

C++ Programming, Comprehensive

Jagadeesh Vasudevamurthy Ph. D

Instructor

UCSC Extension in Silicon Valley

Santa Clara, CA 95054 USA

jvasudev@ucsc.edu

C++ Programming, Comprehensive

(3.0 quarter units)

C++ is a general-purpose object-oriented programming language that offers portability, speed, and modularity, as well as compatibility with C and other languages. Because most automation, embedded applications, gaming, and many large data processing applications are written in C++, it is essential that software developers understand and master it. Topics include object-oriented concepts; structure and input/output streams; declarations, identifiers, pointers, and arguments; memory management, constructors, and destructors; enumeration type, as constructor parameter; character strings, file I/O, functions; inheritance, and interaction diagrams; and exception handling, pointers, and functions.

Prerequisite(s): "C Programming for Beginners."
Experience with a high level programming language such as C. Advanced C programming recommended.

JAGADEESH VASUDEVAMURTHY, Ph.D.

CLASSROOM WITH ONLINE MATERIALS
10 meetings | 6–9 pm | April 3–June 5
Fee: \$1020.

To enroll, use Section Number CMPR.X404.(8)

JAGADEESH VASUDEVAMURTHY, Ph.D., has more than 20 years of experience in electronic design automation. He has worked on the design and development of commercial EDA tools at Cadence, Xilinx, Synplicity and Mentor Graphics. He is a senior member of IEEE and a member of ACM.

**Draft 10
April 2018**

SPRING
APRIL-JUNE 2018

नहि ज्ञानेन सदृशं

Nahi Jnanena Sadrusham

Nothing is equivalent to knowledge

This work is dedicated to my mother V.Radhabai
who sacrificed her life to provide me the best possible education.

Contents

1	Introduction	9
1.1	Introduction	9
1.2	Installing Microsoft Visual C++ 2015 compiler	11
1.3	Writing, compiling and debugging first program using Microsoft Visual C++ 2015 compiler	17
1.4	First C++ program revisited	26
1.4.1	<i>util.h</i>	26
1.4.2	<i>class helloworld</i>	28
1.4.3	C++code for <i>class helloworld</i>	30
1.5	Basic data types	34
1.5.1	Boolean number system	34
1.5.2	bool data type	36
1.5.3	char data type	36
1.5.4	int data type	38
1.5.5	float and double data type	41
1.5.6	C++code for illustrating basic data types	41
1.6	User defined data types	47
1.7	Arithmetic operations	49
1.7.1	C++code for illustrating arithmetic operations	50
1.8	Relational operations	56
1.8.1	C++code for illustrating relational operations	57
1.9	Logical operations	61

CONTENTS

1.9.1 C++code for illustrating logical operations	61
1.10 Bitwise operations	65
1.10.1 C++code for illustrating bitwise operations	67
1.11 Assignment operations	72
1.12 Control statements	72
1.12.1 C++code for illustrating control statements	75
1.13 Enumerations(enum)	87
1.13.1 C++code for illustrating enumeration	90
1.14 Introductions to functions	95
1.14.1 C++code for illustrating call by value function	95
1.15 Introductions to pointers	100
1.15.1 C++code for illustrating pointers	103
1.16 Understanding const	108
1.17 Introductions to arrays	111
1.17.1 C++code for illustrating array and pointers to array elements	113
1.18 Basic stream output and stream input	119
1.18.1 C++code for illustrating stream output	121
1.18.2 C++code for illustrating stream input	127
1.19 Problem set	133
2 C Strings	155
2.1 Introduction	155
2.2 Character manipulation	155
2.3 C Strings	162
2.4 Problem set	169
3 Concept of reference	171
3.1 Introduction	171
3.2 Concept of swapping using reference	173
3.3 FAQ on references	175

CONTENTS

3.4	Passing <code>char*</code> by value, address and reference	183
4	Dynamic memory allocation	185
4.1	Introduction	185
4.2	Static and dynamic memory allocation	185
4.3	Allocating two dimensional array	191
4.3.1	Static allocation	191
4.3.2	Dynamic allocation	193
4.4	Problem set	198
5	Function overloading and default function arguments	205
5.1	Introduction	205
5.2	Function overloading	205
5.3	Default function arguments	207
6	C++ class	209
6.1	Introduction	209
6.2	Need for <i>class</i>	209
6.3	<i>class</i> and objects	213
6.4	Accessing class members using dot and arrow operator	213
6.5	Accessing private class members	217
6.6	Separating interface and implementation	219
6.7	Allocating class objects on stack and heaps, phase 1	224
6.8	Shallow copying of objects, phase 1	233
6.9	Creating temporary objects on the stack, phase1	236
6.10	inline function	239
6.11	Understanding <code>const</code>	243
6.12	Need for constructors	246
6.13	Creating temporary objects on the stack, phase 2	253
6.14	Shallow copying of objects, phase 2	256
6.15	Copy constructor and equal operator	259

CONTENTS

6.15.1 Fraction class	271
6.16 Private constructors	275
6.17 Constructor initialization list	277
6.17.1 Initialization versus assignment	277
6.17.2 Order of initialization	277
6.18 Static member of a class	282
6.19 Size of a class	282
6.20 Need for friend	282
6.20.1 friend function	282
6.20.2 friend class	292
6.21 Type conversions	295
6.22 Usage of class string	302
6.23 Object composition	302
6.24 Returning member variables of a class	309
6.25 Problem set	312
7 Operator overloading	327
7.1 Introduction	327
7.2 Overloading an operator as a member function or as a friend	327
7.3 Operators supported by C++ on basic types	339
7.4 Operators that cannot be overloaded	342
7.5 Operators precedence in C++	342
7.6 Int class	344
7.6.1 Unary operators	346
7.6.2 Binary operators	351
7.7 A set class	358
7.8 A long number class	370
7.9 mstring class	371
8 Inheritance	381

CONTENTS

8.1	Introduction	381
8.2	Why Inheritance?	381
8.3	Need for keyword: protected	383
8.4	Overriding or hiding base class member functions	388
8.5	Writing constructor for a derived class	392
8.6	Writing destructor for a derived class	392
8.7	Writing copy constructor and equal operator for a derived class in a BAD way	394
8.8	Writing copy constructor and equal operator for a derived class in a CORRECT way	394
8.9	Writing copy constructor for a derived class	400
8.10	Writing equal operator for a derived class	400
8.11	Writing insertion operator for a derived class	402
8.12	Using base and derived class objects	402
8.13	Problems of static binding and need for polymorphism	413
8.14	Polymorphism through keyword virtual	419
8.15	Rewriting exam program using polymorphism	423
8.16	Abstract class and concept of pure virtual function	432
8.17	Casting from derived to base and base to derived	436
8.18	Implementing polymorphism without using virtual	441
9	Namespaces	445
9.1	Introduction	445
9.2	What is the problem?	445
9.3	Need for namespace	446
10	Template	453
10.1	Introduction	453
10.2	Macros	453
10.3	Function templates	457
10.4	Class templates	462

CONTENTS

10.5 Integer stack implementation	462
10.6 Generic stack implementation using <code>typedef</code>	469
10.7 Stack implementation using inheritance	477
10.7.1 Problem of using objects and static binding	477
10.8 Stack implementation using template	492
11 Exception handling	505
11.1 Introduction	505
11.2 Problem	505
11.3 Try and catch	509
11.4 Stack example	514
12 Standard Template Library	525
12.1 Introduction	525
12.2 <code>util</code> class	525
12.3 <code>complex</code> class	532
12.4 <code>vector</code> class	541
12.5 <code>string</code> class	550
12.6 <code>deque</code> class	556
12.6.1 <code>stack</code> class	567
12.6.2 <code>queue</code> class	571
12.7 <code>unorderedmap</code> or <code>hash</code> class	575
12.8 <code>set</code> class	587

Chapter 1

Introduction

1.1 Introduction

1.1. INTRODUCTION

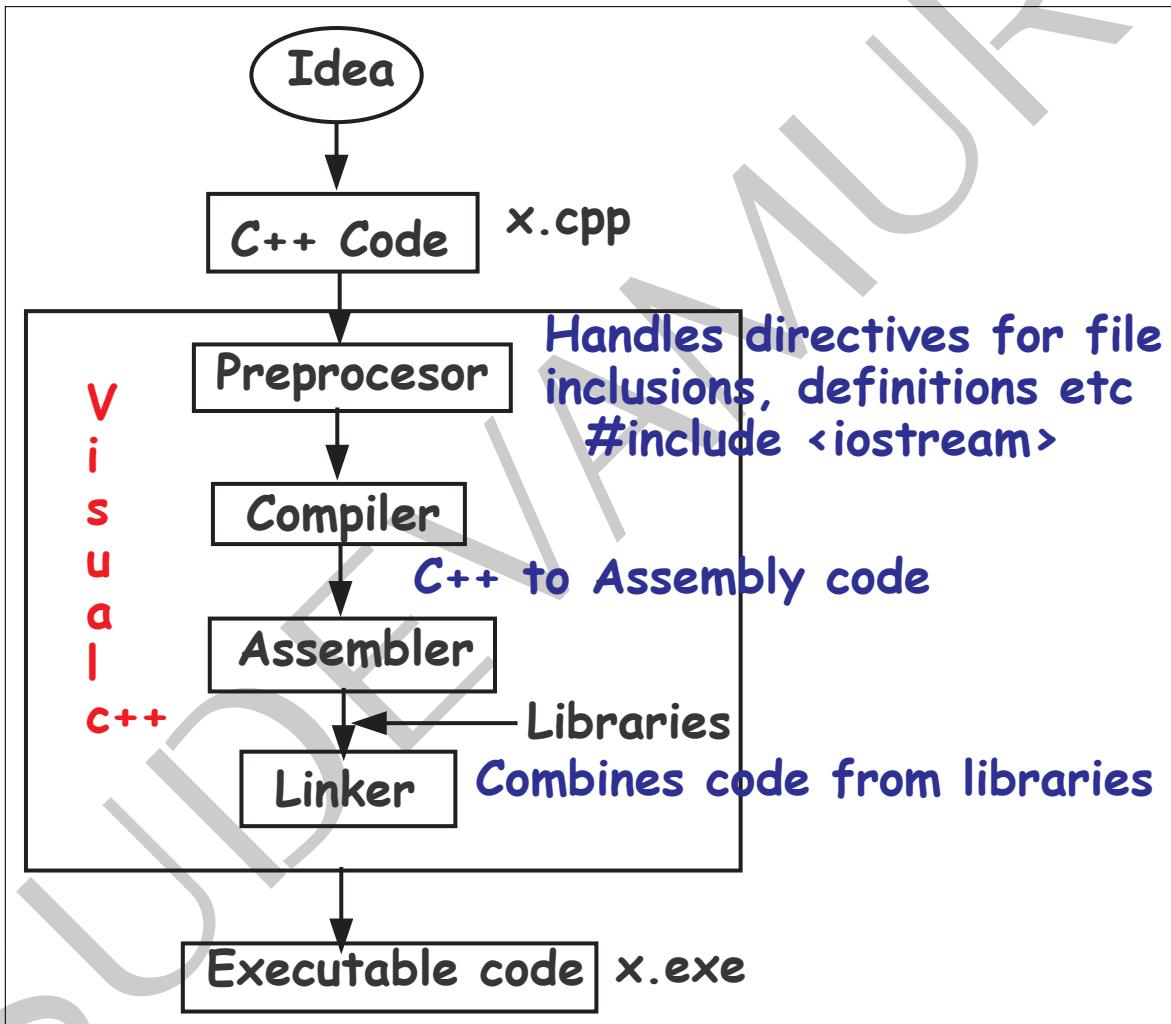


Figure 1.1: Typical C++ environment

1.2 Installing Microsoft Visual C++ 2015 compiler

Getting free Microsoft Visual Studio Community 2015

Jagadeesh Vasudevamurthy

06/22/2016

1. Go this website:

<https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx>



Click on Download



Visual Studio

Community 2015
with Updates

Choose your installation location

C:\Program Files (x86)\Microsoft Visual Studio 14.0

...

Setup requires up to 18 GB across all drives.

By clicking the "Next" button, I acknowledge that I accept the [License Terms](#) and [Privacy Statement](#).

[Cancel](#)

[Next](#)



Community 2015 with Updates

Select features

- Programming Languages
 - Visual C++
 - Common Tools for Visual C++ 2015
 - Microsoft Foundation Classes for C++
 - Windows XP Support for C++
 - Visual F#
 - Python Tools for Visual Studio (March 2016)
- Windows and Web Development
 - ClickOnce Publishing Tools
 - Microsoft SQL Server Data Tools
 - Microsoft Web Developer Tools
 - PowerShell Tools for Visual Studio (3rd Party)

[Select All](#)

[Reset Defaults](#)

Setup requires up to 16 GB across all drives.

[Back](#)

[Next](#)



Community 2015 with Updates

Selected features

MICROSOFT SOFTWARE

C#/.NET (Xamarin v4.1.0)

[License Terms](#)

C#/.NET (Xamarin v4.0.2 or earlier) will be removed

[License Terms](#)

Windows 8.1 SDK and Universal CRT SDK

Common Tools for Visual C++ 2015

Tools for Universal Windows Apps (1.3.2) and Windows 10 SDK (10.0.10586)

THIRD-PARTY SOFTWARE

Android Native Development Kit (R10E, 32 bits)

[License Terms](#)

Android SDK Setup (API Level 23)

[License Terms](#)

Setup requires up to 16 GB across all drives.

By clicking "Install" you agree to the license terms of the software and components you have selected to install. You are responsible for reading and accepting these license terms. Microsoft grants you no rights for any third party software you have selected; it is provided by the third parties indicated in the applicable license terms.

[Back](#)

[Install](#)

**With this installation is you should see this on your desktop
Make a short cut on desktop**



1.3 Writing, compiling and debugging first program using Microsoft Visual C++ 2015 compiler

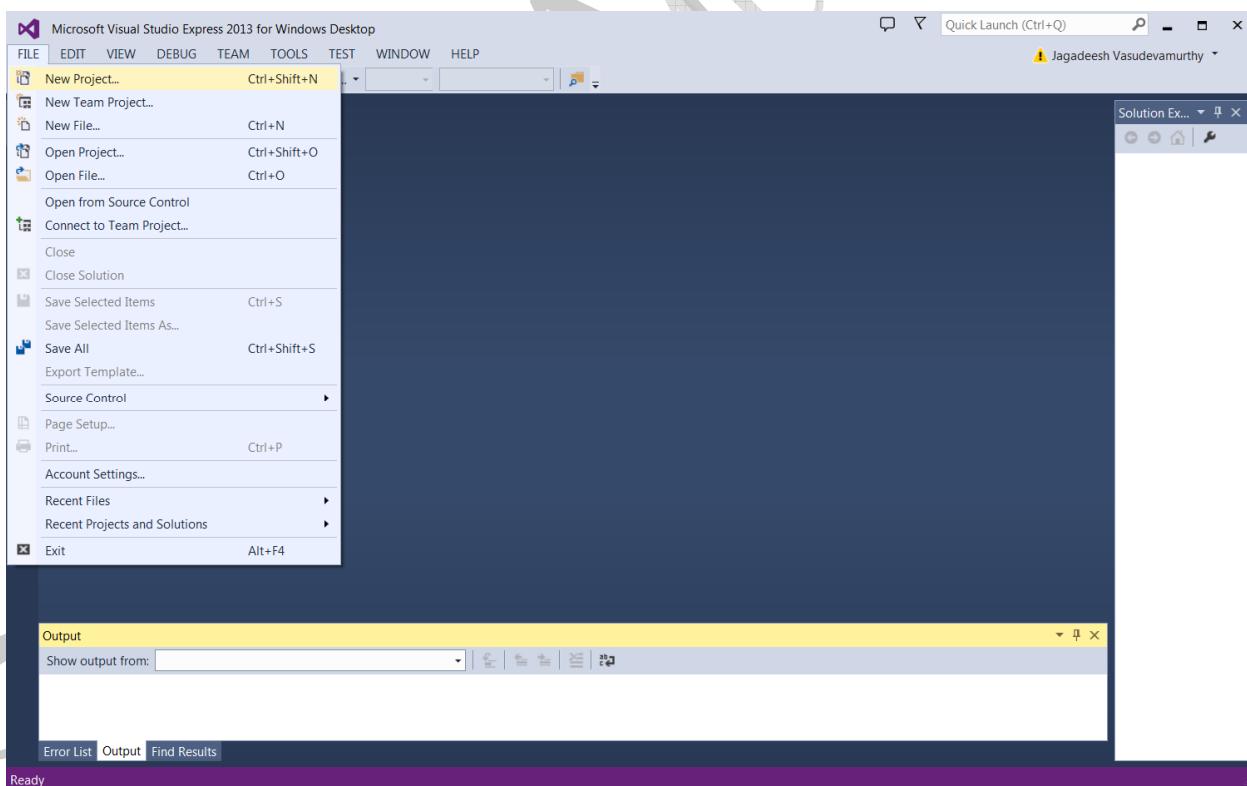
Writing, compiling and debugging first program

Jagadeesh Vasudevamurthy
03/21/2016

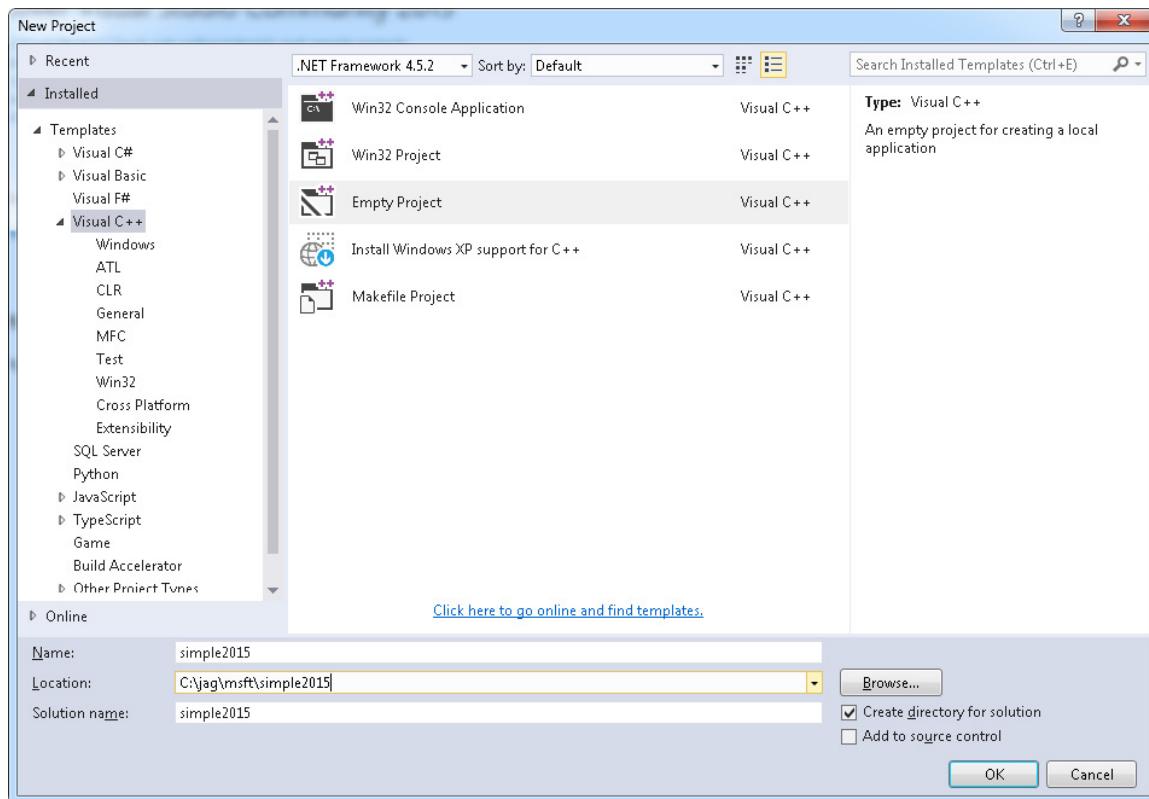
1. Double click on this icon



2. Create a new project

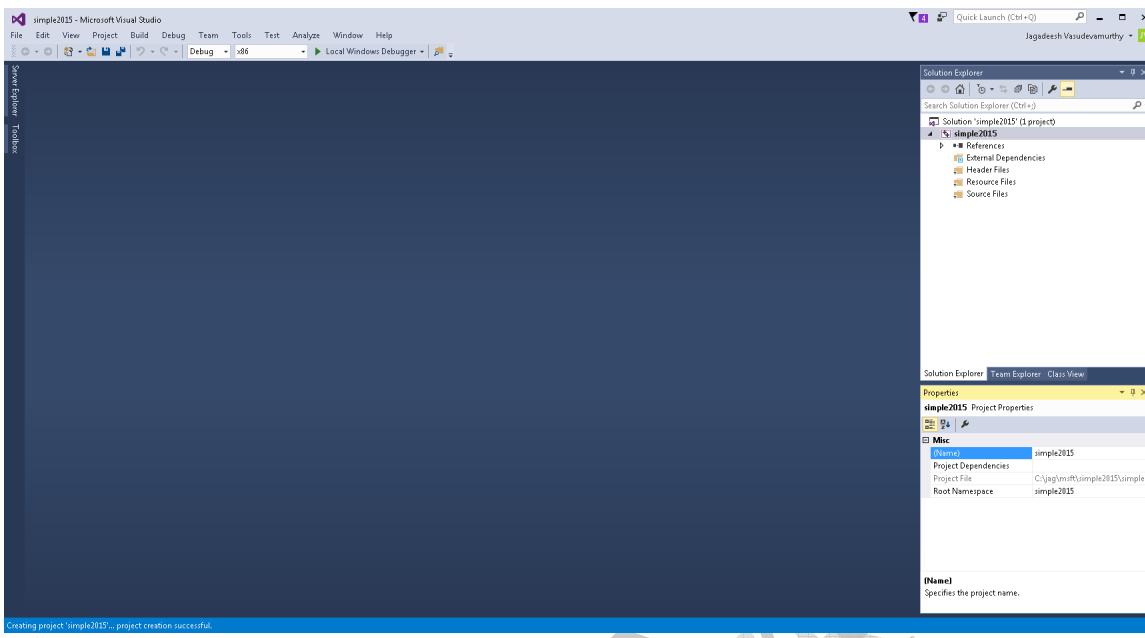


3. Click on New Project



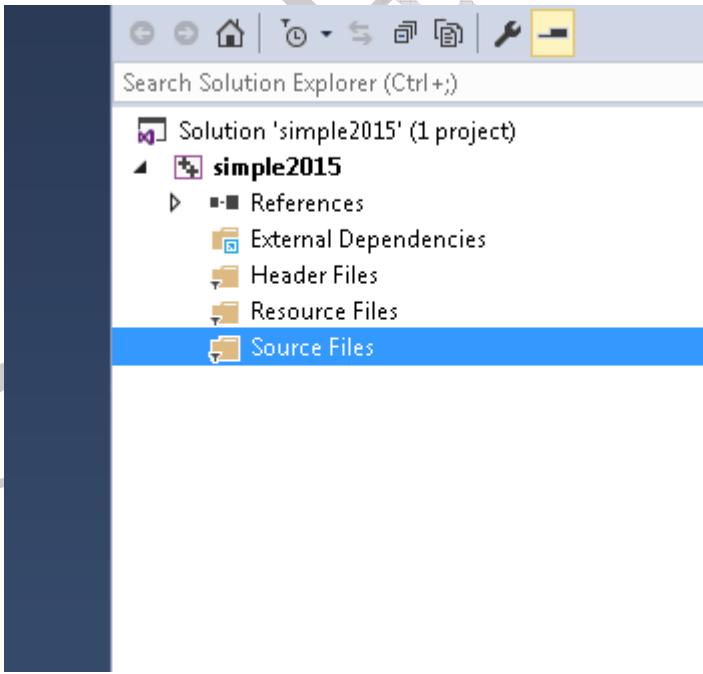
1. Select **EMPTY PROJECT**
2. Name: **simple2015**
3. Location: **C:\jag\msft\2015**
4. Click **OK**

4. You will then see a screen as follows:



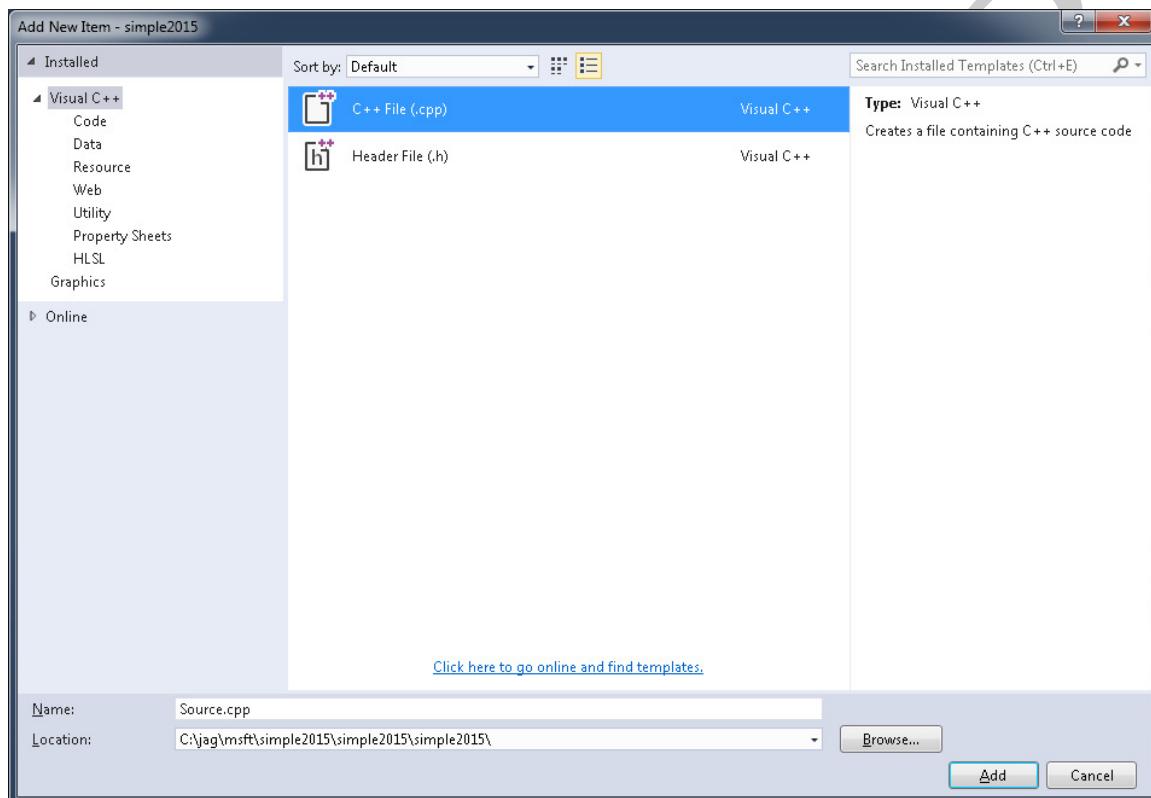
5.

