C++ is fun – Part 10

at Turbine/Warner Bros.!

Russell Hanson

Syllabus

1) First program and introduction to data types and control structures with applications for games learning how to use the programming environment Mar 25-27
2) Objects, encapsulation, abstract data types, data protection and scope April 1-3
3) Basic data structures and how to use them, opening files and performing operations on files – April 8-10
4) Algorithms on data structures, algorithms for specific tasks, simple AI and planning type algorithms, game AI algorithms April 15-17
Project 1 Due – April 17
5) More AI: search, heuristics, optimization, decision trees, supervised/unsupervised learning – April 22-24
6) Game API and/or event-oriented programming, model view controller, map reduce filter – April 29, May 1
7) Basic threads models and some simple databases SQLite May 6-8
8) Graphics programming, shaders, textures, 3D models and rotations May 13-15
Project 2 Due May 15
9) How to download an API and learn how to use functions in that API, Windows Foundation Classes May 20-22
10) Designing and implementing a simple game in C++ May 27-29
11) Selected topics – Gesture recognition & depth controllers like the Microsoft Kinect, Network Programming & TCP/IP, OSC June 3-5
12) Working on student projects - June 10-12
Final project presentations Project 3/Final Project Due June 12

Follow up on App Game Kit

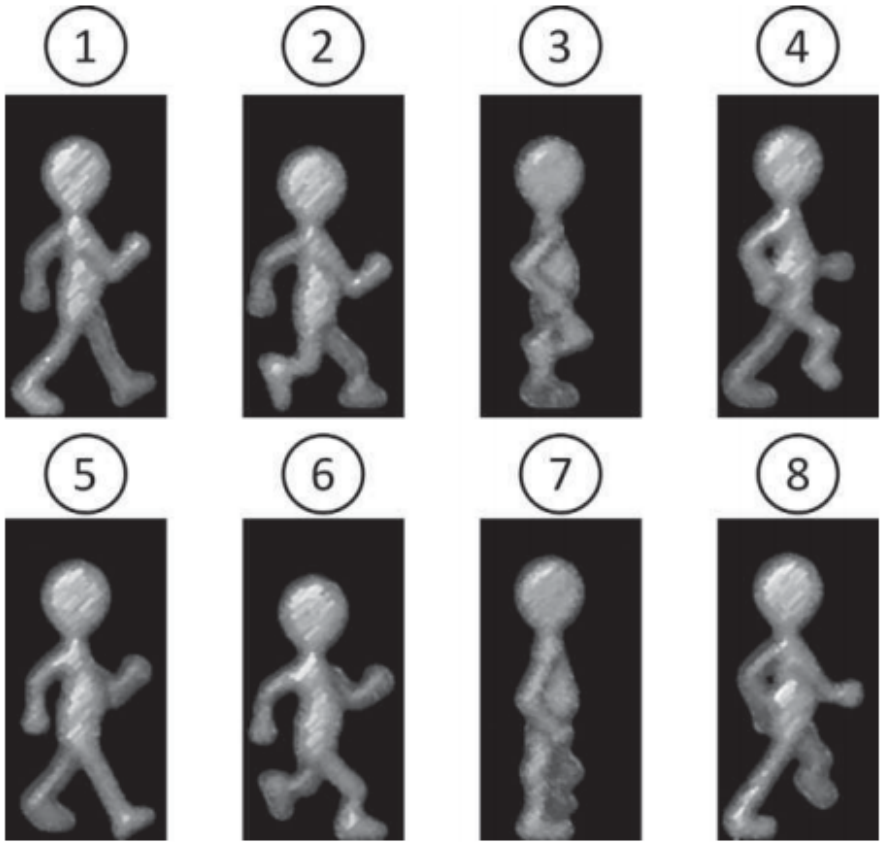
- Animations
- Buttons
- Mouse
- Play MP3's!
- Game Physics and kinematic motion

Sample Game

- narwhal tries to escape evil eskimo that wants its tusk
- so it swims so hard that it flies into space!
- that's level 2
- space level
- so in space it is trying to dodge space trash
- like old soviet satellites
- and impales them with its HORN

Animations

Cel animation images



The eight images shown in Figure 8-9 are named `WalkingMan1.png`, `WalkingMan2.png`, and so forth. Each image is considered a frame in the animation sequence. Program 8-8 loads these images and then displays them one after the other. After the last image is displayed, the program starts over with the first image.

