CS3520 Programming in C++
Graphics

Ken Baclawski

Fall 2016

Outline

- SDL2
  - Installation
  - Main concepts
  - Error handling
  - Bit flags
  - Graphics Memory
  - Colors
  - Default parameters
- The ping package
  - Display class
  - Main program
- Switch statements
- Unions

SDL2

Simple DirectMedia Layer

- Open source cross-platform development library
  - Especially for graphics
  - Also provides access to audio, keyboard, mouse and joystick
- Popular for games and video playback
- Current version is 2.0.5
- Download from `https://www.libsdl.org/`
- On Ubuntu linux install `libsdl2-dev` with apt-get

Compilation

- On the command-line, add `-lSDL2` to your compilation command
- In an IDE, add the SDL2 library to your project properties
- To include the SDL2 header file use

```
#include <SDL.h>
```

or

```
#include <SDL2/SDL.h>
```

depending on where the header file is located

SDL2 Interface

- Not object-oriented
  - Functions are all global (no encapsulation)
  - Structures do not hide any fields (no data hiding)
  - Structures have no behavior (no encapsulation)
  - Use pointers (not safe)
- No namespaces
  - Uses `SDL_` prefix instead
- Object-oriented wrappers for SDL2
  - None are standard
  - Not well-designed

Main SDL Concepts

- `SDL_Window`
- `SDL_Surface`
- `SDL_Renderer`
- `SDL_Texture`
- `SDL_Rect`
- Events

- Constructors

  `SDL_Create...`

- Destructors

  `SDL_Destroy...`

- The first parameter of a function may be regarded as the object

Errors

- C has no support for exceptions
- C libraries use various ways to deal with errors
  - Return an error code
  - Function for obtaining the explanation of the error
- SDL2 uses both of these
  - Obtain the error message with `SDL_GetError()`
- Determining whether a function has failed
  - If the function returns a pointer, a null pointer means that the function call failed
  - If the function returns an int, a normal return is 0 and failure is a nonzero code

Initialization

- `SDL_Init` must be invoked before any other function
  - Specify what services to provide using *bit flags*
  - To provide every service use this

    `SDL_Init(SDL_INIT_EVERYTHING)`
- `SDL_Quit` must be invoked last

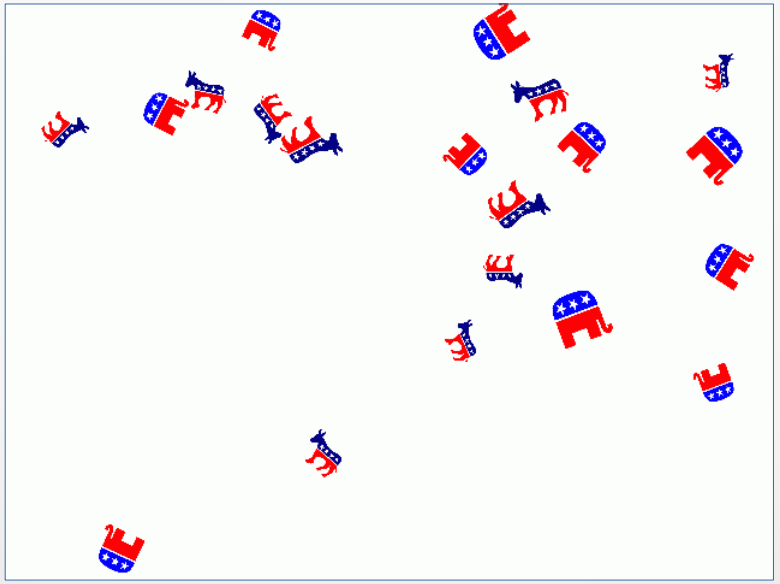
Bit Flags

- C programs often use bit flags
- Efficient technique for specifying many boolean options in a single parameter
- The bit flags are combined with the bitwise or operator

```
SDL_INIT_TIMER           0b00000000000000000001
SDL_INIT_AUDIO           0b00000000000000010000
SDL_INIT_VIDEO           0b00000000000000100000
SDL_INIT_JOYSTICK        0b00000000000001000000
SDL_INIT_HAPTIC          0b00000000100000000000
SDL_INIT_GAMECONTROLLER  0b00000001000000000000
SDL_INIT_EVENTS          0b00000010000000000000
SDL_INIT_EVERYTHING      0b00000011100100011001
SDL_INIT_NOPARACHUTE     0b10000000000000000000
```