1. Right click on Source file
2. select Add -> New Item as follows

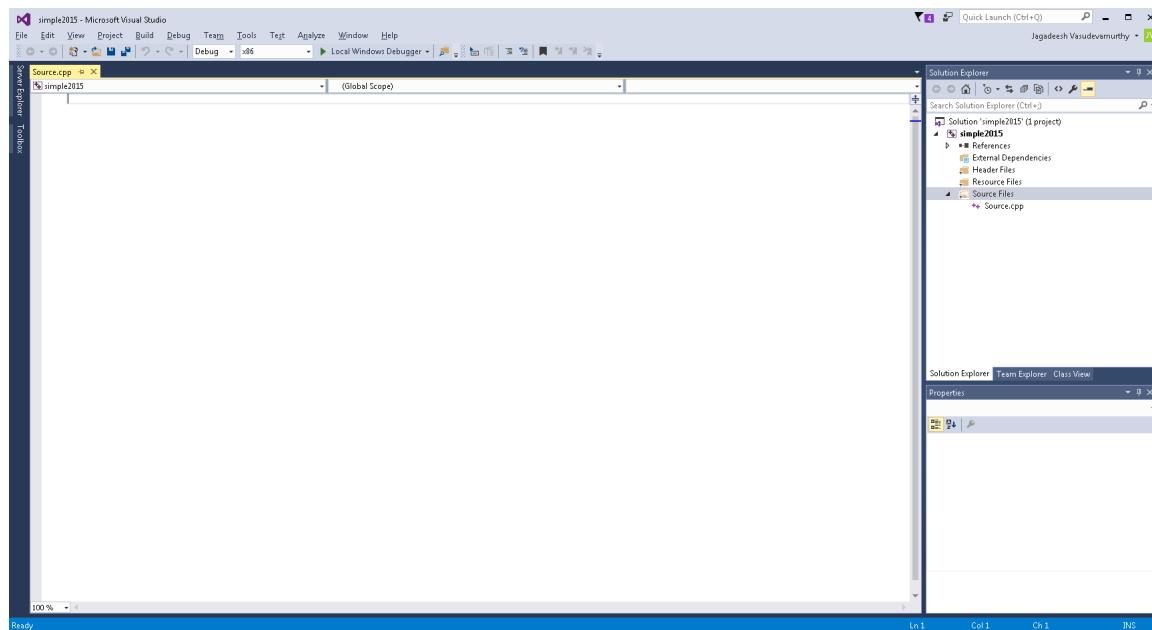


6.

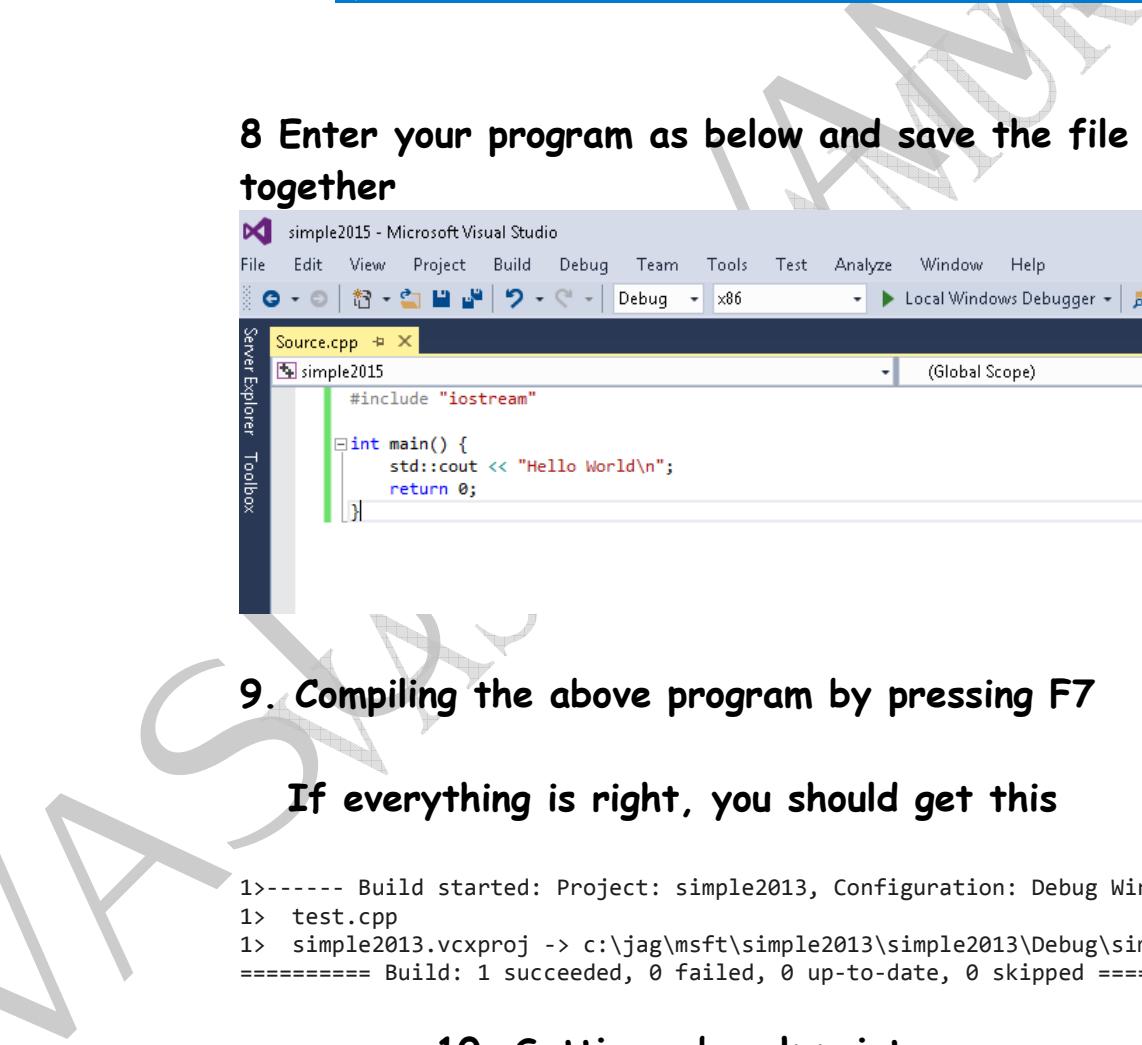
- 1. Select C++ file (cpp)**
- 2. Name: test.cpp**
- 3. Location: C:\test**
- 4. Click on Add**



7 Now you will see a screen as follows:



8 Enter your program as below and save the file with Ctrl S together



simple2015 - Microsoft Visual Studio

File Edit View Project Build Debug Team Tools Test Analyze Window Help

Source.cpp (Global Scope)

```
#include <iostream>
int main() {
    std::cout << "Hello World\n";
    return 0;
}
```

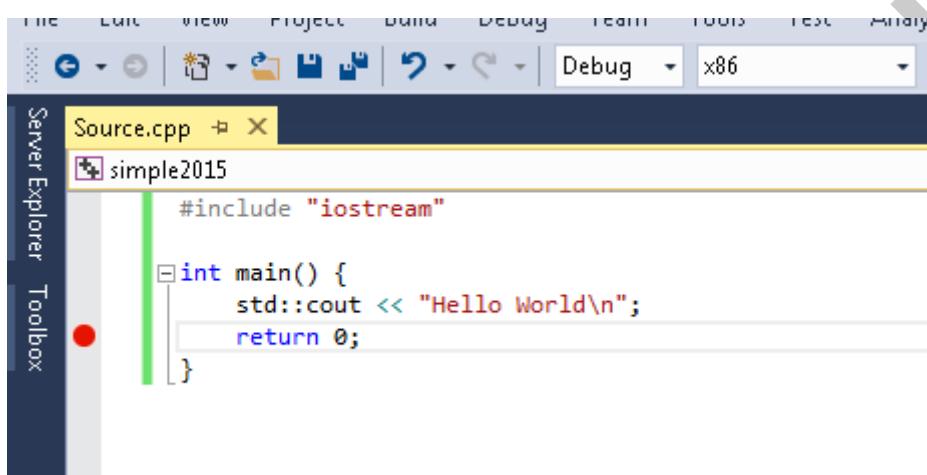
9. Compiling the above program by pressing F7

If everything is right, you should get this

```
1>----- Build started: Project: simple2013, Configuration: Debug Win32 -----
1> test.cpp
1> simple2013.vcxproj -> c:\jag\msft\simple2013\simple2013\Debug\simple2013.exe
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
```

10. Setting a break point

Go to last line }
Left click on }
Press F9
You will see a red dot as follows



A screenshot of the Microsoft Visual Studio IDE. The window title is "Source.cpp" and the project name is "simple2015". The code editor shows a simple "Hello World" program. A red circular breakpoint icon is visible on the left margin at the end of the main function's closing brace "}".

```
#include <iostream>
int main() {
    std::cout << "Hello World\n";
    return 0;
}
```

11. Running the program

Press F5

You will see output as follows:

A screenshot of Microsoft Visual Studio showing a 'Hello World' application. The title bar says 'Source.cpp' and 'simple2015'. The code editor contains the following C++ code:

```
#include <iostream>

int main() {
    std::cout << "Hello World\n";
    return 0;
}
```

The output window on the left shows the text 'Hello World'.

Compiling using gnu g++ compiler on Linux system

Program: test.cpp

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

int main() {
    cout << "Welcome to CE589B course\n" ;
    return 0 ;
}
```

Compiling using g++
g++ test.cpp -o test

Running the program
.test

Welcome to CE589B course

1.4. FIRST C++ PROGRAM REVISTED

1.4 First C++ program revisted

The diagram shows the code for a "Hello World" program with various annotations:

```
1 //file: 1.cpp          //Comment for humans. Ignored by compiler
2 #include <iostream>    //Tells the preprocessor to include this file that has I/O operations
3 using namespace std;   //Use cout object that is defined in std namespace
4 int main() {           Begin of main
5     cout << "Hello World" << endl; //Send the message "Hello World" to the object cout, so that it will be written on your screen
6     return 0;             After you write "Hello World", enter a line feed
7 }                      Name of the procedure that will called first by the compiler
8 }                      End of main
This procedure returns an integer at the end so that the caller can use it.
0 means the procedure exited successfully
1 means failure
```

Note : after every line of code except #include

```
1 //file: 1.cpp
2 #include <iostream>
3 //using namespace std;
4 int main() {
5     cout << "Hello World" << endl;
6     return 0;
7 }
8
```

Compiler will no longer understand cout and endl

1.cpp
..\\..\\..\\jag\\c++\\course\\code\\ch1\\1.cpp(5) : error C2065: 'cout' : undeclared identifier
..\\..\\..\\jag\\c++\\course\\code\\ch1\\1.cpp(5) : error C2065: 'endl' : undeclared identifier

Figure 1.2: Understanding first C++ program

1.4.1 util.h

```
1 /*-----  
2 Copyright (c) 2012 Author: Jagadeesh Vasudevamurthy  
3 file: util.h  
4 -----*/  
5  
6 /*-----  
7 include this file for all the programs  
8 -----*/  
9 #ifndef UTIL_H  
10 #define UTIL_H  
11  
12 /*-----  
13 Basic include files  
14 -----*/  
15 #include <iostream>  
16 #include <fstream>  
17  
18 #include <iomanip> // std::setprecision  
19 using namespace std;  
20  
21 #ifdef _WIN32  
22 #include <cassert>  
23 #include <ctime>  
24 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector  
25 #else  
26 #include <assert.h>  
27 #include <time.h>  
28 #include <string.h> //for strlen,strcat and strcpy on linix  
29 #endif  
30  
31 // 'sprintf' : This function or variable may be unsafe. Consider using sprintf_s instead. To disable  
// deprecation,  
32 //use _CRT_SECURE_NO_WARNINGS  
33 //To overcome above warning  
34 #ifdef _MSC_VER  
35 #pragma warning(disable: 4996) /* Disable deprecation */  
36 #endif  
37  
38  
39 /*-----  
40 STL  
41 -----*/  
42 #include <stdexcept> //Without this catch will NOT work on Linux  
43 #include <vector>  
44  
45  
46  
47  
48 #endif  
49  
50 //EOF  
51
```

1.4. FIRST C++ PROGRAM REVISTED

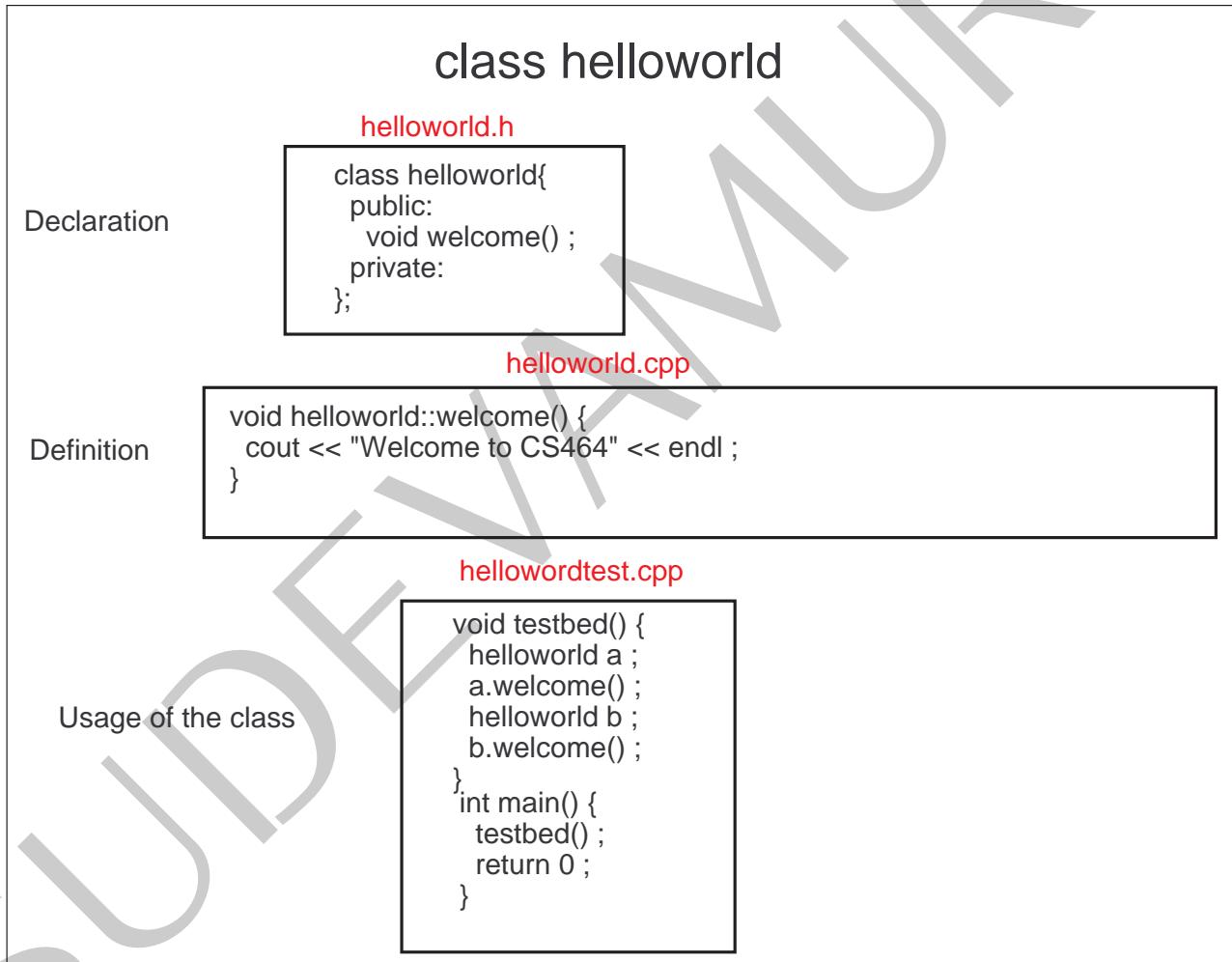
util.h

```
#ifndef UTIL_H
#define UTIL_H
    #include <iostream>
    #ifdef WIN32
    #include <fstream>
    #include <cassert>
    #include <ctime>
    #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector
    #else
    #include <fstream>
    #include <assert.h>
    #include <time.h>
    #endif
    #include <iomanip> // std::setprecision
    using namespace std;
#endif
//sprintf' : This function or variable may be unsafe. Consider using sprintf_s instead. To disable deprecation,
//use _CRT_SECURE_NO_WARNINGS
//To overcome above warning

#ifndef _MSC_VER
#pragma warning(disable: 4996) /* Disable deprecation */
#endif
```

Figure 1.3: *util.h*

1.4.2 class helloworld

Figure 1.4: *class helloworld*

1.4.3 C++code for *class helloworld*

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: helloworld.h  
4 -----*/  
5  
6 /*-----  
7 This file has helloworld class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef HELLOWORLD_H  
14 #define HELLOWORLD_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of helloworld class  
20 -----*/  
21 class helloworld{  
22     public:  
23         void welcome() ;  
24     private:  
25  
26 };  
27  
28 #endif  
29  
30
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: helloworld.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has helloworld class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "helloworld.h"  
14  
15 /*-----  
16 Definition of welcome routine of helloworld class  
17 -----*/  
18 void helloworld::welcome() {  
19     cout << "Welcome to CS464" << endl ;  
20 }  
21  
22 //EOF  
23  
24  
25
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: helloworldtest.cpp  
4  
5 On linux:  
6 g++ helloworld.cpp helloworldtest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test helloworld object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "helloworld.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     helloworld a ;  
25     a.welcome() ;  
26     helloworld b ;  
27     b.welcome() ;  
28 }  
29  
30 /*-----  
31 main  
32 -----*/  
33 int main() {  
34     testbed() ;  
35     return 0 ;  
36 }  
37  
38 //EOF  
39  
40
```

