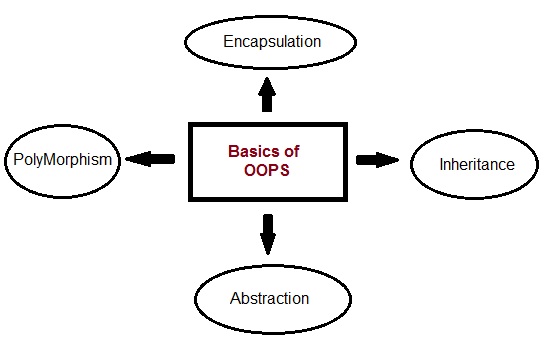
2

Object Oriented Approach and Design in Games

In this chapter, we will cover the object oriented way to program games. Let’s look at the recipes that will be covered in this chapter:

* Using classes for data encapsulation and abstraction
* Using polymorphism to reuse code
* Using operator overloading to re-use operators
* Using function overloading to re-use functions
* Using files for input and output
* Creating your first simple text based game
* Templates: When to use them?

# Introduction



Insert Image B04929\_02\_01.png

The above diagram shows the main concepts of OOP. Let us consider that we need to make a car racing game. So a car can be composed of an engine, wheels, and chassis and so on. All these parts can be considered as individual components which can be used for other cars as well. Similarly every car’s engine can be different and so we can add differently functionalities, states and properties to each of each individual components.

All these can be achieved by object oriented programming.

# Using classes for data encapsulation and abstraction

In this recipe we will see how easy it is to create a game framework using object oriented programming in C++.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp, CEnemy.h and CEnemy.cpp
5. Add the following lines of code.

**Souce.cpp**

#include "CEnemy.h"

#include <iostream>

#include <string>

#include <conio.h>

#include "vld.h"

using namespace std;

int main()

{

CEnemy\* pEnemy = new CEnemy();

int iAge;

int iHealth;

string sName;

string sArmour;

iAge = pEnemy->GetAge();

iHealth = pEnemy->TotalHealth();

sArmour = pEnemy->GetArmourName();

sName = pEnemy->GetName();

cout << "Name of the enemy is :" << sName << endl;

cout << "Name of " << sName << "'s armour is :" << sArmour << endl;

cout << "Health of " << sName << " is :" << iHealth << endl;

cout << sName << "'s age is :" << iAge;

delete pEnemy;

\_getch();

}

**CEnemy.h**

#ifndef \_CENEMY\_H

#define \_CENEMY\_H

#include <string>

using namespace std;

class CEnemy

{

public:

string GetName()const;

int GetAge()const;

string GetArmourName()const;

int TotalHealth()const;

//ctors

CEnemy();

//dtors

~CEnemy();

private:

int m\_iAge;

int m\_iHealth;

string m\_sName;

string m\_sArmour;

};

#endif

**CEnemy.cpp**

#include <iostream>

#include <string>

#include "CEnemy.h"

using namespace std;

CEnemy::CEnemy()

{

m\_iAge = 50;

m\_iHealth = 100;

m\_sArmour = "GOLD\_ARMOUR";

m\_sName = "Dr. EVIL";

}

int CEnemy::GetAge()const

{

return m\_iAge;

}

int CEnemy::TotalHealth()const

{

return m\_iHealth;

}

string CEnemy::GetArmourName()const

{

return m\_sArmour;

}

string CEnemy::GetName()const

{

return m\_sName;

}

## How it works...

For creating an object oriented program, we need to create classes and objects. Although we can write the definition and declaration of the class in the same file, it is advisable to have two separate files for definition and declaration. A declaration class file is called a header file whereas a definition class file is called a source file.

In the CEnemy header file, we define the member variables and the functions that we need. In a class, we have the option to separate out the variables as public, protected and private. A public state indicates that they are accessible from outside the class, a protected state indicates that only the child class that inherits from the current base class, has access to it whereas a private state indicates that they are only accessible within the class. By default everything in a C++ class is private. So we have created all the member functions as public so that we can access them from the driver class which in this example is Source.cpp . The member variables in the header file are all private as they should not be directly accessible from outside the class. This is what we called as abstraction. We define a string type variable for name and armour and an integer type for health and age. It is also advisable to create a constructor and destructor even if we do not have any functionality for them at present.