SDL_Window

- The graphics window on a monitor
  - Undocumented
- Constructor: `SDL_CreateWindow`
- Destructor: `SDL_DestroyWindow`



SDL_Renderer

- Renders graphic objects in the window
- Constructor: `SDL_CreateRenderer`
- Destructor: `SDL_DestroyRenderer`
- Use of the renderer
  1. Clear the window to a background color
  2. Draw graphic objects
  3. `SDL_RenderPresent` updates the window
  4. Repeat if one is animating

Graphics Memory

- Two-dimensional graphics is specified by a 2D array of pixel values
- SDL has two ways of specifying a 2D array of pixel values
  - `SDL_Surface` is stored in main memory
    - Can be accessed like any other memory
    - Slower than graphics device memory
  - `SDL_Texture` is stored in graphics device memory
    - Can only be accessed using SDL functions
    - Faster than main memory for graphics operations

Graphics Memory

- SDL has many functions for constructing surfaces and textures
- One can convert from surface to texture
- Deallocation is important
  - A surface allocates memory in the free store
  - A texture allocates both free store memory and graphics device memory
  - If not deallocated resources will no longer be unavailable
- Destroy them with `SDL_FreeSurface` and `SDL_DestroyTexture`

Colors

- There are 40+ formats
- One of the most common is RGBA8888
  - RGBA is a *color space*
  - The 8's specify how many bits each component uses
- RGBA stands for red, green, blue, alpha
  - Each component has a value from 0 to 1, represented using unsigned integers with a specified number of bits
  - The alpha component is the *transparency*
- For example, in RGBA8888, opaque white has every component equal to 1, so every component has the value 255.

Default Parameters

- A function parameter can have a default value
  - The arguments may be omitted in a function call
- Only trailing parameters may have defaults
  - This is the only way to guarantee no ambiguity
- For example, here is a constructor declaration:

```
Display(int width = 640, int height = 480);
```

- A display object can be constructed in three ways:

```
Display display;  // Use both defaults
Display display(700);  // width is 700 and height is 480
Display display(700, 500);  // width is 700 and height is 500
```

The ping package

The ping classes

- `Display` encapsulates all access to SDL2
- `Sprite` encapsulates an image with a particular size, position in the window, angle, linear velocity and rotational speed
- The `Sprites` class encapsulates the collection of sprites in the window
- `ReleventEvent` is an enumeration class for the relevant events

Display Interface

```
#ifndef PING_DISPLAY_H
#define PING_DISPLAY_H
#include <vector>
#include "RelevantEvent.h"
class SDL_Window;
class SDL_Renderer;
class SDL_Texture;

namespace ping {

/**
 * SDL Display.  The purpose of this class is to
 * encapsulate all uses of SDL2 by this program.
 * Note that SDL.h was not included so that other
 * classes will not have direct access to SDL2.
 * All uses of SDL2 must be made through this
 * class.
 * @author Ken Baclawski
 */
```

These inform the compiler that these are all classes, but do not specify what they are. In this file, one can only have references or pointers to objects of these classes.

Display Interface

```
class Display {
public:
  /**
   * Create a graphical display with specified size.
   * @throw domain_error if the display could not
   * be created and initialized.
   */
  Display(/** Display width. */ int width = 640,
          /** Display height. */ int height = 480);

  /**
   * Destruct the graphical display.  This closes
   * the display.
   */
  ~Display();
```

These are default values for parameters which can be omitted.

This is the syntax for a destructor. Destructors must be defined only when the behavior is nontrivial as it is for this class.