1.5 Basic data types

1.5.1 Boolean number system

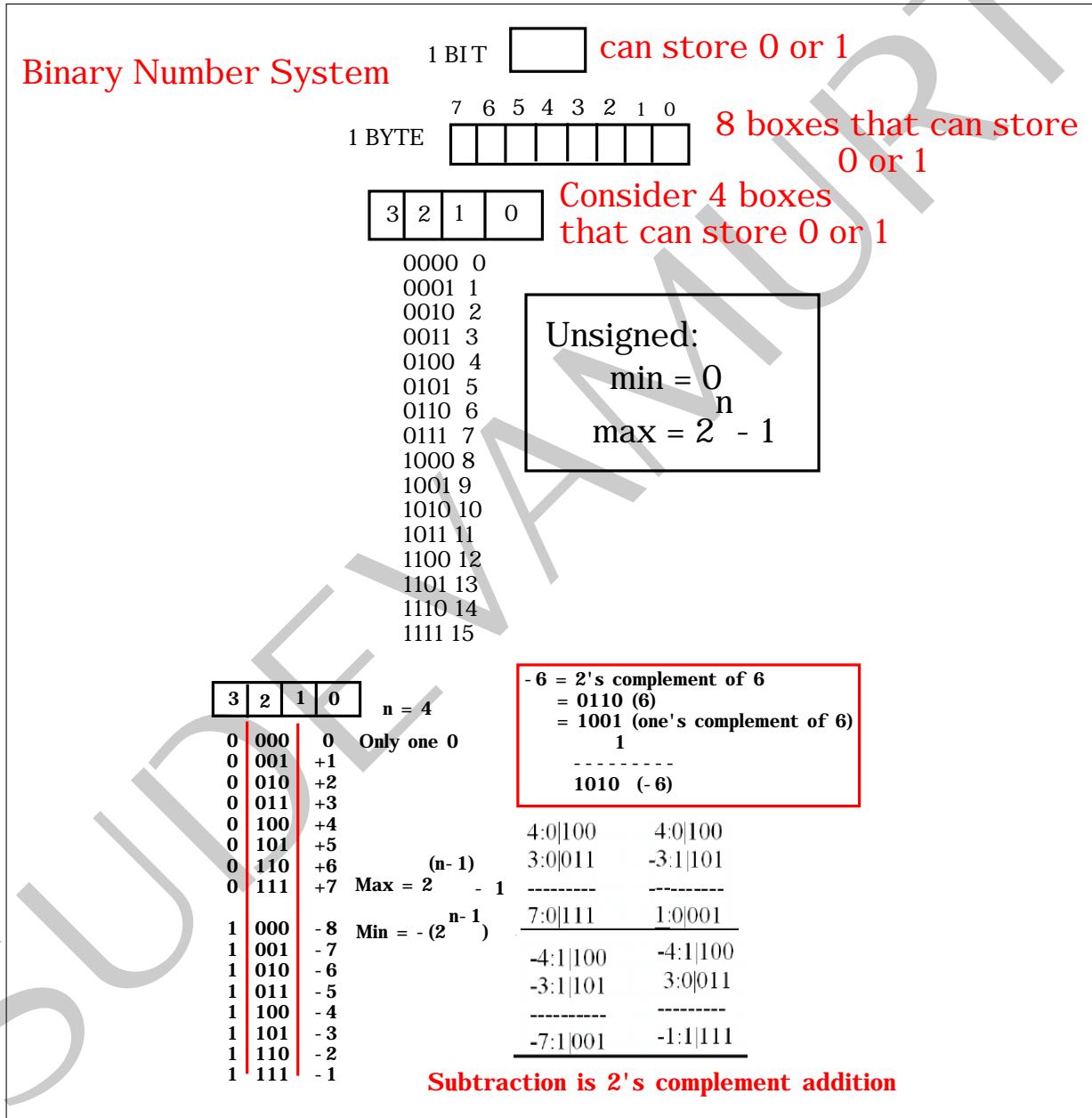


Figure 1.5: Boolean numbers

1.5. BASIC DATA TYPES

1.5.2 bool data type

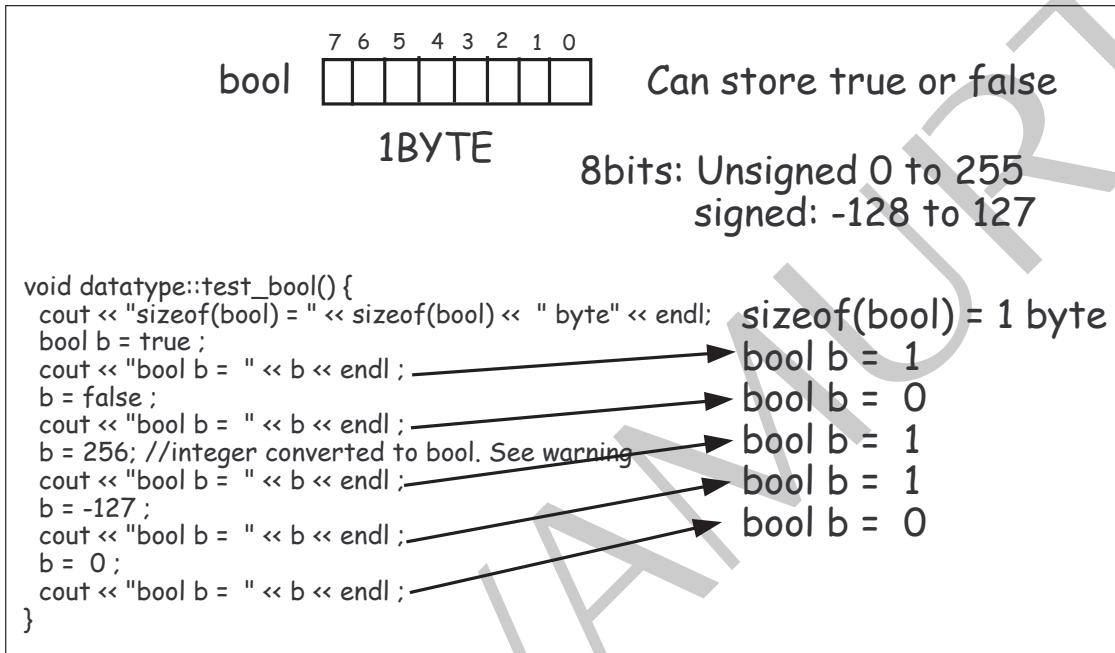


Figure 1.6: **bool** data type

1.5.3 char data type

CHAPTER 1. INTRODUCTION

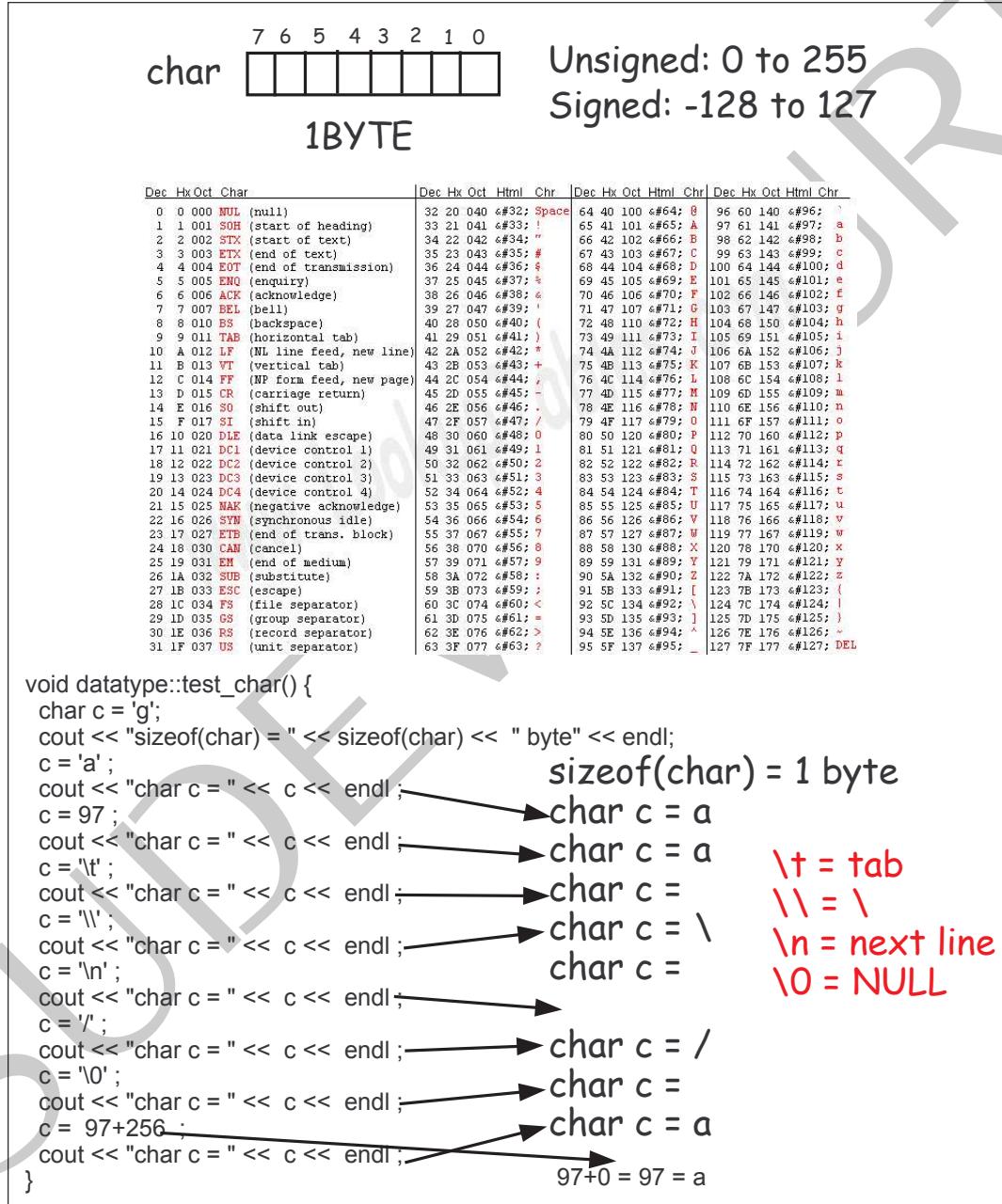


Figure 1.7: `char` data type

1.5. BASIC DATA TYPES

1.5.4 int data type

CHAPTER 1. INTRODUCTION

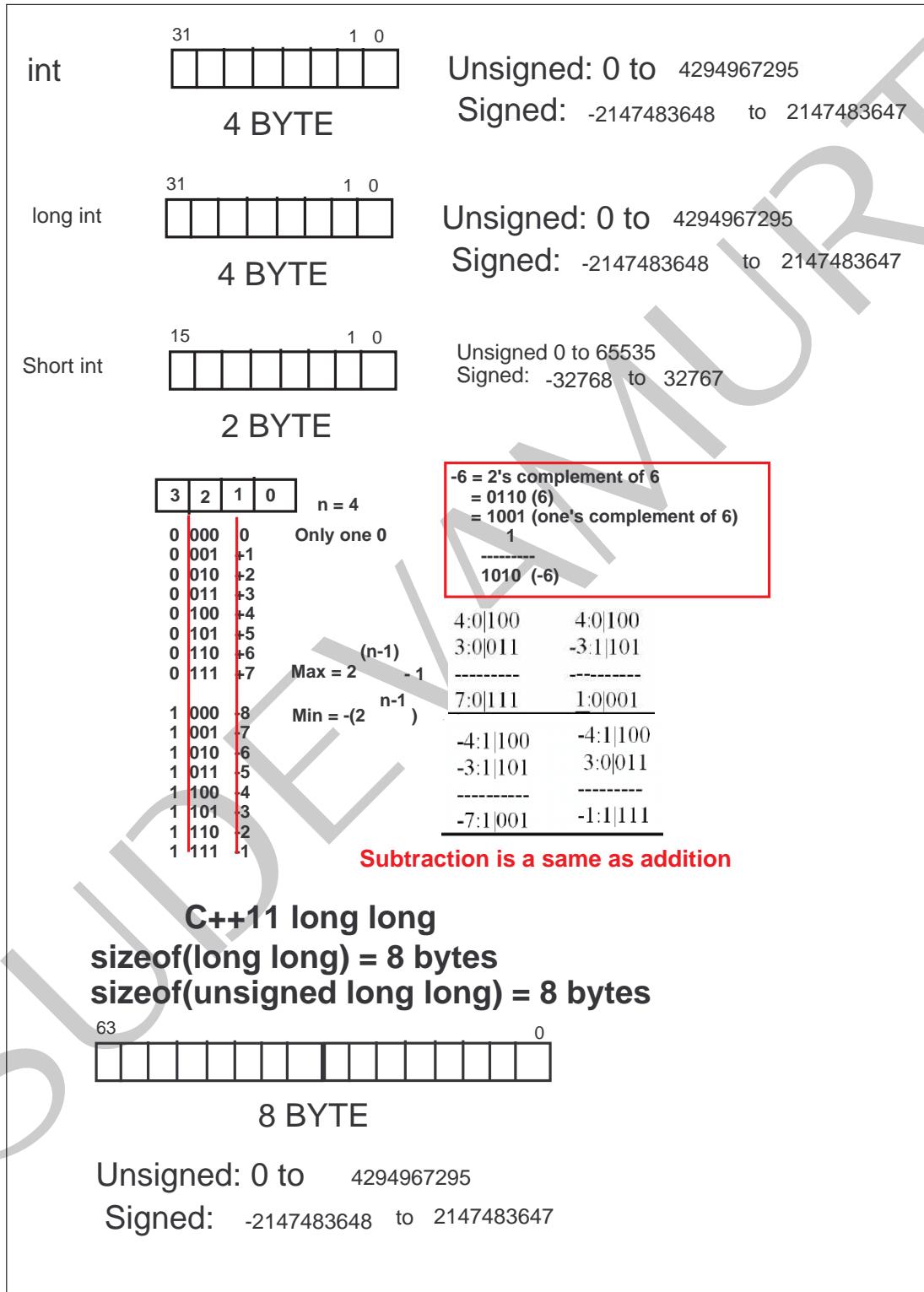


Figure 1.8: Maximum and minimum number that can be represented in an integer

1.5. BASIC DATA TYPES

```

void datatype::test_int() {
    cout << "sizeof(unsigned short int) = " << sizeof(unsigned short int) << " bytes" << endl;
    cout << "sizeof(unsigned int) = " << sizeof(unsigned int) << " bytes" << endl;
    cout << "sizeof(unsigned long int) = " << sizeof(unsigned long int) << " bytes" << endl;
    cout << "sizeof(short int) = " << sizeof(short int) << " bytes" << endl;
    cout << "sizeof(int) = " << sizeof(int) << " bytes" << endl;
    cout << "sizeof(long int) = " << sizeof(long int) << " bytes" << endl;

    unsigned short short_unsigned_max = 0xFFFF ;
    cout << "short_unsigned_max= " << short_unsigned_max << endl ;
    unsigned short short_unsigned_min = 0 ;
    cout << "short_unsigned_min = " << short_unsigned_min << endl ;

    short int short_int_max = 0x7FFF ;
    cout << "short_int_max = " << short_int_max << endl ;
    short int short_int_min = 0x8000 ;
    cout << "short_int_min = " << short_int_min << endl ;

    unsigned int unsigned_int_max = 0xFFFFFFFF ;
    cout << "unsigned_int_max= " << unsigned_int_max << endl ;
    unsigned int unsigned_int_min = 0 ;
    cout << "unsigned_int_min = " << unsigned_int_min << endl ;

    int int_max = 0xFFFFFFFF ;
    cout << "int_max = " << int_max << endl ;
    int int_min = 0x80000000 ;
    cout << "int_max = " << int_min << endl ;

    int i = 2147483648 ;
    cout << "int i = " << i << endl ;           int i = -2147483648

    unsigned long unsigned_long_max = 0xFFFFFFFF ;
    cout << "unsigned_long_max= " << unsigned_long_max << endl ;
    unsigned long unsigned_long_min = 0 ;
    cout << "unsigned_long_min = " << unsigned_long_min << endl ;

    long long_max = 0x7FFFFFFF ;
    cout << "long_max = " << long_max << endl ;
    long long_min = 0x80000000 ;
    cout << "long_min = " << long_min << endl ;

    long long max = 0x7FFFFFFF;
    unsigned long long umax = 0xFFFFFFFF;
}

```

sizeof(unsigned short int) = 2 bytes
 sizeof(unsigned int) = 4 bytes
 sizeof(unsigned long int) = 4 bytes
 sizeof(short int) = 2 bytes
 sizeof(int) = 4 bytes
 sizeof(long int) = 4 bytes

short_unsigned_max= 65535
 short_unsigned_min = 0

short_int_max = 32767
 short_int_min = -32768

unsigned_int_max= 4294967295
 unsigned_int_min = 0

int_max = 2147483647
 int_max = -2147483648

unsigned_long_max= 4294967295
 unsigned_long_min = 0

long_max = 2147483647
 long_min = -2147483648

long long max = 2147483647
 long long min = -2147483648
 unsigned long long max = 4294967295

Figure 1.9: **int** data type

1.5.5 float and double data type

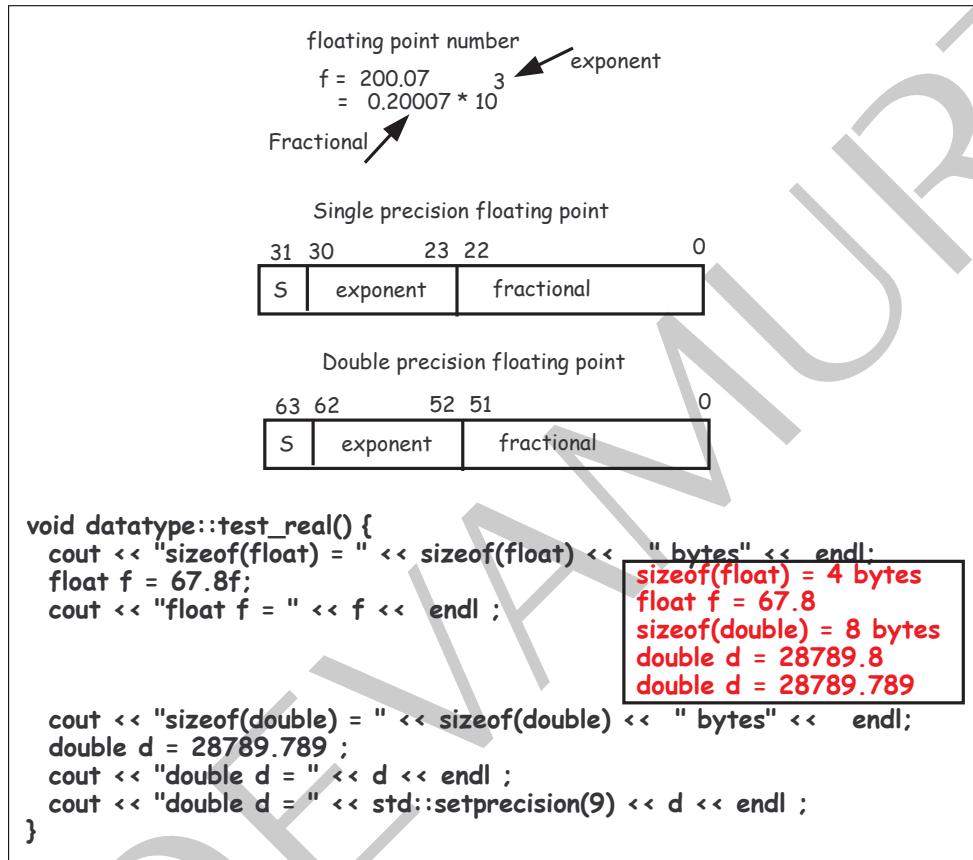


Figure 1.10: float and double data type

1.5.6 C++ code for illustrating basic data types

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: datatype.h  
4 -----*/  
5  
6 /*-----  
7 This file has datatype class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef DATATYPE_H  
14 #define DATATYPE_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of datatype class  
20 -----*/  
21 class datatype{  
22 public:  
23     void test_bool() ;  
24     void test_char() ;  
25     void test_int() ;  
26     void test_real() ;  
27     void test_long_long(); //C++11  
28 private:  
29  
30 };  
31  
32 #endif  
33  
34
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: datatype.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has datatype class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "datatype.h"  
14  
15 /*-----  
16 Definition of routines of datatype class  
17 -----*/  
18  
19 /*-----  
20 sizeof(bool) = 1 byte  
21 bool b = 1  
22 bool b = 0  
23 bool b = 1  
24 bool b = 1  
25 bool b = 0  
26 -----*/  
27 void datatype::test_bool() {  
28     cout << "sizeof(bool) = " << sizeof(bool) << " byte" << endl;  
29     bool b = true ;  
30     cout << "bool b = " << b << endl ;  
31     b = false ;  
32     cout << "bool b = " << b << endl ;  
33     b = 256; //integer converted to bool. See warning  
34 //To prove any number other than 0 is true(1)  
35     cout << "bool b = " << b << endl ;  
36     b = -127 ;  
37     cout << "bool b = " << b << endl ;  
38     b = 0 ;  
39     cout << "bool b = " << b << endl ;  
40     b = -127 ;  
41     cout << "bool b = " << b << endl ;  
42 }  
43  
44 /*-----  
45 sizeof(char) = 1 byte  
46 char c = a  
47 char c = a  
48 char c =  
49 char c = \  
50 char c =  
51  
52 char c = /  
53 char c =  
54 char c = a  
55 -----*/  
56 void datatype::test_char() {  
57     char c = 'g';  
58     cout << "sizeof(char) = " << sizeof(char) << " byte" << endl;  
59     c = 'a' ;  
60     cout << "char c = " << c << endl ;  
61     c = 97 ;  
62     cout << "char c = " << c << endl ;  
63     c = '\t' ;  
64     cout << "char c = " << c << endl ;  
65     c = '\\';  
66     cout << "char c = " << c << endl ;
```

```
67 c = '\n' ;
68 cout << "char c = " << c << endl ;
69 c = '/';
70 cout << "char c = " << c << endl ;
71 c = '\0' ;
72 cout << "char c = " << c << endl ;
73 c = 97+256 ;
74 cout << "char c = " << c << endl ;
75 }
76
77 /*-----
78 sizeof(unsigned short int) = 2 bytes
79 sizeof(unsigned int) = 4 bytes
80 sizeof(unsigned long int) = 4 bytes
81 sizeof(short int) = 2 bytes
82 sizeof(int) = 4 bytes
83 sizeof(long int) = 4 bytes
84 short_unsigned_max= 65535
85 short_unsigned_min = 0
86 short_int_max = 32767
87 short_int_min = -32768
88 unsigned_int_max= 4294967295
89 unsigned_int_min = 0
90 int_max = 2147483647
91 int_max = -2147483648
92 int i = -2147483648
93 unsigned_long_max= 4294967295
94 unsigned_long_min = 0
95 long_max = 2147483647
96 long_min = -2147483648
97 -----*/
98 void datatype::test_int() {
99 cout << "sizeof(unsigned short int) = " << sizeof(unsigned short int) << " bytes" << endl;
100 cout << "sizeof(unsigned int) = " << sizeof(unsigned int) << " bytes" << endl;
101 cout << "sizeof(unsigned long int) = " << sizeof(unsigned long int) << " bytes" << endl;
102 cout << "sizeof(short int) = " << sizeof(short int) << " bytes" << endl;
103 cout << "sizeof(int) = " << sizeof(int) << " bytes" << endl;
104 cout << "sizeof(long int) = " << sizeof(long int) << " bytes" << endl;
105
106 unsigned short short_unsigned_max = 0xFFFF ;
107 cout << "short_unsigned_max= " << short_unsigned_max << endl ;
108 unsigned short short_unsigned_min = 0 ;
109 cout << "short_unsigned_min = " << short_unsigned_min << endl ;
110
111 short int short_int_max = 0x7FFF ;
112 cout << "short_int_max = " << short_int_max << endl ;
113 short int short_int_min = 0x8000 ;
114 cout << "short_int_min = " << short_int_min << endl ;
115
116 unsigned int unsigned_int_max = 0xFFFFFFFF ;
117 cout << "unsigned_int_max= " << unsigned_int_max << endl ;
118 unsigned int unsigned_int_min = 0 ;
119 cout << "unsigned_int_min = " << unsigned_int_min << endl ;
120
121 int int_max = 0x7FFFFFFF ;
122 cout << "int_max = " << int_max << endl ;
123 int int_min = 0x80000000 ;
124 cout << "int_max = " << int_min << endl ;
125
126 int i = 2147483648 ;
127 cout << "int i = " << i << endl ;
128
129 unsigned long unsigned_long_max = 0xFFFFFFFF ;
130 cout << "unsigned_long_max= " << unsigned_long_max << endl ;
131 unsigned long unsigned_long_min = 0 ;
132 cout << "unsigned_long_min = " << unsigned_long_min << endl ;
```

```
133 long long_max = 0xFFFFFFFF ;
134 cout << "long_max = " << long_max << endl ;
135 long long_min = 0x80000000 ;
136 cout << "long_min = " << long_min << endl ;
137 }
138 }
139 */
140 /**
141 sizeof(float) = 4 bytes
142 float f = 67.8
143 sizeof(double) = 8 bytes
144 double d = 28789.8
145 double d = 28789.789
146 */
147 void datatype::test_real() {
148     cout << "sizeof(float) = " << sizeof(float) << " bytes" << endl;
149     float f = 67.8f;
150     cout << "float f = " << f << endl ;
151
152     cout << "sizeof(double) = " << sizeof(double) << " bytes" << endl;
153     double d = 28789.789 ;
154     cout << "double d = " << d << endl ;
155     cout << "double d = " << std::setprecision(9) << d << endl ;
156 }
157 */
158 /**
159 sizeof(long long) = 8 bytes
160 sizeof(unsigned long long) = 8 bytes
161 unsigned long long is the same as unsigned long long int.
162 Its size is platform-dependent, but guaranteed
163 to be at least 64 bits
164 */
165 void datatype::test_long_long(){
166     cout << "sizeof(long long) = " << sizeof(long long) << " bytes" << endl;
167     cout << "sizeof(unsigned long long) = " << sizeof(unsigned long long) << " bytes" << endl;
168     long long x = 10000000000;
169     cout << "x = " << x << endl;
170     long long y = 10000000000LL;
171     cout << "y = " << y << endl;
172     //long long z = 0xFFFFFFFF;
173     long long z = 0x7FFFFFFFFFFFFFFF;
174     cout << "z = " << z << endl;
175     long long z1 = z + 1;
176     cout << "z1 = " << z1 << endl;
177     unsigned long long uz = 0xFFFFFFFFFFFFFFFFFFFF;
178     cout << "uz = " << uz << endl;
179     unsigned long long uz1 = uz + 1;
180     cout << "uz1 = " << uz1 << endl;
181     unsigned long long uz2 = uz + 5;
182     cout << "uz2 = " << uz2 << endl;
183 }
184 */
185 //EOF
186 }
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: datatypeptest.cpp  
4  
5 On linux:  
6 g++ datatype.cpp datatypeptest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test datatype object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "datatype.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     datatype a ;  
25     a.test_bool() ;  
26     a.test_char() ;  
27     a.test_int() ;  
28     a.test_real() ;  
29     a.test_long_long();  
30 }  
31  
32 /*-----  
33 main  
34 -----*/  
35 int main() {  
36     testbed() ;  
37     return 0 ;  
38 }  
39  
40 //EOF  
41
```

1.6 User defined data types

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: uobject.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7 //sprintf' : This function or variable may be unsafe.Consider using sprintf_s instead.To disable  
8 //deprecation,  
9 //use _CRT_SECURE_NO_WARNINGS  
10 //To overcome above warning  
11 #ifndef _MSC_VER  
12 #pragma warning(disable: 4996) /* Disable deprecation */  
13 #endif  
14  
15 struct book {  
16     char name[20] ;  
17     int cost ;  
18     bool in ;  
19 };  
20  
21 /*-----  
22 b1: Name = Algorithms Cost = 23 In  
23 -----*/  
24 int main() {  
25     book b1;  
26     strcpy(b1.name,"Algorithms" );  
27     b1.cost = 23;  
28     b1.in = true ;  
29     cout << "b1: Name = " << b1.name << " Cost = " << b1.cost ;  
30     (b1.in) ? cout << " In " : cout << " Out " ;  
31     cout << endl ;  
32     return 0 ;  
33 }  
34
```

