Contents

Chapter 1

Basics of C++ Programming

1.1 Namespace

```cpp
#include <iostream>

using namespace std;

namespace foo {
int bar = 10;
}

int
main(void)
{
	cout << foo::bar << endl;

	return 0;
}
```

1.2 Manipulators

```cpp
#include <iostream>
#include <iomanip>

using namespace std;

int
main(void)
{
	float foo;
	string bar;

```

```
        foo = 123.456;
        bar = "buzz";

        cout << setprecision(4) << foo << endl;
        cout << setw(10) << bar << endl;

        return 0;
}

/*
Output:

123.5
      buzz
*/
```

1.3 Dynamic memory allocation

```
#include <iostream>

using namespace std;

int
main(void)
{
        int *foo;

        foo = new int;
        *foo = 5;

        cout << *foo << endl;

        delete foo;

        return 0;
}
```

1.4 Functions

1.4.1 Default Arguments

- If value for the parameter is not passed when a function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead.

```
#include <iostream>
```

```

using namespace std;

int
add(int num1, int num2 = 1)
{
	return num1 + num2;
}

int
main(void)
{
	int num;

	num = 10;

	cout << add(num, 5) << endl;		/* NOT using the default parameter */
	cout << add(num) << endl;		/* Using the default parameter */

	return 0;
}
```

1.4.2 Inline Functions

- A function which is expanded inline by the compiler each time its call is appeared instead of jumping to the called function as usual is called inline function.

```
#include <iostream>

using namespace std;

inline void
print_hello(void)
{
	cout << "Hello, World!" << endl;
}

int
main(void)
{
	print_hello();

	return 0;
}
```

1.4.3 Function overloading – Different types of arguments

```
#include <iostream>

using namespace std;

int
add(int num1, int num2)
{
        return num1 + num2;
}

float
add(float num1, float num2)
{
        return num1 + num2;
}

int
main(void)
{
        cout << add(5, 2) << endl;
        cout << add((float)5.5, (float)3.2) << endl;

        return 0;
}
```

1.4.4 Function overloading – Different number of arguments

```
#include <iostream>

using namespace std;

int
add(int num1, int num2)
{
        return num1 + num2;
}

int
add(int num1, int num2, int num3)
{
        return num1 + num2 + num3;
}

int
main(void)
```

```cpp
{
        cout << add(4, 2) << endl;
        cout << add(6, 4, 2) << endl;

        return 0;
}
```

1.4.5 Pass by value

```cpp
#include <iostream>

using namespace std;

void
exchange(int num1, int num2)
{
        int temp_num;

        temp_num = num1;
        num1 = num2;
        num2 = temp_num;
}

int
main(void)
{
        int num1, num2;

        num1 = 5;
        num2 = 7;

        cout << "Before exchange: " << num1 << "-" << num2 << endl;

        exchange(num1, num2);

        cout << "After exchange: " << num1 << "-" << num2 << endl;

        return 0;
}
```

1.4.6 Pass by reference

```cpp
#include <iostream>

using namespace std;

```

```
void
exchange(int &num1, int &num2)
{
        int temp_num;

        temp_num = num1;
        num1 = num2;
        num2 = temp_num;
}

int
main(void)
{
        int num1, num2;

        num1 = 5;
        num2 = 7;

        cout << "Before exchange: " << num1 << "-" << num2 << endl;

        exchange(num1, num2);

        cout << "After exchange: " << num1 << "-" << num2 << endl;

        return 0;
}
```

1.4.7 Pass by pointer

```
#include <iostream>

using namespace std;

void
exchange(int *num1, int *num2)
{
        int temp_num;

        temp_num = *num1;
        *num1 = *num2;
        *num2 = temp_num;
}

int
main(void)
{
        int num1, num2;

        num1 = 5;
        num2 = 7;

        cout << "Before exchange: " << num1 << "-" << num2 << endl;

        exchange(&num1, &num2);

        cout << "After exchange: " << num1 << "-" << num2 << endl;

        return 0;
}
```

1.4.8 Return by value

```
#include <iostream>

using namespace std;

int
largest(int num1, int num2)
{
        return num1 > num2 ? num1 : num2;
}

int
main(void)
{
        int num1, num2, largest_num;

        num1 = 5;
        num2 = 7;

        largest_num = largest(num1, num2);
        cout << largest_num << endl;

        return 0;
}
```

1.4.9 Return by reference

```
#include <iostream>

using namespace std;

int &
```

```cpp
largest(int &num1, int &num2)
{
	return num1 > num2 ? num1 : num2;
}

int
main(void)
{
	int num1, num2;

	num1 = 5;
	num2 = 7;

	cout << num1 << "-" << num2 << endl;

	largest(num1, num2) = 0;

	cout << num1 << "-" << num2 << endl;

	return 0;
}
```