In the CEnemy source file, we have the initialisation of the member variables and also the declarations of the functions. We have used the const keyword at the end of each function because we do not want the function to change the contents of the member variables. We just want them to return the values that are already assigned. As a rule of the thumb, we should always use the const keyword when necessary. It makes the code more secure, organised and readable. We have initialised the variables in the constructor; we could have also created parameterised constructors and assigned values to them from the driver program. Alternatively we can also have Set functions to assign values.

From the driver program, we create a pointer object of the type CEnemy. When the object is initialised, it calls its appropriate constructors and the values are assigned them. Then we call the functions by dereferencing the pointer using the “->” operator. So when we call the function p->function, it is same as (\*p).function. As we are dynamically allocating memory, we should also delete the object or else we will get a memory leak. We have used vld to check for memory leaks. This program does not have any as we have used the delete keyword. Just comment out the line “delete pEnemy;” and you will notice that the program has few memory leaks on exiting.

# Using polymorphism to reuse code

In this recipe we will see how we can use the same function and override them with different functionalities based on the need. Also we will see how we can share values across base and derived classes.

## Getting ready

You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp and 3 header files called Enemy.h, Dragon.h and Soldier.h
5. Add the following lines of code.

**Enemy.h**

#ifndef \_ENEMY\_H

#define \_ENEMY\_H

#include <iostream>

using namespace std;

class CEnemy {

protected:

int m\_ihealth,m\_iarmourValue;

public:

CEnemy(int ihealth, int iarmourValue) : m\_ihealth(ihealth), m\_iarmourValue(iarmourValue) {}

virtual int TotalHP(void) = 0;

void PrintHealth()

{

cout << "Total health is " << this->TotalHP() << '\n';

}

};

#endif

**Dragon.h**

#ifndef \_DRAGON\_H

#define \_DRAGON\_H

#include "Enemy.h"

#include <iostream>

using namespace std;

class CDragon : public CEnemy {

public:

CDragon(int m\_ihealth, int m\_iarmourValue) : CEnemy(m\_ihealth, m\_iarmourValue)

{

}

int TotalHP()

{

cout << "Dragon's ";

return m\_ihealth\*2+3\*m\_iarmourValue;

}

};

endif

**Soldier.h**

#ifndef \_SOLDIER\_H

#define \_SOLDIER\_H

#include "Enemy.h"

#include <iostream>

using namespace std;

class CSoldier : public CEnemy {

public:

CSoldier(int m\_ihealth, int m\_iarmourValue) : CEnemy(m\_ihealth, m\_iarmourValue) {}

int TotalHP()

{

cout << "Soldier's ";

return m\_ihealth+m\_iarmourValue;

}

};

#endif

**Source.cpp**

// dynamic allocation and polymorphism

#include <iostream>

#include <conio.h>

#include "vld.h"

#include "Enemy.h"

#include "Dragon.h"

#include "Soldiers.h"

int main()

{

CEnemy\* penemy1 = new CDragon(100, 50);

CEnemy\* penemy2 = new CSoldier(100, 100);

penemy1->PrintHealth();

penemy2->PrintHealth();

delete penemy1;

delete penemy2;

\_getch();

return 0;

}

## How it works...

Polymorphism is the ability to have different forms. So in this example, we have an Enemy interface which does not have any functionality for calculating total health. However we know that all types of enemy should have a function to calculate total health. So we have made the function in the base class as pure virtual function by assigning it to 0.

This enables or rather forces all the child classes to have their own implementation of calculating total health. So the CSoldier class and CDragon class have their own implementation of TotalHP. The advantage of such a structure is that we can create a pointer object of the child from the base and when it resolves, it calls the correct function of the child class.

If we do not create a virtual function, then the functions in the child classes would have hidden the function of the base class. The way the compiler resolves the functions at run time is by a technique called dynamic dispatch. Most languages use dynamic dispatch. C++ uses single cast dynamic dispatch. It does so with the help of virtual tables. When the class CEnemy defines the virtual function TotalHP , the compiler add a hidden member variable to the class which points to an array of pointers to functions called the virtual method table (VMT or Vtable). At runtime these pointers will be set to point to the right function, because at compile time, it is not yet known if the base function is to be called or a derived one implemented by CDragon and CSoldier.

The member variables in the base class are protected. This means that the derived class also have access to the member variables. From the driver program, because we have allocated memory dynamically, we should also delete or else we will have memory leaks.