1.7 Arithmetic operations

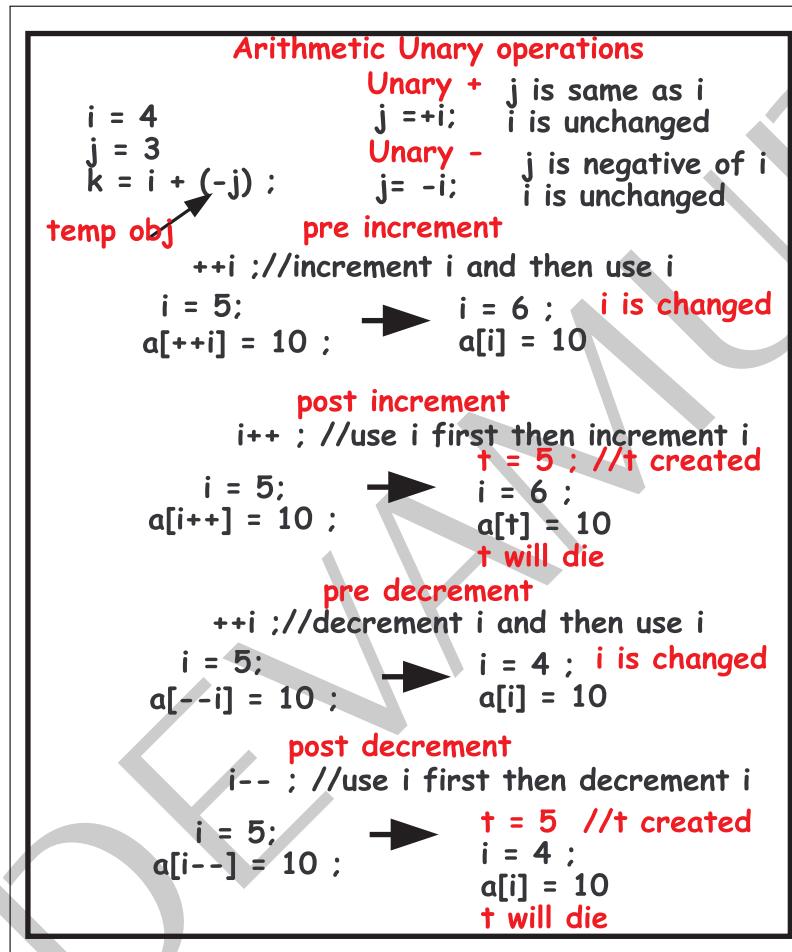


Figure 1.11: Arithmetic unary operators

1.7. ARITHMETIC OPERATIONS

1.7.1 C++ code for illustrating arithmetic operations

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: arithmetic.h  
4 -----*/  
5  
6 /*-----  
7 This file has arithmetic class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef ARITHMETIC_H  
14 #define ARITHMETIC_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of arithmetic class  
20 -----*/  
21 class arithmetic{  
22 public:  
23     void test_unary();  
24     void test_increment();  
25     void test_decrement();  
26     void test_int();  
27     void test_float();  
28     void test_double();  
29  
30 private:  
31  
32 };  
33  
34 #endif  
35 //EOF  
36
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: arithmetic.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has arithmetic class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "arithmetic.h"  
14  
15 /*-----  
16 Definition of routines of arithmetic class  
17 -----*/  
18  
19 /*-----  
20 a = -10  
21 t = 10  
22 Note that a is not changed after the operation -a  
23 After -a, a = -10  
24 -t = -10 and t = 10  
25 -----*/  
26 void arithmetic::test_unary() {  
27     int a = -10 ;  
28     cout << "a = " << a << endl ;  
29     int t = -a ;  
30     cout << "t = " << t << endl ;  
31     cout << "Note that a is not changed after the operation -a\n" ;  
32     cout << "After -a, a = " << a << endl ;  
33     cout << "-t = " << -t << " and t = " << t << endl ;  
34 }  
35  
36 /*-----  
37 1. c = 5  
38 2. c = 6  
39 3. c = 6  
40 4. c = 7  
41 5. c = 8  
42 6. c = 8  
43 -----*/  
44 void arithmetic::test_increment() {  
45     int c = 5 ;  
46     cout << " 1. c = " << c << endl ;  
47     c++ ;  
48     cout << " 2. c = " << c << endl ;  
49     cout << " 3. c = " << c++ << endl ;  
50     cout << " 4. c = " << c << endl ;  
51     cout << " 5. c = " << ++c << endl ;  
52     cout << " 6. c = " << c << endl ;  
53 }  
54  
55 /*-----  
56 1. c = 5  
57 2. c = 4  
58 3. c = 4  
59 4. c = 3  
60 5. c = 2  
61 6. c = 2  
62 -----*/  
63 void arithmetic::test_decrement() {  
64     int c = 5 ;  
65     cout << " 1. c = " << c << endl ;  
66     c-- ;
```

```
67 cout << " 2. c = " << c << endl ;
68 cout << " 3. c = " << c-- << endl ;
69 cout << " 4. c = " << c << endl ;
70 cout << " 5. c = " << --c << endl ;
71 cout << " 6. c = " << c << endl ;
72 }
73
74 /*-
75 a = 10 b = 3
76 1. a+b = 13
77 2. a-b = 7
78 3. a*b = 30
79 4. a/b = 3
80 5. a%b = 1
81 a = 10 b = 3
82 7. c = 13
83 8. c = 39
84 9. c = 36
85 10. c = 108
86 -----*/
87 void arithmetic::test_int() {
88     int a = 10 ;
89     int b = 3 ;
90     cout << "a = " << a << " b = " << b << endl ;
91     cout << " 1. a+b = " << a+b << endl ;
92     cout << " 2. a-b = " << a-b << endl ;
93     cout << " 3. a*b = " << a*b << endl ;
94     cout << " 4. a/b = " << a/b << endl ;
95     cout << " 5. a%b = " << a%b << endl ;
96     cout << "a = " << a << " b = " << b << endl ;
97     int c = 10 ;
98     c += 3 ;
99     cout << " 7. c = " << c << endl ;
100    c *= 3 ;
101    cout << " 8. c = " << c << endl ;
102    c -= 3 ;
103    cout << " 9. c = " << c << endl ;
104    c *= 3 ;
105    cout << " 10. c = " << c << endl ;
106 }
107
108 /*-
109 a = 10.3 b = 3.21
110 1. a+b = 13.51
111 2. a-b = 7.09
112 3. a*b = 33.063
113 4. a/b = 3.20872
114 -----*/
115 void arithmetic::test_float() {
116     float a = 10.3f ;
117     float b = 3.21f ;
118     cout << "a = " << a << " b = " << b << endl ;
119     cout << " 1. a+b = " << a+b << endl ;
120     cout << " 2. a-b = " << a-b << endl ;
121     cout << " 3. a*b = " << a*b << endl ;
122     cout << " 4. a/b = " << a/b << endl ;
123 //cout << " 5. a%b = " << a%b << endl ;
124 // error C2296: '%' : illegal, left operand has type 'float'
125 }
126
127 /*-
128 a = 10.3 b = 3.21
129 1. a+b = 13.51
130 2. a-b = 7.09
131 3. a*b = 33.063
132 4. a/b = 3.20872
```

```
133 -----*/
134 void arithmetic::test_double() {
135     double a = 10.3 ;
136     double b = 3.21 ;
137     cout << "a = " << a << " b = " << b << endl ;
138     cout << " 1. a+b = " << a+b << endl ;
139     cout << " 2. a-b = " << a-b << endl ;
140     cout << " 3. a*b = " << a*b << endl ;
141     cout << " 4. a/b = " << a/b << endl ;
142 //cout << " 5. a%b = " << a%b << endl ;
143 // error C2296: '%' : illegal, left operand has type 'float'
144 }
145
146
147 //EOF
148
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: arithmetictest.cpp  
4  
5 On linux:  
6 g++ arithmetic.cpp arithmetictest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test arithmetic object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "arithmetic.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     arithmetic a ;  
25     a.test_unary() ;  
26     a.test_increment();  
27     a.test_decrement();  
28     a.test_int();  
29     a.test_float();  
30     a.test_double();  
31 }  
32  
33 /*-----  
34 main  
35 -----*/  
36 int main() {  
37     testbed() ;  
38     return 0 ;  
39 }  
40  
41 //EOF  
42
```

1.8. RELATIONAL OPERATIONS

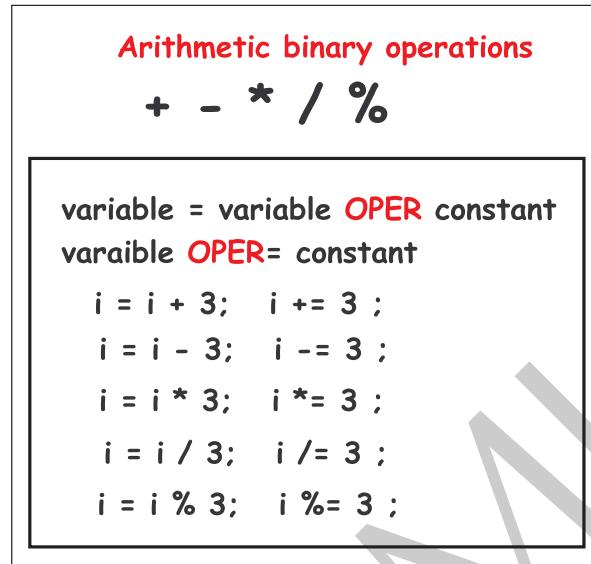


Figure 1.12: Arithmetic binary operators

1.8 Relational operations

1.8.1 C++ code for illustrating relational operations

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: relational.h  
4 -----*/  
5  
6 /*-----  
7 This file has relational class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef RELATIONAL_H  
14 #define RELATIONAL_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of relational class  
20 -----*/  
21 class relational{  
22 public:  
23     void test_relational();  
24     void test_ternery();  
25  
26 private:  
27  
28 };  
29  
30 #endif  
31 //EOF  
32
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: relational.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has relational class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "relational.h"  
14  
15 /*-----  
16 Definition of routines of relational class  
17 -----*/  
18  
19 /*-----  
20 a = 7 b = 12  
21 1. a < b = true  
22 2. a <= b = true  
23 3. a > b = false  
24 4. a >= b = false  
25 5. a == b = false  
26 6. a != b = true  
27 -----*/  
28 void relational::test_relational() {  
29     int a = 7;  
30     int b = 12 ;  
31     cout << boolalpha ; //displays boolean as true or false  
32     cout << "a = " << a << " b = " << b << endl ;  
33     cout << "1. a < b = " << (a < b) << endl ;  
34     cout << "2. a <= b = " << (a <= b) << endl ;  
35     cout << "3. a > b = " << (a > b) << endl ;  
36     cout << "4. a >= b = " << (a >= b) << endl ;  
37     cout << "5. a == b = " << (a == b) << endl ;  
38     cout << "6. a != b = " << (a != b) << endl ;  
39     cout.unsetf(ios::boolalpha) ; //unset boolalpha  
40 }  
41  
42 /*-----  
43 a = 10 b = 20  
44 1. c = 77  
45 a = 20 b = 10  
46 2. c = 5  
47 -----*/  
48 void relational::test_ternary() {  
49     int a = 10 ;  
50     int b = 20 ;  
51     cout << "a = " << a << " b = " << b << endl ;  
52     int c = (a < b) ? (a + 67) : (b -5) ;  
53     cout << "1. c = " << c << endl ;  
54     a = 20 ;  
55     b = 10 ;  
56     cout << "a = " << a << " b = " << b << endl ;  
57     c = (a < b) ? (a + 67) : (b -5) ;  
58     cout << "2. c = " << c << endl ;  
59 }  
60  
61  
62 //EOF  
63  
64
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: relationaltest.cpp  
4  
5 On linux:  
6 g++ relational.cpp relationaltest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test relational object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "relational.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     relational a ;  
25     a.test_relational();  
26     a.test_ternery();  
27 }  
28  
29 /*-----  
30 main  
31 -----*/  
32 int main() {  
33     testbed() ;  
34     return 0 ;  
35 }  
36  
37 //EOF  
38
```

1.9 Logical operations

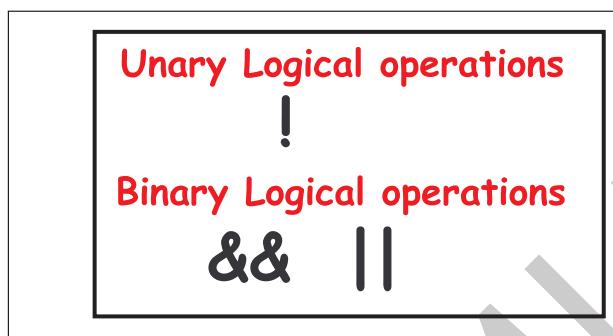


Figure 1.14: Logical operations

1.9.1 C++ code for illustrating logical operations

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: logical.h  
4 -----*/  
5  
6 /*-----  
7 This file has logical class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef logical_H  
14 #define logical_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of logical class  
20 -----*/  
21 class logical{  
22 public:  
23     void test_logical();  
24     void test_comma();  
25  
26 private:  
27  
28 };  
29  
30 #endif  
31 //EOF  
32
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: logical.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has logical class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "logical.h"  
14  
15 /*-----  
16 Definition of routines of logical class  
17 -----*/  
18  
19 /*-----  
20 a = 7 b = 12 c = 5 d = 0  
21 1. (a < b) && (b > c) = true  
22 2. (a < b) && (b < c) = false  
23 3. (a < b) || (b < c) = true  
24 4. (a > b) || (b < c) = false  
25 5. (!a) = false  
26 6. (d) = 0  
27 7. (d) = false  
28 8. (!d) = true  
29 Note that d is not changed after !d. d = 0  
30 -----*/  
31 void logical::test_logical() {  
32     int a = 7;  
33     int b = 12 ;  
34     int c = 5 ;  
35     int d = 0 ;  
36     cout << boolalpha ; //displays boolean as true or false  
37     cout << "a = " << a << " b = " << b << " c = " << c << " d = " << d << endl ;  
38     cout << "1. (a < b) && (b > c) = " << ((a < b) && (b > c)) << endl ;  
39     cout << "2. (a < b) && (b < c) = " << ((a < b) && (b < c)) << endl ;  
40     cout << "3. (a < b) || (b < c) = " << ((a < b) || (b < c)) << endl ;  
41     cout << "4. (a > b) || (b < c) = " << ((a > b) || (b < c)) << endl ;  
42     cout << "5. (!a) = " << (!a) << endl ;  
43     cout << "6. (d) = " << (d) << endl ;  
44     cout << "7. (d) = " << bool(d) << endl ;  
45     cout << "8. (!d) = " << (!d) << endl ;  
46     cout << "Note that d is not changed after !d. d = " << d << endl ;  
47     cout.unsetf(ios::boolalpha) ; //unset boolalpha  
48 }  
49  
50 /*-----  
51 a = 7 b = 12  
52 1. k=(a,b) = 12 //Evaluated from left to right;final 12 is assigned to k  
53 2. k=(b,a) = 7  
54 -----*/  
55 void logical::test_comma() {  
56     int a = 7;  
57     int b = 12 ;  
58     cout << "a = " << a << " b = " << b << endl ;  
59     int k = (a,b) ;  
60     cout << "1. k=(a,b) = " << k << endl ;  
61     k = (b,a) ;  
62     cout << "2. k=(b,a) = " << k << endl ;  
63 }  
64  
65 //EOF  
66
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: logicaltest.cpp  
4  
5 On linux:  
6 g++ logical.cpp logicaltest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test logical object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "logical.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     logical a ;  
25     a.test_logical();  
26     a.test_comma();  
27 }  
28  
29 /*-----  
30 main  
31 -----*/  
32 int main() {  
33     testbed() ;  
34     return 0 ;  
35 }  
36  
37 //EOF  
38  
39
```

Relational operations

< <= > >= == !=

Ternary operation

```
if (s)
    f = a ; f = (s) ? a : b
else
    f = b ;
```

Figure 1.13: Relational operations

1.10 Bitwise operations

Bitwise Unary operations

$\sim j$ Unary bitwise complement

j will not be changed. The answer on the RHS will be changed

```
int a = 0 ; //a is 0
int t = ~a ; // t is ffffffff
//a is 0 at this point
```

Bitwise Binary operations

$a \& b$	Bitwise AND
$a b$	Bitwise OR
$a ^ b$	Bitwise XOR

Bitwise Shift operators

$a << k$ Left shift a , k times. Fill lost space by 0

$a >> k$ Right shift a , k times. Lost space on MSB
is 0 or 1(arithmetic shift)

```
int a = 1 ; //a is 1
int t = a << 3 ; //t is 8
//a is 1 at this point
```

$int a = -3 ;$
3 is 011. 2' complement is: 100

1	

101	
1111 1111 1111 1111 1111 1111 1111 1101	
Right shift by 1	
1111 1111 1111 1111 1111 1111 1111 1110	

Figure 1.15: Bitwise operations

1.10.1 C++ code for illustrating bitwise operations

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: bitwise.h  
4 -----*/  
5  
6 /*-----  
7 This file has bitwise class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef bitwise_H  
14 #define bitwise_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of bitwise class  
20 -----*/  
21 class bitwise{  
22 public:  
23     void test_bitwise();  
24     void test_shift();  
25  
26 private:  
27  
28 };  
29  
30 #endif  
31 //EOF  
32
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: bitwise.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has bitwise class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "bitwise.h"  
14  
15 /*-----  
16 Definition of routines of bitwise class  
17 -----*/  
18  
19 /*-----  
20 All answers are in hexadecimal  
21 1. a = 0  
22 2. ~a = ffffffff  
23 Note that a NOT is changed,. Only the RHS is changed  
24 2a. After ~a, a = 0  
25 3. a = 1  
26 4. b = 2  
27 5. (a & b) = 0  
28 6. (a ! b) = 3  
29 7. (a ^ b) = 3  
30 8. (a & c) = 1  
31 9. (a ! c) = 3  
32 10. (a ^ c) = 2  
33 -----*/  
34 void bitwise::test_bitwise() {  
35     int a = 0 ;  
36     cout << hex ;  
37     cout << "All answers are in hexadecimal" << endl ;  
38     cout << "1. a = " << a << endl ;  
39     cout << "2. ~a = " << ~a << endl ;  
40     cout << "Note that a NOT is changed,. Only the RHS is changed\n" ;  
41     cout << "2a. After ~a, a = " << a << endl ;  
42     a = 1 ;  
43     int b = 2 ;  
44     cout << "3. a = " << a << endl ;  
45     cout << "4. b = " << b << endl ;  
46     cout << "5. (a & b) = " << (a & b) << endl ;  
47     cout << "6. (a ! b) = " << (a | b) << endl ;  
48     cout << "7. (a ^ b) = " << (a ^ b) << endl ;  
49     int c = 3 ;  
50     cout << "8. (a & c) = " << (a & c) << endl ;  
51     cout << "9. (a ! c) = " << (a | c) << endl ;  
52     cout << "10. (a ^ c) = " << (a ^ c) << endl ;  
53     cout.unset(ios::hex);  
54 }  
55  
56 /*-----  
57 BEGIN a = 1  
58 All answers are in hexadecimal  
59 Note that a << 3 will NOT effect a. The result will be shifted by 3  
60 a = 1  
61 a << 3 = 8  
62 after shifting a = 1  
63 shift a = 1 left by one = 2  
64 shift a = 2 left by one = 4  
65 shift a = 4 left by one = 8  
66 shift a = 8 left by one = 10
```

```
67 shift a = 10 left by one = 20
68 -----
69 shift a = 20 right by one = 10
70 shift a = 10 right by one = 8
71 shift a = 8 right by one = 4
72 shift a = 4 right by one = 2
73 shift a = 2 right by one = 1
74 shift a = ffffffff right by one = ffffffe
75 -----
76 void bitwise::test_shift() {
77     int a = 1 ;
78     cout << "BEGIN a = " << a << endl ;
79
80     cout << hex ;
81     cout << "All answers are in hexadecimal" << endl ;
82     cout << "Note that a << 3 will NOT effect a. The result will be shifted by 3\n" ;
83     cout << "a = " << a << endl ;
84     cout << "a << 3 = " << (a << 3) << endl ;
85     cout << "after shifting a = " << a << endl ;
86     for (int i = 0 ; i < 5 ; i++) {
87         cout << "shift a = " << a << " left by one = " ;
88         cout << (a = a << 1) << endl ;
89     }
90     cout << "-----" << endl ;
91     for (int i = 0 ; i < 5 ; i++) {
92         cout << "shift a = " << a << " right by one = " ;
93         cout << (a = a >> 1) << endl ;
94     }
95     a = -3 ;
96     cout << "shift a = " << a << " right by one = " ;
97     cout << (a = a >> 1) << endl ;
98     cout.unsetf(ios::hex);
99 }
100
101
102
103
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: bitwisetest.cpp  
4  
5 On linux:  
6 g++ bitwise.cpp bitwisetest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test bitwise object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "bitwise.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     bitwise a ;  
25     a.test_bitwise();  
26     a.test_shift();  
27 }  
28  
29 /*-----  
30 main  
31 -----*/  
32 int main() {  
33     testbed() ;  
34     return 0 ;  
35 }  
36  
37 //EOF  
38
```

1.11. ASSIGNMENT OPERATIONS

1.11 Assignment operations

Assignment	
=	a = b
Additional assignment	
+=	a+=b a = a + b
-=	a-=b a = a - b
=	a=b a = a * b
/=	a/=b a = a / b
%=	a%=b a = a % b
Bitwise assignment	
&=	a&=b a = a & b
!=	a!=b a = a b
^=	a^=b a = a ^ b

Figure 1.16: Assignment operations

1.12 Control statements

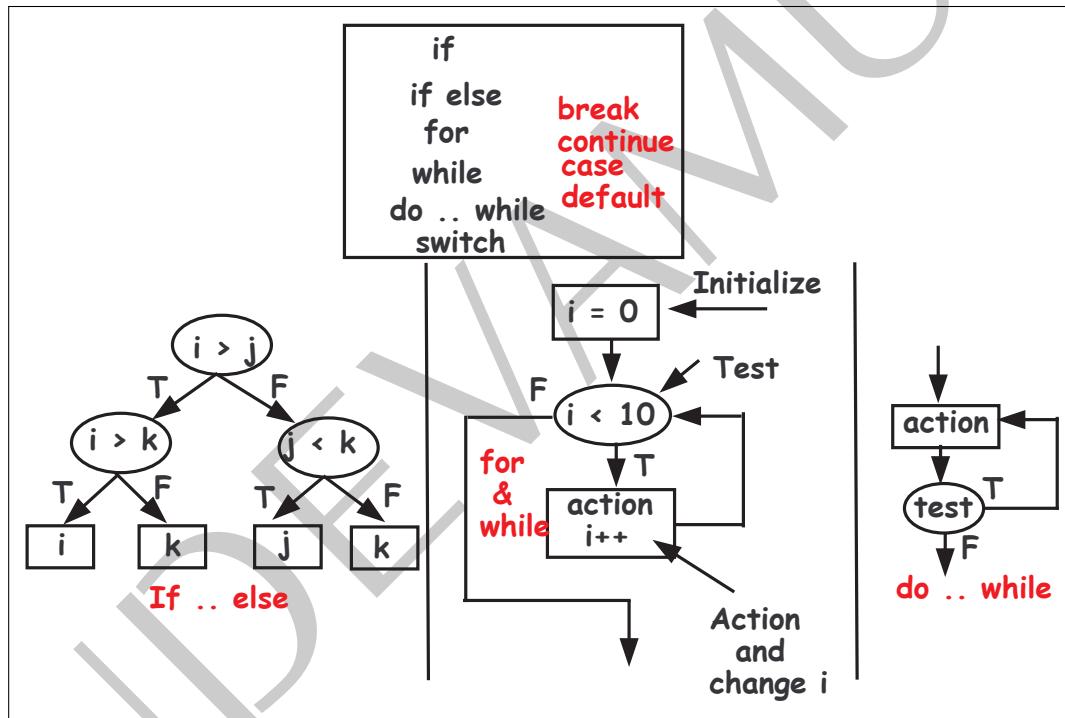


Figure 1.17: Control statements

1.12. CONTROL STATEMENTS

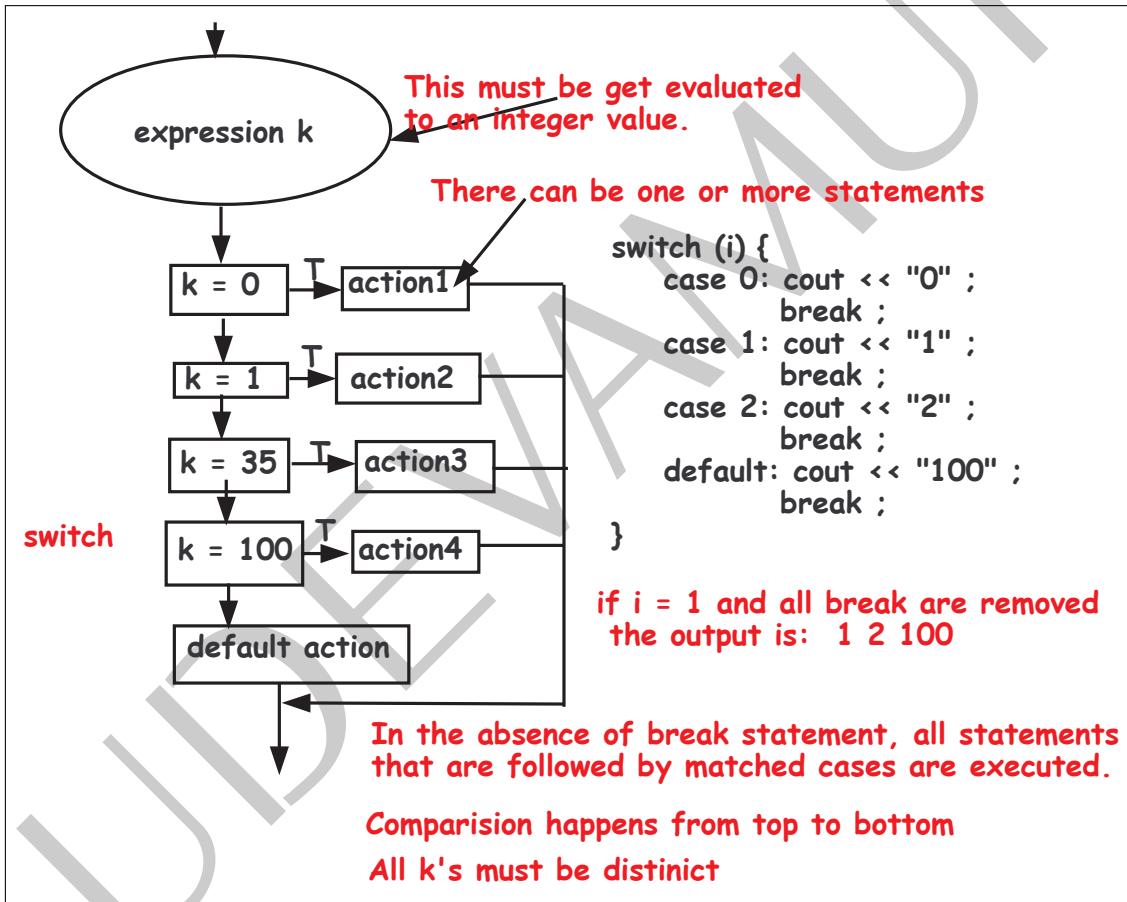


Figure 1.18: switch statement

1.12.1 C++code for illustrating control statements

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: control.h  
4 -----*/  
5  
6 /*-----  
7 This file has control class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef control_H  
14 #define control_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of control class  
20 -----*/  
21 class control{  
22 public:  
23     void test_block();  
24     void test_if();  
25     void test_if_else() ;  
26     void test_for() ;  
27     void test_while() ;  
28     void test_do_while() ;  
29     void test_need_for_switch() ;  
30     void test_switch() ;  
31     void test_switch1() ;  
32 private:  
33 };  
35  
36 #endif  
37 //EOF  
38  
39
```