All materials for the Animations exercise are in the Google Drive in a folder called WalkingMan. 8 .png's and 1 .cpp file.

```
// This program demonstrates cel animation.

// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;

// Constants

const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
const int FIRST_IMAGE = 1;
const int LAST_IMAGE = 8;
const int SPRITE_INDEX = 1;
const float FPS = 8;
const int MODE = 0;

// Begin app, called once at the start
void app::Begin( void )
{
// Set the window title.
agk::SetWindowTitle("Cel Animation");

// Set the virtual resolution.
agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);

// Set the frame rate and mode.
agk::SetSyncRate(FPS, MODE);

// Load the walking man images.
agk::LoadImage(1, "WalkingMan1.png", 1);
agk::LoadImage(2, "WalkingMan2.png", 1);
agk::LoadImage(3, "WalkingMan3.png", 1);
agk::LoadImage(4, "WalkingMan4.png", 1);
agk::LoadImage(5, "WalkingMan5.png", 1);
agk::LoadImage(6, "WalkingMan6.png", 1);
agk::LoadImage(7, "WalkingMan7.png", 1);
agk::LoadImage(8, "WalkingMan8.png", 1);

// Create the sprite using the first frame of animation.
agk::CreateSprite(SPRITE_INDEX, FIRST_IMAGE);

// Calculate the sprite's position.
float spriteWidth = agk::GetSpriteWidth(SPRITE_INDEX);

// Calculate the sprite's position.
float spriteWidth = agk::GetSpriteWidth(SPRITE_INDEX);
float spriteHeight = agk::GetSpriteHeight(SPRITE_INDEX);
float spriteX = SCREEN_WIDTH / 2 - spriteWidth / 2;
float spriteY = SCREEN_HEIGHT / 2 - spriteHeight / 2;

// Set the sprite's position.
agk::SetSpritePosition(SPRITE_INDEX, spriteX, spriteY);
}

// Main loop, called every frame
void app::Loop ( void )
{
// Get the sprite's image number.
int currentImage = agk::GetSpriteImageID(SPRITE_INDEX);

// Update the sprite's image number.
if (currentImage == LAST_IMAGE)
{
currentImage = FIRST_IMAGE;
}
else


{
currentImage++;
}

// Set the sprite's image number.
agk::SetSpriteImage(SPRITE_INDEX, currentImage);

// Refresh the screen.
agk::Sync();
}

// Called when the app ends
void app::End ( void )
{
}
```

The Mouse aka *el raton*

Getting the Mouse Coordinates

You can call the `agk::GetRawMouseX` and `agk::GetRawMouseY` functions to get the current coordinates of the mouse pointer. The following code shows an example. Assume that `x` and `y` are `float` variables.

```
x = agk::GetRawMouseX();
y = agk::GetRawMouseY();
```

After this code executes, `x` will contain the mouse pointer's *X*-coordinate, and `y` will contain the mouse pointer's *Y*-coordinate. Program 8-1 demonstrates how these functions work. The program uses the `app::Loop` function to continually get the mouse pointer's coordinates (lines 32 and 33) and sets the sprite's position to that location (line 37). This is shown in Figure 8-1. As the user moves the mouse pointer, the sprite moves with it.

```
// This program demonstrates pressing
// the left or right mouse buttons.

// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;

// Begin app, called once at the start
void app::Begin( void )
{
 // Set the window title and the virtual resolution.
 agk::SetWindowTitle("Mouse Presses");
 agk::SetVirtualResolution(agk::GetDeviceWidth(), agk::GetDeviceHeight());
}
// Main loop, called every frame
void app::Loop ( void )
{
 // Determine if the left mouse button was pressed.
 if(agk::GetRawMouseLeftPressed())
  {
   // Create a sprite using the "mouse.png" image, and
   // set its position to the current mouse coordinates.
   agk::SetSpritePosition(agk::CreateSprite("mouse.png"),
                                                    agk::GetRawMouseX(),
                                                    agk::GetRawMouseY());
  }

 // Determine if the right mouse button was pressed.
 if(agk::GetRawMouseRightPressed())
  {
   // Create a sprite using the "cat.png" image, and
   // set its position to the current mouse coordinates.
   agk::SetSpritePosition(agk::CreateSprite("cat.png"),
                                                    agk::GetRawMouseX(),
                                                    agk::GetRawMouseY());
  }

 // Refresh the screen.
 agk::Sync();
}
// Called when the app ends
void app::End ( void )
{
}
```

Class Exercise: Mouse.png and cat.png and MousePress.cpp in gDrive folder

Virtual Buttons

8.2 Virtual Buttons

CONCEPT: The AGK allows you to create up to 12 virtual buttons on the screen. Virtual buttons can be used to determine when the user clicks on them with the mouse.

