinates
  Circle mycircle;                     // Declare a circle
  do
  {
    x=ask("Enter x coordinate:");
    y=ask("Enter y coordinate:");
    mycircle.place(x,y);               // Put the circle in place
    mycircle.show();                   // Show the circle
    coordx.say(x);                     // Write coordinates
                                       //      in boxes
    coordy.say(y);
  }
  while (yesno("Wanna try again?"));
}
```

Notice that two new objects are declared, `coordx` and `coordy`. Both objects are of type `Box`, as indicated in their declaration. The box `coordx` will be used to display the *x* coordinate of the circle, and the box `coordy` will be used to display the *y* coordinate of the circle. The boxes could be placed anywhere on the screen; they are automatically positioned by default.

The following lines:

```
x=ask("Enter x coordinate:");
y=ask("Enter y coordinate:");
```

request that you provide values for *x* and *y*. This lets you place the circle anywhere on the screen, instead of at (50,50), as before. The main piece of code is surrounded by the `do/while` repetition:

```
do
{
...
}
while(yesno("wanna try again?"));
```

What does this do? The loop repeats as long as the condition is true. Thus, as long as you choose *yes*, the program will keep asking for new coordinates and drawing circles. If, instead, you choose *no*, the loop will end.

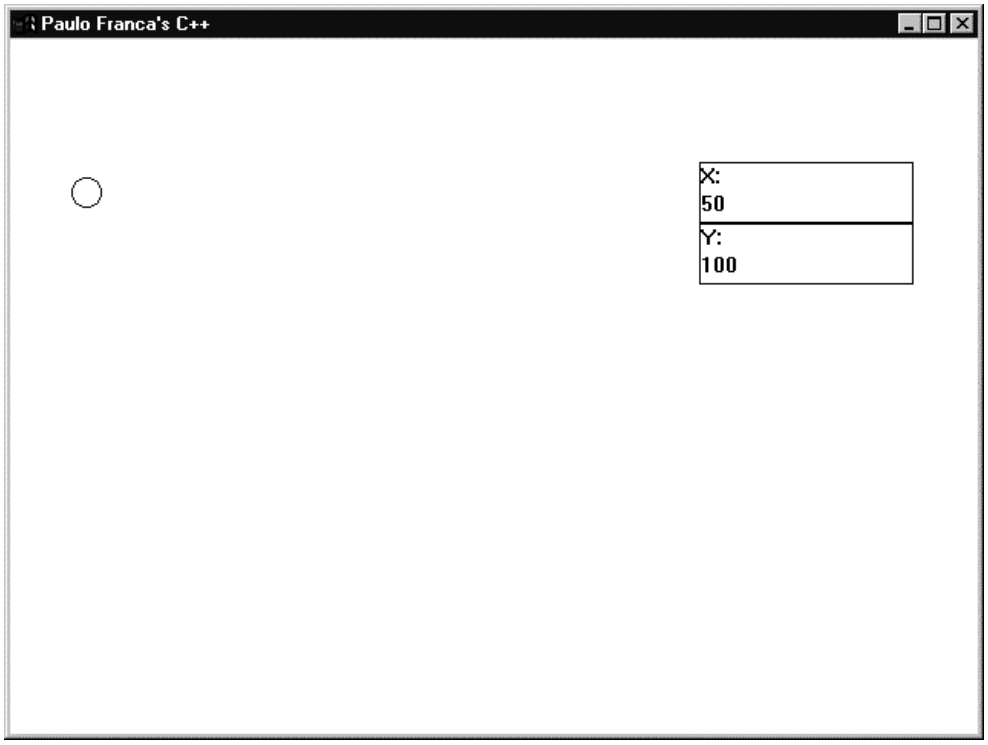


Fig 14.2 Result of c5circ1.cpp

Erasing

Each time we draw a new circle, the old drawing remains. What should we do if we don't want to see it anymore? How about using the `erase` function? Can you try it? It is quite simple! All you have to do is to include the following statement:

```
mycircle.erase();
```

in a convenient place in the program.

What do I mean by *convenient place*?

- You cannot erase the circle immediately after showing it. If you do so, you will not have enough time to see it. You could use this solution by declaring a `Clock` and telling the clock to wait a few seconds (objects of class `Clock` were introduced in Skill 2).
- You cannot erase the circle after placing it at the new coordinates. The `erase` function works by drawing a white circle on top of the old one to erase it. If you move the circle to a new place, the white circle will be drawn in the new place.

Besides using a `Clock` object, you may consider two alternatives:

- You can erase the circle immediately before changing its place. The first time that the program goes through the loop, you will be erasing the circle before actually drawing

it or even giving coordinates to it. There is nothing wrong with erasing before showing. However, erasing before placing the circle is the same as erasing in the wrong place!

Oops! If the circle is at the default (0,0) location and you try to erase it, you will get an *Object out of range* error message. This is because part of the circle will be off the screen.

- You can erase the circle after asking the *yes or no* question, but before restarting the loop! How? In this case, you would pose the question before you end the loop, and then save the answer. As a matter of fact, the `yes` function returns an integer value (1 means *yes*, 0 means *no*). You may use an additional variable to save this answer:

```
int answer;
     ...
     do
     {
     ...
        answer=yes("Wanna try again?");
        mycircle.erase();
     } while (answer);
```

Error Messages

Any attempt to draw a point outside the valid screen region will generate an error message in a dialog box. The *Object out of range* error message is shown in Figure 14.3. The wrong coordinate (*x* or *y*) will be displayed in the upper-left corner of the screen, and you will be requested to check the object that is out of range.

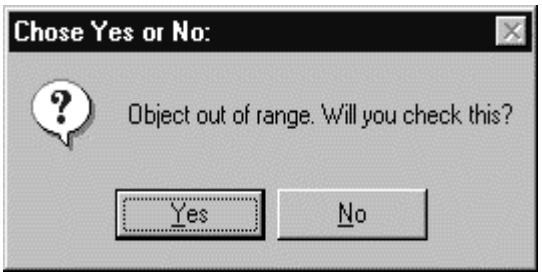


Fig 14.3 Object

Try These for Fun…

- Modify the program `c5circ1.cpp` to erase each circle before drawing the next one.
- Modify the program `c5circ1.cpp` to draw two circles (using two sets of coordinates), instead of drawing one circle. Both circles should be erased before drawing the next two circles.
- Write a program (or modify the program `c5circ1.cpp`) to draw several circles in a horizontal line. You should request the coordinates (*x*,*y*) of the first circle, the number of circles to draw, and the distance between them. (Hint: you can declare only one circle in this case.)
- Include a `Clock` object in `c5circ1.cpp` in the problem above to draw circles at 2-second intervals.

Moving Screen Objects

By now you should know how to draw and erase Screen Objects. The next thing we want to do is to move them around. Just like in movies or in cartoons, the illusion of movement is caused by drawing several images very quickly. This was already done with the athletes in Skill 1.

You can achieve a high-quality animation by displaying 30 frames per second. What does *30 frames per second* mean? In one second, you change your drawing 30 times. In other words, each drawing should last 1/30 of a second! If you have a very complex picture, you need a very fast computer to draw the next picture in less than 1/30 of a second. Since our pictures are simple, I don't expect you to run into this kind of problem.

To get started, look at the program `c5circ2.cpp`, which implements a very simple animation. If you run this program, you should see a circle moving on the screen.

Try running this program using the coordinates x=100 and y=100, and using 300 circles. Experiment with other values, as well.

