PowerBall Lottery Simulator Revamped

Introduction

I read the very well written article by “roachmaster” the other day and thought that it was a good beginner’s project to learn C++. But then I got thinking. Maybe it would make a good guide for an intermediate level project as well. It implements the same rules as “roachmaster’s” simulator, but it differs in design and somewhat in implementation. It uses OO concepts and the STL more extensively.

C++

C++ is a very versatile language, once you start to take advantage of everything it has to offer. The *STL* is great for starters. One of the goals Stroustrup (the inventor of C++) had was to make the *std::vector* class at least as quick as C-style arrays. He has also expressed that the C-style arrays are ridiculous, they don’t even know how many elements they contain. Another important aspect of C++ is the notion of objects. An object is a user defined type that has relationship with other objects. This means that if you got a function that calculates areas in square centimeters you can create a centimeter object that you pass to the function instead of a naked *double* value. Now you are safe from anyone trying to pass an inch object to that particular function. It will generate an error at compile time and you can’t ship software if it isn’t compiled. This is also known as type safety.

This project

This project is aimed at programmers who got the basic C++ syntax and grammar down and want to expand on how to think when designing object oriented programs. The problem we are trying to solve is an easy one which means we can focus more on the code and the concepts behind it.

There is much to be added to this code to make it complete, but the classes do compile and are ready to be put into a project.

So the problem at hand is to draw random lottery numbers, create tickets to a player with random numbers on them and to check and see if the player has won or not.

The rules

1. The lottery draws 6 balls, 5 white and 1 red ball, randomly.
2. You buy any amount of lottery tickets with 6 random numbers on each.
3. If the drawn numbers matches the numbers in your tickets, you win.
4. Bonus is awarded if you match the red ball.

Balls

When I first began designing the new version I thought that the different colored balls would make a perfect chance to show off some object inheritance. So I created a base class *Ball* which implements all code that is congruent for any ball. That is, every ball has a number and every ball has a color. Balls can also be compared with other balls and since all balls has a number they can also be compared with any integer.

```
class Ball
{
public:
        Ball(int i) { number = i; }

        int   getNumber() { return number; }
        bool  isRed()      { return Red; }

        const bool operator == (int i) { return number == i; }
        const bool operator == (Ball &rhs) { return number == rhs.number; }
        const bool operator <  (Ball &rhs) { return number < rhs.number;  }
        const bool operator >  (Ball &rhs) { return number > rhs.number;  }
protected:
        int number;
        bool Red;
};
```

Now we need to differentiate between different kinds of balls. In this lottery there are white balls and red balls. The only real difference is the color. We set a flag to tell if the ball is red or if it is not red. Note also, that we use the base class' ctor to set the number of the ball.

```
class WhiteBall : public Ball
{
public:
        WhiteBall(int i) : Ball(i)
        {
                Red = false;
        }
};

class RedBall : public Ball
{
public:
        RedBall(int i) : Ball(i)
        {
                Red = true;
        }
};
```

Random Numbers

A lottery isn't a lottery without its random numbers. The *RandomNumber* class creates random numbers. The ctor takes 3 integer arguments, *minimum*, *maximum* and *amount*. The *minimum* is the lowest value to generate, the *maximum* is the highest number to generate and *amount* is the amount of numbers to create. We use the new C++11 library <random> to create our random numbers. First we create a *random_device* object. This object uses /dev/urandom on GNU/Linux to generate a random seed to the random generator. I don't know how windows implements this object, but it still works. We use the default random engine (**mersenne_twister_engine** at the time of writing) to generate our numbers.

We see a new C++11 construct here as well. The new for-loop. It makes it easier to traverse arrays and other objects with iterators. It wouldn't do to have duplicate balls. Would we count that as two matches or one match if we got the number on our ticket? If we detect duplicates, regenerate the number and stick the new number into the vector.

```
class RandomNumbers
{
public:
        RandomNumbers(int min, int max, int amount)
        {
                std::random_device rd;
                std::default_random_engine re(rd());
                std::uniform_int_distribution<int> uid(min, max);

                for (int i = 0; i < amount; i++)
                {
                        int num = uid(re);

                        for (int n : numbers)
                        {
                                if (n == num)
                                        num = uid(re);
                        }

                        numbers.push_back(num);
                }
        }

        std::vector<int> getNumbers() { return numbers; }
private:
        std::vector<int> numbers;
};
```

Tickets

Next I decided to create the ticket class. It is very simple. It only holds 6 random numbers between 1 and 58 for us. It also contains logic for displaying our ticket. The most interesting thing about this class is that we are using an object we our self's have created, the *RandomNumbers* object. We also implement the *std::sort* function that can sort arrays, vectors and other containers that contain basic types (e.g. *int*, *char*, *double*, *float*)

```
class Ticket
{
public:
	Ticket()
	{
		RandomNumbers ticketNums(1, 58, 6);
		numbers = ticketNums.getNumbers();
		std::sort(numbers.begin(), numbers.end());
	}

	void display()
	{
		std::cout << "Ticket:  ";
		for (int i : numbers)
		{
			std::cout << std::setw(2) << i << " ";
		}
		std::cout << std::endl;
	}

	std::vector<int> getNumbers() { return numbers; }
private:
	std::vector<int> numbers;
};
```

The Lottery

Now we have everything except the lottery itself. The interesting thing here is the use of the vector *balls*. It contains pointers to the base class *Ball* but we fill it with 5 *WhiteBalls* and a *RedBall*. We can do this since both are of the type *Ball*. Since we use the keyword *new* to create the balls we also need to use *delete* to free the memory allocated by the balls. This is a typical example of how to use the ctor and dtor of a class. The ctor allocates the memory needed for the object and the dtor releases the memory when the object is out of scope.