# Use operator overloading to reuse operators

## Getting ready

1. In this recipe we will see how we can overload an operator and which operators are allowed to be overloaded in C++

## How to do it...

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called Source.cpp, vector3.h and vector3.cpp
5. Add the following lines of code.

**Source.cpp**

#include "vector3.h"

#include <conio.h>

#include "vld.h"

int main()

{

// Vector tests:

// Create two vectors.

CVector3 a(1.0f, 2.0f, 3.0f);

CVector3 b(1.0f, 2.0f, 3.0f);

CVector3 c;

// Zero Vector.

c.Zero();

// Addition.

CVector3 d = a + b;

// Subtraction.

CVector3 e = a - b;

//Scalar Multiplication.

CVector3 f1 = a \* 10;

//Scalar Multiplication.

CVector3 f2 = 10 \* a;

//Scalar Division.

CVector3 g = a / 10;

// Unary minus.

CVector3 h = -a;

// Relational Operators.

bool bAEqualsB = (a == b);

bool bANotEqualsB = (a != b);

// Combined operations +=.

c = a;

c += a;

// Combined operations -=.

c = a;

c -= a;

// Combined operations /=.

c = a;

c /= 10;

// Combined operations \*=.

c = a;

c \*= 10;

// Normalisation.

c.Normalize();

// Dot Product.

float fADotB = a \* b;

// Magnitude.

float fMag1 = CVector3::Magnitude(a);

float fMag2 = CVector3::Magnitude(c);

// Cross product.

CVector3 crossProduct = CVector3::CrossProduct(a, c);

// Distance.

float distance = CVector3::Distance(a, c);

\_getch();

return (0);

}

**Vector3.h**

#ifndef \_\_VECTOR3\_H\_\_

#define \_\_VECTOR3\_H\_\_

#include <cmath>

class CVector3

{

public:

// Public representation: Not many options here.

float x;

float y;

float z;

CVector3();

CVector3(const CVector3& \_kr);

CVector3(float \_fx, float \_fy, float \_fz);

// Assignment operator.

CVector3& operator =(const CVector3& \_kr);

// Relational operators.

bool operator ==(const CVector3& \_kr) const;

bool operator !=(const CVector3& \_kr) const;

// Vector operations

void Zero();

CVector3 operator -() const;

CVector3 operator +(const CVector3& \_kr) const;

CVector3 operator -(const CVector3& \_kr) const;

// Multiplication and division by scalar.

CVector3 operator \*(float \_f) const;

CVector3 operator /(float \_f) const;

// Combined assignment operators to conform to C notation convention.

CVector3& operator +=(const CVector3& \_kr);

CVector3& operator -=(const CVector3& \_kr);

CVector3& operator \*=(float \_f);

CVector3& operator /=(float \_f);

// Normalize the vector

void Normalize();

// Vector dot product.

// We overload the standard multiplication symbol to do this.

float operator \*(const CVector3& \_kr) const;

// Static member functions.

// Compute the magnitude of a vector.

static inline float Magnitude(const CVector3& \_kr)

{

return (sqrt(\_kr.x \* \_kr.x + \_kr.y \* \_kr.y + \_kr.z \* \_kr.z));

}

// Compute the cross product of two vectors.

static inline CVector3 CrossProduct(const CVector3& \_krA,

const CVector3& \_krB)

{

return

(

CVector3(\_krA.y \* \_krB.z - \_krA.z \* \_krB.y,

\_krA.z \* \_krB.x - \_krA.x \* \_krB.z,

\_krA.x \* \_krB.y - \_krA.y \* \_krB.x)

);

}

// Compute the distance between two points.

static inline float Distance(const CVector3& \_krA, const CVector3& \_krB)

{

float fdx = \_krA.x - \_krB.x;

float fdy = \_krA.y - \_krB.y;

float fdz = \_krA.z - \_krB.z;

return sqrt(fdx \* fdx + fdy \* fdy + fdz \* fdz);

}

};

// Scalar on the left multiplication, for symmetry.

inline CVector3 operator \*(float \_f, const CVector3& \_kr)

{

return (CVector3(\_f \* \_kr.x, \_f \* \_kr.y, \_f \* \_kr.z));

}

#endif // \_\_VECTOR3\_H\_\_

**Vector3.cpp**

#include "vector3.h"

// Default constructor leaves vector in an indeterminate state.