Adding a Virtual Button to the Screen

A virtual button is an image of a button that you can display in your AGK program. The user may click the button with the mouse. You can have up to 12 virtual buttons in an AGK program. To add a virtual button, call the `agk::AddVirtualButton` function. Here is the general format of the function:

```
agk::AddVirtualButton(Index, X, Y, Size);
```

This function accepts four arguments. The first argument, *`Index`*, sets the index number that will be used to identify the virtual button, the value used for the index should be an integer in the range of 1 through 12. The second argument, *`X`*, is a floating-point value that sets the *X*-coordinate where the button will appear on the screen and is based on the center of the virtual button. The third argument, *`Y`*, is a floating-point value that sets the *Y*-coordinate where the button will appear on the screen and is based on the center of the virtual button. The fourth argument, *`Size`*, is a floating point value that sets the diameter of the button. Figure 8-5 illustrates how the size of the virtual button is determined based on diameter.

5 The size of a virtual button is based on the diameter of a circle

Changing a Virtual Button's Position on the Screen

If at any time you need to change the position of a virtual button, you can do so by calling the `agk::SetVirtualButtonPosition` function and passing the index number of the virtual button and the new *X* and *Y* values as arguments. For example, the following statement positions a virtual button with an index value of 1 centered at the screen coordinates (120, 120):

```
agk::SetVirtualButtonPosition(1, 120, 120);
```

```
// This program demonstrates virtual buttons.
// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;
// Constants

const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
const int SPRITE_INDEX = 1;
const int SHOW_BUTTON_INDEX = 1;
const int HIDE_BUTTON_INDEX = 2;
const float BUTTON_SIZE = 100.0;

// Begin app, called once at the start
void app::Begin( void )
{
 // Set the window title.
 agk::SetWindowTitle("Virtual Buttons");

 // Set the virtual resolution.
 agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);

 // Create the sprite.
 agk::CreateSprite(SPRITE_INDEX, "frog.png");

 // Calculate the position of the sprite.
 float spriteWidth = agk::GetSpriteWidth(SPRITE_INDEX);
 float spriteX = SCREEN_WIDTH / 2 - spriteWidth / 2;
 float spriteY = 0.0;

 // Set the position of the sprite.
 agk::SetSpritePosition(SPRITE_INDEX, spriteX, spriteY);

 // Calculate the position of the virtual "show" button.
 float showButtonX = SCREEN_WIDTH / 2 - BUTTON_SIZE;
 float showButtonY = SCREEN_HEIGHT - BUTTON_SIZE;

 // Calculate the position of the virtual "hide" button.
 float hideButtonX = SCREEN_WIDTH / 2 + BUTTON_SIZE;
 float hideButtonY = SCREEN_HEIGHT - BUTTON_SIZE;

 // Add the virtual buttons.
 agk::AddVirtualButton(SHOW_BUTTON_INDEX, showButtonX,
                                                  showButtonY, BUTTON_SIZE);
 // Add the virtual buttons.
 agk::AddVirtualButton(SHOW_BUTTON_INDEX, showButtonX,
                                                  showButtonY, BUTTON_SIZE);
 agk::AddVirtualButton(HIDE_BUTTON_INDEX, hideButtonX,
                                                  hideButtonY, BUTTON_SIZE);

 // Set the text of the virtual buttons.
 agk::SetVirtualButtonText(SHOW_BUTTON_INDEX, "Show");
 agk::SetVirtualButtonText(HIDE_BUTTON_INDEX, "Hide");
}

// Main loop, called every frame
void app::Loop ( void )
{
 // Determine if the virtual "show" button was pressed.
 if(agk::GetVirtualButtonPressed(SHOW_BUTTON_INDEX))
  {
   // Show the sprite.
   agk::SetSpriteVisible(SPRITE_INDEX, 1);
  }

 // Determine if the virtual "hide" button was pressed.
 if(agk::GetVirtualButtonPressed(HIDE_BUTTON_INDEX))
  {
   // Hide the sprite.
   agk::SetSpriteVisible(SPRITE_INDEX, 0);
  }

 // Refresh the screen.
 agk::Sync();
}

// Called when the app ends
void app::End ( void )
{
}
```