1.4.10 Return by pointer

```cpp
#include <iostream>

using namespace std;

int *
largest(int *num1, int *num2)
{
	return *num1 > *num2 ? num1 : num2;
}

int
main(void)
{
	int num1, num2;

	num1 = 5;
	num2 = 7;

	cout << num1 << "-" << num2 << endl;

	*(largest(&num1, &num2)) = 0;

	cout << num1 << "-" << num2 << endl;
```

```

        return 0;
}
```

Chapter 2

Classes and Objects

2.1 Constructors

2.1.1 Default Constructor

```cpp
#include <iostream>

using namespace std;

class foo {
public:
	foo()
	{
		cout << "Foo object created" << endl;
	}
};

int
main(void)
{
	foo f;

	return 0;
}
```

2.1.2 Parameterized Constructor

```cpp
#include <iostream>

using namespace std;

class foo {
private:
        int bar;
public:
        foo()
        {
                cout << "No parameter was given" << endl;
        }

        foo(int b)
        {
                cout << "Bar was given" << endl;
                bar = b;
        }
};

int
main(void)
{
        foo f1, f2(5);

        return 0;
}
```

2.1.3 Copy constructor

```
#include <iostream>

using namespace std;

class foo {
private:
        int bar;
public:
        foo(int b)
        {
                cout << "A value to bar is given" << endl;
                bar = b;
        }

        foo(foo &f)
        {
                cout << "A foo object instance is given" << endl;
                bar = f.bar;
        }
};

int
main(void)
```

```
{
        foo f1(5), f2(f1);

        return 0;
}
```

2.1.4 Default copy constructor

```
#include <iostream>

using namespace std;

class foo {
private:
        int bar;
public:
        foo(int b)
        {
                cout << "A value to bar is given" << endl;
                bar = b;
        }

        void
        print_bar(void)
        {
                cout << bar << endl;
        }
};

int
main(void)
{
        foo f1(5), f2(f1);

        f1.print_bar();
        f2.print_bar();

        return 0;
}

/*
Output:

A value to bar is given
5
5
*/
```

2.2 Destructor

```cpp
#include <iostream>

using namespace std;

class foo {
public:
	foo()
	{
		cout << "Foo object is created" << endl;
	}

	~foo()
	{
		cout << "Foo object is destroyed" << endl;
	}
};

int
main(void)
{
	foo f;

	return 0;
}
```

2.3 Objects as Function Arguments

2.3.1 Pass by value

```cpp
#include <iostream>

using namespace std;

class distances {
private:
	int feet;
	int inches;

	void
	recalc(void)
	{
		feet = feet + inches/12;
		inches = inches % 12;
```

```
        }
public:
        distances(void)
        {
                feet = 0;
                inches = 0;
        }

        void
        set_distance(int f, int i)
        {
                feet = f;
                inches = i;

                recalc();
        }

        void
        add_distance(distances d)
        {
                feet += d.feet;
                inches += d.inches;

                recalc();
        }

        void
        print_distances(void)
        {
                cout << "Feet: " << feet << ", Inches: " << inches << endl;
        }
};

int
main(void)
{
        distances d1, d2;

        d1.set_distance(4, 10);
        d2.set_distance(2, 4);
        d2.add_distance(d1);

        d1.print_distances();
        d2.print_distances();

        return 0;
}

```

```
/*
Output:

Feet: 4, Inches: 10
Feet: 7, Inches: 2
*/
```

2.3.2 Pass by reference

```cpp
#include <iostream>

using namespace std;

class foo {
private:
	int bar;
public:
	void
	set_bar(int b)
	{
		bar = b;
	}

	/* This function changes the value of `bar` in the passed `foo` object
	 * with the value of `bar` in the current object. */
	void
	change_bar(foo &f)
	{
		f.bar = bar;
	}

	void
	print_bar(void)
	{
		cout << bar << endl;
	}
};

int
main(void)
{
	foo b1, b2;

	b1.set_bar(5);
	b1.change_bar(b2);

	b1.print_bar();
	b2.print_bar();

	return 0;
}

/*
Output:
5
5
*/
```

2.3.3 Pass by pointer

```cpp
#include <iostream>

using namespace std;

class foo {
private:
	int bar;
public:
	void
	set_bar(int b)
	{
		bar = b;
	}

	/* This function changes the value of `bar` in the passed `foo` object
	 * with the value of `bar` in the current object. */
	void
	change_bar(foo *f)
	{
		f->bar = bar;
	}

	void
	print_bar(void)
	{
		cout << bar << endl;
	}
};

int
main(void)
{
	foo b1, b2;

	b1.set_bar(5);
	b1.change_bar(&b2);

	b1.print_bar();
	b2.print_bar();

	return 0;
}

/*
Output:
5
```

```
5
*/
```