CVector3::CVector3()

{

}

// Copy constructor.

CVector3::CVector3(const CVector3& \_kr)

: x(\_kr.x)

, y(\_kr.y)

, z(\_kr.z)

{

}

// Construct given three values.

CVector3::CVector3(float \_fx, float \_fy, float \_fz)

: x(\_fx)

, y(\_fy)

, z(\_fz)

{

}

// Assignment operator, we adhere to C convention and return reference to the lvalue.

CVector3&

CVector3::operator =(const CVector3& \_kr)

{

x = \_kr.x;

y = \_kr.y;

z = \_kr.z;

return (\*this);

}

// Equality operator.

bool

CVector3::operator ==(const CVector3&\_kr) const

{

return (x == \_kr.x && y == \_kr.y && z == \_kr.z);

}

// Inequality operator.

bool

CVector3::operator !=(const CVector3& \_kr) const

{

return (x != \_kr.x || y != \_kr.y || z != \_kr.z);

}

// Set the vector to zero.

void

CVector3::Zero()

{

x = 0.0f;

y = 0.0f;

z = 0.0f;

}

// Unary minus returns the negative of the vector.

CVector3

CVector3::operator -() const

{

return (CVector3(-x, -y, -z));

}

// Binary +, add vectors.

CVector3

CVector3::operator +(const CVector3& \_kr) const

{

return (CVector3(x + \_kr.x, y + \_kr.y, z + \_kr.z));

}

// Binary –, subtract vectors.

CVector3

CVector3::operator -(const CVector3& \_kr) const

{

return (CVector3(x - \_kr.x, y - \_kr.y, z - \_kr.z));

}

// Multiplication by scalar.

CVector3

CVector3::operator \*(float \_f) const

{

return (CVector3(x \* \_f, y \* \_f, z \* \_f));

}

// Division by scalar.

// Precondition: \_f must not be zero.

CVector3

CVector3::operator /(float \_f) const

{

// Warning: no check for divide by zero here.

float fOneOverA = 1.0f / \_f;

return (CVector3(x \* fOneOverA, y \* fOneOverA, z \* fOneOverA));

}

CVector3&

CVector3::operator +=(const CVector3& \_kr)

{

x += \_kr.x;

y += \_kr.y;

z += \_kr.z;

return (\*this);

}

CVector3&

CVector3::operator -=(const CVector3& \_kr)

{

x -= \_kr.x;

y -= \_kr.y;

z -= \_kr.z;

return (\*this);

}

CVector3&

CVector3::operator \*=(float \_f)

{

x \*= \_f;

y \*= \_f;

z \*= \_f;

return (\*this);

}

CVector3&

CVector3::operator /=(float \_f)

{

float fOneOverA = 1.0f / \_f;

x \*= fOneOverA;

y \*= fOneOverA;

z \*= fOneOverA;

return (\*this);

}

void

CVector3::Normalize()

{

float fMagSq = x \* x + y \* y + z \* z;

if (fMagSq > 0.0f)

{

// Check for divide-by-zero.

float fOneOverMag = 1.0f / sqrt(fMagSq);

x \*= fOneOverMag;

y \*= fOneOverMag;

z \*= fOneOverMag;

}

}

// Vector dot product.

// We overload the standard multiplication symbol to do this.

float

CVector3::operator \*(const CVector3& \_kr) const

{

return (x \* \_kr.x + y \* \_kr.y + z \* \_kr.z);

}

## How it works...

## C++ has built in types: int, char, and float. Each of these types has a number of built-in operators, such as: Addition +, Multiplication \*. C++ allows you to add these operators to your own classes as well. Operators on built-in types (int, float) cannot be overloaded. The precedence order cannot be changed. There are great reasons to proceed with caution when overloading an operator. The goal is to increase usability and understanding. In our example we have overloaded the basic multiplication operators so that we can add, subtract etc our vector3 objects which we create. This is extremely handy as we can find the distance of an object in our game, if we know the position vectors of the two objects. We have used const functions as much as possible. The compiler will enforce the promise to not modify the object. This can be a great way to make sure that your code has no unanticipated side effects.