Music Files

Music files are files that are saved in the MP3 format. The AGK provides many of the same operations for music files as sound files. Let's take a closer look at some of the things you can do with music files.

Loading a Music File

Before you can use a music file, you have to load it into memory by calling the `agk::LoadMusic` function. Here is the general format of how you call the function:

```
agk::LoadMusic(MusicNumber, Filename);
```

`MusicNumber` is an integer number that you are assigning to the music. This can be an integer in the range of 1 through 50. You will use the music number to identify the music when you want to play it or perform other operations with it. *`Filename`* is the name of the music file that you would like to load into the program's memory. For example, the following statement loads the file *myMusic.mp3* as music number 1.

```
agk::LoadMusic(1, "myMusic.mp3");
```

Playing Music

Once you have loaded a music file into memory, you can play it with the `agk::PlayMusic` function. Only one music file may be played at any one time. Here is the general format of how you call the function:

```
agk::PlayMusic(MusicNumber);
```

`MusicNumber` is the number of the music that you want to play. For example, the following statement plays music number 1:

```
agk::PlayMusic(1);
```

Program 8-12 demonstrates playing three different music files when the user clicks on three different virtual buttons.

```
// This program demonstrates playing music
// when a virtual button is clicked by the mouse.
// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;
// Begin app, called once at the start
void app::Begin( void )
{
// Set the window title.
agk::SetWindowTitle("Music Player");
// Set the virtual resolution.
agk::SetVirtualResolution(640, 480);
// Load the music files.
agk::LoadMusic(1, "MusicA.mp3");
agk::LoadMusic(2, "MusicB.mp3");
agk::LoadMusic(3, "MusicC.mp3");
// Add the virtual buttons.
agk::AddVirtualButton(1, 220, 240, 100);
agk::AddVirtualButton(2, 320, 240, 100);
agk::AddVirtualButton(3, 420, 240, 100);
// Set the text for the virtual buttons.
agk::SetVirtualButtonText(1, "1");
agk::SetVirtualButtonText(2, "2");
agk::SetVirtualButtonText(3, "3");
}
// Main loop, called every frame
void app::Loop ( void )
{
// If button 1 was pressed, play music 1.
if(agk::GetVirtualButtonPressed(1))
{ agk::PlayMusic(1);
}
// If button 2 was pressed, play music 2.
if(agk::GetVirtualButtonPressed(2))
{ agk::PlayMusic(2); }
// If button 3 was pressed, play music 3.
if(agk::GetVirtualButtonPressed(3))
{ agk::PlayMusic(3);  }
// Refresh the screen.
agk::Sync(); }
// Called when the app ends
void app::End ( void )
{
}
```

Simulating Falling Objects

CONCEPT: **When an object in the real world falls to Earth, its speed increases as it falls. If you want to write a program that realistically simulates falling objects, you will need to incorporate this acceleration into your program.**

Game programmers often need to simulate moving objects, and in many cases the objects must move realistically. For example, suppose you are writing a program that shows an object falling toward the ground. You could design a loop that merely moves the object down the screen's Y-axis the same amount each time the loop iterates. The resulting animation would not be realistic, however, because in the world, objects do not fall at a steady speed.

moving steadily at 12 meters per second. However, a falling object does not move at a constant speed. A falling object speeds up as it falls. As a result, a falling object travels downward an increasingly greater distance each second that it falls. We can use the following formula to calculate the distance that a falling object falls:

$$d = \frac{1}{2}gt^2$$

In this formula, d is the distance, g is 9.8, and t is the number of seconds that the object has been falling. Going back to the brick example, we can use the formula to calculate the following distances:

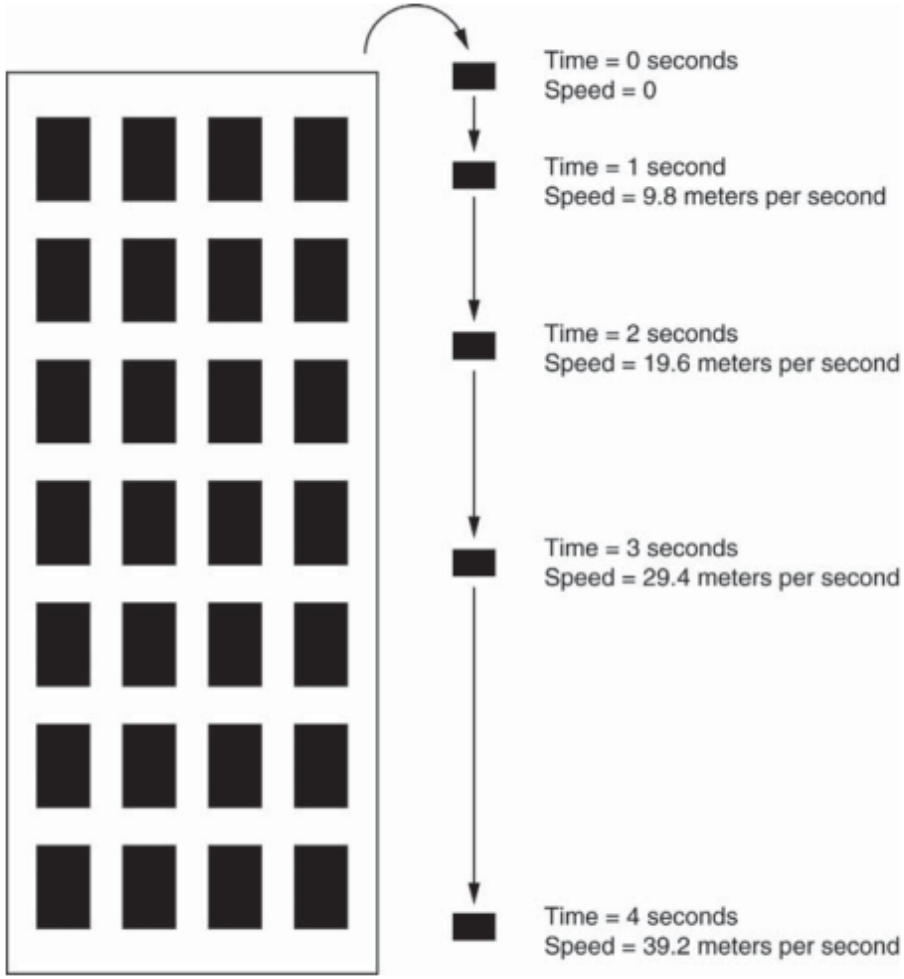
- At one second the brick is falling at a speed of 4.9 meters.
- At two seconds the brick is falling at a speed of 19.6 meters.
- At three seconds the brick is falling at a speed of 44.1 meters.
- At four seconds the brick is falling at a speed of 78.4 meters.
- And so forth.

air will slow the brick down a small amount, we can calculate how fast the brick will be traveling each second as it plummets to the ground:

- At one second the brick is falling at a speed of 9.8 meters per second.
- At two seconds the brick is falling at a speed of 19.6 meters per second.
- At three seconds the brick is falling at a speed of 29.4 meters per second.
- At four seconds the brick is falling at a speed of 39.2 meters per second.
- And so forth.

This is illustrated in Figure 9-10.