2.4 Structure

2.4.1 Showing all the extensions to structure

```
#include <iostream>

using namespace std;

struct foo {
private:
        int bar;
public:
        void
        set_bar(int b)
        {
                bar = b;
        }

        void
        print_bar(void)
        {
                cout << bar << endl;
        }
};

int
main(void)
{
        foo f;

        f.set_bar(5);
        f.print_bar();

        return 0;
}
```

2.5 Static data members in class

```
#include <iostream>
using namespace std;

```

```
class foo {
private:
        static int foo_count;
public:

        foo()
        {
                foo_count++;
        }

        void
        print_count(void)
        {
                cout << foo_count << endl;
        }
};

int foo::foo_count = 0;

int
main(void)
{
        foo f1, f2;

        f1.print_count();

        return 0;
}
```

2.6 Member functions defined outside the class

```cpp
#include <iostream>
using namespace std;

class foo {
private:
        int bar;
public:
        void
        set_bar(int f);

        void
        print_bar(void);
};

void
foo::set_bar(int f)
{
        bar = f;
}

void
foo::print_bar(void)
{
        cout << bar << endl;
}

int
main(void)
{
        foo f;

        f.set_bar(5);
        f.print_bar();

        return 0;
}
```

Chapter 3

Operator Overloading

3.1 Overloading unary operator

Prefix and Postfix

```
#include <iostream>
using namespace std;

class rectangle {
private:
        int length;
        int breadth;

public:
        void
        set_data(int l, int b)
        {
                length  = l;
                breadth = b;
        }

        void
        print_data(void)
        {
                cout << "Length: " << length << ", Breadth: " << breadth
                     << endl;
        }

        /* pre-increment */
        void
        operator ++(void)
        {
                ++length;
                ++breadth;
```

```cpp
        }

        /* post-increment */
        void
        operator ++(int)
        {
                length++;
                breadth++;
        }
};

int
main(void)
{
        rectangle r1, r2;

        r1.set_data(5, 6);
        r2.set_data(7, 3);

        r1++;
        ++r2;

        r1.print_data();
        r2.print_data();

        return 0;
}
```

3.2 Overloading binary operator

Plus

```cpp
#include <iostream>
using namespace std;

class rectangle {
private:
        int length;
        int breadth;

public:
        void
        set_data(int l, int b)
        {
                length  = l;
                breadth = b;
        }

        void
        print_data(void)
        {
                cout << "Length: " << length << ", Breadth: " << breadth
                     << endl;
        }

        /* plus */
        rectangle
        operator +(rectangle r)
        {
                rectangle temp_r;

                temp_r.length  = length + r.length;
                temp_r.breadth = breadth + r.breadth;

                return temp_r;
        }
};

int
main(void)
{
        rectangle r1, r2, r3;

        r1.set_data(5, 6);
        r2.set_data(7, 3);

        r3 = r1 + r2;
        r3.print_data();

        return 0;
}
```

Comparison

```
#include <iostream>
using namespace std;

class length {
private:
        int data;
public:
        void
        set_data(int d)
```

```
	{
		data = d;
	}

	void
	print_data(void)
	{
		cout << "Length = " << data << endl;
	}

	int
	operator >(length l)
	{
		return l.data > data;
	}
};

int
main(void)
{
	length r1, r2;

	cout << (r1 > r2 ? "r1 is greater" : "r2 is greater") << endl;

	return 0;
}
```

Assignment

```
#include <iostream>
using namespace std;

class length {
private:
	int data;
public:
	void
	set_data(int d)
	{
		data = d;
	}

	void
	print_data(void)
	{
		cout << data << endl;
	}

	void
	operator =(length &l)
	{
		data = l.data - 1;
	}
};

int
main(void)
{
	length l1, l2;

	l1.set_data(5);
	l2 = l1;

	l1.print_data();
	l2.print_data();
}
```

3.3 Overloading binary operator with friend function

```
#include <iostream>
using namespace std;

class rectangle {
private:
	int length;
	int breadth;
public:
	void
	set_data(int l, int b)
	{
		length = l;
		breadth = b;
	}

	void
	print_data(void)
	{
		cout << "Length: " << length << ", Breadth: " << breadth << endl;
	}

	friend rectangle
	operator +(rectangle r1, rectangle r2);
};
```

```

rectangle
operator +(rectangle r1, rectangle r2)
{
        rectangle temp_r;

        temp_r.length = r1.length + r2.length;
        temp_r.breadth = r1.breadth + r2.breadth;

        return temp_r;
}

int
main(void)
{
        rectangle r1, r2, r3;

        r1.set_data(5, 6);
        r2.set_data(2, 5);

        r3 = r1 + r2;
        r3.print_data();

        return 0;
}
```