```
c:\work\software\course\objects\control\control.cpp 1
1  /*
2  Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy
3  file: control.cpp
4  */
5
6  /*
7  This file has control class definition
8  */
9
10 /*
11 All includes here
12 */
13 #include "control.h"
14
15 /*
16 Definition of routines of control class
17 */
18
19 /*
20 i = 88
21 i = 77
22 i = 88
23 */
24 void control::test_block() {
25     int i = 88 ;
26     cout << " i = " << i << endl;
27     {
28         int i = 77 ;
29         cout << " i = " << i << endl;
30         int j = 20 ;
31     }
32     cout << " i = " << i << endl;
33     //cout << " j = " << j << endl; //error C2065: 'j' : undeclared identifier
34 }
35
36 /*
37 i = 81
38 80 < 85
39 i = 81
40 */
41 void control::test_if() {
42     int i = 80 ;
43     if (i > 85)
44         cout << i << " > " << "85\n" ;
45     i++ ;
46     cout << "i = " << i << endl ;
47     i-- ;
48     if (i < 85) {
49         cout << i << " < " << "85\n" ;
```

```
c:\work\software\course\objects\control\control.cpp
50     i++ ;
51 }
52 cout << "i = " << i << endl ;
53 }
54
55 /*
56 max = 50
57 */
58 void control::test_if_else() {
59     int i = 30;
60     int j = 10 ;
61     int k = 50 ;
62     int max = -1 ;
63     if (i > j) {
64         // i is greater
65         if (i > k) {
66             // i is greater
67             max = i ;
68         }else {
69             //k is greater
70             max = k ;
71         }
72     }else {
73         //j is greater
74         if (j > k) {
75             // j is greater
76             max = j;
77         }else {
78             //k is greater
79             max = k ;
80         }
81     }
82     cout << "max = " << max << endl ;
83 }
84
85 /*
86 100 101 102 103 104 105 106 107 108 109
87 100 103 106 109
88
89 100 99 98 97 96 95 94 93 92 91
90 50 52 54 56
91 50 52 54 56
92 50 52 54 56
93 */
94 void control::test_for() {
95     for (int i = 100; i < 110; i++) {
96         cout << i << " " ;
97     }
98     cout << endl ;
```

```
c:\work\software\course\objects\control\control.cpp
99   for (int i = 100; i < 110; i=i+3){
100     cout << i << " " ;
101   }
102   cout << endl ;
103
104 //to demonstrate i is dead here
105 //cout << i << endl ; //error C2065: 'i' : undeclared identifier
106
107 #if 0
108   //What happens ?
109   for (int i = 100; i < 110; i--){
110     cout << i << " " ;
111   }
112   cout << endl ;
113 #endif
114
115   for (int i = 100; i < 90; i--) {
116     cout << i << " " ;
117   }
118   cout << endl ;
119
120   for (int i = 100; i > 90; i--) {
121     cout << i << " " ;
122   }
123   cout << endl ;
124 {
125   int i = 50 ;
126   for (; i < 57; i=i+2) {
127     cout << i << " " ;
128   }
129   cout << endl ;
130 }
131 {
132   int i = 50 ;
133   for (; i < 57;) {
134     cout << i << " " ;
135     i = i + 2 ;
136   }
137   cout << endl ;
138 }
139 {
140   for (int i = 0; i < 10; i++) {
141     cout << " i = " << i << endl ;
142     if (i == 5) {
143       break ;
144     }
145   }
146
147 }
```

c:\work\software\course\objects\control\control.cpp

```
148  {
149      int i = 50 ;
150      for (;;) {
151          cout << i << " " ;
152          i = i + 2 ;
153          if (i >= 57) {
154              break ;
155          }
156      }
157      cout << endl ;
158  }
159 }
160
161 /*-----
162 100 101 102 103 104 105 106 107 108 109
163 100 103 106 109
164
165 100 99 98 97 96 95 94 93 92 91
166 50 52 54 56
167 50 52 54 56
168 50 52 54 56
169 -----*/
170 void control::test_while() {
171     {
172         int i = 100 ;
173         while (i < 110) {
174             cout << i << " " ;
175             i++ ;
176         }
177         cout << endl ;
178     }
179     {
180         int i = 100 ;
181         while (i < 110) {
182             cout << i << " " ;
183             i = i + 3 ;
184         }
185         cout << endl ;
186     }
187
188 #if 0
189 //What happens ?
190     int i = 100 ;
191     while (i < 110) {
192         cout << i << " " ;
193         i-- ;
194     }
195     cout << endl ;
196 #endif
```

```
197  {
198      int i = 100 ;
199      while (i < 90) {
200          cout << i << " " ;
201          i-- ;
202      }
203      cout << endl ;
204  }
205  {
206      int i = 100 ;
207      while ( i > 90) {
208          cout << i << " " ;
209          i-- ;
210      }
211      cout << endl ;
212  }
213  {
214      int i = 50 ;
215      while (i < 57) {
216          cout << i << " " ;
217          i = i + 2;
218      }
219      cout << endl ;
220  }
221  {
222      int i = 50 ;
223      while (i < 57) {
224          cout << i << " " ;
225          i = i + 2 ;
226      }
227      cout << endl ;
228  }
229  {
230      int i = 50 ;
231      while(true) {
232          cout << i << " " ;
233          i = i + 2 ;
234          if (i >= 57) {
235              break ;
236          }
237      }
238      cout << endl ;
239  }
240 }
241
242 /*-----*
243 100 101 102 103 104 105 106 107 108 109
244 100 103 106 109
245 100
```

c:\work\software\course\objects\control\control.cpp

```
246 100 99 98 97 96 95 94 93 92 91
247 50 52 54 56
248 50 52 54 56
249 50 52 54 56
250 -----
251 void control::test_do_while() {
252 {
253     int i = 100 ;
254     do {
255         cout << i << " " ;
256         i++ ;
257     } while (i < 110) ;
258     cout << endl ;
259 }
260 {
261     int i = 100 ;
262     do {
263         cout << i << " " ;
264         i = i + 3 ;
265     } while (i < 110);
266     cout << endl ;
267 }
268
269 #if 0
270     //What happens ?
271     int i = 100 ;
272     do {
273         cout << i << " " ;
274         i-- ;
275     }while (i < 110);
276     cout << endl ;
277 #endif
278 {
279     int i = 100 ;
280     do{
281         cout << i << " " ;
282         i-- ;
283     } while (i < 90) ;
284     cout << endl ;
285 }
286 {
287     int i = 100 ;
288     do{
289         cout << i << " " ;
290         i-- ;
291     } while ( i > 90) ;
292     cout << endl ;
293 }
294 {
```

```
295     int i = 50 ;
296     do {
297         cout << i << " " ;
298         i = i + 2;
299     } while (i < 57);
300     cout << endl ;
301 }
302
303 {
304     int i = 50 ;
305     do {
306         cout << i << " " ;
307         i = i + 2 ;
308         if (i >= 57) {
309             break ;
310         }
311     } while(true) ;
312     cout << endl ;
313 }
314 }
315
316 /*-----
317 Marks = 7 Grade = f bonus = 0
318 -----*/
319 void control::test_need_for_switch() {
320     int marks = 7 ;
321     int bonus = -1;
322     char grade = ' ' ;
323     if (marks > 90) {
324         grade = 'a' ;
325         bonus = 5 ;
326     } else if (marks > 80){
327         grade = 'b' ;
328         bonus = 4 ;
329     } else if (marks > 70) {
330         grade = 'c' ;
331         bonus = 3 ;
332     } else if (marks > 60){
333         grade = 'd' ;
334         bonus = 2 ;
335     } else if (marks > 50) {
336         grade = 'e' ;
337         bonus = 1 ;
338     } else {
339         grade = 'f' ;
340         bonus = 0 ;
341     }
342     cout << "Marks = " << marks << " Grade = " << grade << " bonus = " << bonus << endl ;
```

```
c:\work\software\course\objects\control\control.cpp
343 }
344
345 /*-----*
346 Marks = 7 Grade = f bonus = 0
347 -----*/
348 void control::test_switch() {
349     int marks = 7 ;
350     int bonus = -1;
351     char grade = ' ' ;
352     int k = (marks -1) /10 ; //Must get evaluated to an integer value
353     switch (k) {
354     case 9:
355         grade = 'a' ;
356         bonus = 5 ;
357         break ;
358     case 8:
359         grade = 'b' ;
360         bonus = 4 ;
361         break ;
362     case 7:
363         grade = 'c' ;
364         bonus = 3 ;
365         break ;
366     case 6:
367         grade = 'd' ;
368         bonus = 2 ;
369         break ;
370     case 5:
371         grade = 'e' ;
372         bonus = 1 ;
373         break ;
374     default:
375         grade = 'f' ;
376         bonus = 0 ;
377         break ;
378     }
379     cout << "Marks = " << marks << " Grade = " << grade << " bonus = " << bonus <
380     << endl ;
381 }
382 /*-----*
383 Marks = 7 Grade = d bonus = 2
384 -----*/
385 void control::test_switch1() {
386     int marks = 7 ;
387     int bonus = -1;
388     char grade = ' ' ;
389     int k = 8 ;
390     switch (k) {
```

```
391 case 9:  
392     grade = 'a' ;  
393     bonus = 5 ;  
394     break ;  
395 case 8:  
396     grade = 'b' ;  
397     bonus = 4 ;  
398 case 7:  
399     grade = 'c' ;  
400     bonus = 3 ;  
401 case 6:  
402     grade = 'd' ;  
403     bonus = 2 ;  
404     break ;  
405 case 5:  
406     grade = 'e' ;  
407     bonus = 1 ;  
408     break ;  
409 default:  
410     grade = 'f' ;  
411     bonus = 0 ;  
412     break ;  
413 }  
414 cout << "Marks = " << marks << " Grade = " << grade << " bonus = " << bonus ↵  
415     << endl ;  
416  
417 }
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: controltest.cpp  
4  
5 On linux:  
6 g++ control.cpp controltest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test control object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "control.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     control a ;  
25     a.test_block();  
26     a.test_if();  
27     a.test_if_else();  
28     a.test_for();  
29     a.test_while();  
30     a.test_do_while();  
31     a.test_need_for_switch();  
32     a.test_switch();  
33     a.test_switch1();  
34 }  
35  
36 /*-----  
37 main  
38 -----*/  
39 int main() {  
40     testbed();  
41     return 0 ;  
42 }  
43  
44 //EOF  
45
```

1.13 Enumerations(enum)

1.13. ENUMERATIONS(ENUM)

```
static void using_magic_numbers(int dir) {  
    switch (dir) {  
        case 1: cout << "Go north\n" ;  
        break ;  
        case 2: cout << "Go south\n" ;  
        break ;  
        case 3: cout << "Go west\n" ;  
        break ;  
        case 4: cout << "Go east\n" ;  
        break ;  
    }  
}
```

Need to document dir
Use of magic numbers

```
#define NORTH 1  
#define SOUTH 2  
#define EAST 4  
#define WEST 3
```

No type checking
Handled by c++ reprocessor

```
static void using_define(int dir) {  
    switch (dir) {  
        case NORTH: cout << "Go north\n" ;  
        break ;  
        case SOUTH: cout << "Go south\n" ;  
        break ;  
        case WEST: cout << "Go west\n" ;  
        break ;  
        case EAST: cout << "Go east\n" ;  
        break ;  
    }  
}
```

Figure 1.19: Need for enumeration

```
const int NORTH = 1;
const int SOUTH = 2 ;
const int EAST = 4 ;
const int WEST = 3 ;
```

Data type: Int
Information not together

```
static void using_consts(int dir) {
    switch (dir) {
        case NORTH: cout << "Go north\n";
                     break ;
        case SOUTH: cout << "Go south\n";
                     break ;
        case WEST: cout << "Go west\n";
                     break ;
        case EAST: cout << "Go east\n";
                     break ;
    }
}
```

```
enum direction {NORTH=1, SOUTH, EAST, WEST} ;
```

```
static void using_enum(direction dir) {
    switch (dir) {
        case NORTH: cout << "Go north\n";
                     break ;
        case SOUTH: cout << "Go south\n";
                     break ;
        case WEST: cout << "Go west\n";
                     break ;
        case EAST: cout << "Go east\n";
                     break ;
    }
}
```

Create new data type
All info in one place

Figure 1.20: Enumeration:your own data tpe

1.13. ENUMERATIONS(ENUM)

1.13.1 C++ code for illustrating enumeration

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: enumeration.h  
4 -----*/  
5  
6 /*-----  
7 This file has enumeration class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef enumeration_H  
14 #define enumeration_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of enumeration class  
20 -----*/  
21 class enumeration{  
22 public:  
23     enum direction{NORTH=1, SOUTH, EAST, WEST} ;  
24     enum day{  
25         mon = 3, tue = 8, wed = 2, thu, fri, sat, sun  
26     };  
27     enum day_of_months{  
28         jan = 31, feb = 28, mar = 31, apr = 30, may = 31, june = 30,  
29         july = 31, aug = 31, sep = 30, oct = 31, nov = 30, dec = 31  
30     } ;  
31     enum Days {           // Declare enum type Days  
32         saturday,          // saturday = 0 by default  
33         sunday = 0,          // sunday = 0 as well  
34         monday,              // monday = 1  
35         tuesday,             // tuesday = 2  
36         wednesday,           // etc.  
37         thursday,  
38         friday  
39     };  
40  
41     void go_in_this_direction(enumeration::direction dir) ;  
42     void day_of_the_week(enumeration::day d) ;  
43     void which_month(enumeration::day_of_months m) ;  
44     void test_default_enum_starts_with_zero() ;  
45 private:  
46  
47 } ;  
48  
49 #endif  
50 //EOF  
51  
52
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: enumeration.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has enumeration class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "enumeration.h"  
14  
15 /*-----  
16 Definition of routines of enumeration class  
17 -----*/  
18  
19 /*-----  
20 if dir is passed as WEST  
21  
22 Go west  
23 -----*/  
24 void enumeration::go_in_this_direction(enumeration::direction dir) {  
25     switch (dir) {  
26         case NORTH: cout << "Go north\n" ;  
27             break ;  
28         case SOUTH: cout << "Go south\n" ;  
29             break ;  
30         case WEST: cout << "Go west\n" ;  
31             break ;  
32         case EAST: cout << "Go east\n" ;  
33             break ;  
34     }  
35 }  
36  
37 /*-----  
38 if d is passed as sat  
39  
40 Enjoy weekend  
41 go to school  
42 8  
43 2  
44 wed 2 thu = 3 fri 4 sat 5 sun 6  
45 -----*/  
46 void enumeration::day_of_the_week(enumeration::day d) {  
47     if (d == sat || d == sun) {  
48         cout << "Enjoy weekend\n" ;  
49     }  
50     int k = sat - 1 ;  
51     if (k == fri) {  
52         cout << "go to school\n" ;  
53     }  
54     cout << tue << endl ;  
55     int x = tue ;  
56     cout << wed << endl ;  
57     cout << "wed " << wed << " thu = " << thu << " fri " << fri << " sat " << sat << " sun " << sun <<  
58     endl ;  
59 }  
60  
61 /*-----  
62 if 'd' is passed as feb  
63 I am in feb  
64 I am in may  
65 -----*/
```

```
66 void enumeration::which_month(enumration::day_of_months d) {
67     //d++ ; error C2676: binary '++' : 'test_month::day_of_months' does not define this operator or a
68     conversion to a type acceptable to the predefined operator
69     if (d == feb) {
70         cout << "I am in feb\n" ;
71     }
72     enumration::day_of_months m = may ;
73     if (m == may) {
74         cout << "I am in may\n" ;
75     }
76
77 /*-----*/
78 Not Monday
79 -----*/
80 void enumeration::test_default_enum_starts_with_zero() {
81     enumration::Days today = sunday;
82     switch (today) {
83     case 1:
84         cout << "It's Monday" << endl;
85         break;
86     default:
87         cout << "Not Monday" << endl;
88     }
89 }
90
91 //EOF
92
93
```

c:\work\software\course\objects\enumeration\enumerationtest.cpp

1

```
1  /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: enumerationtest.cpp  
4  
5 On linux:  
6 g++ enumeration.cpp enumerationtest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test enumeration object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "enumeration.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     enumeration a ;  
25     a.go_in_this_direction(enum::direction::WEST);  
26     a.day_of_the_week(enum::day::sat) ;  
27     a.which_month(enum::day_of_months::feb) ;  
28     a.test_default_enum_starts_with_zero() ;  
29 }  
30  
31 /*-----  
32 main  
33 -----*/  
34 int main() {  
35     testbed() ;  
36     return 0 ;  
37 }  
38  
39 //EOF  
40
```

1.14 Introductions to functions

PASS BY VALUE

```
int callbyvalue::call_by_value_and_return_function(int coffee, int tea) {
    int temp = coffee ; temp [ ] [ ]
    coffee = tea ; coffee [ ] [ ]
    tea = temp ; tea [ ] [ ]
    cout << "7 : coffee = " << coffee << " tea = " << tea << endl ;
    int x = coffee + tea ;
    return x ;
} → temp, coffee, tea and x dies here
```

```
void callbyvalue::test_f3() {
    int coffee = 10 ; coffee [ ] [ ]
    int tea = 20 ; tea [ ] [ ]
    cout << "5: coffee = " << coffee << " tea = " << tea << endl ;
    int x = call_by_value_and_return_function(coffee,tea) ;
    cout << "8 : coffee = " << coffee << " tea = " << tea << endl ;
    cout << "9 : x = " << x << endl ;
}
```

1. You cannot change the values of the arguments
 2. You can only get ONE returned value

```
int main() {
    callbyvalue a;
    a.test_f3();
    return 0 ;
}
```

Figure 1.21: Call by value

1.14.1 C++ code for illustrating call by value function

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: callbyvalue.h  
4 -----*/  
5  
6 /*-----  
7 This file has callbyvalue class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef callbyvalue_H  
14 #define callbyvalue_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of callbyvalue class  
20 -----*/  
21 class callbyvalue{  
22 public:  
23     void no_parameters_function();  
24     void test_f1();  
25     void call_by_value_function(int coffee, int tea);  
26     void test_f2();  
27     int call_by_value_and_return_function(int coffee, int tea);  
28     void test_f3();  
29 private:  
30 };  
31  
32  
33 #endif  
34 //EOF  
35  
36
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: callbyvalue.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has enumeration class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "callbyvalue.h"  
14  
15 /*-----  
16 Definition of routines of callbyvalue class  
17 -----*/  
18  
19 /*-----  
20 global variable to this file. BAD way  
21 -----*/  
22 static int global_i = 10 ;  
23  
24 /*-----  
25 This function has no input and output  
26 -----*/  
27 void callbyvalue::no_parameters_function() {  
28     cout << "global_i = " << global_i++ << endl ;  
29 }  
30  
31 /*-----  
32 global_i = 10  
33 global_i = 11  
34 -----*/  
35 void callbyvalue::test_f1() {  
36     no_parameters_function();  
37     no_parameters_function();  
38 }  
39  
40 /*-----  
41 This function has no output, but has 2 inputs  
42 -----*/  
43 void callbyvalue::call_by_value_function(int coffee, int tea) {  
44     cout << "2 : coffee = " << coffee << " tea = " << tea << " global_i " << global_i << endl ;  
45     int temp = coffee ;  
46     coffee = tea ;  
47     tea = temp ;  
48     global_i += 20 ;  
49     cout << "3 : coffee = " << coffee << " tea = " << tea << " global_i " << global_i << endl ;  
50 }  
51  
52 /*-----  
53 1: coffee = 10 tea = 20 global_i 10  
54 2 : coffee = 10 tea = 20 global_i 10  
55 3 : coffee = 20 tea = 10 global_i 30  
56 4 : coffee = 10 tea = 20 global_i 30  
57 -----*/  
58 void callbyvalue::test_f2() {  
59     global_i = 10 ;  
60     int coffee = 10 ;  
61     int tea = 20 ;  
62     cout << "1: coffee = " << coffee << " tea = " << tea << " global_i " << global_i << endl ;  
63     call_by_value_function(coffee,tea) ;  
64     cout << "4 : coffee = " << coffee << " tea = " << tea << " global_i " << global_i << endl ;  
65 }  
66
```