Speed of a falling brick at various time intervals (not drawn to scale)



```
// This program simulates a falling ball.
// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;
// Global Constants
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
const int BALL_IMAGE = 1;
const int BALL_SPRITE = 1;
const float ACCELERATION = 0.98;
// Global variables
float g_time = 0;
float g_distance = 0;
// Begin app, called once at the start
void app::Begin( void ) {
// Set the virtual resolution.
agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);
// Set the window title.
agk::SetWindowTitle("Free Fall");
// Load the ball image.
agk::LoadImage(BALL_IMAGE, "ball.png");
// Create the ball sprite.
agk::CreateSprite(BALL_SPRITE, BALL_IMAGE);
// Set the starting position of the ball sprite.
agk::SetSpriteX(BALL_SPRITE, SCREEN_WIDTH / 2 -
agk::GetSpriteWidth(BALL_SPRITE) / 2);
}
// Main loop, called every frame
void app::Loop ( void ) {
// Get the Y-coordinate of the ball sprite.
float y = agk::GetSpriteY(BALL_SPRITE);
// If the ball is above the bottom of the screen,
// then update its position.
if (y < SCREEN_HEIGHT - agk::GetSpriteHeight(BALL_SPRITE))
{
// Calculate the object's distance using the
// distance formula.
g_distance = 0.5 * ACCELERATION * g_time * g_time;
// Set the Y-coordinate of the ball sprite to
// the distance.
y = g_distance;
// Increment time.
g_time++;
}
// Else, set the Y-coordinate of the ball sprite at
// the bottom of the screen.
else
{
y = SCREEN_HEIGHT - agk::GetSpriteHeight(BALL_SPRITE);
}

// Update the Y-coordinate of the ball sprite.
agk::SetSpriteY(BALL_SPRITE, y);

// Refresh the screen.
agk::Sync();
}

// Called when the app ends
void app::End ( void )
{
}
```

FreeFall.cpp and ball.png in the gDrive

13.5 Abstract Classes and Pure `virtual` Functions

When we think of a class as a type, we assume that programs will create objects of that type. However, there are cases in which it's useful to define *classes from which you never intend to instantiate any objects*. Such classes are called **abstract classes**. Because these classes normally are used as base classes in inheritance hierarchies, we refer to them as **abstract base classes**. These classes cannot be used to instantiate objects, because, as we'll soon see, abstract classes are *incomplete*—derived classes must define the "missing pieces" before objects of these classes can be instantiated. We build programs with abstract classes in Section 13.6.

An abstract class provides a base class from which other classes can inherit. Classes that can be used to instantiate objects are called **concrete classes**. Such classes define or inherit implementations for every member function they declare. We could have an *abstract* base class `TwoDimensionalShape` and derive such *concrete* classes as `Square`, `Circle` and `Triangle`. We could also have an *abstract* base class `ThreeDimensionalShape` and derive such *concrete* classes as `Cube`, `Sphere` and `Cylinder`. Abstract base classes are *too generic* to define real objects; we need to be *more specific* before we can think of instantiating objects. For example, if someone tells you to "draw the two-dimensional shape," what shape would you draw? Concrete classes provide the *specifics* that make it reasonable to instantiate objects.

An inheritance hierarchy does not need to contain any abstract classes, but many object-oriented systems have class hierarchies headed by abstract base classes. In some cases, abstract classes constitute the top few levels of the hierarchy. A good example of this is the shape hierarchy in Fig. 12.3, which begins with abstract base class `Shape`. On the

Pure Virtual Functions

A class is made abstract by declaring one or more of its `virtual` functions to be "pure." A **pure `virtual` function** is specified by placing "`= 0`" in its declaration, as in

```
virtual void draw() const = 0; // pure virtual function
```

The "`= 0`" is a **pure specifier**. Pure `virtual` functions do *not* provide implementations. Every concrete derived class *must override all* base-class pure `virtual` functions with concrete implementations of those functions. The difference between a `virtual` function and a pure `virtual` function is that a `virtual` function has an implementation and gives the derived class the *option* of overriding the function; by contrast, a pure `virtual` function does *not* provide an implementation and *requires* the derived class to override the function for that derived class to be concrete; otherwise the derived class remains *abstract*.

Pure `virtual` functions are used when it does *not* make sense for the base class to have an implementation of a function, but you want all concrete derived classes to implement the function. Returning to our earlier example of space objects, it does not make sense for the base class `SpaceObject` to have an implementation for function `draw` (as there is no way to draw a generic space object without having more information about what type of space object is being drawn). An example of a function that would be defined as `virtual` (and not pure `virtual`) would be one that returns a name for the object. We can name a generic `SpaceObject` (for instance, as `"space object"`), so a default implementation for this function can be provided, and the function does not need to be pure `virtual`. The function is still declared `virtual`, however, because it's expected that derived classes will override this function to provide *more specific* names for the derived-class objects.

```
52  // print Employee's information (virtual, but not pure virtual)
53  void Employee::print() const
54  {
55     cout << getFirstName() << ' ' << getLastName()
56        << "\nsocial security number: " << getSocialSecurityNumber();
57  } // end function print
```

Fig. 13.10 | `Employee` class implementation file. (Part 2 of 2.)