3.4 Data Conversion

3.4.1 From basic to basic

```
#include <iostream>
using namespace std;

int
main(void)
{
        float f_num;
        int i_num;

        f_num = 1234.567;
        i_num = (int)f_num;

        cout << i_num << endl;

        return 0;
}
```

3.4.2 From basic to user defined

```cpp
#include <iostream>
using namespace std;

class distances {
private:
        int feet;
        int inches;
public:
        distances(int i)
        {
                feet = i / 12;
                inches = i % 12;
        }

        void
        print_data(void)
        {
                cout << "Feet = " << feet << ", Inches = " << inches << endl;
        }
};

int
main(void)
{
        distances d(14);

        d.print_data();

        return 0;
}
```

3.4.3 From user defined to basic

```cpp
#include <iostream>
using namespace std;

class height {
private:
        int feet;
        int inches;
public:
        height(int f, int i)
        {
                feet = f;
                inches = i;
        }

        operator int(void)
        {
                int temp_i;

                temp_i = (feet * 12) + inches;

                return temp_i;
        }
};

int
main(void)
{
        height h(3, 7);
        int inches;

        inches = (int)h;

        cout << inches << endl;

        return 0;
}
```

3.4.4 From user defined to user defined

Routine in the source class

```
#include <iostream>
using namespace std;

class length_centimeter {
private:
        int centimeter;
public:
        void
        set_centimeter(int c)
        {
                centimeter = c;
        }

        void
        print_centimeter(void)
        {
                cout << centimeter << endl;
        }
};

class length_meter {
private:
        int meter;
public:
        void
        set_meter(int m)
        {
                meter = m;
        }

        operator length_centimeter(void)
        {
                length_centimeter temp_c;

                temp_c.set_centimeter(meter * 100);

                return temp_c;
        }
};

int
main(void)
{
        length_centimeter cm;
        length_meter m;

        m.set_meter(3);
        cm = (length_centimeter)m;

        cm.print_centimeter();

        return 0;
}
```

Routine in destination class

```
#include <iostream>
using namespace std;

class length_meter {
private:
        int meter;
public:
        void
        set_meter(int m)
```

```cpp
	{
		meter = m;
	}

	int
	get_meter(void)
	{
		return meter;
	}
};

class length_centimeter {
private:
	int centimeter;
public:
	length_centimeter(length_meter m)
	{
		centimeter = m.get_meter() * 100;
	}

	void
	print_centimeter(void)
	{
		cout << centimeter << endl;
	}
};

int
main(void)
{
	length_meter m;

	m.set_meter(3);

	length_centimeter cm(m);
	cm.print_centimeter();

	return 0;
}
```

ALTERNATIVE CODE

```cpp
#include <iostream>
using namespace std;

class length_meter {
private:
        int meter;
public:
        void
        set_meter(int m)
        {
                meter = m;
        }

        int
        get_meter(void)
        {
                return meter;
        }
};

class length_centimeter {
private:
        int centimeter;
public:
        length_centimeter(void)
        {
        }

        length_centimeter(length_meter m)
        {
                centimeter = m.get_meter() * 100;
        }

        void
        print_centimeter(void)
        {
                cout << centimeter << endl;
        }
};

int
main(void)
{
        length_meter m;
        length_centimeter cm;

        m.set_meter(3);
        cm = m;

        cm.print_centimeter();

        return 0;
}
```

Both in one

```
#include <iostream>
using namespace std;

class foo {
public:
	int data;
};

class bar {
public:
	int data;

	bar(void)
	{}

	/* routine in source */
	operator
	foo(void)
	{
		foo temp_f;

		temp_f.data = data;

		return temp_f;
	}

	/* routine in destination */
	bar(foo f)
	{
		data = f.data;
	}
};

int
main(void)
{
	foo f;
	bar b;

	/* routine in source */
	b.data = 5;
	f = (foo)b;
	cout << f.data << endl;

	/* routine in destination */
	f.data = 6;
```

```
        b = (bar)f;
        cout << b.data << endl;

        return 0;
}
```

Chapter 4

Inheritance

4.1 Overriding member functions

```
#include <iostream>
using namespace std;

class foo {
public:
	void
	say_hello(void)
	{
		cout << "Hello from foo" << endl;
	}
};

class bar: public foo {
public:
	void
	say_hello(void)
	{
		cout << "Hello from bar" << endl;
	}
};

int
main(void)
{
	bar b;

	b.foo::say_hello();

	return 0;
}
```

4.2 Types of inheritance

4.2.1 Single

```
#include <iostream>
using namespace std;

class foo {
public:
        foo(void)
        {
                cout << "Class foo initialized" << endl;
        }
};

class bar: public foo {
public:
        bar(void)
        {
                cout << "Class bar initialized" << endl;
        }
};

int
main(void)
{
        bar b;

        return 0;
}
```