* 1. All functions that accept vectors accept a constant reference to a vector.We have to remember that passing an argument by value to a function invokes a constructor. Inheritance will not be very useful to the vector class as we know CVector3 is speed critical.The V-table adds 25% to the class size so it is not advisable.
  2. Also data hiding does not make too much sense as we need the values of the vector class. Some operators can be overloaded in C++. The operators which C++ does not allow us to overload are :

. (Member Access or Dot operator),?: (Ternary or Conditional Operator),:: (Scope Resolution Operator),.\* (Pointer-to-member Operator),sizeof (Object size Operator) and typeid (Object type Operator).

# Use function overloading to reuse functions

In this recipe we will learn how to overload a function and use the same function name and overload them.

## Getting ready

For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

1. Open Visual Studio.
2. Create a new C++ project
3. Select a win32 console application
4. Add a source file called main.cpp, Cspeed.h/cpp
5. Add the following lines of code.

**Main.cpp**

#include <iostream>

#include <conio.h>

#include "CSpeed.h"

using namespace std;

//This is not overloading as the function differs only

//in return type

/\*int Add(float x, float y)

{

return x + y;

}\*/

int main()

{

CSpeed speed;

cout<<speed.AddSpeed(2.4f, 7.9f)<<endl;

cout << speed.AddSpeed(4, 5)<<endl;

cout << speed.AddSpeed(4, 9, 12)<<endl;

\_getch();

return 0;

}

**CSpeed.cpp**

#include "CSpeed.h"

CSpeed::CSpeed()

{

}

CSpeed::~CSpeed()

{

}

int CSpeed::AddSpeed(int x, int y, int z)

{

return x + y + z;

}

int CSpeed::AddSpeed(int x, int y)

{

return x + y;

}

float CSpeed::AddSpeed(float x, float y)

{

return x + y;

}

**CSpeed.h**

#ifndef \_VELOCITY\_H

#define \_VELOCITY\_H

class CSpeed

{

public:

int AddSpeed(int x, int y, int z);

int AddSpeed(int x, int y);

float AddSpeed(float x, float y);

CSpeed();

~CSpeed();

private:

};

#endif

## How it works...

Overloading a function is a type of functional polymorphism. A function can be overloaded only by the number of parameters in the argument list and the type of parameter. A function cannot be overloaded only by the return type.

We have created a class to calculate the sum of speeds. We can use the function to add 2 speeds, 3 speeds or speeds of different data types. The compiler will resolve which function to call based on the signature. One might argue that we could create different objects with different speeds and then add them using operator overloading or use templates and write one function. However we have to remember that in simple templates the implementation will remain same, however in function overloading we can change the implementation of each function as well.

# Using files for input and output

In this recipe we will find out how to use file handling operations in C++ to write or read from a text file. We can even use C++ operations to create binary files.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

## Open Visual Studio.

## Create a new C++ project

1. 3. Select a win32 console application

## 4. Add a source file called Source.cpp and File.h/.cpp

## 5. Add the following lines of code.

**Source.cpp**

1. #include <conio.h>
2. #include "File.h"
3. int main() {
4. CFile file;
6. file.WriteNewFile("Example.txt");
7. file.WriteNewFile("Example.txt", "Logging text1");
8. file.AppendFile("Example.txt", "Logging text2");
9. file.ReadFile("Example.txt");
10. \_getch();
11. return 0;
12. }
13. **File.cpp**
14. #include "File.h"
15. #include <string>
16. #include <fstream>
17. #include <iostream>
18. using namespace std;
19. CFile::CFile()
20. {
21. Text = "This is the initial data";
22. }
23. CFile::~CFile()
24. {
25. }
26. void CFile::WriteNewFile(string Filename)const
27. {
28. ofstream myfile(Filename);
29. if (myfile.is\_open())
30. {
31. myfile << Text;
33. myfile.close();
34. }
35. else cout << "Unable to open file";
36. }
37. void CFile::WriteNewFile(string Filename,string Text)const
38. {
39. ofstream myfile(Filename);
40. if (myfile.is\_open())
41. {
42. myfile << Text;
44. myfile.close();
45. }
46. else cout << "Unable to open file";
47. }
48. void CFile::AppendFile(string Filename, string Text)const
49. {
50. ofstream outfile;
51. outfile.open(Filename, ios\_base::app);
52. outfile << Text;
    * + - 1. outfile.close();
53. }
54. void CFile::ReadFile(string Filename)const
55. {
56. string line;
57. ifstream myfile(Filename);
58. if (myfile.is\_open())
59. {
60. while (getline(myfile, line))
61. {
62. cout << line << '\n';
63. }
64. myfile.close();
65. }
66. else cout << "Unable to open file";
67. }
68. **File.h**