```
67 /*-----  
68 This function has a output, but has 2 inputs  
69 -----*/  
70 int callbyvalue::call_by_value_and_return_function(int coffee, int tea) {  
71     cout << "6 : coffee = " << coffee << " tea = " << tea << " global_i " << global_i << endl ;  
72     int temp = coffee ;  
73     coffee = tea ;  
74     tea = temp ;  
75     global_i += 20 ;  
76     cout << "7 : coffee = " << coffee << " tea = " << tea << " global_i " << global_i << endl ;  
77     int x = coffee + tea ;  
78     return x ;  
79 }  
80  
81 /*-----  
82 global_i 10  
83 5: coffee = 10 tea = 20 global_i 24  
84 6 : coffee = 10 tea = 20 global_i 10  
85 7 : coffee = 20 tea = 10 global_i 30  
86 8 : coffee = 10 tea = 20 global_i 25  
87 9 : x = 30  
88 global_i 30  
89 -----*/  
90 void callbyvalue::test_f3() {  
91     global_i = 10 ;  
92     cout << "global_i " << global_i << endl ;  
93     {  
94         int coffee = 10 ;  
95         int tea = 20 ;  
96         int global_i = 24 ;  
97         cout << "5: coffee = " << coffee << " tea = " << tea << " global_i " << global_i++ << endl ;  
98         int x = call_by_value_and_return_function(coffee,tea) ;  
99         cout << "8 : coffee = " << coffee << " tea = " << tea << " global_i " << global_i << endl ;  
100        cout << "9 : x = " << x << endl ;  
101    }  
102    cout << "global_i " << global_i << endl ;  
103 }  
104  
105 //EOF  
106  
107
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: callbyvaluetest.cpp  
4  
5 On linux:  
6 g++ callbyvalue.cpp callbyvaluetest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test callbyvalue object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "callbyvalue.h"  
19  
20  
21 /*-----  
22 test bed  
23 -----*/  
24 void testbed() {  
25     callbyvalue a ;  
26     a.test_f1() ;  
27     a.test_f2() ;  
28     a.test_f3() ;  
29 }  
30  
31 /*-----  
32 main  
33 -----*/  
34 int main() {  
35     testbed() ;  
36     return 0 ;  
37 }  
38  
39 //EOF  
40  
41  
42
```

1.15 Introductions to pointers

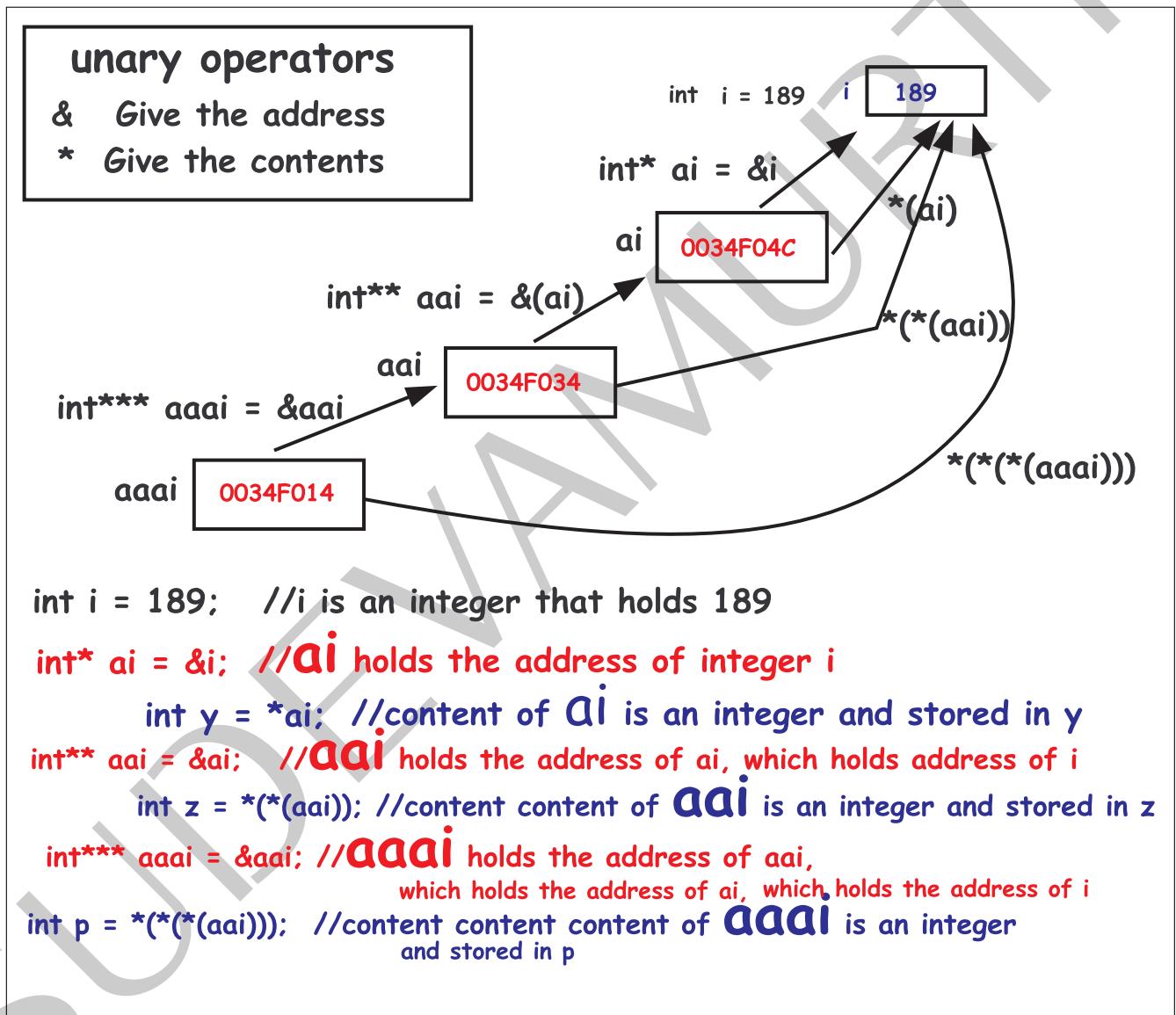


Figure 1.22: Pointers

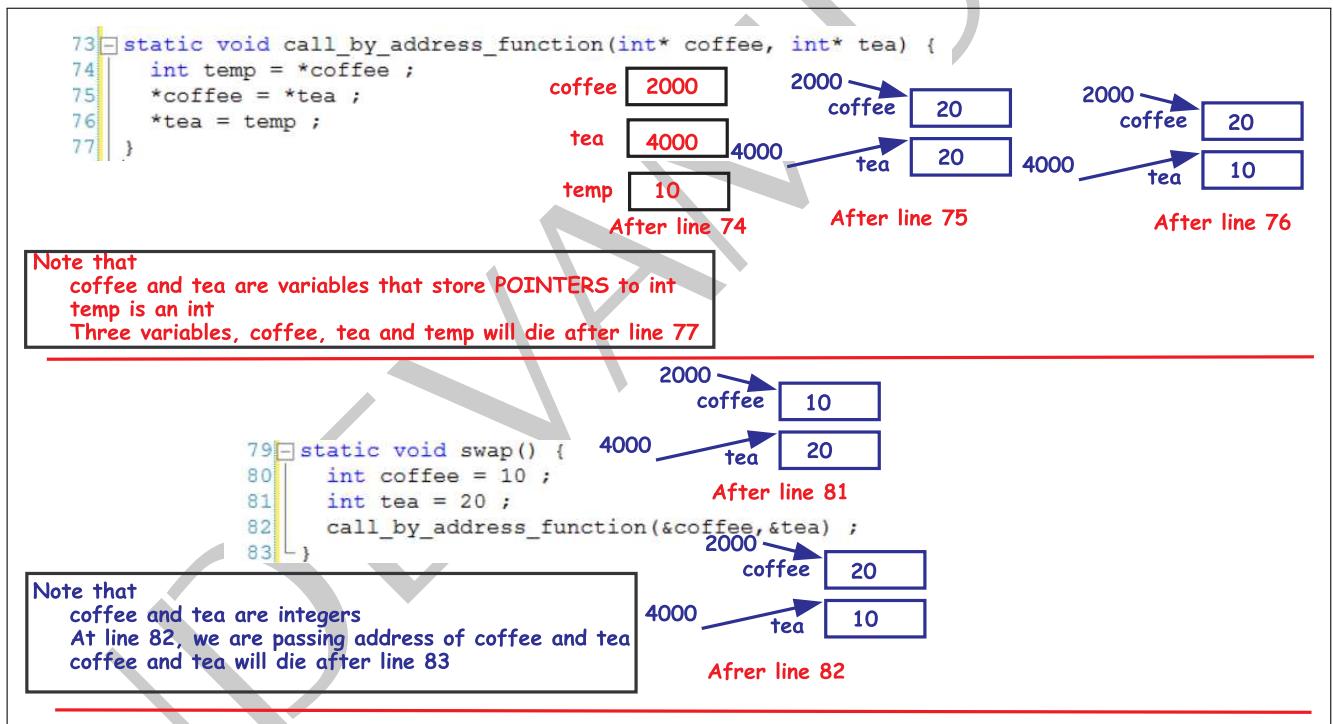


Figure 1.23: Swapping contents of two variables

is ``int* p;'' right or is ``int *p;'' right?

Both are "right" in the sense that both are valid C and C++ and both have exactly the same meaning.

As far as the language definitions and the compilers are concerned we could just as well say ``int*p;'' or ``int * p;''

A ``typical C programmer'' writes ``int *p;''
and explains it ``*p is what is the int''

A ``typical C++ programmer'' writes ``int* p;''
and explains it ``p is a pointer to an int''

In this course we use this notation:
We read from right to left

int x ; x is an integer
int* p ; p is a pointer to integer
int** p ; p is a pointer, pointing to another pointer, pointing to integer
int& p; p is an alias to an integer object

Figure 1.24: int *p or int* p

1.15.1 C++code for illustrating pointers

c:\work\software\course\objects\pointers\pointers.h

```
1  /*-----  
2 Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3 file: pointers.h  
4 -----*/  
5  
6 /*-----  
7 This file has pointers class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef pointers_H  
14 #define pointers_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of pointers class  
20 -----*/  
21 class pointers{  
22 public:  
23     void p1();  
24     void swap_that_does_not_swap();  
25     void swap() ;  
26     void why_pointer_to_pointer_exists() ;  
27  
28 private:  
29     void _p2(int* i);  
30     void _p3(int** i);  
31     void _p4(int*** i);  
32     void _call_by_value_function(int coffee, int tea);  
33     void _call_by_address_function(int* coffee, int* tea);  
34     void _f2(const char*** y);  
35     void _f1(const char** x);  
36 };  
37  
38 #endif  
39 //EOF  
40
```

1

```
1  /*
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3  file: pointers.cpp
4  */
5
6  /*
7  This file has enumeration class definition
8  */
9
10 /*
11 All includes here
12 */
13 #include "pointers.h"
14
15 /*
16 Definition of routines of pointers class
17 */
18
19 /*
20 */
21 /*
22 void pointers::_p4(int*** i) {
23     cout << "in p4: " << "address of address of address of i = " << i;
24     cout << " and i = " << *(*(*(i))) << endl ;
25 }
26
27 /*
28 */
29 /*
30 void pointers::_p3(int** i) {
31     cout << "in p3: " << "address of address of i = " << i;
32     cout << " and i = " << *i << endl ;
33     int*** aaai = &i ;
34     _p4(aaai) ;
35 }
36
37 /*
38 */
39 /*
40 void pointers::_p2(int* i) {
41     cout << "in p2: " << "address of i = " << i;
42     cout << " and i = " << *i << endl ;
43     int** aai = &i ;
44     _p3(aai) ;
45 }
46
47 /*
48 in p1: i = 189
49 in p2: address of i = 0034F04C and i = 189
```

C:\work\software\course\objects\pointers\pointers.cpp

```

50 in p3: address of address of i = 0034F034 and i = 189
51 in p4: address of address of address of i = 0034F014 and i = 189
52 -----
53 void pointers::p1() {
54     int i = 189 ;
55     cout << "in p1: " << "i = " << i << endl ;
56     int* ai = &i ;
57     _p2(ai) ;
58 }
59
60 /*
61 This function has no output, but has 2 inputs
62 -----
63 void pointers::_call_by_value_function(int coffee, int tea) {
64     cout << "2 : coffee = " << coffee << " tea = " << tea << endl ;
65     int temp = coffee ;
66     coffee = tea ;
67     tea = temp ;
68     cout << "3 : coffee = " << coffee << " tea = " << tea << endl ;
69 }
70
71 /*
72 1: coffee = 10 tea = 20
73 2 : coffee = 10 tea = 20
74 3 : coffee = 20 tea = 10
75 4 : coffee = 10 tea = 20
76 -----
77 void pointers::swap_that_does_not_swap() {
78     int coffee = 10 ;
79     int tea = 20 ;
80     cout << "1: coffee = " << coffee << " tea = " << tea << endl ;
81     _call_by_value_function(coffee,tea) ;
82     cout << "4 : coffee = " << coffee << " tea = " << tea << endl ;
83 }
84
85 /*
86 This function has no output, but has 2 inputs
87 -----
88 void pointers::_call_by_address_function(int* coffee, int* tea) {
89     cout << "2 : coffee = " << *(coffee) << " tea = " << *(tea) << endl ;
90     int temp = *coffee ;
91     *coffee = *tea ;
92     *tea = temp ;
93     cout << "3 : coffee = " << *(coffee) << " tea = " << *(tea) << endl ;
94 }
95
96 /*
97 1: coffee = 10 tea = 20
98 2 : coffee = 10 tea = 20

```

```
1  /*-----  
2 Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3 file: pointerstest.cpp  
4  
5 On linux:  
6 g++ pointers.cpp pointerstest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test pointers object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "pointers.h"  
19  
20  
21 /*-----  
22 test bed  
23 -----*/  
24 void testbed() {  
25     pointers a ;  
26     a.p1() ;  
27     a.swap_that_does_not_swap();  
28     a.swap() ;  
29     a.why_pointer_to_pointer_exists();  
30 }  
31  
32 /*-----  
33 main  
34 -----*/  
35 int main() {  
36     testbed() ;  
37     return 0 ;  
38 }  
39  
40 //EOF  
41  
42  
43
```

1.16 Understanding const

```
char* p = "jag"; //Non constant pointer, Non constant data  
const char* p = "jag"; //Non constant pointer, constant data  
char* const p = "jag"; //constant pointer, non constant data  
const char* const p = "jag"; //constant pointer, constant data
```

What's pointed to
is constant

char *
const char *
char *
const char *

Pointer is
constant

p = "jag"
p = "jag"
const p = "jag"
const p = "jag"

Use const whenever possible

Figure 1.25: Use **const** whenever possible

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: const.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 integer test  
11 -----*/  
12 static void int_test()  
13 {  
14     int x = 4;  
15     x = 10;  
16     //const int y = 2;  
17     //y = 89;  
18     //FAILS: error C3892: 'y' : you cannot assign to a variable that is const  
19 }  
20  
21 /*-----  
22 Generic print routine  
23 -----*/  
24 static void print(const char* title, const char* s)  
25 {  
26     cout << title << endl;  
27     int l = strlen(s);  
28     for (int i = 0; i < l; i++) {  
29         cout << s[i];  
30     }  
31     cout << endl;  
32 }  
33  
34 /*-----  
35 Non constant pointer Non constant data  
36 -----*/  
37 static void ncp_ncd()  
38 {  
39     char* p = new char[4];  
40     strcpy(p,"NPU");  
41     print("ncp_ncd BEFORE: ", p);  
42     p[0] = 'L';  
43     print("ncp_ncd AFTER: ", p);  
44     delete [] p;  
45 }  
46  
47 /*-----  
48 Non constant pointer Constant data  
49 -----*/  
50 static void ncp_cd()  
51 {  
52     const char* p = "MIT";  
53     cout << "p = " << p << endl;  
54     //p[0] = 'N';  
55     //error C3892: 'p' : you cannot assign to a variable that is const  
56     const char* q = "UCSC";  
57     p = q;  
58 }
```

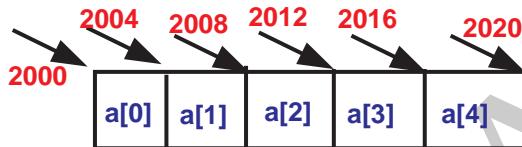
```
56 /*-----  
57 Constant pointer Non constant data  
58 -----*/  
59 static void cp_ncd(){  
60     char* const p = new char[4];  
61     strcpy(p,"NPU");  
62     print("cp_ncd BEFORE: ", p);  
63     p[0] = 'L';  
64     print("cp_ncd AFTER: ", p);  
65  
66     char* const p1 = new char[4];  
67     strcpy(p1,"MIT");  
68     print("cp_ncd BEFORE2: ", p1);  
69 //p = p1;  
70 // error C3892: 'p' : you cannot assign to a variable that is const  
71     delete [] p;  
72     delete [] p1;  
73 }  
74  
75 /*-----  
76 Constant pointer Constant data  
77 -----*/  
78 static void cp_cd(){  
79     const char* const p = "MIT";  
80 //p[0] = 'K';  
81 //error C3892: 'p' : you cannot assign to a variable that is const  
82  
83     const char* q = "UCSC";  
84 //p = q;  
85 //error C3892: 'p' : you cannot assign to a variable that is const  
86 }  
87  
88 /*-----  
89 Main  
90 -----*/  
91 int main(){  
92     int_test();  
93     cout << "_____" << endl;  
94     ncp_ncd();  
95     cout << "_____" << endl;  
96     ncp_cd();  
97     cout << "_____" << endl;  
98     cp_ncd();  
99     cout << "_____" << endl;  
100    cp_cd();  
101    cout << "_____" << endl;  
102    return 0;  
103 }  
104 }
```

1.17 Introductions to arrays

```
int a[5]; //a is an array of 5 int elements starting from 0 to 4
'size', in this case 5, must be known before. CANNOT be changed
```



We should carry size of array, 5 in this case.
 There is no way of getting size, after declaration
 You can do a[23] and get crash/unpredictable results



**a is an array
pa is variable**

```
int* pa = a; //pa is a pointer to the a[0]
int x = *(pa); //Gives the content of a[0]
```

→ int* pa = &a[0];

Note that pointers are not integer

```
pa++; //Not 2001, but 2004 (size of integer is 4)
int x = *(pa); //Gives the content of a[1]
pa = pa+2; //Goes to (2004 + (2 * 4)) = 2012
x = *(pa); //Gives the content of a[3]
&(a[0]) = pa = 2000
&(a[1]) = 2000 + (1 * 4) = 2004 = pa++
&(a[2]) = 2000 + (2 * 4) = 2008 = pa+=2
&(a[3]) = 2000 + (3 * 4) = 2012 = pa+=3
&(a[4]) = 2000 + (4 * 4) = 2016 = pa+=4
```

**you cannot do
a++, but pa++**

Figure 1.26: Arrays and pointers to array elements

1.17. INTRODUCTIONS TO ARRAYS

Array's are NEVER passed by value

```

int a[5]           Declaring array
int* pa = a ;    pointer to 0th position of a
int* pa = &(a[0]) ; //same as above
int b = a ; //NOT POSSIBLE
int* pa3 = &(a[3]); //Pointer to 3rd position of a

2004   2008   2012   2016   2020
2000   [a[0] a[1] a[2] a[3] a[4]]      a is an array
                                         pa is variable

static void int_array() {
    const int SIZE = 5 ;
    int a[SIZE] ;
    int total_size = sizeof(a) ; //Here it knows the size of array
    int array_size = sizeof(a)/sizeof(int) ; //WOW.#of elements in a
}

```

CASE 1 PASSING ARRAY - CASE 1

```

static void print_array( int b[] , int size) {
    int k = sizeof(b) ;
    //HERE IT DOES NOT KNOW. It thinks as a pointer and hence k = 4
    //That means you need to pass correct size all the time.
    for (int i = 0; i < size; i++) {
        cout << b[i] << " ";
    }
    cout << endl ;
}

static void f1() {
    const int SIZE = 5 ;
    int a[SIZE] ;
    print_array(a,SIZE) ;
}

```

CASE 2 PASSING ARRAY - CASE 2

```

static void inc_array( int* b , int size, int incr_amount) {
    int k = sizeof(b) ;
    //HERE IT DOES NOT KNOW. It thinks as a pointer and hence k = 4
    //That means you need to pass correct size all the time.
    for (int i = 0; i < size; i++) {
        b[i] = b[i] + incr_amount;
    }
}

static void f2() {
    const int SIZE = 5 ;
    int a[SIZE] ;
    inc_array(a,SIZE,40) ;
}

```

Figure 1.27: Passing array as a parameter of a function

1.17.1 C++ code for illustrating array and pointers to array elements

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: intarray.h  
4 -----*/  
5  
6 /*-----  
7 This file has intarray class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef intarray_H  
14 #define intarray_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of intarray class  
20 -----*/  
21 class intarray{  
22 public:  
23     void array_basic1();  
24     void array_basic2();  
25 private:  
26     void _print_array(const char* title, int b[], int size);  
27     void _print_in_reverse_order(const char* title, int b[], int size);  
28     void _inc_array(int* b, int size, int incr_amount);  
29     void _dec_array(int b[], int size, int decr_amount);  
30 };  
31  
32 #endif  
33 //EOF  
34  
35
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: intarray.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has enumeration class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "intarray.h"  
14  
15 /*-----  
16 Definition of routines of intarray class  
17 -----*/  
18  
19 /*-----  
20 sizeof(a)= 20  
21 sizeof(a)/sizeof(int) = 5  
22 a != b  
23 a == b  
24 a[0]= 23 a[1]= 78 a[2]= 90 a[3]= -45  
25 a[0]= 23 a[1]= 78 a[2]= 90 a[3]= -45  
26 -----*/  
27 void intarray::array_basic1() {  
28     {  
29         int size = 5 ;  
30         //SIZE MUST BE KNOWN at COMPILATION TIME. MUST BE A CONSTANT  
31         //int a[size] ;  
32         //error C2057: expected constant expression  
33     }  
34     {  
35         const int size = 5 ;  
36         int a[size] ;  
37         int k = sizeof(a) ;  
38         cout << "sizeof(a)= " << k << endl ;  
39         int num_element = sizeof(a)/sizeof(int) ;  
40         cout << "sizeof(a)/sizeof(int) = " << num_element << endl ;  
41         for (int i = 0; i < size; ++i) {  
42             a[i] = i ;  
43         }  
44         int b[size] ;  
45         //YOU cannot make b = a  
46         //b = a ;  
47         //error C2106: '=' : left operand must be l-value  
48         for (int i = 0; i < size; ++i) {  
49             b[i] = a[i];  
50         }  
51         if (a == b) {  
52             cout << "a == b" << endl ;  
53         }else {  
54             cout << "a != b" << endl ;  
55         }  
56         bool eq = true ;  
57         for (int i = 0; i < size; ++i) {  
58             if (b[i] != a[i]) {  
59                 eq = false ;  
60                 break ;  
61             }  
62         }  
63         if (eq) {  
64             cout << "a == b" << endl ;  
65         }else {  
66             cout << "a != b" << endl ;  
67         }  
68     }  
69 }
```

```

67     }
68 }
69 {
70 //This shows how to initialize the array
71 int a[] = {23,78,90,-45};
72 int num_elements = sizeof(a)/sizeof(int) ;
73 for (int i = 0; i < num_elements; ++i) {
74     cout << " a[" << i << "] = " << a[i] ;
75 }
76 cout << endl ;
77 int* p = a;
78 for (int i = 0; i < num_elements; ++i) {
79     cout << " a[" << i << "] = " << *p++ ;
80 }
81 cout << endl ;
82 }
83 {
84 //This shows how to initialize the array
85 const char* a[] = { "Jan","feb","mar","april" };
86 int num_elements = sizeof(a) / sizeof(const char*);
87 for (int i = 0; i < num_elements; ++i) {
88     cout << " a[" << i << "] = " << a[i];
89 }
90 cout << endl;
91 const char** p = a;
92 for (int i = 0; i < num_elements; ++i) {
93     cout << " a[" << i << "] = " << *p++;
94 }
95 cout << endl;
96 }
97 }
98 */
99 /*-----*
100 print the content of array 'b' that has size 'size'
101 Note that
102 1. array is passed by address
103 2. size is passed by value
104 -----*/
105 void intarray::_print_array(const char* title, int b[], int size) {
106     int k = sizeof(b) ;
107     cout << "k = " << k << endl ;
108 //HERE IT DOES NOT KNOW. It thinks as a pointer and hence k = 4
109 //That means you need to pass correct size all the time.
110     cout << title << ":" ;
111     for (int i = 0; i < size; i++) {
112         cout << b[i] << " " ;
113     }
114     cout << endl ;
115 }
116 */
117 /*-----*
118 address of b[9]= 0032EBBC content of b[9]= 14
119 -----*/
120 void intarray::_print_in_reverse_order(const char* title, int b[], int size) {
121     int k = sizeof(b) ;
122     cout << "k = " << k << endl ;
123 //HERE IT DOES NOT KNOW. It thinks as a pointer and hence k = 4
124 //That means you need to pass correct size all the time.
125     cout << title << ":" << endl;
126     b = b + size - 1; //note that size is (size * 'size of integer')
127     for (int i = size-1; i >= 0; i--) {
128         cout << "address of b[" << i << "] = " << b << " content of b[" << i << "] = " << *b << endl ;
129         b-- ; //b is not decremented by 1, Will decrement b by 'size of integer' which is 4
130     }
131 }
132 
```