4.2.2 Multiple

```
#include <iostream>
using namespace std;

class foo {
public:
        foo(void)
        {
                cout << "Class foo initialized" << endl;
        }
};

class bar {
public:
```

```cpp
        bar(void)
        {
                cout << "Class bar initialized" << endl;
        }
};

class buzz: public foo, public bar {
public:
        buzz(void)
        {
                cout << "Class buzz initialized" << endl;
        }
};

int
main(void)
{
        buzz b;

        return 0;
}
```

4.2.3 Hierarchical

```cpp
#include <iostream>
using namespace std;

class vehicle {
public:
        vehicle(void)
        {
                cout << "Vehicle initialized" << endl;
        }
};

class bike: public vehicle {
public:
        bike(void)
        {
                cout << "Bike initialized" << endl;
        }
};

class car: public vehicle {
public:
        car(void)
        {
                cout << "Car initialized" << endl;
        }
};

int
main(void)
{
        bike b;
        car c;

        return 0;
}
```

4.2.4 Multilevel

```
#include <iostream>
using namespace std;

class vehicle {
public:
        vehicle(void)
        {
                cout << "Vehicle initialized" << endl;
        }
};

class bike: public vehicle {
public:
        bike(void)
        {
                cout << "Bike initialized" << endl;
        }
};

class xpulse: public bike {
public:
        xpulse(void)
        {
                cout << "Xpulse initialized" << endl;
        }
};

int
main(void)
{
        xpulse x;

        return 0;
}
```

4.2.5 Hybrid

Jhew lagyo

4.3 Ambiguity in multiple inheritance

```
#include <iostream>
using namespace std;

class foo {
public:
        int data;
};

class bar {
public:
        int data;
};

class buzz: public foo, public bar {
};

int
main(void)
{
        buzz b;

        /* b.data = 5; */
        b.foo::data = 2;
        b.bar::data = 3;

        return 0;
}
```

4.4 Ambiguity in multipath inheritance

```
#include <iostream>
using namespace std;

class A {
public:
        int a;
};

class B: virtual public A {
public:
        int b;
};

class C: virtual public A {
public:
        int c;
};

class D: public B, public C {
public:
        int d;
};

int
main(void)
{
        D o;

        o.a = 1;
        o.b = 2;
        o.c = 3;
        o.d = 4;

        cout << o.a << endl;
        cout << o.b << endl;
        cout << o.c << endl;
        cout << o.d << endl;
}
```

4.5 Aggregation

```
#include <iostream>
using namespace std;

class employee {
private:
        string e_id;
        string e_name;
public:
        void
        set_data(string id, string name)
        {
                e_id = id;
                e_name = name;
        }

        string
        get_id(void)
        {
                return e_id;
        }

        string
        get_name(void)
        {
                return e_name;
        }
};

class college {
private:
        employee e;

        string c_id;
        string c_name;
public:
        void
        set_data(string id, string name)
        {
                c_id = id;
                c_name = name;
        }

        void
        set_employee_data(string id, string name)
        {
                e.set_data(id, name);
        }

        void
        print_data(void)
        {
                cout << "College ID: " << c_id << ", Name: " << c_name << endl;
                cout << "Employee ID: " << e.get_id() << ", Name: " <<
 ↪  e.get_name() << endl;
        }
};

int
```

```
main(void)
{
        college c;

        c.set_data("12", "foo");
        c.set_employee_data("34", "bar");

        c.print_data();

        return 0;
}
```

Chapter 5

Virtual function, Polymorphism and Miscellaneous C++ Features

5.1 Virtual function

```
#include <iostream>
using namespace std;

class Player {
public:
        virtual void
        say_hello(void)
        {
                cout << "Hello from Player" << endl;
        }
};

class Ram: public Player {
public:
        void
        say_hello(void)
        {
                cout << "Hello from Ram" << endl;
        }
};

class Shyam: public Player {
public:
        void
        say_hello(void)
        {
                cout << "Hello from Shyam" << endl;
        }
};
```

```

class Hari: public Player {
public:
        void
        say_hello(void)
        {
                cout << "Hello from Hari" << endl;
        }
};

class Sita: public Player {
public:
        void
        say_hello(void)
        {
                cout << "Hello from Sita" << endl;
        }
};

int
main(void)
{
        Player *p;

        Ram r;
        Shyam s;
        Hari h;
        Sita si;

        p = &r;
        p->say_hello();

        p = &s;
        p->say_hello();

        p = &h;
        p->say_hello();

        p = &si;
        p->say_hello();

        return 0;
}
```