```
#include "franca.h" // c5circ2.cpp

// This program moves a circle on the screen.
void drawing(Circle &mycircle,int x,int y,Box &coordx,Box &coordy)
{
 // This function draws one frame:
 mycircle.place(x,y);               // Put the circle in place
 mycircle.show();                   // Show the circle
 coordx.say(x);                     // Write coordinates in boxes
 coordy.say(y);
}

void mainprog()
{
 // Declaration of objects and variables:
 Box coordx(ìX:"),coordy(ìY:î);     // Use boxes for x and y
 int x,y;                           // Declare coordinates
 Circle mycircle;                   // Declare a circle
 Clock mytimer;                     // And a clock
 int howmany;

 do
 {
   x=ask("Enter x coordinate:");
   y=ask("Enter y coordinate:");
   howmany=ask("How many circles?");
   for(int k=1;k<=howmany;k++)
   {
     drawing(mycircle,x,y,coordx,coordy);
     mytimer.wait(0.033);
     mycircle.erase();
     x++;
   }
 }
 while (yesno("Wanna try again?"));
}
```

How Does This Program Work?

The program above uses three Screen Objects:

- `coordx`

- `coordy`
- `mycircle`

The objects `coordx` and `coordy` are boxes that display the current coordinates of the circle. The object `mycircle` is a circle that is moved on the screen.

A `drawing()` function produces a frame. This function does as follows:

- Receives the circle, the coordinates, and the boxes as parameters
- Places the circle at coordinates (*x*,*y*)
- Shows the circle
- Writes the coordinates in the appropriate boxes

The main program consists of two essential parts:

- Initialization
- Loop

The initialization declares the objects and labels the boxes.

The loop is repeated as long as the user replies *yes* to the *yes or no* question at the end. This loop requests that the user provide the *x* and *y* coordinates, as well as a number to specify how many circles (each circle corresponds to a frame) will be drawn. An inner loop calls the `drawing()` function several times to produce the frames.

Another alternative would be to declare and use a `Clock` inside the `drawing()` function. In this case, the function could have the following code:

```
void drawing(Circle &mycircle,int x,int y,
                    Box &coordx,Box &coordy)
{
 Clock mytimer;
 // This function draws one frame:
 mycircle.place(x,y); // Put the circle in place
 mycircle.show();     // Show the circle
 coordx.say(x);       // Write the coordinates in boxes
 coordy.say(y);
 mytimer.watch(.033);
 mycircle.erase();
}
```

The `drawing()` function keeps the circle displayed for .033 seconds, and erases it afterward. The `wait()` and the `erase()` could then be removed from the main function. As you can see, there are many ways to design and implement your functions.

> & To achieve a successful animation, keep the time that elapses between erasing an object and showing it again to the minimum duration possible.

Try These for Fun…

- Modify the program `c5circ2.cpp` to make the circle move vertically, instead of horizontally.
- Modify the program `c5circ2.cpp` to start with a circle of diameter 2, and increase the diameter at each iteration.
- Modify the program `c5circ2.cpp` to draw the circle with a different color.

Changing Coordinate Systems

It may not be convenient all the time to deal with coordinates in pixels. You may want to use your knowledge of geometry to work with a different system of coordinates.

Changing the Point of Origin

The simplest thing to do is to change the point of origin. If you don't wish to refer all the points to the upper-left corner, you can redefine the point of origin for your coordinate system.

Suppose that you want your new point of origin to be 100 pixels horizontally and 150 pixels vertically from the upper-left corner. This is the same as saying that the coordinates of your new point of origin are (100,150). If you want to place an object at the coordinates (*x*,*y*) using your new point of origin, the only thing you have to do is to perform a simple operation on (*x*,*y*) before you place the object on the screen.

In this case, the transformations are very simple. You can obtain a new set of coordinates (*x1*,*y1*) as follows:

```
x1=x+100;
y1=y+150;
```

In more general terms, if the coordinate of your new origin is (*x0*,*y0*), you can use the following expressions:

```
x1=x+x0;
y1=y+y0;
```

Notice that every time you want to draw an object in the position (*x*,*y*), you actually have to draw it in the position (*x+x0*,*y+y0*).

Changing Scales

You may want to work with distances in centimeters, inches, or anything besides pixels. In other words, you may want to change the scale.

This is also a simple operation. Suppose that in a space of 100 pixels, you want to represent 1,000 of your new units (for example, if you want to represent 1,000 feet in a space of 100 pixels). What do you do? Since your scale is now 1,000 units per 100 pixels—or 10 units per pixel—all you have to do is to divide your new coordinate by 10 to obtain the pixel coordinate. Therefore, if your horizontal scale is *scalex* and your vertical scale is *scaley*, you can obtain new coordinates by using the following expressions:

```
x2=x/scalex;
y2=y/scaley;
```

> & You cannot represent anything that is smaller than 1 pixel on the screen. In the example above, 10 feet will be represented by 1 pixel, which means that the 10-foot mark, the 12-foot mark, and the 18-foot mark will all be placed at the same location.

If you want to combine the change of the point of origin with the change of the scale, you can use the following expressions:

```
x3=(x+x0)/scalex;
y3=(y+y0)/scaley;
```

> & The factor for your new scale is obtained by dividing the number of units in your new coordinate system by the equivalent number of units in the original coordinate system.

Changing Orientation

Finally, how can we make the vertical coordinate increase upward, instead of downward? This is particularly relevant because most mathematical formulas use this notation.

What we want to do in this case is to simply change the sign of the vertical coordinate. This can easily be done by multiplying it by a negative number. Why not use a negative scale then?

A value of –1 for *scaley* would be enough to invert the orientation of the vertical axis.

As an example, let's use a different coordinate system that is located closer to the bottom of the screen, and that has a vertical-axis orientation from bottom to top.

Here is what we have to do:

- Determine the coordinates of our new origin. Suppose that we decide to use the point (50,400) as the new origin.
- Determine the scale and use a negative value for the vertical scale. Suppose that we want each unit to be 60 pixels. We may then use *scalex=1/60.* and *scaley=–1/60.*
- Translate our coordinate to the computer coordinate by using the expressions above every time an object is to be placed on the screen.

If we want to modify the program `c5circ1.cpp` to handle our new coordinates, the solution is quite easy. Since we must make sure that the object is placed at the transformed coordinates, we may simply correct the following statement:

```
mycircle.place(x,y);  // Put the circle in place
```

This statement should now be as follows:

```
mycircle.place((x+50)/60.,(y+400)/(-60).);
```

> & Do not forget to use both of the decimal points in the fraction! If the fraction has no floating point numbers, the result will be only the integer part, which in this case is zero!

This solution would work fine. However, it is a very simple program that places only one object in only one spot. If many other statements placed objects on the screen, you would have to include this new expression in all of them.

Also, if you decide later that either the point of origin or the scale was not well chosen, you will again have to go through all those statements to use the new values. A better idea would be to use functions to transform the coordinates. If you assume that you declare the following variables globally:

```
float scalex=1./60.,scaley=-1/60.;
float x0=50.,y0=150.;
```

the functions could be as follows:

```
float newx(float oldx)
{
   return (oldx+x0)/scalex;
}
float newy(float oldy)
{
  return (oldy+y0)/scaley;
}
```

and the circle should be placed using the following statement:

```
mycircle.place(newx(x),newy(y));
```

Try These for Fun…

- Modify the program `c5circ1.cpp`, which draws circles at given coordinates, to use a new coordinate system with a point of origin at (50,400), and to scale 0.01 in each direction. The vertical axis should increase upward.
- Modify the program `c5circ1.cpp` to request that you input the new point of origin and scales.

No Need for New Functions

The good news is that you do not need to write functions or to modify your programs to use a different coordinate system. A mechanism for changing the coordinate system is already embedded in the Screen Objects.

By telling a Screen Object to `scale()`, you can change the scales for the horizontal and vertical axes. Of course, if you supply a negative scale for the vertical axis, you will also change the orientation of *y*. Notice that you will be changing the scale for all objects, not only for the object to which you sent the message! Any drawings that are already on the screen will remain unaffected, but any other object that you move, erase, or show will be subject to the new scale.

Similarly, you can change the point of origin of your coordinate system by telling any screen object to `origin()`. For example:

```
Circle ball;
ball.scale(1.,-1.);
ball.origin(20,400);
```

causes the new origin to be located at (20,400), and causes the orientation of the vertical axis to increase upward. The new coordinates are always expressed in pixels, using the original scale and orientation. If an object is invoked to change either scale or origin, the new change will again affect all objects.

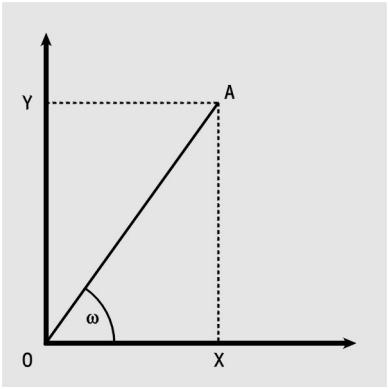
> & If you set different values for the horizontal and vertical scales, squares and circles will be distorted! Also, you should place boxes in their appropriate locations when you redefine scales. Otherwise, the mechanism that automatically creates one box under another may assign a location that is off the screen to your boxes.

The *Grid* Object

`Grid` is an object that can represent the horizontal and vertical axes. Declare `Grid` like you declare any other Screen Object. When the grid is shown, the horizontal and vertical axes will be drawn to intersect at the point of origin (0,0).

Try This for Fun…

- Modify the program `c5circ1.cpp` to accept a given point of origin and scale, as well as to display the *x*-*y* axis.



Using Polar Coordinates

Another coordinate system that we may be interested in using is the polar-coordinate system. In this system, instead of expressing coordinates by the distances to the *x* and *y* axes, the coordinates are expressed by the distance to the point of origin (called the *radius*), and by the angle made with the horizontal axis.

Figure 14.4 shows both coordinate systems. Point *A* can be located by giving the (*x*,*y*) coordinates, or by giving the distance from the origin (*O*) to *A* and the angle that the segment *OA* forms with the horizontal axis.

Fig
14.4 Polar

Polar coordinates are very useful and easy to deal with when describing circular motions. On the other hand, it is easier to manipulate objects on the computer screen using screen coordinates. Since the screen coordinates are expressed in (*x*,*y*) form, it is interesting to learn how to transform polar coordinates into (*x*,*y*) coordinates.

Given a point *A*, defined by the following polar coordinates:

Distance to origin: r=OA

Angle: w=xOA

we may compute as follows:

```
x = r* cos (w);
y = r* sin (w);
```

It is common to have the angle denoted by θ (the Greek letter *theta*).

If point *O* (the origin of the polar coordinate) is located at the coordinates (*x0*,*y0*) in the (*x*,*y*) system, the expressions above can be modified as follows:

```
x = r* cos (w) + x0;
y = r* sin (w) + y0;
```

It is then possible to write a C++ function to transform a point from polar coordinates to screen (*x*,*y*) coordinates. You may choose to write one function to obtain the *x* coordinate and another function to obtain the *y* coordinate:

```
float coordx(float r,float theta, float x0)
{
   return r*cos(theta)+x0;
}
float coordy(float r,float theta,float y0)
{
   return r*sin(theta)+y0;
}
```

Or, you may choose to write a single function that computes the two values, and returns them in one of the arguments:

```
void polarxy (float r, float theta, float x0,float y0,
              float &x, float &y)
{
  x=r*cos(theta)+x0;
  y=r*sin(theta)+y0;
}
```

Either solution will work fine.

> **&** The last two parameters, *x* and *y,* are passed by reference (they are preceded by &). If this is not done, the function will not be able to alter the values of the real arguments!

Are You Experienced?

Now you can…

Locate a point on the screen by using coordinates

Use Screen Objects such as Circles, Squares, and Boxes

Place, show, move, and erase Screen Objects anywhere on the screen

Change coordinate systems

SKILL

FIFTEEN

CREATING ANIMATIONS

- Drawing function graphs
- Simulating movements on the computer screen
- Handling several Screen Objects at one time
- Completing a short project—Sun, Earth, Moon

Now that you know how to manipulate numbers to compute the coordinates of objects, you can move them to produce an animation. The Screen Objects that you learned to use in Skill 14 can be moved according to your instructions to produce an animation.

While you learn how to create animations, you will also learn how to use Screen Objects to draw the graph of a mathematical function. This is a very simplified application of animation. You don't really have to be concerned with the speed at which you are moving, and the vertical coordinate *y* is given by a mathematical function that depends on the value of the horizontal coordinate *x*. While you learn how to draw functions, you will also learn how to pass a function name as a parameter to another function.

The process of animation is very simple, and it essentially consists of repeating the following steps:

1. Place the objects that you want on the screen in the desired location.
2. Show them for an appropriate amount of time.
3. Erase them.

When you have to deal with several Screen Objects that move together, the program may become lengthy and repetitive. A new kind of Screen Object—the `Stage` object—will help you to manipulate several objects as if they are only one object.

Drawing Mathematical Functions

Since you now know how to place and to move objects on the screen, and you also know how to change coordinates and scales, it will be very easy to draw the graph of a function. Given a mathematical function y=f(x), we can display the graph of this function by moving a dot (a small circle) along the coordinates (x,f(x)).

Suppose that our function is y=x2–x+1, and that we want to view the graph between the points x=–5 and x=20. It is a good idea to define the mathematical function as a C++ function:

```
float f(float x)
{
  return x*x-x+1;
}
```

This is a good solution, because if we want to compute the function somewhere else in the program, we may simply refer to f(x), instead of writing the complete expression. Also, it will make it much easier to draw the graph of a different function, as we will see later.

We must now place the dot at the initial point, and then move it to the final point. Of course, we should not erase the points that we draw!

Moving a Dot

The procedure for drawing the function is very easy:

- Declare `dot` as a small circle.
- Repeat the steps below for different values of *x*, ranging from an initial value to a final value, at small intervals (such as 0.01).
    - Place `dot` at new coordinates.
    - Show `dot` on screen.

In C++, this could be written as follows:

```
Circle dot;                       // Declare dot a circle
dot.color(7,7);                   // Use the color
dot.resize(2);                    // Make dot small
for (x=xstart;x<=xend;x=x+0.01) // Loop from xstart to xend
{
   dot.place(x,f(x));
// Place dot at x,y
   dot.show();
// Show dot
}
```

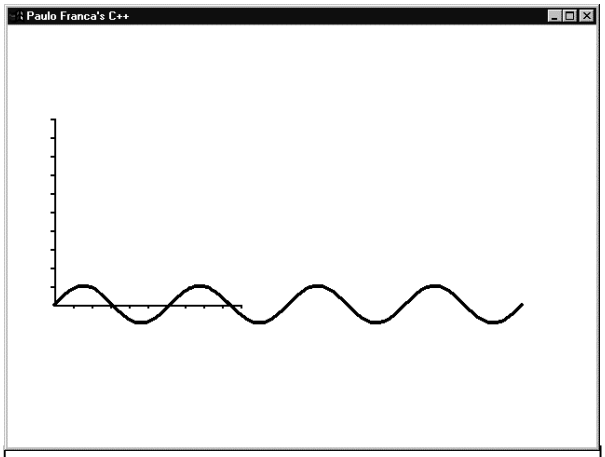


Fig 15.1 The result of

You must make sure that you do not attempt to draw an object outside the boundaries of your screen. Always transform your expressions or use scales to make sure that the ranges for *x* and *y* are respected:

- *x* must be between 0 and 640 (both inclusive).
- *y* must be between 0 and 480 (both inclusive).

Of course, if you want to use scales and to draw the *x* and *y* axes, it has to be done before the function is drawn. It may also be a good idea to transform the piece of program above into a function. Why? Functions are easier to reuse at a later time. If you want to write another program to draw a mathematical function, you can reuse the same function and save time. Shall we explore some alternatives?

The *draw()* Function

One possible idea is to take the code above and transform it into a C++ function. Initial and final values for *x* can be specified as arguments, and we may assume that the coordinate system is already set. This would result in the following code:

```
void draw(float xstart,float xend,int dotcolor=7)
{
  Circle dot;                     // Declare dot a circle
  dot.color(dotcolor);            // Use appropriate color
  dot.absize(4);                  // Make dot small
  for (x=xstart;x<=xend;x=x+0.01) // Loop
  {
     dot.place(x,f(x));           // Place dot at x,y
     dot.show();                  // Show dot
  }
}
```

In this case, the main program could be as follows:

```
void mainprog()
{
 ...
   draw(-5.,20.);
}
```

A complete program is supplied to draw the sine function. The resulting screen is shown in Figure 15.1.

Here is the program `c5sin.cpp`:

```
#include "franca.h"
#include <math.h>
            // c5sin.cpp
            // Program to draw graph of sin(x).
float f(float x)
{
  return sin(x);
}

void draw (float firstx,float lastx,int color=7)
{
 float x;
 Circle dot;                    // Declare dot a circle
 dot.color(color,color);        // Use a colored dot
 dot.absize(4);                 // Make dot small
 for (x=firstx;x<=lastx;x=x+0.1) // Loop
 {
   dot.place(x,f(x));           // Place dot at x,y
   dot.show();                  // Show dot
 }
}

void mainprog()
{
  // Use a grid, set scales and origin:
  Grid mygrid;
  mygrid.scale(20.,-20);
  mygrid.origin(50.,300.);
  mygrid.resize(10.);
  mygrid.show();
  const float pi=3.14159;
  float xstart=0.;
  float xlimit=8.*pi;
  draw(xstart,xlimit);
}
```

Using the `draw()` function is a reasonable solution to this problem. However, you must always ask yourself whether there is anything you can do to make your solution more general and more reusable. In this case, if you want to reuse the `draw()` function in the future, what kind of obstacles might you face?

Two issues may arise:

- Scale—couldn't the `draw()` function include a scale and axis definition?
- Function name—what if you want to draw a function that has a different name?

The scale problem is relatively easy to fix. You might request the data of the new coordinate system as arguments to the `draw()` function, and declare your own `Grid` object in the function. You may want to think twice before doing this. It would be a better idea to request that the coordinate system is established before the function call.

The function-name problem is more interesting. At first, you may not think it is very relevant. After all, you could simply change the name of the function and reuse exactly the same code. However, this is not true. There may be cases in which you want to draw the graphs of more than one function in the same program. How would you do it?

It is simple—pass the function name as an argument to the`draw()` function.

Passing Functions as Arguments

C++ allows you to pass a function as an argument to another function. In other words, if you have two functions, f1(x) and f2(x), that you want to use to draw a graph, it is possible to redesign the `draw()` function to accept three arguments, instead of two. The new argument is the name of the function. When you call the new `drawf()` function, you would use statements such as the following statements:

```
drawf(f1,xstart,xend);
drawf(f2,xstart,xend);
```

This is not very different from the usual function call. We are providing three arguments: the function to be used (f1 or f2), the starting value of *x* (xstart), and the ending value of *x* (xend). The only thing that will be different is the declaration of the function `drawf()`. The original `draw()` function was declared as follows:

```
void draw(float xstart,float xend)
```

The new `drawf()` function must have another parameter that specifies the function—but the question is, What is the type of this argument? We know it is neither an `int` nor a `float`. What could it be? This parameter is a function.

Functions as Parameters

Function types are declared by summarizing the appearance of the function. To do this, you must include the following items:

- The return type (for example, `float`, `void`, etc.)
- A symbolic name with which to refer to the function (for example, `func`)
- The list of argument types inside parentheses (for example, `(float)`)

In our example, the description of the `drawf()` function could be as follows:

```
void drawf ( float func(float),float firstx,float lastx)
```

Notice that this function has three parameters:

- A function that takes a floating point variable as an argument, and that returns `float`. This function is denoted by the symbolic name `func`.
- Two floating point variables. These variables are denoted by the symbolic names `firstx` and `firsty`.

The complete function listing is shown below.

```
void drawf (float func (float), float firstx,float lastx)
{
  float x;
  Circle dot;                       // Declare dot a circle
  dot.color(7,7);                   // Use a black dot
  dot.resize(2);                    // Make dot small
  for (x=firstx;x<=lastx;x=x+0.01) // Loop
  {
     dot.place(x,func(x));          // Place dot at x,y
     dot.show();                    // Show dot
  }
}
```

When we place the dot using arguments `x` and `func(x)`, the function whose nickname is `func` will be called with `x` as an argument. The result will then be used as the vertical coordinate to place the dot.

Try This for Fun…

- Develop a program to draw the graph of the sine and cosine functions. The sine should be drawn in red, and the cosine should be drawn in blue. Hint: the sine and cosine functions are available if you include the header file `math.h`. The resulting screen should look like Figure 15.2.

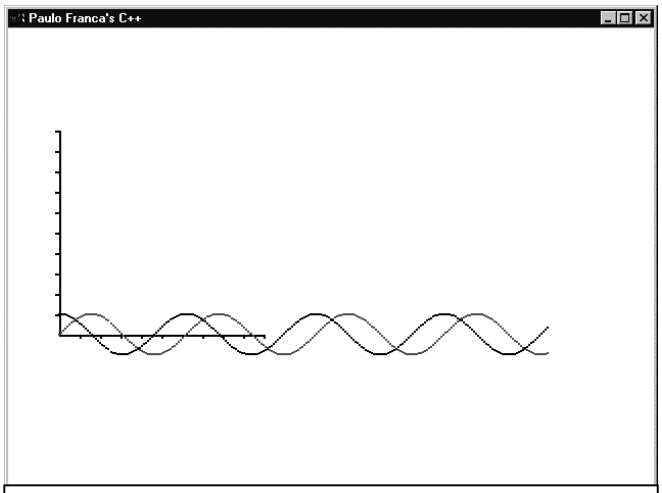


Fig 15.2 Sine and Co-sine

Producing Animations

We will now use Screen Objects to produce an animation. A simple animation was already introduced with the program `c5circ2.cpp`. This section shows you how to use Screen Objects to illustrate some of the movement equations you may know from physics.

Creating the Illusion of Movement

It is possible to simulate movement on the computer screen in the same way it is done in motion pictures—by providing different images after small intervals of time. This approach enables you to produce several kinds of animations.

One of the easiest animations is to express all movements by a mathematical formula. This is the case in many physical phenomena. Several of the phenomena studied in physics have equations that can determine the position of a given body.

Simulating Composite Movements

Suppose that there is a body in movement, and that you know the equations x=dist(t) and y=height(t) that specify the horizontal distance to the origin (*x*) and the height above the ground (*y*).

As they usually do in physics, these equations express *x* and *y* as functions of the time (*t*).

Some examples follow.

Uniform horizontal movement

Here is an example of uniform horizontal movement:

```
x=speedx*t+x0;
y=y0;
```

in which `speedx` is a constant horizontal speed, and `y0` is any constant value. A constant `x0` is included to set an initial horizontal position for the body.

Uniform horizontal and vertical movement

Here is an example of uniform horizontal and vertical movement:

```
x=speedx*t+x0;
y=speedy*t+h0;
```

in which `speedx` and `speedy` are constant speeds in the horizontal and vertical directions, respectively.

Uniformly accelerated movement

If the body is subject to a constant acceleration, the following lines may also be included in the equations:

```
x=accx*t*t/2+speedx*t+x0;
y=accy*t*t/2+speedy*t+y0;
```

If you look carefully, you might notice that as long as you provide the functions `dist(t)` and `height(t)`, the movement simulation will be exactly the same no matter what kind of movement we are trying to simulate.

If the functions are properly written, we can develop a program to produce an animation using the following strategy:

- Declare a Screen Object to represent the moving body (a circle, for example).
- Declare a `Clock` object to keep track of the time.
- Perform a loop causing a standard interval of time to elapse from one frame drawing to the next (for a smooth animation, use 1/30 of a second). This loop will consist of the following items:
    1. Compute the new coordinates as functions of *time*.
    2. Move the body to the new position.
    3. Show the body.
    4. Wait for the time interval to exhibit the next frame (1/30 of a second).
    5. Erase the body from the screen.

Our next piece of program will do all of the above.

A body in free fall

The program `c5body.cpp` assumes two functions `dist(t)` and `height(t)` to compute the *x* and *y* coordinates of a moving body. The current coordinates of the moving body and the current value of `time` are displayed in appropriate boxes.