```
133 /*-----  
134 Increment the content of array 'b' that has size 'size' by 'incr_amount'  
135 Note that  
136 1. array is passed by address  
137 2. size is passed by value  
138 3. incr_amount is passed by value  
139 -----*/  
140 void intarray::_inc_array(int* b, int size, int incr_amount) {  
141     int k = sizeof(b) ;  
142     cout << "k = " << k << endl ;  
143     //HERE IT DOES NOT KNOW. It thinks as a pointer and hence k = 4  
144     //That means you need to pass correct size all the time.  
145     for (int i = 0; i < size; i++) {  
146         b[i] = b[i] + incr_amount;  
147     }  
148 }  
149  
150 /*-----  
151 decrement the content of array 'b' that has size 'size' by 'decr_amount'  
152 Note that  
153 1. array is passed by address  
154 2. size is passed by value  
155 3. decr_amount is passed by value  
156 4. b++, which increment by 'size of integer'  
157 -----*/  
158 void intarray::_dec_array(int b[], int size, int decr_amount) {  
159     for (int i = 0; i < size; i++) {  
160         cout << "address of b[" << i << "] = " << b << " content of b[" << i << "] = " << *b << endl ;  
161         *b = (*b) - decr_amount ;  
162         b++ ; //b is not incremented by 1, Will increment b by 'size of integer' which is 4  
163     }  
164 }  
165  
166 /*-----  
167 -----*/  
168 -----*/  
169 void intarray::array_basic2() {  
170     const int SIZE = 10 ;  
171     int a[SIZE] ;  
172     int total_size = sizeof(a) ; //Here it knows the size of array  
173     cout << "total size = " << total_size << endl ;  
174     int array_size = sizeof(a)/sizeof(int) ; //WOW. number of element in a  
175     cout << "array size = " << array_size << endl ;  
176     for (int i = 0; i < SIZE; i++) {  
177         a[i] = i + 5 ;  
178     }  
179     _print_array("int_array", a,SIZE) ;  
180     _inc_array(a,SIZE,22) ;  
181     _print_array("After incrementing by 22", a,SIZE) ;  
182     _dec_array(a,SIZE,22) ;  
183     _print_array("After decrementing by 22", a,SIZE) ;  
184     _print_in_reverse_order("After reversing",a,SIZE) ;  
185 }  
186  
187  
188 //EOF  
189  
190
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: intarraytest.cpp  
4  
5 On linux:  
6 g++ intarray.cpp intarraytest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test intarray object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "intarray.h"  
19  
20  
21 /*-----  
22 test bed  
23 -----*/  
24 void testbed() {  
25     intarray a ;  
26     a.array_basic1();  
27     a.array_basic2();  
28 }  
29  
30 /*-----  
31 main  
32 -----*/  
33 int main() {  
34     testbed() ;  
35     return 0 ;  
36 }  
37  
38 //EOF  
39  
40
```

1.18 Basic stream output and stream input

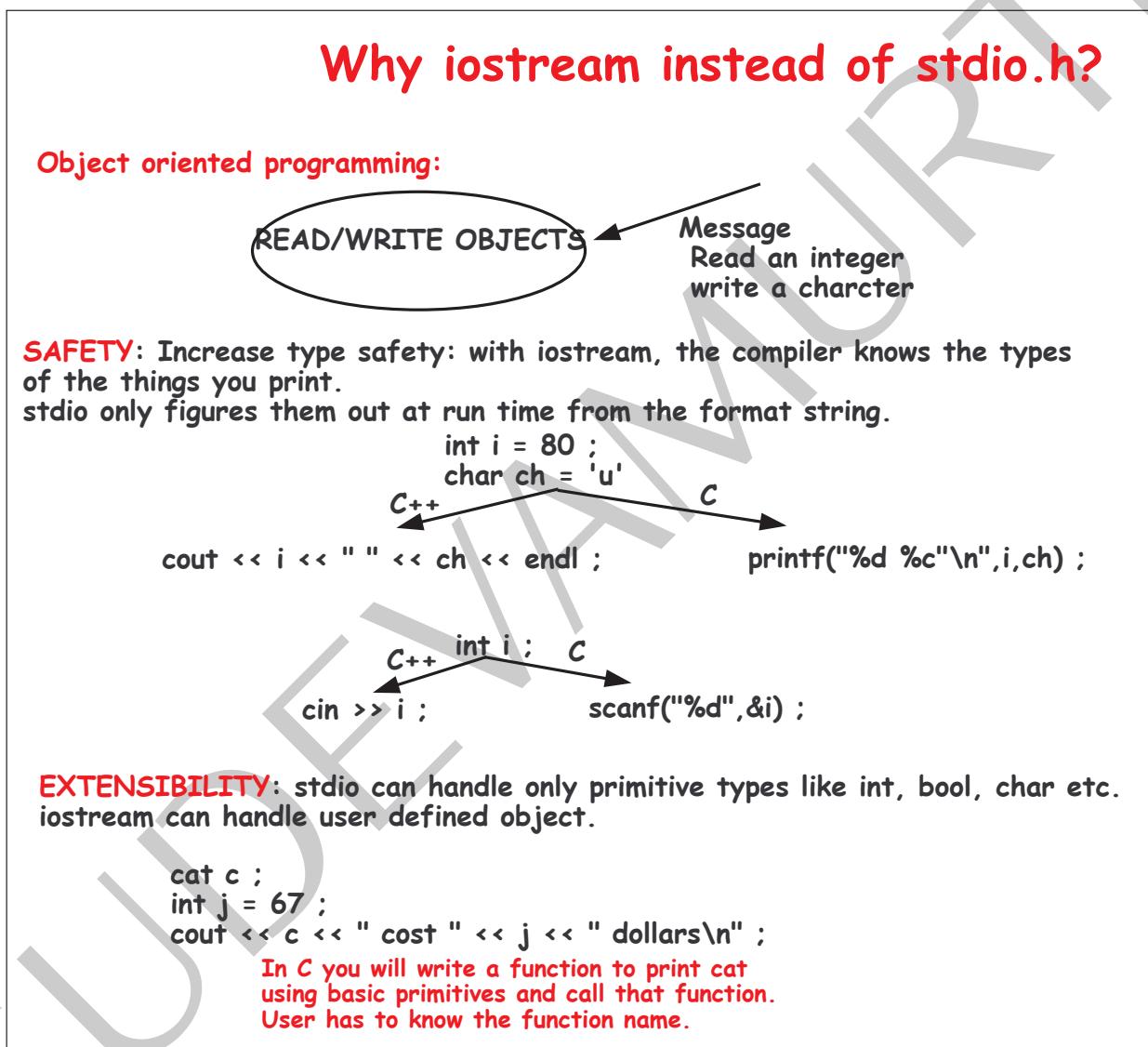


Figure 1.28: Why switch to something new?

How to write and read?

```
#include <iostream>
#include <fstream>

#include <iomanip>      // std::setprecision
using namespace std;
```

class ostream

ostream class can be used only with the instance of the class.

WRITE TO SCREEN

```
ostream cout ;
```

This object is already instantiated at global scope under namespace std

```
ostream cout ;
```

```
ostream cerr ;
```

```
ostream clog ;
```

```
int i ;
```

```
dog d ;
```

```
char ch ;
```

```
cout << i << " I am here " << d << ch << endl ;
```

class ofstream

```
ofstream out("c:\\jag\\c\\data.dat") ;
out << i << " I am here " << d << ch << endl ;
```

WRITE TO A FILE

READ FROM KEYBOARD

class istream

```
istream cin;
int i ; cin >> i ; //read int
char ch; cin >> ch ; //read
```

READ FROM A FILE

class ifstream

```
ifstream in("c:/jag/myoutput/in.dat") ;
int i ; in << i ; //Read integer from file
```

Figure 1.29: How to write and read data?

1.18.1 C++code for illustrating stream output

```
1 /*-----  
2 All includes here  
3 -----*/  
4 #ifndef output_H  
5 #define output_H  
6  
7 #include "../util/util.h"  
8  
9 /*-----  
10 Declaration of output class  
11 -----*/  
12 class output{  
13 public:  
14     void default_output_of_6() ;  
15     void how_to_format_real_numbers() ;  
16     void how_to_display_integers_in_proper_base() ;  
17     void how_to_set_width_and_fill_character() ;  
18     void file_default_output_of_6() ;  
19     void how_to_alternate_screen_and_file_output();  
20 private:  
21  
22 };  
23  
24 #endif  
25 //EOF  
26
```

```
1 /*-
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy
3 file: output.cpp
4 -----
5
6 /*
7 This file has enumeration class definition
8 -----
9
10 /*
11 All includes here
12 -----
13 #include "output.h"
14
15 /*
16 Definition of routines of output class
17 -----
18
19 /*
20 Note that default precision for float or double is 6
21 double d = 56.00009 is printed as d = 56.0001 (TOTAL WIDTH IS 6)
22
23 Note that float f0 = 56 and float d0 = 89 is printed with no decimal point
24
25 n = 65 f = 56.87 d = 56.0001 ch = a b = 1
26 f0 = 56 d0 = 89
27
28 -----
29 void output::default_output_of_6() {
30     int n = 65 ;
31     float f0 = 56;
32     float f = 56.87f ;
33     float d0 = 89 ;
34     double d = 56.00009 ;
35     char ch = 'a' ;
36     bool b = true ;
37
38     cout << "n = " << n << " " ;
39     cout << "f = " << f << " " << " d = " << d ; //Total width (number.number) is 6
40     cout << " ch = " << ch << " b = " << b << endl ;
41     cout << "f0 = " << f0 << " d0 = " << d0 << endl ;
42     cout << "-----" << endl ;
43 }
44
45 /*
46 f = 56.86999893 d = 56.00009000
47 -----
48 f = 57. d = 56. f0 = 56. d0 = 89.
49
50 f = 56.87 d = 56.00 f0 = 56.00 d0 = 89.00
51
52 f = 56.87 d = 56 f0 = 56 d0 = 89
53
54 -----
55 void output::how_to_format_real_numbers() {
56     float f0 = 56;
57     float f = 56.87f ;
58     float d0 = 89 ;
59     double d = 56.00009 ;
60
61     cout.precision(10) ;//Total width (number.number) is 10
62     cout.setf(ios::showpoint) ;
63     cout << "f = " << f << " " << " d = " << d ;
64     cout << "\n-----" << endl ;
65
66     cout.precision(2) ; //Total width (number.number) is 2
```

```
67 cout.setf(ios::showpoint) ;
68 cout << "f = " << f << " " << " d = " << d ;
69 cout << " f0 = " << f0 << " d0 = " << d0 << endl ;
70 cout << "-----" << endl ;
71
72 cout.precision(4) ; //Total width (number.number) is 4
73 cout.setf(ios::showpoint) ;
74 cout << "f = " << f << " " << " d = " << d ;
75 cout << " f0 = " << f0 << " d0 = " << d0 << endl ;
76 cout << "-----" << endl ;
77
78 cout.unsetf(ios::showpoint) ;
79 cout << "f = " << f << " " << " d = " << d ;
80 cout << " f0 = " << f0 << " d0 = " << d0 << endl ;
81 cout << "-----" << endl ;
82 }
83
84 /-----
85 In Decimal x = 65
86 In Octal x = 0101
87 In Hex x = 0x41
88 -----
89 void output::how_to_display_integers_in_proper_base() {
90     int x = 65 ;
91     cout.setf(ios::showbase) ;
92
93     cout << "In Decimal x = " ;
94     cout << x << endl ;
95
96     cout << "In Octal x = " ;
97     cout << oct << x << endl ;
98
99     cout << "In Hex x = " ;
100    cout << hex << x << endl ;
101    cout << "-----" << endl ;
102    cout.unsetf(ios::showbase) ;
103    cout << dec ; //Set the state back to decimal
104 }
105
106 /-----
107 65
108 89
109 00065
110 ***89
111 80
112 -----
113 void output::how_to_set_width_and_fill_character() {
114     int x = 65 ;
115     int y = 89 ;
116     cout << setw(5) << x << endl ;
117     cout << setw(5) << y << endl ;
118
119     cout << setw(5) << setfill('0') << x << endl ;
120     cout << setw(5) << setfill('*') << y << endl ;
121
122     int z = 80 ;
123     cout << z << endl ;
124 }
125
126 /-----
127 Note that default precision for float or double is 6
128 double d = 56.00009 is printed as d = 56.0001 (TOTAL WIDTH IS 6)
129
130 Note that float f0 = 56 and float d0 = 89 is printed with no decimal point
131 -----
132 n = 65 f = 56.87 d = 56.0001 ch = a b = 1
```

```
133 f0 = 56 d0 = 89
134 -----
135 -----*/
136 void output::file_default_output_of_6() {
137 #ifdef WIN32
138   ofstream out("c:\\jag\\myoutput\\junk.dat") ;
139 #else
140   ofstream out("c:/jag/myoutput/junk.dat") ;
141 #endif
142   if (!out) {
143     cout << "Cannot open junk.dat for writing" << endl ;
144   }
145   int n = 65 ;
146   float f0 = 56;
147   float f = 56.87f ;
148   float d0 = 89 ;
149   double d = 56.00009 ;
150   char ch = 'a' ;
151   bool b = true ;
152
153   out << "n = " << n << " " ;
154   out << "f = " << f << " " << " d = " << d ; //Total width (number.number) is 6
155   out << " ch = " << ch << " b = " << b << endl ;
156   out << "f0 = " << f0 << " d0 = " << d0 << endl ;
157   out << "-----" << endl ;
158   out.flush() ;
159   out.close() ;
160 #ifdef WIN32
161   cout << "Answer is in: c:\\jag\\myoutput\\junk.dat\\n" ;
162 #else
163   cout << "Answer is in: c:/jag/myoutput/junk.dat\\n" ;
164 #endif
165 }
166
167 /*-
168 Writing to screen or to file
169 -----*/
170 void output::how_to_alternate_screen_and_file_output() {
171 #ifdef WIN32
172   ofstream fout("c:\\jag\\myoutput\\junk.dat") ;
173 #else
174   ofstream fout("c:/jag/myoutput/junk.dat") ;
175 #endif
176   if (!fout) {
177     cout << "Cannot open junk.dat for writing" << endl ;
178   }
179   for (int i = 0 ; i < 10; ++i) {
180     ostream* out = &cout ; //Even numbers on screen
181     if (i%2) {
182       out = &fout ; //Odd numbers in file
183     }
184     *out << "Writing number " << i << endl ;
185   }
186   fout.close() ;
187 #ifdef WIN32
188   cout << "Answer is in: c:\\jag\\myoutput\\junk.dat\\n" ;
189 #else
190   cout << "Answer is in: c:/jag/myoutput/junk.dat\\n" ;
191 #endif
192 }
193 //EOF
194
195
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: outputtest.cpp  
4  
5 On linux:  
6 g++ output.cpp outputtest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test output object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "output.h"  
19  
20 /*-----  
21 -----*/  
22 /*-----*/  
23 void test1() {  
24     output o ;  
25     o.default_output_of_6() ;  
26     o.how_to_format_real_numbers() ;  
27     o.how_to_display_integers_in_proper_base() ;  
28     o.how_to_set_width_and_fill_character() ;  
29     o.file_default_output_of_6();  
30     o.how_to_alternate_screen_and_file_output();  
31 }  
32  
33 /*-----  
34 test bed  
35 -----*/  
36 void testbed() {  
37     test1() ;  
38 }  
39  
40 /*-----  
41 main  
42 -----*/  
43 int main() {  
44     testbed() ;  
45     return 0 ;  
46 }  
47  
48 //EOF  
49
```

1.18.2 C++code for illustrating stream input

```
1 /*-----  
2 All includes here  
3 -----*/  
4 #ifndef input_H  
5 #define input_H  
6  
7 #include "../util/util.h"  
8  
9 /*-----  
10 Declaration of input class  
11 -----*/  
12 class input{  
13 public:  
14     void basic() ;  
15     void how_to_enter_in_loop();  
16     void how_to_check_error() ;  
17     void how_to_read_reliably();  
18     void how_to_read_from_file() ;  
19 private:  
20  
21 };  
22  
23 #endif  
24 //EOF  
25
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: input.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has enumeration class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "input.h"  
14  
15 /*-----  
16 Definition of routines of input class  
17 -----*/  
18  
19 /*-----  
20 You can space before or after.  
21 If U make error problem  
22 -----*/  
23 void input::basic()  
24 {  
25     cout << "Enter a number: " ;  
26     int x ;  
27     cin >> x ;  
28     cout << "You entered:" << x << endl ;  
29 }  
30  
31 {  
32     cout << "Enter a character: " ;  
33     char x ;  
34     cin >> x ;  
35     cout << "You entered:" << x << endl ;  
36 }  
37  
38 {  
39     cout << "Enter a float: " ;  
40     float x ;  
41     cin >> x ;  
42     cout << "You entered:" << x << endl ;  
43 }  
44  
45 {  
46     cout << "Enter a double: " ;  
47     double x ;  
48     cin >> x ;  
49     cout << "You entered:" << x << endl ;  
50 }  
51 }  
52  
53 /*-----  
54 enter a number: 8  
55 You entered:8  
56 enter a number: 78  
57 You entered:78  
58 enter a number: 678  
59 You entered:678  
60 enter a number: 6780  
61 You entered:6780  
62 enter a number: ^Z  
63  
64  
65 WINDOWS: <CONTROL>Z  
66 UNIX : <CONTROL>D
```

```
67 -----*/
68 void input::how_to_enter_in_loop() {
69     bool isend = false ;
70     do {
71         cout << "enter a number: " ;
72         int x ;
73         cin >> x ;
74         isend = cin.eof() ;
75         if (!isend) {
76             cout << "You entered:" << x << endl ;
77         }
78     }while (!isend) ;
79 }
80 */
81 <CONTROL>Z will set eof to true
82
83 enter a number: y
84 eof() = false
85 good = false
86 bad = false
87 fail = true
88 */
89 -----
90 void input::how_to_check_error() {
91     cout << "enter a number: " ;
92     int x ;
93     cin >> x ;
94
95     cout << boolalpha ; //displays boolean as true or false
96
97     cout << "eof() = " << (cin.eof()) << endl ;
98     cout << "good = " << (cin.good()) << endl ;
99     cout << "bad = " << (cin.bad()) << endl ;
100    cout << "fail = " << (cin.fail()) << endl ;
101
102    cout.unsetf(ios::boolalpha) ; //unset boolalpha
103 }
104
105 */
106
107 -----
108 void input::how_to_read_reliably() {
109     const INT MAX = 1024 ;
110     bool error = false ;
111     do {
112         error = false ;
113         cout << "Enter integer: " ;
114         char buffer[MAX] ;
115         cin.getline(buffer,MAX) ;
116         int i = 0 ;
117         int num = 0 ;
118         char ch = 'a' ;
119         do {
120             ch = buffer[i++] ;
121             if (ch != '\0') {
122                 if (ch >= '0' && ch <= '9') {
123                     num = num*10 + (ch - '0') ;
124                 }else {
125                     error = true ;
126                 }
127             }
128         }while (!error && (ch != '\0')) ;
129         if (error) {
130             cout << "You entered garbage:" << buffer << endl ;
131         }else {
132             cout << "Correct Input:" << num << endl ;
```

```
133     }
134   }while (error) ;
135 }
136
137
138 /*-----*
139 in.dat
140 34 a
141 67 h
142 78 j
143 90 k
144 89 k
145 -----*/
146 void input::how_to_read_from_file() {
147 #ifdef WIN32
148   ifstream in("c:\\jag\\myoutput\\in.dat") ;
149 #else
150   ifstream in("c:/jag/myoutput/in.dat") ;
151 }
152 #endif
153 if (!in) {
154   cout << "Cannot open in.dat for reading" << endl ;
155 }
156 do{
157   int n = -1;
158   char ch = 'z' ;
159   in >> n >> ch ;
160   if (!in.eof()) {
161     cout << "Read: " << n << " " << ch << endl ;
162   }else {
163     break ;
164   }
165 }while(1) ;
166 }
167
168 //EOF
169
170
171
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: inputtest.cpp  
4  
5 On linux:  
6 g++ input.cpp inputtest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test input object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "input.h"  
19  
20 /*-----  
21 -----*/  
22 /*-----*/  
23 void test1() {  
24     input i ;  
25     i.basic() ;  
26     i.how_to_enter_in_loop();  
27     i.how_to_check_error() ;  
28     i.how_to_read_reliably() ;  
29     i.how_to_read_from_file() ;  
30 }  
31  
32 /*-----  
33 test bed  
34 -----*/  
35 /*-----*/  
36 void testbed() {  
37     test1() ;  
38 }  
39  
40 /*-----  
41 main  
42 -----*/  
43 int main() {  
44     testbed() ;  
45     return 0 ;  
46 }  
47  
48 //EOF  
49
```

1.19 Problem set

Problem 1.19.1. Write a *class* called **temperature** that performs the following tasks:

- Write a public function that converts Fahrenheit temperature to Centigrade temperature using the equation

$$C = (5/9) * (F - 32)$$

- Write a public function that converts Centigrade temperature to Fahrenheit temperature using the equation

$$F = ((9/5) * T) + 32;$$

- Test your program from a Fahrenheit temperature of -50 degrees to +50 degrees, in a step of five degrees. Convert the centigrade answer you got to Fahrenheit temperature. At what temperature the Fahrenheit temperature matches with the Centigrade temperature.

Problem 1.19.2. Write a *class* called **p1** as described in Fig 1.30.

1.19. PROBLEM SET

class p1

<p>print_usa</p> <pre>XXX XXX XXXX X XXX X X X XX X X X X X X X X X X X X X X X XXX X X X X X XX X X X X X X X XXXXX X X X X X X X XX XX XX X X X X XXX X XXXX XXX XXX</pre>	<p>print_n_n2_n3</p> <pre>n n^2 n^3 1 1 1 2 4 8 3 9 27 4 16 64 5 25 125 6 36 216 7 49 343 8 64 512 9 81 729</pre>	<p>p1test.cpp</p> <pre>void testbed0 { p1 a ; a.print_usa(); a.print_n_n2_n3(); a.a_power_b(); a.two_power_n(); a.a1(1, 6); a.a2(1, 6); a.a3(1, 6); a.a4(1, 6); } start < end 0 to 9 only only one character int main() { testbed0 ; return 0 ; }</pre>																																																																																																																																																							
<p>a_power_b</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>a</th> <th>b</th> <th>a^b</th> </tr> </thead> <tbody> <tr><td>1</td><td>2</td><td>1</td></tr> <tr><td>2</td><td>3</td><td>8</td></tr> <tr><td>3</td><td>4</td><td>81</td></tr> <tr><td>4</td><td>5</td><td>1024</td></tr> <tr><td>5</td><td>6</td><td>15625</td></tr> <tr><td>6</td><td>7</td><td>279936</td></tr> <tr><td>7</td><td>8</td><td>5764801</td></tr> </tbody> </table>	a	b	a^b	1	2	1	2	3	8	3	4	81	4	5	1024	5	6	15625	6	7	279936	7	8	5764801	<p>two_power_n</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>n</th> <th>2^n</th> </tr> </thead> <tbody> <tr><td>0</td><td>1</td></tr> <tr><td>1</td><td>2</td></tr> <tr><td>2</td><td>4</td></tr> <tr><td>3</td><td>8</td></tr> <tr><td>4</td><td>16</td></tr> <tr><td>5</td><td>32</td></tr> <tr><td>6</td><td>64</td></tr> <tr><td>7</td><td>128</td></tr> <tr><td>8</td><td>256</td></tr> <tr><td>9</td><td>512</td></tr> <tr><td>10</td><td>1024</td></tr> <tr><td>11</td><td>2048</td></tr> <tr><td>12</td><td>4096</td></tr> <tr><td>13</td><td>8192</td></tr> <tr><td>14</td><td>16384</td></tr> <tr><td>15</td><td>32768</td></tr> <tr><td>16</td><td>65536</td></tr> <tr><td>17</td><td>131072</td></tr> <tr><td>18</td><td>262144</td></tr> <tr><td>19</td><td>524288</td></tr> <tr><td>20</td><td>1048576</td></tr> </tbody> </table>	n	2^n	0	1	1	2	2	4	3	8	4	16	5	32	6	64	7	128	8	256	9	512	10	1024	11	2048	12	4096	13	8192	14	16384	15	32768	16	65536	17	131072	18	262144	19	524288	20	1048576	<p>a1</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tbody> <tr><td>1</td><td>2</td></tr> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> </tbody> </table> <p>a2</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tbody> <tr><td>1</td></tr> <tr><td>2</td><td>1</td></tr> <tr><td>3</td><td>2</td><td>1</td></tr> <tr><td>4</td><td>3</td><td>2</td><td>1</td></tr> <tr><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td></tr> <tr><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td></tr> </tbody> </table> <p>a3</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tbody> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>1</td><td>2</td></tr> <tr><td>1</td></tr> </tbody> </table> <p>a4</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tbody> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td></tr> <tr><td>1</td><td>2</td><td>3</td></tr> <tr><td>1</td><td>2</td></tr> <tr><td>1</td></tr> </tbody> </table>	1	2	1	2	3	1	2	3	4	1	2	3	4	5	1	2	3	4	5	6	1	2	1	3	2	1	4	3	2	1	5	4	3	2	1	6	5	4	3	2	1	1	2	3	4	5	6	1	2	3	4	5	1	2	3	4	1	2	3	1	2	1	1	2	3	4	5	6	1	2	3	4	5	1	2	3	4	1	2	3	1	2	1
a	b	a^b																																																																																																																																																							
1	2	1																																																																																																																																																							
2	3	8																																																																																																																																																							
3	4	81																																																																																																																																																							
4	5	1024																																																																																																																																																							
5	6	15625																																																																																																																																																							
6	7	279936																																																																																																																																																							
7	8	5764801																																																																																																																																																							
n	2^n																																																																																																																																																								
0	1																																																																																																																																																								
1	2																																																																																																																																																								
2	4																																																																																																																																																								
3	8																																																																																																																																																								
4	16																																																																																																																																																								
5	32																																																																																																																																																								
6	64																																																																																																																																																								
7	128																																																																																																																																																								
8	256																																																																																																																																																								
9	512																																																																																																																																																								
10	1024																																																																																																																																																								
11	2048																																																																																																																																																								
12	4096																																																																																																																																																								
13	8192																																																																																																																																																								
14	16384																																																																																																																																																								
15	32768																																																																																																																																																								
16	65536																																																																																																																																																								
17	131072																																																																																																																																																								
18	262144																																																																																																																																																								
19	524288																																																																																																																																																								
20	1048576																																																																																																																																																								
1	2																																																																																																																																																								
1	2	3																																																																																																																																																							
1	2	3	4																																																																																																																																																						
1	2	3	4	5																																																																																																																																																					
1	2	3	4	5	6																																																																																																																																																				
1																																																																																																																																																									
2	1																																																																																																																																																								
3	2	1																																																																																																																																																							
4	3	2	1																																																																																																																																																						
5	4	3	2	1																																																																																																																																																					
6	5	4	3	2	1																																																																																																																																																				
1	2	3	4	5	6																																																																																																																																																				
1	2	3	4	5																																																																																																																																																					
1	2	3	4																																																																																																																																																						
1	2	3																																																																																																																																																							
1	2																																																																																																																																																								
1																																																																																																																																																									
1	2	3	4	5	6																																																																																																																																																				
1	2	3	4	5																																																																																																																																																					
1	2	3	4																																																																																																																																																						
1	2	3																																																																																																																																																							
1	2																																																																																																																																																								
1																																																																																																																																																									

Figure 1.30: Various functions of class P1

CHAPTER 1. INTRODUCTION

Problem 1.19.3. Write a *class* called **tax** for the **income tax** problem described in Fig 1.31.

1.19. PROBLEM SET

2010 Federal Income Tax Table			
If taxable income is:			
Married Filing Jointly:			
Over	But Not Over	The Tax Is	Of The Amount Over
\$0	\$16,750	\$0 + 10%	\$0
\$16,750	\$68,000	\$1,675 + 15%	\$16,750
\$68,000	\$137,300	\$9,362.50 + 25%	\$68,000
\$137,300	\$209,250	\$26,687.50 + 28%	\$137,300
\$209,250	\$373,650	\$46,833.50 + 33%	\$209,250
\$373,650	And Over	\$101,085.50 + 35%	\$373,650
Single:			
Over	But Not Over	The Tax Is	Of The Amount Over
\$0	\$8,375	\$0 + 10%	\$0
\$8,375	\$34,000	\$837.50 + 15%	\$8,375
\$34,000	\$82,400	\$4,681.25 + 25%	\$34,000
\$82,400	\$171,850	\$16,781.25 + 28%	\$82,400
\$171,850	\$373,650	\$41,827.25 + 33%	\$171,850
\$373,650	And Over	\$108,421.25 + 35%	\$373,650