5.2 Pure virtual function

```cpp
#include <iostream>
using namespace std;

class Player {
public:
        virtual void
        say_hello(void) = 0;
};

class Ram: public Player {
public:
        void
        say_hello(void)
        {
                cout << "Hello from Ram" << endl;
        }
};

class Shyam: public Player {
public:
        void
        say_hello(void)
        {
                cout << "Hello from Shyam" << endl;
        }
};

class Hari: public Player {
public:
        void
        say_hello(void)
        {
                cout << "Hello from Hari" << endl;
        }
};

class Sita: public Player {
public:
        void
        say_hello(void)
        {
                cout << "Hello from Sita" << endl;
        }
};

```

```
int
main(void)
{
        Player *p;

        Ram r;
        Shyam s;
        Hari h;
        Sita si;

        p = &r;
        p->say_hello();

        p = &s;
        p->say_hello();

        p = &h;
        p->say_hello();

        p = &si;
        p->say_hello();

        return 0;
}
```

5.3 Virtual destructor

```
#include <iostream>
using namespace std;

class foo {
public:
        virtual ~foo(void)
        {
                cout << "Foo is destroyed" << endl;
        }
};

class bar: public foo {
public:
        ~bar(void)
        {
                cout << "Bar is destroyed" << endl;
        }
};

```

```
int
main(void)
{
        foo *f = new bar;

        delete f;

        return 0;
}
```

5.4 Friend class

```
#include <iostream>
using namespace std;

class foo {
private:
        int data;
public:
        void
        set_data(int d)
        {
                data = d;
        }

        friend class bar;
};

class bar {
private:
        int data;
public:
        void
        print_foo_data(foo f)
        {
                cout << f.data << endl;
        }
};

int
main(void)
{
        foo f;
        bar b;

        f.set_data(5);
        b.print_foo_data(f);

        return 0;
}
```

5.5 Friend Function

```
#include <iostream>
using namespace std;

class foo {
private:
        int bar;
public:
        friend void
        set_bar(foo &f, int b);

        friend void
        print_bar(foo f);
};

void
set_bar(foo &f, int b)
{
        f.bar = b;
}

void
print_bar(foo f)
{
        cout << f.bar << endl;
}

int
main(void)
{
        foo f;

        set_bar(f, 14);
        print_bar(f);
}
```

5.6 Static function

```
#include <iostream>
using namespace std;

class foo {
private:
        int data;
        static int foo_count;
public:
        foo(void)
        {
                foo_count++;
        }

        static void
        print_foo_count(void)
        {
                cout << foo_count << endl;
        }
};

int foo::foo_count = 0;

int
main(void)
{
        foo f1, f2, f3;

        foo::print_foo_count();

        return 0;
}
```

Chapter 6

Function Templates and Exception Handling

6.1 Function templates

```
#include <iostream>
using namespace std;

template <class T>
T my_max(T x, T y)
{
        return (x > y) ? x : y;
}

int
main(void)
{
        cout << my_max(30, 20) << endl;
        cout << my_max(2.5, 10.3) << endl;
        cout << my_max('c', 'x') << endl;

        return 0;
}
```

6.2 Function templates with multiple arguments

```
#include <iostream>
using namespace std;

template <class T, class U>
T my_max(T x, U y)
{
        return (x > y) ? x : y;
}

int
main(void)
{
        cout << my_max(5, 3.4) << endl;
        cout << my_max(10.3, 4) << endl;
        cout << my_max(100, 'y') << endl;
}
```

6.3 Class template

```
#include <iostream>
using namespace std;

template <class T>
class foo {
private:
        T data;
public:
        void
        set_data(T d)
        {
                data = d;
        }

        void
        print_data(void)
        {
                cout << data << endl;
        }
};

int
main(void)
{
        foo<int> f;

        f.set_data(4.2);
        f.print_data();
}
```

6.4 Inheritance in template class

```
#include <iostream>
using namespace std;

template <class T>
class foo {
private:
        T data;
public:
        void
        set_data(T d)
        {
                data = d;
        }

        void
        print_data(void)
        {
                cout << data << endl;
        }
};

template <class T>
class bar: public foo<T> {
private:
        T data;
public:
        void
        set_data(T d1, T d2)
        {
                data = d1;
                foo<T>::set_data(d2);
        }

        void
        print_data(void)
        {
                cout << data << endl;
                foo<T>::print_data();
        }
};

int
main(void)
{
        bar<int> b1;
        bar<double> b2;

        b1.set_data(3, 5);
        b2.set_data(1.3, 6.2);

        b1.print_data();
        b2.print_data();
}
```

6.5 Exception Handling

```
#include <iostream>
using namespace std;

int
main(void)
{
        int a, b;

        a = 4;
        b = 0;

        try {
                if (b == 0)
                        throw b;
                else
                        cout << "a / b = " << a/b << endl;
        }
        catch(int x)
        {
                cout << "Divided by zero" << endl;
        }
}
```