#ifndef \_FILE\_H

#define \_FILE\_H

#include <iostream>

#include <string.h>

using namespace std;

class CFile

{

public:

CFile();

~CFile();

void WriteNewFile(string Filename)const;

void WriteNewFile(string Filename, string Text)const;

void AppendFile(string Filename, string Text)const;

void ReadFile(string Filename)const;

private:

string Text;

};

1. #endif

## How it works...

We can use file handling for a variety of reasons. The most important of them might be to log data while the game is running, to load data from a text file to be used in the game or encrypt save data or load data of a game.

We have created a class called CFile. This class helps us to write data to a new file, to append to a file and read from a file. We use the fstream header file to load all the file handling operations.

Everything in a file is written and read in terms of streams. While doing C++ programming, we must write information to a file from your program using the stream insertion operator (<<) just as we use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

We have created a constructor to have an initial data if a file is created without any data in it. If we just create or write to a file, every time a new file will be created with the new data. This is sometimes useful if we just want to write the most updated or latest data. However if we want to add data to the already existing file, we can use the append function. The append function starts writing to an existing file from the last file-position pointer position.

The read function starts reading data from the file until it has reach the last line of written data. We can display the result to the screen or if needed, we could then write the contents to another file. We also must remember to close the file after each operation or it might lead to ambiguity in code. We can also use the seekp and seekg functions to reposition the file-position pointer.

# Creating your first simple game

In this recipe we will learn how to create a simple luck based lottery game.

## Getting ready

To step through this recipe, you will need a machine running Windows. No other prerequisites are required. You need to have a working copy of Visual Studio installed on your Windows machine.

## How to do it...

1. Open Visual Studio.
   1. Create a new C++ project
   2. Select a win32 console application
   3. Add the following files: Source.cpp
   4. Add the following lines of code.

#include <iostream>

#include <cstdlib>

#include <ctime>

int main(void) {

srand(time(NULL)); // To not have the same numbers over and over again.

while (true) { // Main loop.

// Initialize and allocate.

int inumber = rand() % 99 + 2; // System number is stored in here.

int iguess; // User guess is stored in here.

int itries = 0; // Number of tries is stored here.

char canswer; // User answer to question is stored here.

while (true) { // Get user number loop.

// Get number.

std::cout << "Enter a number between 1 and 100 (" << 20 - itries << " tries left): ";

std::cin >> iguess;

std::cin.ignore();

// Check is tries are taken up.

if (itries >= 20) {

break;

}

// Check number.

if (iguess > inumber) {

std::cout << "Too high! Try again.\n";

}

else if (iguess < inumber) {

std::cout << "Too low! Try again.\n";

}

else {

break;

}

// If not number, increment tries.

itries++;

}

// Check for tries.

if (itries >= 20) {

std::cout << "You ran out of tries!\n\n";

}

else {

// Or, user won.

std::cout << "Congratulations!! " << std::endl;

std::cout << "You got the right number in " << itries << " tries!\n";

}

while (true) { // Loop to ask user is he/she would like to play again.

// Get user response.

std::cout << "Would you like to play again (Y/N)? ";

std::cin >> canswer;

std::cin.ignore();

// Check if proper response.

if (canswer == 'n' || canswer == 'N' || canswer == 'y' || canswer == 'Y') {

break;

}

else {

std::cout << "Please enter \'Y\' or \'N\'...\n";

}

}

// Check user's input and run again or exit;

if (canswer == 'n' || canswer == 'N') {

std::cout << "Thank you for playing!";

break;

}

else {

std::cout << "\n\n\n";

}

}

// Safely exit.

std::cout << "\n\nEnter anything to exit. . . ";

std::cin.ignore();

return 0;

}

## How it works...

The game works by creating random number from 1 to 100 and asks the user to guess that number. Hints are provided as to whether a number guessed is higher or lower than the actual number. Also the user is given just 20 tries to guess the number. We first need a pseudo seeder based on which we are going to generate a random number. The pseudo-seeder in this case is srand. We have chosen TIME as a value to generate our random range.