```
#include "franca.h"                 // c5body.cpp
#include "math.h"
Grid mygrid;
// Use standard coordinates:
const float v0x=10.,v0y=20.;
float dist (float t)                // Function to determine
{                                   //      the x coordinate
   return v0x*t+10;
}

float height (float t)              // Function to determine
{                                   //      the y coordinate
   return v0y*t+50.;
}

void mainprog()
{
  // Declare and label boxes for displaying
  //      x, y, and time:
  Box boxx("x:"),boxy("y:"),boxt("Time:");
  mygrid.show();                    // Show axis
  float t;                          // t is the time
  Circle body;                      // Declare the body
  body.resize(12);                  // Size it at 12
  body.color(3,3);                  // Color it
  Clock timer;                      // Declare a timer clock
  Clock mywatch;                    // Declare another clock
  // The loop below will be repeated
  //      as long as the time is less than 15 seconds.
  //      Note that the current time is copied into
  //      the variable t:
  while ((t=mywatch.time())<15.)
  {
     boxx.say(dist(t));             // Update value of x, y,
     boxy.say(height(t));           //      and time in the
     boxt.say(t);                   //      appropriate boxes
     body.place(dist(t),height(t)); // Place the body in
                                    //      current location
     body.show();                   // Show the body
     timer.watch(.033);             // Wait until timer reaches
     timer.reset();                 //      .033 seconds and reset
     body.erase();                  // Erase the body
   }
}
```

The program simply follows the general strategy mentioned above, and, by now, you should be able to understand how it works. However, it may be worthwhile to explore a few items.

Using Two Clocks

There are two objects of type `Clock` declared, `mywatch` and `timer`. We use one as a regular watch (such as your wristwatch) and the other as a stopwatch (a timer). The stopwatch is really handy because it will make it easier for us to draw a new frame every .033 seconds.

The *while* Expression

There is a very tricky expression used in the `while` statement. We want to keep looping while `time` is less than 15, but we also want to copy the value of `time` into the `t` variable. We could solve this problem by using two separate steps:

```
while(mywatch.time()<15)
{
   t=mywatch.time();
   ...
```

In this case, this would work fine. However, you may notice that the value assigned to `t` is a little greater than the value tested in the `while` expression.

You might be tempted to use an expression as follows:

```
while (t=mywatch.time()<15)
```

instead of using:

```
while ((t=mywatch.time())<15.)
```

Unfortunately, the first expression will produce an incorrect result! Expressions are evaluated in C++, and then the value is assigned to the variable on the left side of the equal sign. First, `time` is compared with 15, and this comparison produces a result—the integer *1*, which stands for *true*. Then, this result is assigned to `t`. As you can see, as long as `time` is less than 15, the value of `t` will be 1.

Using the `watch` member function is more convenient for simulating than using the `wait` member function, because you can make the frames appear at more precise intervals. The time that it takes for the computer to prepare the next frame is implicitly added when you `wait`, but not when you `watch`.

Handling More Generic Movements

The example above uses functions that represent the equations of uniform movement in both coordinates. As a result, you will see the circle move in a straight line at a constant speed when you run the program. However, the same program can be used for any kind of movement. All we have to do is to change the function definitions!

As an example, let's simulate a body that is dropping while moving horizontally at a constant speed. In this case, the equations for the *x* coordinate will still be the same as above, but the equations for the *y* coordinate will be as follows:

$$y=h_0 + v_{0y} t - {}^1/_2 g t^2$$

or, in C++:

```
y=h0+v0y+0.5*g*t*t ;
```

The following items are assumed:

- h0 is the initial height at time t=0.
- v0y is the initial vertical speed at time t=0.
- g is the gravity acceleration (9.81 m/s^2 or 32.18 ft/s^2).
- The body is dropped with no vertical speed—v0y=0.

The function `height(t)` could then be defined as follows:

```
float height(float t)
{
   const float h0=200.,g=32.18;
   return h0+0.5*g*t*t;
}
```

ON GLOBALLY DEFINED CONSTANTS AND VARIABLES…

The functions we are using in this section can have variables other than the time (for example, h0, V0, and g). If so, try to use these variables as arguments to the function, as well. A previous example used V0x and V0y as globally defined constants to avoid passing them as arguments. Although the practice of using global constants may not be harmful to your programming habits, the practice of using global variables should be avoided.

Substitute the definition above for the function `height(t)` in the program `c5body.cpp`, and then run it to see the results. But wait—if you do only this, your body will drop until it falls off the screen. After all, there is no floor to hold it! It may be a good idea to check the height, and to stop the program when the height becomes negative. There are two alternatives:

- You can expand the condition in the `while` statement so the loop is repeated while the time is less than 15 *and* while the height is greater than or equal to zero. For example:

```
while (((t=mywatch.time(t))<15.)&&(height(t)>=0))
```

Notice the parentheses in the statement above!

- You can test `height(t)` in the loop, and then break out of the loop if the result is negative. For example:

```
if (height(t)<=0) break;
```

Try This for Fun…

- Modify the program `c5body.cpp` to simulate a body in free fall. Stop the simulation when the body reaches the height of zero.

Efficiency Matters

It is a well-known fact that computers are very fast at their work. However, it is up to the programmer to avoid unnecessary work for the computer, so all the computations can be done in the appropriate time. If you look in the program `c5body.cpp`, you may notice that the functions `dist(t)` and `height(t)` are called twice in the loop. There is nothing wrong with that, except that both times the functions are called with the same value of `t`—both calls will result in the same value!

Is that a smart way to use the computer? Not really. If you compute a value and expect to use it several times in the program, you may be better off to keep this value in a variable, and then to avoid computing it over and over again. In this case, you could declare two additional variables:

```
float x,y;
```

and then modify the loop as follows:

```
while ((t=mywatch.time())<15.)
  {
     x=dist(t);
     y=height(t);
     boxx.say(x);       // Update value of x, y,
     boxy.say(y));      //      and time in the
     boxt.say(t);       //      appropriate boxes
     body.place(x,y);   // Place body in
                        //      current location
```

If you want to be really fancy, you can assign the values to `x` and `y`, and tell the boxes to `say`:

```
boxx.say(x=dist(t));
boxy.say(y=height(t));
```

This second alternative does not bring significant savings, and it may make your program less legible.

Another issue is the use of expressions that use constants, such as 1./2. The computer will actually divide 1 by 2 every time this expression is found in the program. You can save time by using the equivalent value of 0.5. You can also define a constant or a variable that has the value you want to use. For example:

```
const float pi2=3.14159/2;
```

Then, use `pi2` in the program.

Simulating a Cannonball

Our next simulation deals with a cannon firing at different angles. As the cannonball starts to move, it has a velocity in the direction that the cannon is firing. This direction is given by an angle *theta*. As you may know from physics, you can deconstruct this velocity into components:

velocx=veloc*cos(theta)

velocy=veloc*sin(theta)

The movement of the cannonball can then be expressed by the functions `dist(t)` and `height(t)`. Horizontally, the cannonball moves at a constant speed (uniform movement), whereas vertically, its movement is subject to the acceleration of gravity.

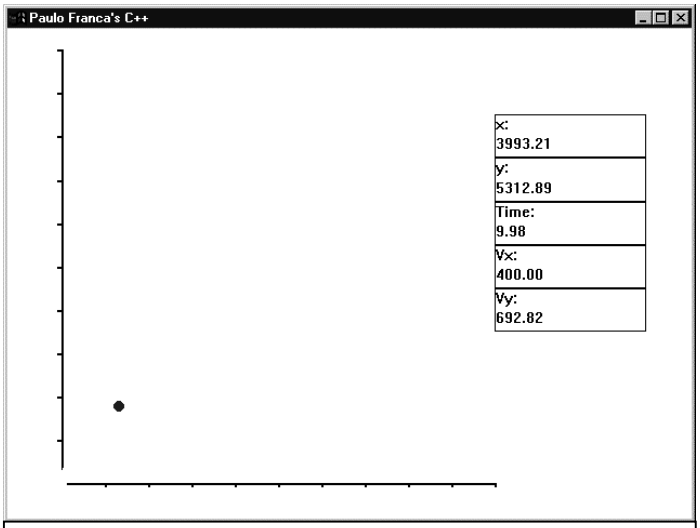


Fig 15.3 c5cannon at a certain point in time

The functions could be defined as follows:

```
const float g=32.18;          // Gravity
// Functions to compute coordinates:
float dist(float velocx,float t)
{
   return velocx*t;
}
float height(float velocy,float t)
{
   return velocy*t - 0.5 * g * t * t;
}
```

You can define `g` as a global constant.

The main program should request that you give the angle *theta* and the initial velocity *veloc*. The program then computes the values of `velocx` and `velocy`. After doing this, the program can simulate the movement as seen in previous programs.

Remember that trigonometric functions require that angles are expressed in radians, so we should convert them before we use them.

Computing the Cannonball's Coordinates

The program `c5cannon.cpp` implements the simulation of the cannonball being fired. The program is straightforward, and it includes the following items:

- Functions that determine distance and height
- Setting of initial conditions
- Simulation loop

The functions have already been explained. The setting of initial conditions determines scales, draws the x-y axis, creates the circle that represents the cannonball, and initializes the clocks needed for the simulation. Only one of these issues may confuse you—the setting of scales.

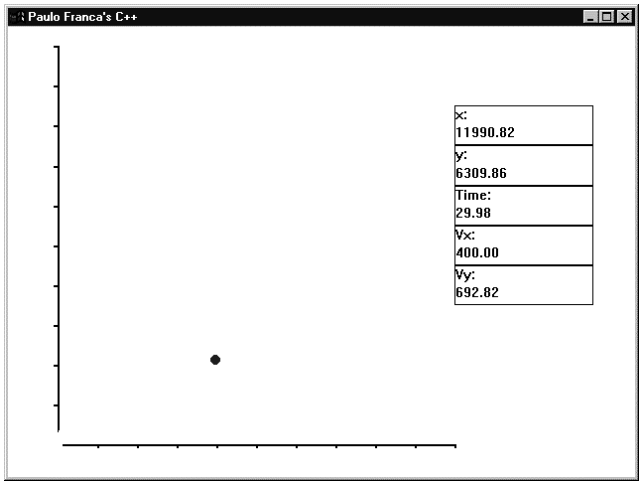


Fig 15.4 c5cannon at a later time

The scales should be set to accommodate all the objects we want to draw on the screen. It is necessary to look at the `height()` and `dist()` functions to determine the maximum values that will be used for *x* and *y*. Unfortunately, these values depend on the values that are input for the velocity and the angle. One thing that makes computer programming difficult is that you don't always know how your programs are going to be used!

The best we can do is to make a reasonable guess. If we assume a speed such as 1,000 feet per second, we may find that the horizontal distance can reach a little less than 30,000 feet, with a 45-degree angle (which gives the maximum range). If the cannon is fired at a 90-degree angle (gosh—who is in charge of this cannon?), the maximum height will be a little more than 15,000 feet.

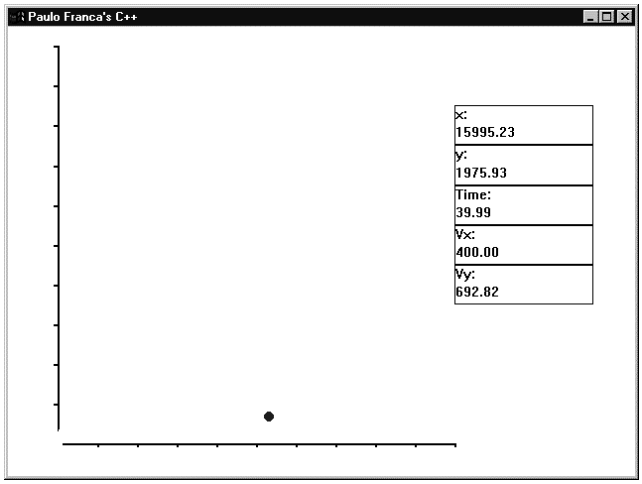


Fig 15.5 c5cannon yet even later

It is a good idea to use the same scale for both axes, because both axes represent distances. Therefore, we can assume a maximum of 30,000 feet for both height and distance.

Calculating for VGA

On the other hand, even though our screen can usually accommodate 640◊480 pixels, it may be a good idea to restrict the pictures to a smaller area—for example, 400◊400. This means that the 30,000 feet we will show in the simulation should fit in 400 dots on the screen. As a result, our scale, which expresses how many dots there are per foot, will be 400/30,000.

If you ever become confused while computing scales, it will help you to consider the units. In the case above, we had 30,000 feet to correspond to 400 dots. At first, you may be confused: should you use 400/30,000 or 30,000/400? If you include the units, which would be dots per foot in the first case and feet per dot in the second, you will notice that the second alternative gives you square feet per dot as a result when you multiply the distance (in feet) by your scale. Of course, this is wrong!

The Simulation Loop

The most interesting part of the program appears in its second part—the simulation loop. The loop is repeated while the time, as reported by the sidereal clock, is less than 50 seconds. In the loop, new coordinates are computed for the new value of *time*. The circle is then erased from the old position, placed in the new position, and then shown. The stopwatch is then instructed to wait for 0.033 seconds, so we can see the frame on the screen. After this, the stopwatch is reset, and the loop resumes.

Figures 15.3, 15.4, and 15.5 show the results of executing the cannonball simulation program at different times.

The listing for the `c5cannon.cpp` program is shown below.