The `virtual` function `print` (Fig. 13.10, lines 53–57) provides an *implementation* that will be *overridden* in each of the derived classes. Each of these functions will, however, use the abstract class's version of `print` to print information *common to all classes* in the `Employee` hierarchy.

`SalariedEmployee` *Class Member-Function Definitions*

Figure 13.12 contains the member-function implementations for `SalariedEmployee`. The class's constructor passes the first name, last name and social security number to the `Employee` constructor (line 10) to initialize the `private` data members that are inherited from the base class, but not directly accessible in the derived class. Function `earnings` (lines 32–35) overrides pure `virtual` function `earnings` in `Employee` to provide a *concrete* implementation that returns the `SalariedEmployee`'s weekly salary. If we did not implement `earnings`, class `SalariedEmployee` would be an *abstract* class, and any attempt to instantiate an object of the class would result in a compilation error (and, of course, we want `SalariedEmployee` here to be a concrete class). In class `SalariedEmployee`'s header, we declared member functions `earnings` and `print` as `virtual` (lines 18–19 of Fig. 13.11)—actually, placing the `virtual` keyword before these member functions is *redundant*. We defined them as `virtual` in base class `Employee`, so they remain `virtual` functions throughout the class hierarchy. Explicitly declaring such functions `virtual` at every level of the hierarchy can promote program clarity. Not declaring `earnings` as pure `virtual` signals our intent to provide an implementation in this concrete class.

```
7  #include "Employee.h"
8  #include "SalariedEmployee.h"
9  #include "CommissionEmployee.h"
10 #include "BasePlusCommissionEmployee.h"
11 using namespace std;
12
13 void virtualViaPointer( const Employee * const ); // prototype
14 void virtualViaReference( const Employee & ); // prototype
15
16 int main()
17 {
18    // set floating-point output formatting
19    cout << fixed << setprecision( 2 );
20
21    // create derived-class objects
22    SalariedEmployee salariedEmployee(
23       "John", "Smith", "111-11-1111", 800 );
24    CommissionEmployee commissionEmployee(
25       "Sue", "Jones", "333-33-3333", 10000, .06 );
26    BasePlusCommissionEmployee basePlusCommissionEmployee(
27       "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
28
29    cout << "Employees processed individually using static binding:\n\n";
30
31    // output each Employee's information and earnings using static binding
32    salariedEmployee.print();
33    cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
34    commissionEmployee.print();
35    cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
36    basePlusCommissionEmployee.print();
37    cout << "\nearned $" << basePlusCommissionEmployee.earnings()
38       << "\n\n";
39
40    // create vector of three base-class pointers
41    vector < Employee * > employees( 3 );
42
43    // initialize vector with Employees
44    employees[ 0 ] = &salariedEmployee;
45    employees[ 1 ] = &commissionEmployee;
46    employees[ 2 ] = &basePlusCommissionEmployee;
47
48    cout << "Employees processed polymorphically via dynamic binding:\n\n";
49
50    // call virtualViaPointer to print each Employee's information
51    // and earnings using dynamic binding
52    cout << "Virtual function calls made off base-class pointers:\n\n";
53
54    for ( size_t i = 0; i < employees.size(); ++i )
55       virtualViaPointer( employees[ i ] );
56
57    // call virtualViaReference to print each Employee's information
58    // and earnings using dynamic binding
59    cout << "Virtual function calls made off base-class references:\n\n";
```

```
60
61    for ( size_t i = 0; i < employees.size(); ++i )
62       virtualViaReference( *employees[ i ] ); // note dereferencing
63 } // end main
64
65 // call Employee virtual functions print and earnings off a
66 // base-class pointer using dynamic binding
67 void virtualViaPointer( const Employee * const baseClassPtr )
68 {
69    baseClassPtr->print();
70    cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
71 } // end function virtualViaPointer
72
73 // call Employee virtual functions print and earnings off a
74 // base-class reference using dynamic binding
75 void virtualViaReference( const Employee &baseClassRef )
76 {
77    baseClassRef.print();
78    cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
79 } // end function virtualViaReference
```

```
Employees processed individually using static binding:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00

Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00
```