We need to execute the program in an infinite loop so that the program breaks only when tries are over or when the user correctly guess a number. We have set a variable for tries and increment for every guess a user takes. The random number is generated by the rand function. We use rand%99 +2 so that the random number is in the range 1 to 100. We ask the user input for the guessed number and then we check whether is number is less than or greater than or equal to the randomly generated number. Accordingly we display the correct message. If the user has guessed correctly or the number of tries are over, the program should break out of the main loop. At this point we ask the user if he wants to play the game again.

Then depending on his answer he go back into the main loop and start the process of selecting a random number.

# Templates: When to use them

In this recipe we will find out the importance of templates, how to use them and the advantage it provides to us.

## Getting ready

1. For this recipe, you will need a Windows machine with a working copy of Visual Studio.

## How to do it...

## Open Visual Studio.

## Create a new C++ project

## Add a source file called Source.cpp and Stack.h

## Add the following lines of code.

1. **Source.cpp**

#include <iostream>

#include <conio.h>

#include <string>

#include "Stack.h"

using namespace std;

template<class T>

void Print(T array[], int array\_size)

{

for (int nIndex = 0; nIndex < array\_size; ++nIndex)

{

cout << array[nIndex] << "\t";

}

cout << endl;

}

int main()

{

int iArray[5] = { 4, 5, 6, 6, 7 };

char cArray[3] = { 's', 's', 'b' };

string sArray[3] = { "Kratos", "Dr.Evil", "Mario" };

//Printing any type of elements

Print(iArray, sizeof(iArray) / sizeof(\*iArray));

Print(cArray, sizeof(cArray) / sizeof(\*cArray));

Print(sArray, sizeof(sArray) / sizeof(\*sArray));

Stack<int> iStack;

//Pushes an element to the of the stack

iStack.push(7);

cout << iStack.top() << endl;

for (int i = 0; i < 10; i++)

{

iStack.push(i);

}

//Removes an element from the top of the stack

iStack.pop();

//Prints the top of stack

cout << iStack.top() << endl;

\_getch();

}

1. **Stack.h**

#include <vector>

using namespace std;

template <class T>

class Stack {

private:

vector<T> elements; // elements

public:

void push(T const&); // push element

void pop(); // pop element

T top() const; // return top element

bool empty() const{ // return true if empty.

return elements.empty();

}

};

template <class T>

void Stack<T>::push(T const& elem)

{

// append copy of passed element

elements.push\_back(elem);

}

template <class T>

void Stack<T>::pop()

{

if (elements.empty()) {

throw out\_of\_range("Stack<>::pop(): empty stack");

}

// remove last element

elements.pop\_back();

}

template <class T>

T Stack<T>::top() const

{

if (elements.empty()) {

throw out\_of\_range("Stack<>::top(): empty stack");

}

// return copy of last element

return elements.back();

}

## How it works...

## Templates are the foundation of generic programming of C++. If the implementation of a function or a class is same but we need them to operate on different data types, it is advisable to use templates instead of writing a new class or function. One can argue that we can overload a function to achieve the same thing. One must keep in mind that while overloading a function, we can change the implementation based on data type and we are still writing a new function. With templates the implementation has to be same for all the data type. This is the advantage of templates that writing one function is enough. With advanced templates and C++11 features, we can even change the implementation but we will reserve that discussion for later.

We have used function templates and class templates in this example. The function template is defined in Source.cpp itself. On top of the function print we have added the line template<class T>. The keyword class could be replaced by typename as well. The reason for two keywords is a historic one and we do not need to discuss. The remaining part of the function definition is normal except instead of using any particular data type, we have used T. So when we call the function from main, T gets replaced with the correct data type. So by just using one function, we can print all types of data type. We can even create our own data type and pass it to the function.

Stack.h is an example of class template as the data type that the class uses is a generic one. We have selected a stack as it is a very popular data structure in game programming. It’s a LIFO(Last in First Out) structure. So we can display the latest content from the stack as per our requirement. The push function pushes an element onto the stack, whereas a pop removes an element from the stack. The top function displays the top most element of the stack and empty function empties the stack. By using this generic stack class, we can store and display any data type of our choice.

One thing to be kept in mind while using templates is that the compile must know at compile time the correct implementation of the template. So generally template definition and declaration are both done in the header file. However if you want to separate out the two, you can do so by two popular methods. One method is to have another header file and list the implementation at the end of it. Other implementation is to create an .ipp or .tpp file extension and have the implementation in those files.