Display Interface

```
/**
 * Close the graphical display and release the
 * resources.
 */
void close() noexcept;
/**
 * Add an image to the collection.
 */
void addImage(/** The location of the file. */
              const std::string& fileLocation) noexcept;
/**
 * Get the number of images.
 */
unsigned int getImageCount() const noexcept;
```

The `close` method is invoked by the destructor and also whenever an exception is thrown. It is designed to be *idempotent*: Invoking it more than once is the same as invoking it once.

Display Interface

```
/**
 * Check for relevant events as specified in the
 * RelevantEvent enumeration.  If quit is
 * requested, the display is closed and deleted.
 * @return The relevant event that occurred or
 * None if no relevant event occurred.  If the
 * Quit event occurred, then the display is
 * closed and deleted.
 */
RelevantEvent checkForRelevantEvent() noexcept;

/**
 * Refresh the display.
 * @throw domain_error if the display could not
 * be refreshed.
 */
void refresh(/** The sprites to display. */
             const class Sprites& sprites);
```

Events include mouse clicks, mouse motion, keyboard presses and releases, joystick motions, etc.

The display is animated by periodically modifying it. This is called a *refresh*.

Display Interface

```
private:
  /** The display window. */
  SDL_Window* window_ = nullptr;

  /** The display rendering tool. */
  SDL_Renderer* renderer_ = nullptr;

  /** The collection of images. */
  std::vector<SDL_Texture*> images_;

  /** The width of the window. */
  const int width_ = 0;

  /** The height of the window. */
  const int height_ = 0;0

  /** Clear the window to white. */
  void clearBackground();
};
}
#endif
```

The name for a null pointer is `nullptr`. Do not use `NULL` or `0` because they are deprecated.

The SDL structures cannot be directly used: Only pointers to them are permitted because the SDL header file was not included.

Display Source

```
#include <SDL2/SDL.h>
#include <stdexcept>
#include <iostream>

#include "Sprite.h"
#include "Sprites.h"
#include "Display.h"

using namespace std;
using namespace ping;

Display::Display(int width, int height)
  : width_(width), height_(height) {

  // Initialize SDL2

  if (SDL_Init(SDL_INIT_EVERYTHING) != 0) {
    throw domain_error(string("SDL Initialization failed due to: ")
                       + SDL_GetError());
  }
```

Include the SDL header file

These parameters have default values, but default parameter values are specified in the header file, not in the source file

Initialize SDL before using any other SDL functions

Display Source

```
// Construct the screen window

window_ = SDL_CreateWindow("Display",
            SDL_WINDOWPOS_UNDEFINED,
             SDL_WINDOWPOS_UNDEFINED,
             width_, height_, SDL_WINDOW_SHOWN);
if (!window_) {
  close();
  throw domain_error(string("Unable to create the window due to: ")
                     + SDL_GetError());
}
```

The window contains everything that is being displayed. One can specify whether the window is centered on the screen or placed by the window manager.

The `window_` variable is tested for being null by treating it as if it was a boolean variable. This is an example of a conversion operation from a pointer to a boolean. One should not compare a pointer with `nullptr`.

Display Source

```
  // Construct the renderer

  renderer_ = SDL_CreateRenderer(window_, -1,
                SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
  if (!renderer_) {
    close();
    throw domain_error(string("Unable to create the renderer due to: ")
                       + SDL_GetError());
  }

  // Clear the background
  clearBackground();
}
```

The *renderer* draws graphic images in the window. The second parameter is for specifying a rendering driver when one needs very low level control of rendering.

The third parameter is another example of bit flags:

`SDL_RENDERER_ACCELERATED` requests hardware acceleration

`SDL_RENDERER_PRESENTVSYNC` means that the calls to the refresh method will be synchronized with the monitor's refresh rate.