We get to acquaint our self's further with C++11 as we see in the line that sorts our balls. *Std::sort* does not know how to sort balls yet, but it allows us to submit a custom function to show how we want to sort them. As seen in the ball's class, a ball knows how to compare itself with another ball. But it feels cumbersome to write a function just to test if a ball has a lower value than another. C++11 has the answer to this. Lambdas. A lambda-expression is a short piece of code that behaves like a function, but can be fitted nicely into a line of code and used as an argument to functions that takes a function as one of its arguments. It is also useful for functions that we aren't likely to reuse but still need for some reason.

```
class Lottery
{
public:
	Lottery()
	{
		RandomNumbers RandomWhite(1, 58, 5);
		RandomNumbers RandomRed(1, 34, 1);

		for (int n : RandomWhite.getNumbers())
		{
			balls.push_back(new WhiteBall(n));
		}

		std::sort(balls.begin(), balls.end(), [](Ball* a, Ball* b){ return *a
< *b; });

		balls.push_back(new RedBall(RandomRed.getNumbers()[0]));
	}

	~Lottery()
	{
		for (auto ball : balls)
		{
			delete ball;
		}
	}

	void display()
	{
		std::cout << "Lottery: ";

		for (auto ball : balls)
		{
			if (ball->isRed())
				std::cout << "Red number: " << ball->getNumber() <<
std::endl;
			else
				std::cout << std::setw(2) << ball->getNumber() << " ";
		}
	}
```

```
        std::vector<Ball*> getBalls() { return balls; }
private:
        std::vector<Ball*> balls;
};
```

Winning conditions

Ok, so we got our balls, our tickets and our lottery. Now we need to see if the player has won anything. The logic is pretty straight forward even if the code may look a bit cluttered. If you use a good IDE it will help you highlight which curly braces are around what. I use Microsoft Visual Studio Express 2013, but there are many other very competent IDEs out there so make sure to pick one that suits your needs.

```
class Winning
{
public:
	Winning(std::vector<Ticket*> tickets, std::vector<Ball*> balls)
	{
		for (auto ticket : tickets)
		{
			int matches = 0;
			bool hasRed = false;

			for (int number : ticket->getNumbers())
			{
				for (auto ball : balls)
				{
					if (*ball == number)
					{
						matches++;

						if (ball->isRed())
							hasRed = true;
					}
				}
			}

			winnsPerTicket.push_back(matches);
			hasRedTicket.push_back(hasRed);
		}
	}

	int getWinnings()
	{
		for (size_t i = 0; i < winnsPerTicket.size(); i++)
		{
			std::cout << "Got " << winnsPerTicket[i] << " matches.";

			if (hasRedTicket[i])
				std::cout << " And has got the red ball!" <<
std::endl;
			else
				std::cout << " But has not got the red ball." <<
std::endl;
		}

		return 0;
	}

private:
	std::vector<int> winnsPerTicket;
	std::vector<bool> hasRedTicket;
};
```

The Game

We're almost done now. The only thing that remains is to create a menu for buying tickets and implement the logic needed to play. Again we need to be sure to *delete* the objects we've created with *new* in the dtor.

```
class Game
{
public:
	Game() {};
	~Game()
	{
		for (auto ticket : tickets)
		{
			delete ticket;
		}
	}

	void Menu()
	{
		int numTic = 0;
		std::cout << "Welcome to the PowerBall Lottery!" << std::endl;
		std::cout << "To play you need to purchase a ticket at $2. More
tickets increase the odds to win." << std::endl;
		std::cout << "How many tickets would you like? " << std::endl;

		do
		{
			std::cout << "Enter amount of tickets you would like to
purchase: ";
			std::cin >> numTic;
			std::cin.sync();

			if ((numTic < 1) || (numTic > 100))
			{
				std::cout << "Input invalid. Needs to be a number
between 1 and 100. Please try again" << std::endl;
			}
		} while ((numTic < 1) || (numTic > 100));

		createTickets(numTic);
		std::cout << "Your tickets are registered. Thank you for playing the
PowerBall lottery!" << std::endl;
	}

	void Play()
	{
		std::cout << "Let\'s see this weeks PowerBall lottery numbers!" <<
std::endl;
		lotto.display();

		for (auto ticket : tickets)
		{
			ticket->display();
		}

		Winning w(tickets, lotto.getBalls());
		w.getWinnings();
	}

private:
	std::vector<Ticket*> tickets;
```

```
        Lottery lotto;

        void createTickets(int numTic)
        {
                for (int i = 0; i < numTic; i++)
                {
                        tickets.push_back(new Ticket);
                }
        }
};
```

Epilogue

That's it! Almost. I've left some things out for you to implement, but the code compiles and the lottery is playable with the code in this tutorial. You might want to add some function to keep track of the wins and losses and maybe give the player a wallet object to keep money in. One could also think of ways to save the state of the game to keep the stats between sessions. But as I've already said, that's for you to implement.

Oh, before I forget. You'll need to include some headers to make this code work.

```
#include <iostream>  // For std::cout, std::cin
#include <iomanip>        // For std::setw
#include <random>         // For all random generation stuff
#include <algorithm> // For std::sort
#include <vector>         // For std::vector
```


"roachmaster"'s article http://www.cplusplus.com/articles/4yhv0pDG/

Author Tomas Landberg (tomas.landberg@gmail.com)

Good Luck!