6.6 Multiple catch blocks for a single try block

```
#include <iostream>
using namespace std;

void
test(int x)
{
        try {
                if (x > 0) {
                        throw x;
```

```
		} else {
			throw 'x';
		}
	} catch(int a) {
		cout << "Its a digit: " << a << endl;
	} catch(char a) {
		cout << "Its a character: " << a << endl;
	}
}

int
main(void)
{
	test(4);
	test(0);
}
```

6.7 Catch all exceptions

```
#include <iostream>
using namespace std;

int
main(void)
{
	int num1, num2, result;
	char op;

	try {
		cout << "Enter the first number: ";
		cin >> num1;

		cout << "Enter the operator: ";
		cin >> op;

		cout << "Enter the second number: ";
		cin >> num2;

		switch (op) {
		case '+':
			result = num1 + num2;
			break;
		case '-':
			result = num1 - num2;
			break;
		case '*':
```

```
                result = num1 * num2;
                break;
            case '/':
                if (num2 == 0) {
                    throw num2;
                }
                result = num1 / num2;
                break;
            default:
                throw op;
        }

        cout << result << endl;
    } catch(...) {
        cout << "Oh no! Exception occurred!" << endl;
    }
}
```

Chapter 7

File handling and Streams

7.1 Read contents of a file character by character

```
#include <iostream>
#include <fstream>
using namespace std;

int
main(void)
{
        char ch;
        ifstream fin;

        fin.open("text");
        while (!fin.eof()) {
                fin.get(ch);
                cout << ch;
        }

        fin.close();
        return 0;
}
```

7.2 Writing as well as reading from the file

```
#include <iostream>
#include <fstream>
using namespace std;

int
main(void)
{
        char name[100];
        int age;

        /* = PUTTING CONTENTS INTO THE FILE = */
        ofstream outfile;
        outfile.open("text");       /* Opening the file we want to write to */

        cout << "Enter your name: ";
        cin.getline(name, 100);     /* same as 'cin >> name;' */
        outfile << name << endl;    /* Writing the name with a new line into the file */

        cout << "Enter your age: ";
        cin >> age;                 /* we cannot use the getline method because age is an int */
        outfile << age << endl;     /* Appending the age and a new line into the file */

        outfile.close();            /* Closing the opened file stream */

        /* = READING CONTENTS FROM THE FILE = */
        ifstream infile;
        infile.open("text");        /* Opening the file we want to read from */

        infile.getline(name, 100);  /* same as 'infile >> name;' */
        cout << name << endl;

        infile >> age;              /* read the age */
        cout << age << endl;

        infile.close();             /* Closing the opened file stream */

        return 0;
}
```

7.3 Write and Read to/from a single object

```
#include <iostream>
#include <fstream>
using namespace std;

class foo {
private:
        int data;
public:
        void
        set_data(int d)
        {
                data = d;
        }

        void
        print_data(void)
        {
                cout << data << endl;
        }
};

int
main(void)
{
        foo f, f2;

        /* = WRITE OUT THE OBJECT TO THE FILE = */
        ofstream fout;

        fout.open("text");

        f.set_data(4);
        fout.write((char *)&f, sizeof(foo));

        fout.close();

        /* = READ ABOUT THE OBJECT FROM THE FILE = */
        ifstream fin;

        fin.open("text");

        fin.read((char *)&f2, sizeof(foo));
        f2.print_data();

        fin.close();
}
```

7.4 Write multiple objects

```
#include <iostream>
#include <fstream>
using namespace std;

class foo {
private:
        int data;
public:
        void
        take_data(void)
        {
                cout << "Enter a data: ";
                cin >> data;
        }

        void
        print_data(void)
        {
                cout << data << endl;
        }
};

int
main(void)
{
        foo f;
        ofstream fout;

        fout.open("text");

        for (int i = 0; i < 5; i++) {
                f.take_data();
                fout.write((char *)&f, sizeof(foo));
        }

        fout.close();
        return 0;
}
```

7.5 Read multiple objects

```
#include <iostream>
#include <fstream>
using namespace std;

class foo {
private:
        int data;
public:
        void
        set_data(void)
        {
```

```
		cout << "Enter a data: " << endl;
		cin >> data;
	}

	void
	print_data(void)
	{
		cout << data << endl;
	}
};

int
main(void)
{
	foo f;
	ifstream fin;

	fin.open("writtentext");

	for (int i = 0; i < 5; i++) {
		fin.read((char *)&f, sizeof(foo));

		f.print_data();
	}

	fin.close();
}
```

7.6 Overloading insertion operator

```
#include <iostream>
using namespace std;

class foo {
private:
	int data;
public:
	void
	set_data(int d)
	{
		data = d;
	}

	friend ostream &
	operator <<(ostream &o, foo f);
};
```

```

ostream &
operator <<(ostream &os, foo f)
{
        os << f.data;

        return os;
}

int
main(void)
{
        foo f;

        f.set_data(4);

        cout << f << endl;
}
```