Display Source

```
Display::~Display() {
  close();
}
void Display::close() {
  for (SDL_Texture* image : images_) {
    if (image) {
      SDL_DestroyTexture(image);
    }
  }
  images_.clear();
  if (renderer_) {
    SDL_DestroyRenderer(renderer_);
    renderer_ = nullptr;
  }
  if (window_) {
    SDL_DestroyWindow(window_);
    window_ = nullptr;
  }
  SDL_Quit();
}
```

Destructor definition.

The window, renderer and texture should be destroyed in reverse order to their creation followed by calling SDL_Quit.

Clear the images collection to ensure idempotence.

Testing for not null and setting the pointers to null ensure that the `close` method is idempotent.

Display Source

```
void Display::addImage(const string& fileLocation) noexcept{
  if (renderer_) {
    SDL_Surface* imageSurface = SDL_LoadBMP(fileLocation.c_str());
    if (imageSurface) {
      SDL_Texture* imageTexture =
        SDL_CreateTextureFromSurface(renderer_, imageSurface);
      if (imageTexture) {
        images_.push_back(imageTexture);
      } else {
        cerr << "Unable to load the image file at " << fileLocation
             << " due to: " << SDL_GetError() << endl;
      }
      SDL_FreeSurface(imageSurface);
    } else {
      cerr << "Unable to load the image file at " << fileLocation
           << " due to: " << SDL_GetError() << endl;
    }
  }
}
```

See next slide for more about this code

Display Source

SDL_LoadBMP creates a surface, but a texture is needed for rendering using the graphic device memory

`SDL_LoadBMP` requires a C-style string

Check that pointers are not null before using them.

The image surface is no longer needed so its resources must be freed to prevent a memory leak

```
void Display::addImage(const string& fileLocation) noexcept{
  if (renderer_) {
    SDL_Surface* imageSurface = SDL_LoadBMP(fileLocation.c_str());
    if (imageSurface) {
      SDL_Texture* imageTexture =
        SDL_CreateTextureFromSurface(renderer_, imageSurface);
      if (imageTexture) {
        images_.push_back(imageTexture);
      } else {
        cerr << "Unable to load the image file at " << fileLocation
             << " due to: " << SDL_GetError() << endl;
      }
      SDL_FreeSurface(imageSurface);
    } else {
      cerr << "Unable to load the image file at " << fileLocation
           << " due to: " << SDL_GetError() << endl;
    }
  }
}
```

Display Source

```
RelevantEvent Display::checkForRelevantEvent() {
  RelevantEvent result = RelevantEvent::NONE;
  SDL_Event event;
  while (SDL_PollEvent(&event) != 0) {
    if (event.type == SDL_QUIT) {
      close();
      return RelevantEvent::QUIT;
    }
  }
  return result;
}
```

The parameter of `SDL_PollEvent` is a pointer not a reference so one must take the address of the argument object.

SDL places events in a queue in the order they occur. The `SDL_PollEvent` function removes the events from the queue in LIFO order. In this program, the only relevant event is the `SDL_QUIT` event which occurs when the window is Xed out.

Display Source

Always start a refresh by clearing the background

An `SDL_Rect` is used for specifying where a sprite is to be drawn in the window

```
void Display::refresh(const Sprites& sprites) {
  if (renderer_) {
    clearBackground();
    for (const Sprite& sprite : sprites.getList()) {
      SDL_Rect destination = { sprite.getXCoordinate(), sprite.getYCoordinate(),
                               sprite.getDiameter(), sprite.getDiameter() };
      unsigned int imageIndex = sprite.getImageIndex();
      if (imageIndex >= 0 && imageIndex < images_.size()) {
        SDL_Texture* imageTexture = images_.at(imageIndex);
        if (imageTexture) {
          if (SDL_RenderCopyEx(renderer_, imageTexture, nullptr,
                               &destination, sprite.getAngle(),
                               nullptr, SDL_FLIP_NONE) != 0) {
            ...
```

Check that the index is valid

SDL_RenderCopyEx resizes an image to the size of the destination rectangle, rotating and flipping it if requested. The third parameter is the part of the image that is to be copied. The sixth parameter is the center of rotation if not the center of the image.