```
void mainprog()           // Part 1
{

  Grid mygrid;
  mygrid.origin(50.,420);
  mygrid.scale(400./30000,-400./30000);
  // Declare and label boxes for displaying
  //      x, y, and time:
  Box boxx("x:"),boxy("y:"),boxt("Time:");
  Box vx("Vx:"),vy("Vy:");
  mygrid.resize(30000.);
  mygrid.show();            // Show axes
  float t;                  // t is the time
  Circle body;              // Declare the body
  body.absize(10.);         // Size it
  body.color(3,3);          // Color it
  Clock timer;              // Declare a timer clock
  Clock watch;              // Declare another clock
  float theta,veloc,velx,vely;
  // Request data:
  theta=ask("Enter angle of firing:");
  theta=theta*3.14159/180.;// Convert to radians
  veloc=ask("Enter initial velocity:");
  velx=veloc*cos(theta);    // Deconstruct into x and y
  vely=veloc*sin(theta);
  vx.say(velx);
  vy.say(vely);
 float x,y;                 // c5cannon.cpp—Part 2
```

```
  // The loop below will be repeated
  //      as long as the time is less than 15 seconds.
  //      Note that current time is copied into
  //      the variable "t":
  timer.reset();
  watch.reset();
  while ((t=watch.time())<50.)
  {
    y=height(vely,t);
    if (y<0) break;
    x=dist(velx,t);
    boxx.say(x);           // Update values of x, y,
    boxy.say(y);           //      and time in the
    boxt.say(t);           //      appropriate boxes
    body.erase();
    body.place(x,y);       // Place the body in current location
    body.show();           // Show the body
    timer.watch(.033);     // Wait until timer reaches
    timer.reset();         //      .033 secs and reset
  }
 }
```

Try These for Fun…

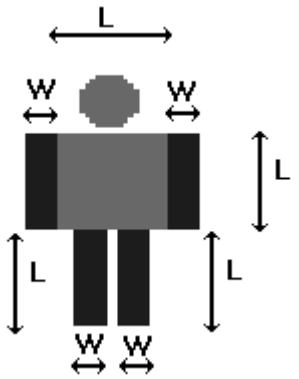
- Use the `dist(t)` and `height(t)` functions to write a program that finds the maximum value of height for given values of velocity and angle. Use boxes to display the current values of distance, height, and time, and to display the maximum value that was observed of height. You can use the general structure of the cannonball program.
- Modify the `c5cannon.cpp` program to find the maximum distance reached by the cannonball, which is the value of *x* when height drops to zero.

Handling Multiple Screen Objects—the *Stage* Class

There are cases in which you have an object that can be drawn as a set of other Screen Objects. Remember the athletes? How can you draw one of them?

The athlete figure, like the figure of most real people, is represented by the following parts:

- Head—represented by a circle
- Trunk—represented by a square
- Left and right arms—represented by rectangles
- Left and right legs—represented by rectangles



For the sake of simplicity, we use only two values for sizing the body parts. We can call these values *L* and *W*, as shown in Figure 15.6. The head is a circle whose radius is given by the value of *L*. The arms and the legs are rectangles whose width is given by the value of *W*, and whose length is given by the value of *L*. The trunk is a square whose sides are each given by the value of *L*.

Fig 15.6 An

Building an Athlete Piece by Piece

Can you draw an athlete? There's the hard way, and there's the easy way. You may start by declaring the objects that represent the body parts (the hard way!). A little later, you will learn how to use the `Stage` class (the easy way!).

```
Circle head;
  Square trunk;
  Square leftleg,rightleg;
  Square leftarm,rightarm;
```

Since we know how to deal with circles and squares, all we have to do now is to position these objects in the right places, and to assign them the appropriate colors and sizes.

Assume that the athlete is enclosed by a square whose center has the coordinates *x* and *y*. Furthermore, let's declare some constants to hold the values of *W* and *L*, using the names `armwidth` for *W* and `armsize` for *L*.

```
const int armsize=20;
const int armwidth=6;
```

Coloring Your Athlete

The following statements can be used to color the objects:

```
head.color(5,5);
trunk.color(5,5);
leftarm.color(3,3);
rightarm.color(3,3);
leftleg.color(3,3);
rightleg.color(3,3);
```

Scaling Your Athlete

Next, you can make all the objects assume the appropriate sizes:

```
head.resize(2*armwidth);
trunk.resize(armsize);
leftarm.resize(armsize,armwidth);
rightarm.resize(armsize,armwidth);
leftleg.resize(armsize,armwidth);
rightleg.resize(armsize,armwidth);
```

Placing Your Athlete

Then, you can place all the objects:

```
head.place(x,y-armsize/2.-armwidth);
trunk.place(x,y);
leftarm.place((x-(armsize+armwidth)/2.),y);
rightarm.place((x+(armsize+armwidth)/2.),y);
leftleg.place(x-(armsize/2.-armwidth),y+armsize);
rightleg.place(x+(armsize/2.-armwidth),y+armsize);
```

Showing Your Athlete

Finally, you can now show all the objects:

```
head.show();
trunk.show();
leftarm.show();
rightarm.show();
leftleg.show();
rightleg.show();
```

You are welcome to try these statements to draw an athlete. However, you may want to be patient and learn how to use the `Stage` class in the next section.

Using the *Stage* Class

The `Stage` class handles a group of Screen Objects as if they are all bonded together.

Objects of class `Stage` are also of class `ScreenObj`. Therefore, `Stage` has coordinates, and can be shown and erased. However, `Stage` objects do not have a shape. Instead, you can insert several other Screen Objects in the `Stage` object, and when you instruct the `Stage` object with `show()`, all the objects that were inserted will be shown.

In addition to the usual operations you can perform with a Screen Object, you can perform an operation called `insert` in a `Stage` object. By using `insert`, you can insert any Screen Object into the `Stage` object.

> & You can also insert a `Stage` object into another `Stage` object.

As you know by now, an athlete is composed of several objects. If you want to move the athlete on the screen, you will have to perform the following operations on each object that makes up the athlete:

- Erase the object from the old location.
- Move the object to the new location.
- Show the object in the new location.

Stage Saves Time and Trouble

If we use a `Stage` class, we can perform the same operations with all the objects in the `Stage` object by sending the appropriate message only once to the `Stage` object! In other words, instead of showing the head, left arm, right arm, etc., all we have to do is to show the athlete.

```
Stage body;
body.place(x,y);
```

It is important to place the `Stage` object appropriately, because when the `Stage` object is placed somewhere else, all the objects in the `Stage` object will also be moved!

All we have to do now is to insert each object in the `Stage` object (just once):

```
body.insert(trunk);
body.insert(leftarm);
body.insert(rightarm);
body.insert(leftleg);
body.insert(rightleg);
```

If we want to move the whole `Stage` object to a new location *x1,y1*, we can use the following statements:

```
body.erase();
body.place(x,y);
body.show();
```

Of course, this is much simpler than having to erase, place, and show each part individually.

A Short Project—Sun, Earth, Moon

This short project for Skill 15 simulates a simplified planetary system. Suppose that you were hired by your physics instructor to develop a visual animation of Earth revolving around the Sun, while the Moon revolves around Earth.

This software should represent the movement of these bodies on the computer screen. It makes no sense to try to keep the planet sizes and the distances between them proportional, because Earth and the Moon would not be visible on the screen!

Instead, you can choose an arbitrary size (diameter) for each body, as well as an arbitrary distance from one body to another body. We will also represent these movements as circular, instead of as elliptical as in real life.

However, it is important to keep the time in proportion. For example, 1 second in the simulation could correspond to 1 day in real life. This may still be too slow for our purposes—a complete simulation involving a 365-day year would take 365 seconds to complete. Your fellow students may be too impatient to sit and watch for that long! You may try to simulate each 10th of a second as a day in real life.

> & While you work through this project, remember that Earth takes 365.25 days to complete a revolution around the Sun, and that the Moon takes 28 days to complete a revolution around Earth.

Once you have declared and initialized the Sun, Earth, and the Moon as objects of class `Circle`, the simulation itself is relatively easy to carry out.

The position of each body is merely a function of time. Each body can be erased from its position, placed in the new position, and then redrawn. Polar coordinates come in really handy in this case. Since the angular speed is known for Earth and the Moon, the angle can be determined as a function of time.

Useful Objects for Your Planetary Project

It is easy to imagine that we can use objects of class `Circle` to represent the Sun, Earth, and the Moon. You can declare these objects as follows:

```
Circle Sun, Earth, Moon;
```

These objects must be initialized with an appropriate size, color, and initial position.

Figure 15.7 displays a convenient initial position. In this case, all the bodies are horizontally aligned when the simulation starts. As stated before, the distances between them are arbitrary, and it is a good idea to use a named constant to experiment with these distances and sizes.

You may try the following statements:

```
const float earthradius=160.,moonradius=40.;
const float earthsize= 20., moonsize=8., sunsize = 40.;
```

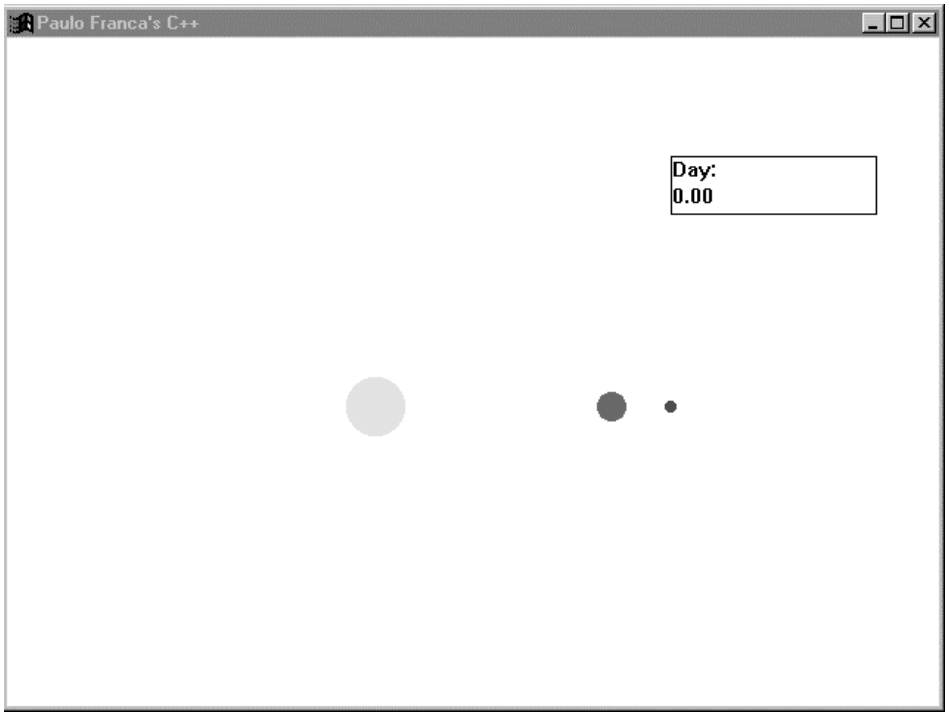


Fig 15.7 Initial positions for Sun, Earth and Moon

Don't Forget the *Stage* Class!

It may not be obvious at first, but you may also consider using an object of class `Stage`. During the simulation, you will have to erase and redraw Earth and the Moon. If you include all the objects that are to be erased and redrawn in a `Stage` object, you can save yourself some lines of code. Here is what I suggest:

```
Stage universe;
universe.insert(Sun);
universe.insert(Earth);
universe.insert(Moon);
```

You will also need a `Clock` object to keep track of time. In fact, it would be convenient to use two clocks. One clock can be used to keep track of the simulated time and can be continuously displayed, so the users know the day of the year. Another clock can be used just to pace the frames at 30th of a second intervals.

```
Clock universal, clock stopwatch;
```

Initialization

To get started, we have to declare, size, place, and color our planets. In other words, these Screen Objects have to be initialized.

Scaling

Since all the distances were arbitrarily chosen, there is no need for a specific scale. It may be convenient to use a negative vertical scale, so you can work with the convention that is used in geometry. It may also be a good idea to set the point of origin near the center of the screen, and to locate the Sun in that position.

Any object can be used to set up the scale and the point of origin. For example:

```
universe.scale(1.,-1.);
universe.origin(250.,250.);
```

Positions

The initial cartesian coordinates could be set as follows:

```
Sun:  x=0;                          y=0;
Earth x=earthradius;                y=0;
Moon  x=earthradius+moonradius;     y=0;
```

The complete initialization is implemented below.

```
const float earthradius=160.,moonradius=40.;
const float earthsize= 20., moonsize=8., sunsize = 40.;
Stage universe;
universe.origin(250.,250.);
universe.scale(1.,-1.);
// Set Sun's data:
Circle sun;                          // Declare the Sun's shape
float xsun=0,ysun=0;                 // Initial coordinates
sun.resize(sunsize);
sun.color(6,6);
sun.place(xsun,ysun);
universe.insert(sun);

// Set Earth's data:
Circle earth;                        // Declare Earth's shape
float xearth=earthradius,yearth=0.; // Initial coordinates
float wearth=(2*pi)/365.25;          // Angular speed in radians per day
earth.resize(earthsize);
earth.color(5,5);
earth.place(xsun+earthradius,ysun);
universe.insert(earth);

// Set Moon's data:
Circle moon;
float xmoon=moonradius,ymoon=0;
float wmoon=(2*pi)/27.;              // Angular speed
moon.resize(moonsize);
moon.color(1,1);
moon.place(xsun+earthradius+moonradius,ysun);
universe.insert(moon);
```

The Simulation Loop

As usual, the simulation loop consists of producing frames at regular intervals. Earth and the Moon have to be moved to their new locations before they are redrawn.

Here is the general procedure in the simulation loop:

- Erase the universe.
- Compute Earth's new location.
- Place Earth in new location.
- Compute the Moon's new location.
- Place the Moon in new location.
- Show the universe.
- Display the time.
- Wait for the time to show the next frame.

Where Is Earth?

Can you figure out where Earth is at any given time *t*? Remember that Earth takes 365.25 days to complete its revolution around the Sun.

You can compute the angular speed either in degrees:

```
w = 365.25 / 360  degrees per day;
```

or in radians:

```
w = 365.25 / (2*pi)  radians per day;
```

The angular speed `wearth` can be used to compute the angular position of Earth on any given day, represented by `time`:

```
thetaearth= wearth * time;
```

Now, since the Sun is at the point of origin, you can use `earthradius` (distance from the sun) and this angle as polar coordinates to compute the cartesian coordinates.

If you simply issue a call to the `polarxy` function:

```
polarxy(earthradius,wearth*time,xsun,ysun,xearth,yearth);
```

you will return the cartesian coordinates in `xearth` and `yearth`.

Where Is the Moon?

In a similar way, you can determine the cartesian coordinates of the Moon. Remember that all that is needed is to find out the angle. When you call the `polarxy` function, consider your point of origin to be Earth:

```
polarxy(moonradius,wmoon*time,xearth,yearth,xmoon,ymoon);
```

The piece of program below implements the simulation loop.

```
for (;time<365.25;)
 {
   universe.erase();
   polarxy(earthradius,wearth*time,
           xsun,ysun,xearth,yearth);
   earth.place(xearth,yearth);
   polarxy(moonradius,wmoon*time,
           xearth,yearth,xmoon,ymoon);
   moon.place(xmoon,ymoon);
   day.say(time);
   universe.show();
   stopwatch.watch(.033);
   stopwatch.reset();
   time=sidereal.time()*timescale;
 }
```

For this simulation loop, there is a variable `timescale` that can make the simulation run faster or slower. `timescale` is defined as 10, which means that each day in the simulation lasts a 10th of a second. This loop only checks whether the simulation time is greater than 365.25 (the end of the year).

The complete implementation can be found in the program `c5stars.cpp`.

Try These for Fun…

- Modify the `c5stars.cpp` program so the Moon's orbit is shown. To do this, simply avoid erasing the Moon in the loop.
- Modify the `c5stars.cpp` program so the background is shown in blue instead of in white

Are You Experienced?

Now you can…

Use a moving object to draw a function graph

Move an object with appropriate timing to simulate a real-life movement

Use *Stage* objects to manipulate several Screen Objects at one time

Develop a simulation of Earth and the Moon revolving around the Sun