7.7 Overloading extraction operator

```
#include <iostream>
using namespace std;

class foo {
private:
        int data;
public:
        void
        print_data(void)
        {
                cout << data << endl;
        }

        friend istream &
        operator >>(istream &is, foo &f);
};

istream &
operator >>(istream &is, foo &f)
{
        is >> f.data;

        return is;
}

```

```
int
main(void)
{
        foo f;

        cin >> f;
        f.print_data();
}
```

7.8 Overloading both insertion and extraction operator

```
#include <iostream>
using namespace std;

class foo {
private:
        int data;
public:
        friend ostream &
        operator <<(ostream &os, foo f);

        friend istream &
        operator >>(istream &is, foo &f);
};

ostream &
operator <<(ostream &os, foo f)
{
        os << f.data;

        return os;
}

istream &
operator >>(istream &is, foo &f)
{
        is >> f.data;

        return is;
}

int
main(void)
{
        foo f;

        cin >> f;
        cout << f << endl;
}
```

Chapter 8

Extra

8.1 Reverse string

```
#include <iostream>
using namespace std;

string
reverse(string msg)
{
	string rev;

	for (int i = 0; i < msg.length(); i++) {
		rev += msg[msg.length() - 1 - i];
	}

	return rev;
}

int
main(void)
{
	string msg, rev_msg;

	msg = "foobar";
	rev_msg = reverse(msg);

	cout << rev_msg << endl;
}
```

8.2 Stack implementation using templates

```
#include <iostream>
using namespace std;

template <class T>
class my_stack {
private:
        T data[999];
        int cur_index;
public:
        my_stack(void)
        {
                cur_index = 0;
        }

        void
        push(T d)
        {
                data[cur_index++] = d;
        }

        T
        pop(void)
        {
                return data[--cur_index];
        }
};

int
main(void)
{
        my_stack<int> numbers;
        my_stack<string> strings;

        numbers.push(5);
        numbers.push(6);
        numbers.push(7);

        cout << numbers.pop() << endl;
        cout << numbers.pop() << endl;
        cout << numbers.pop() << endl;

        strings.push("foo");
        strings.push("bar");
        strings.push("buzz");

        cout << strings.pop() << endl;
        cout << strings.pop() << endl;
        cout << strings.pop() << endl;
}
```

8.3 2078 - 1

```
#include<iostream>
using namespace std;

class Account {
private:
        string acc_no;
        int balance;
        static int min_balance;

public:
        void
        set_values(void)
        {
                string acc_no;
                int balance;

                cout << "Enter the account number: ";
                cin >> acc_no;

                cout << "Enter the balance: ";
                cin >> balance;

                this->acc_no = acc_no;
                this->balance = balance;

                if (min_balance == 0 || min_balance > balance) {
                        min_balance = balance;
                }
        }

        void
        print_values(void)
        {
                cout << "Account no = " << acc_no << ", Balance = " << balance <<
    ↪   endl;
        }

        static void
        print_min_balance(void)
        {
                cout << min_balance << endl;
        }
};

int Account::min_balance = 0;

int
main(void)
{
        Account a[5];

        for (int i = 0; i < 5; i++) {
                a[i].set_values();
        }

        for (int i = 0; i < 5; i++) {
                a[i].print_values();
        }

        Account::print_min_balance();
}
```

8.4 2076 - 1

```
#include <iostream>
using namespace std;

class Teacher {
private:
        string tid;
        string subject;
public:
        void
        set_values(void)
        {
                cout << "Enter the teacher id: ";
                cin >> tid;

                cout << "Enter the teacher subject: ";
                cin >> subject;
        }

        void
        print_values(void)
        {
```

```cpp
                cout << "TID = " << tid << ", Subject = " << subject << endl;
        }
};

class Staff {
private:
        string sid;
        string position;
public:
        void
        set_values(void)
        {
                cout << "Enter the staff id: ";
                cin >> sid;

                cout << "Enter the staff position: ";
                cin >> position;
        }

        void
        print_values(void)
        {
                cout << "SID = " << sid << ", Position = " << position << endl;
        }
};

class Coordinator: public Teacher, public Staff {
private:
        string department;
public:
        void
        set_values(void)
        {
                Staff::set_values();
                Teacher::set_values();

                cout << "Enter the department: ";
                cin >> department;
        }

        void
        print_values(void)
        {
                Staff::print_values();
                Teacher::print_values();

                cout << "Department = " << department << endl;
        }
```

```
};

int
main(void)
{
	Coordinator c1, c2;

	c1.set_values();
	c2.set_values();

	c1.print_values();
	c2.print_values();

	return 0;
}
```