Display Source

```
...
            close();
            throw domain_error(string("Unable to render a sprite due to: ")
                               + SDL_GetError());
          }
        } else {
          close()
          throw domain_error("Missing image texture at index "
                             + to_string(imageIndex));
        }
      } else {
        close();
        throw domain_error("Invalid image index "
                           + to_string(imageIndex));
      }
    }
    SDL_RenderPresent(renderer_);
  }
}
```

The rest of the refresh method. All rendering up to the last call is actually done to an internal buffer. The buffer is displayed on the monitor when `SDL_RenderPresent` is called

Display Source

```
void Display::clearBackground() {
  if (renderer_) {

    // Clear the window to opaque white

    if (SDL_SetRenderDrawColor(renderer_, 0xff, 0xff, 0xff, 0xff) != 0) {
      close();
      throw domain_error(string("Unable to set the background color due to: ")
                         + SDL_GetError());
    }
    if (SDL_RenderClear(renderer_) != 0) {
      close();
      throw domain_error(string("Unable to set the background color due to: ")
                         + SDL_GetError());
    }
  }
}
```

This was made a private method because it is used more than once in methods.
It also makes it easy to change the background color if desired.

The Sprite and Sprites Classes

- This contains some complex code for the physics of elastic collisions
- However, none of the code involves any graphics
  - The SDL2 functionality was encapsulated in the Display class
- You are encouraged to examine the code if your project will involve elastic collisions.

Main Program

```
int main() {
  try {

    // Initialize the graphical display

    Display display;

    // Add some images to the display

    display.addImage("graphics/image1.bmp");
    display.addImage("graphics/image2.bmp");

    // Construct the sprite collection

    Sprites sprites(display.getImageCount());
...
```

The display uses the default parameter values

One could add as many images as desired. If a file does not exist, a warning is printed but the program continues.

Main Program

```
...
    // Run until quit.

    for (;;) {

      // Check for relevant events.

      switch (display.checkForRelevantEvent()) {
      case RelevantEvent::NONE:
        break;
      case RelevantEvent::QUIT:
        return 0;
      default:
        cerr << "Unexpected event" << endl;
        return 1;
      }
...
```

Example of a switch statement using an enumeration class. Every case must end with break or return or an explanation for why there is no break or return. There should always be a default case.

Main Program

```
...
      // Move the sprites and draw the new ones.

      sprites.evolve();
      display.refresh(sprites);
    }
  } catch (const exception& e) {
    cerr << e.what() << endl;
    return 1;
  }
}
```

The animation consists of evolving the sprites and refreshing the screen. The display will be destructed and all resources deallocated when the main function returns.

Switch Statements

Enumerations

- An `enum class` is a finite collection of named constants
  - The collection is the *enumeration*
  - The constants are the *enumerators*
- Traditionally, enumerators were integer codes and so are not type safe
- Enumerators of an `enum class` are type safe.
- Coding style requirement: Enumerators are in uppercase
- Enumerations are most commonly used in switch statements.

Switch Statement

- Useful alternative to a series of `if`, `else if` conditions.
- Especially well suited to enumerations.
- Every case of a switch *must* end with one of these:
  - A `break` statement, or
  - A `return` statement, or
  - A detailed explanation why there is no `break` statement
- Every `switch` statement should end with the `default` case.

Union Structures

Union structure

- A union is a structure that looks syntactically like a struct or class but only having data members.
- Unlike a struct or class, a union only represents *one* of its data members.
- This union will have 8 bytes
  - x.integer will interpret the 8 bytes as a 64-bit integer
  - x.real will interpret the 8 bytes as a double
- This violates type safety
- Conclusion: Never use a union.

```
union number {
  long long integer;
  double real;
};

number x;
x.integer = 5;
cout << x.real << endl;
```

The result is:

```
2.47033e-323
```

Next Class

- Lambda Expressions
- Graphs
- Assignment #7