```
void compute_tax(int salary, bool single) {
    int tax ;
    //Write code
    cout << "TAX for " << salary << " is = " << tax << endl ;
}
```

Run your program for atleast 10 different salaries.
Try to cover corner cases.

Figure 1.31: Income tax computation

CHAPTER 1. INTRODUCTION

Problem 1.19.4. Write a *class* called **collatz** for the **Collatz conjecture** described in Fig 1.32.

1.19. PROBLEM SET

Collatz conjecture

Consider the following operation on an arbitrary positive [integer](#):

- If the number is even, divide it by two.
- If the number is odd, triple it and add one.

$$f(n) = \begin{cases} n/2 & \text{if } n \text{ is even} \\ 3n + 1 & \text{if } n \text{ is odd} \end{cases}$$

For instance, starting with $n = 6$, one gets the sequence $6, 3, 10, 5, 16, 8, 4, 2, 1$.

Run your program for $n = 11$ and $n = 27$

Figure 1.32: Collatz conjecture

CHAPTER 1. INTRODUCTION

Problem 1.19.5. Write a *class* called **caesar** for the Caesar cipher described in Fig 1.33. You must write two files **caesar.h** and **caesar.cpp** and test your class using the provided file **caesartest.cpp**. Your program must work for all the cases provided in **caesartest.cpp**. You cannot change anything in the file **caesartest.cpp**. **email** only **caesar.h** and **caesar.cpp**. Also include screen shot of the program output as a pdf file.

```
1 /*-----
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy
3 file: caesartest.cpp
4
5 On linux:
6 g++ caesar.cpp caesartest.cpp
7 valgrind a.out
8 -----*/
9
10 /*-----
11 This file test caesar object
12 -----*/
13
14 /*-----
15 All includes here
16 -----*/
17 #include "caesar.h"
18
19 /*-----
20
21 -----*/
22 void test1(int k, const char* s) {
23     cout << "-----\n";
24     if (k < 0) {
25         cout << "Number of rotation cannot be negative\n";
26     }
27     else {
28         char eans[100];
29         char dans[100];
30         caesar a;
31         a.encrypt(k, s, eans);
32         a.decrypt(k, eans, dans);
33         cout << "Rotating by " << k << endl;
34         cout << "original string = " << s << endl;
35         cout << "Encrypted string = " << eans << endl;
36         cout << "Decrypted string = " << dans << endl;
37         assert(strcmp(s, dans) == 0);
38     }
39     cout << "-----\n";
40 }
41 /*-----
42 test bed
43 -----*/
44 void test2(const char* s) {
45     test1(0, s); //No rotation
46     test1(26, s); //No rotation
47     test1(1, s); //One rotation
48     test1(6, s); //six rotation
49     test1(13, s);
50     test1(-1, s);
51     test1(45, s);
52     test1(31, s);
53     test1(30, s);
54     test1(420, s);
55 }
56
57 /*-----
58 test bed
59 -----*/
60 void testbed() {
61     test2("abcdefghijklmnopqrstuvwxyz");
62     test2("ABCDEFGHIJKLMNOPQRSTUVWXYZ");
63     test2("Be sure to drink your Ovaltine3!");
64     test2("Pack My Box With Five Dozens LIQUOR jugs");
65     test2("I am going to fail YOU");
```

```
67 test2("A Quick BROWN fox jumps over a lazy DOG");
68 }
69
70 /*-----
71 main
72 -----*/
73 int main() {
74     testbed();
75     return 0;
76 }
77
78 //EOF
79
80
```

1.19. PROBLEM SET

Caesar cipher



In cryptography, a Caesar cipher, also known as Caesar's cipher, the shift cipher, Caesar's code, is one of the simplest and most widely known encryption techniques. It is a type of substitution cipher in which each letter in the plaintext is replaced by a letter some fixed number of positions down the alphabet.

For example, with a shift of 1, A would be replaced by B, B would become C, and so on. The method is named after Julius Caesar, who used it in his private correspondence

Example: With a shift of 13, k = 13,

original string = I am going to fail YOU

Encrypted string = V nz tbvat gb snvy LBH

Decrypted string = I am going to fail YOU

With k = 420,

original string = A Quick BROWN fox jumps over a lazy DOG

Encrypted string = E Uymgo FVSAR jsb nyrtw sziv e pedc HSK

Decrypted string = A Quick BROWN fox jumps over a lazy DOG

Your task is to write a class called:

```
class caesar{  
private:  
public:  
    void encrypt(int k, const char* s, char* ans);  
    void decrypt(int k, const char* s, char* ans);  
};
```

Number of rotation

You must write code in: caesar.h and caesar.cpp
and test your code using caesartest.cpp which is provided to you.

YOU CANNOT CHANGE ANYTHING IN caesartest.cpp.

E-mail caesar.h and caesar.cpp.

Figure 1.33: Caesar cipher

CHAPTER 1. INTRODUCTION

Problem 1.19.6. Write a *class* called **reverse1** as explained in Figure 1.34

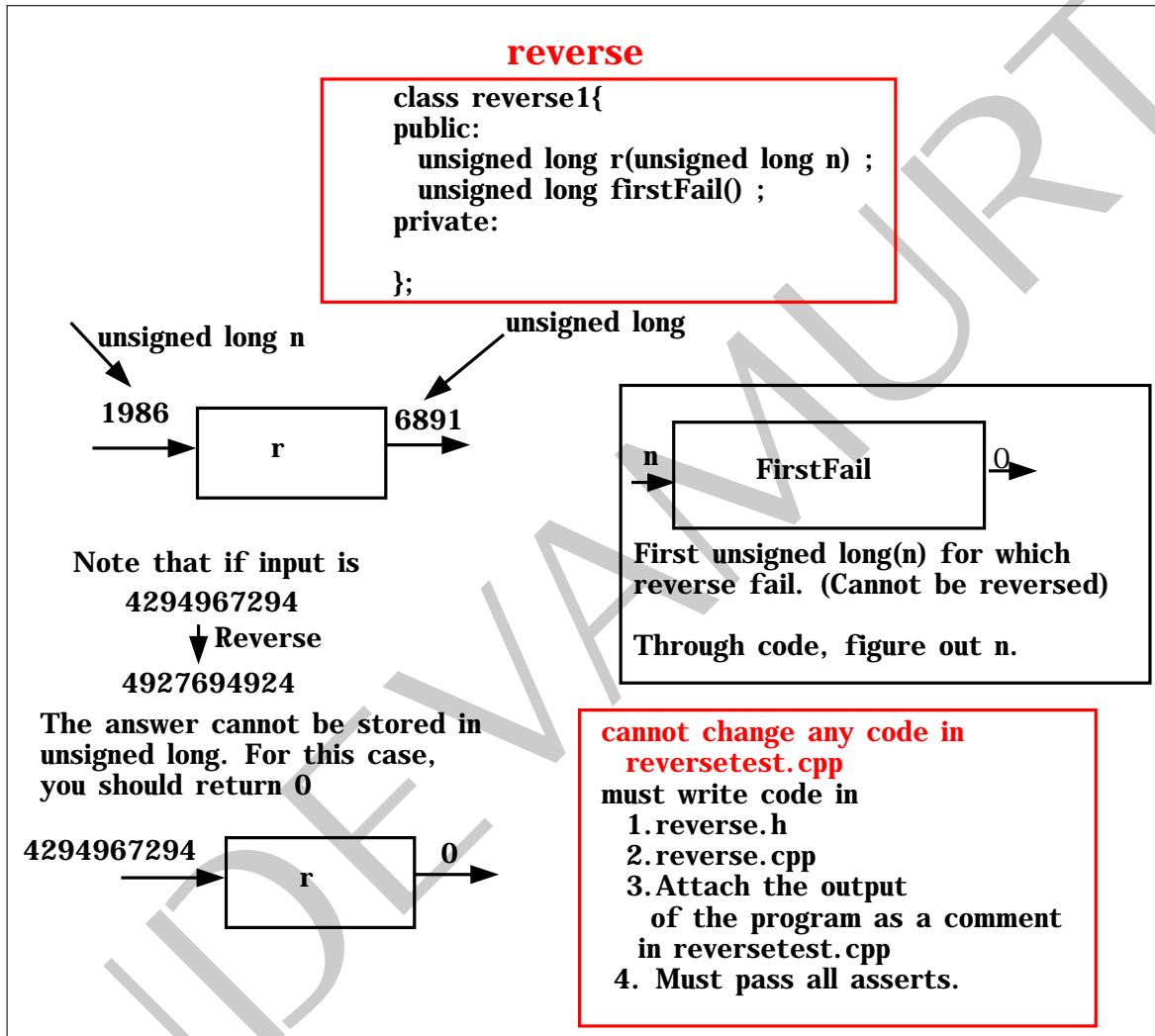


Figure 1.34: Reverse an *unsigned long* number

1.19. PROBLEM SET

Problem 1.19.7. Write a *class* called **luhn** for validating a credit card as described in Fig 1.35. You must write two files **luhn.h** and **luhn.cpp** and test your class using the provided file **luhntest.cpp**. Your program must work for all the cases provided in **luhntest.cpp**. You cannot change anything in the file **luhntest.cpp**. **email** only **luhn.h** and **luhn.cpp**. Also include screen shot of the program output as a pdf file.

Validating Credit Card Numbers Using LUHN Algorithm

4012888888881881

Step 1 - Starting with the check digit double the value of every other digit (right to left every 2nd digit)

4	0	1	2	8	8	8	8	8	8	8	1	8	8	1
x2	1													
8	2	16	16	16	16	16	2	16						

Step 2 - If doubling of a number results in a two digits number, add up the digits to get a single digit number. This will result in eight single digit numbers.

4	0	1	2	8	8	8	8	8	8	8	1	8	8	1
x2	1													
8	2	7	7	7	7	7	7	7	2	7	1			

Step 3 - Now add the un-doubled digits to the odd places

4	0	1	2	8	8	8	8	8	8	8	1	8	8	1
x2	1													
8	2	7	7	7	7	7	7	2	7	8	1			
0	2	8	8	8	8	8	8	8	8	8	1			

Step 4 - Add up all the digits in this number

4	0	1	2	8	8	8	8	8	8	8	1	8	8	1
x2	1													
8	2	7	7	7	7	7	7	2	7	8	1			
0	2	8	8	8	8	8	8	8	8	8	1			

8+0+2+7+8+7+8+7+8+7+2+8+7+1
= 90

If the final sum is divisible by 10, then the credit card number is valid. If it is not divisible by 10, the number is invalid.

Test Credit Card Account Numbers

Credit Card Type	Credit Card Number
American Express	378282246310005
American Express	371449635398431
American Express Corporate	378734493671000
Australian BankCard	5610591081018250
Diners Club	30569309025904
Diners Club	38520000023237
Discover	6011111111111117
Discover	6011000990139424
JCB	3530111333300000
JCB	3566002020360505
MasterCard	5555555555554444
MasterCard	5105105105105100
Visa	4111111111111111
Visa	401288888881881

Figure 1.35: Validating a credit card number using Luhn algorithm

```
1 /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 file: luhntest.cpp  
4  
5 On linux:  
6 g++ luhn.cpp luhntest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test luhn object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "luhn.h"  
19  
20 /*-----  
21 https://www.paypalobjects.com/en_US/vhelp/paypalmanager_help/credit_card_numbers.htm  
22 -----*/  
23 void test1() {  
24     static const long long list[] = { 378282246310005, 371449635398431, 378734493671000, 5610591081018250, ↵  
         30569309025904, 38520000023237, 60111111111117, 6011000990139424, 3530111333300000, ↵  
         5105105105100, 411111111111111, 401288888881881 };  
25     int s = sizeof(list) / sizeof(long long);  
26     for (int i = 0; i < s; ++i) {  
27         luhn a;  
28         assert(a.check(list[i]));  
29     }  
30 }  
31  
32 /*-----  
33 test bed  
34 -----*/  
35 void testbed() {  
36     luhn a;  
37     assert(a.check(401288888881881));  
38     assert(a.check(378282246310005));  
39     assert(a.check(401288888881880) == false);  
40     assert(a.check(4222222222222));  
41     test1();  
42 }  
43  
44 /*-----  
45 main  
46 -----*/  
47 int main() {  
48     testbed();  
49     return 0;  
50 }  
51  
52 //EOF  
53  
54
```

1.19. PROBLEM SET

Problem 1.19.8. Repeat the problem **luhn algorithm** shown in figure 1.19.7. Now, you cannot use any **array** in your file **luhn.cpp**. Create a new directory called **luhn1**. Copy **luhn.h**, **luhn.cpp** and **luhn1test.cpp** from the directory **luhn**. You cannot change anything in **luhn1test.cpp**. **luhn.h** must have **check** routine and any other helper functions, as private routines, to implement **check** routine. Rewrite **check** routine in **luhn.cpp** without using any **array**. That means you need to compute validity of the credit card without using temporary array. **email** only **luhn.cpp**. Also include screen shot of the program output as a pdf file.

Problem 1.19.9. Repeat the problem **collatz conjecture**, 1.19.4, in a new directory **collatz1**. Now your **collatz** procedure should compute the **collatz** sequence and store the results in an array. Write another procedure that takes the computed array and prints the **collatz** sequence. Test your program for numbers from zero to hundred. **email** **collatz.h**, **collatz.cpp** and **collatz1test.cpp**. Also include screen shot of the program output as a pdf file.

Problem 1.19.10. Write a *class* called **funnychar** that outputs the following figure:

```
#  
##  
###  
####  
#####  
######  
#######  
########  
#########  
##########  
###########
```

- Use only **for** statements. You can use only one

```
cout << '#'; as a print statement
```
- Use only **while** statements
- Use only **do while** statements
- Use only **if** statement and cannot have any loop statements like **for**, **while** and **do while**. Hint: Use **goto** statement.

Problem 1.19.11. What is the output of the program below. Can you prove why your answer is correct?

CHAPTER 1. INTRODUCTION

```
void problem5() {  
    int n = 10;  
    int sum = 0 ;  
    for (int i = 0 ; i < n ; i++)  
        for (int j = 0 ; j < i ; j++)  
            for (int k = 0 ; k < j ; k++)  
                sum++ ;  
    cout << "sum = " << sum << endl ;  
}
```

Write a *class* called **funnyloop** that does the following.

- Rewrite the program above, using only decrement operator
- Rewrite the program using only **while** statements
- Rewrite the program using only **do while** statements
- Use only **if** statement and cannot have any loop statements like **for**, **while** and **do while**.

Problem 1.19.12. Write a *class* called **array** as decribed in Fig 1.36.

1.19. PROBLEM SET

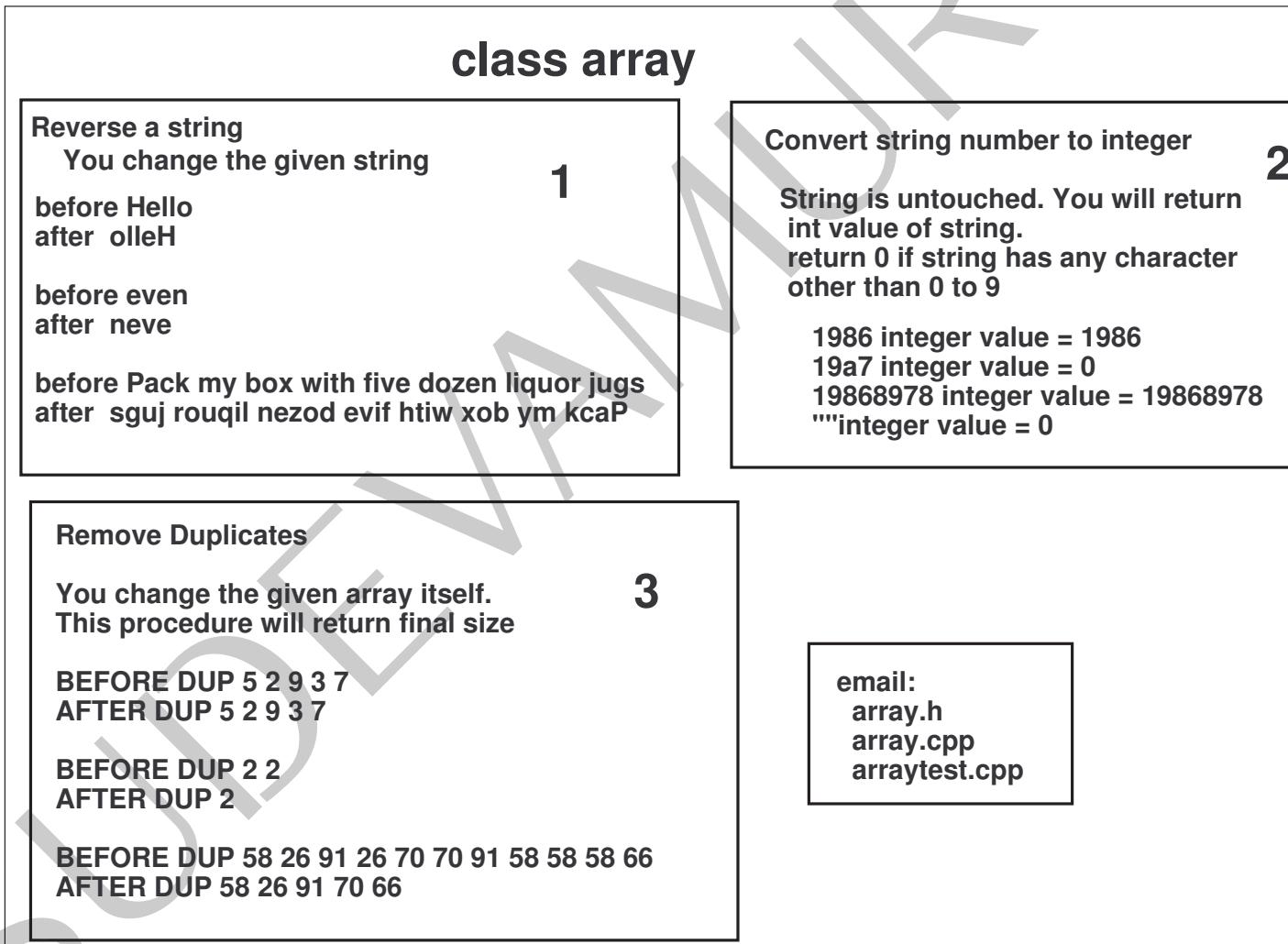


Figure 1.36: Various functions of class array

CHAPTER 1. INTRODUCTION

Problem 1.19.13. Write a *class* called **PerpetualCalendar** that can find the day of the events given in figure 1.37 using the chart **Perpetual Calendar**

Use perpetual calender to find the day of the following events

- October 12, 1493 - Christopher Columbus sights land in the Americas.
- May 29, 1493 - Constantinople falls. End of Byzantine empire.
- January 30, 1649 - King Charles I of England is executed,
- July 4, 1776 - Declaration of independence officially adopted. First modern
- July 14, 1789 - Storming of the Bastille. Turning point in the French Revolution.
- June 18, 1815 - Napoleon defeated at Battle of Waterloo.
- June 28, 1914 - Archduke Ferdinand assassinated.
- October 25, 1917 - Start of Bolshevik rebellion.
- June 26, 1919 - Treaty of Versailles signed
- March 23, 1933 - Hitler becomes dictator of Germany.
- August 6, 1945 - US drops atomic bomb on Hiroshima.
- October 22, 1960 - Cuban missile crisis.
- November 9, 1989 - Berlin wall falls.
- May 2, 2011 - Osama bin Laden killed by US Seals
- Adolf Hitler committed suicide by gunshot on 30 April 1945

Leap Day - Famous Birthdays

- 1960 - Richard Ramirez, American serial killer
- 1964 - Lyndon Byers, Canadian hockey player
- 1972 - Antonio Sabato Jr, Italian-born actor
- 1976 - Ja Rule, American rapper and actor
- 1980 - Chris Conley, American musician and songwriter/composer
- 1896 - Morarji Desai, former Indian prime minister
- 1924 - Carlos Humberto Romero, former president of El Salvador

Figure 1.37: What day of the week this events fall?

Perpetual Calendar

These charts enable you to find the day of the week in any year from 1755 through 2033.

Instructions

STEP 1

Find the year that you are interested in below and note the letter that follows it.

YEARS											
1789D	1824K	1859F	1894A	1929B	1964J	1999E					
1755C	1790E	1825F	1860N	1895B	1930C	1965E	2000M				
1756K	1791F	1826G	1861B	1896J	1931D	1966F	2001A				
1757F	1792N	1827A	1862C	1897E	1932L	1967G	2002B				
1758G	1793B	1828I	1863D	1898F	1933G	1968H	2003C				
1759A	1794C	1829D	1864L	1899G	1934A	1969C	2004K				
1760I	1795D	1830E	1865G	1900A	1935B	1970D	2005F				
1761D	1796L	1831P	1866A	1901B	1936J	1971E	2006G				
1762E	1797G	1832N	1867B	1902C	1937E	1972M	2007A				
1763F	1798A	1833B	1868J	1903D	1938F	1973A	2008I				
1764N	1799B	1834C	1869E	1904L	1939G	1974B	2009D				
1765B	1800C	1835D	1870F	1905G	1940H	1975C	2010E				
1766C	1801D	1836L	1871G	1906A	1941C	1976K	2011F				
1767D	1802E	1837G	1872H	1907B	1942D	1977F	2012N				
1768L	1803F	1838A	1873C	1908J	1943E	1978G	2013B				
1769G	1804N	1839B	1874D	1909E	1944M	1979A	2014C				
1770A	1805B	1840J	1875E	1910F	1945A	1980I	2015D				
1771B	1806C	1841E	1876M	1911G	1946B	1981D	2016L				
1772J	1807D	1842F	1877A	1912H	1947C	1982E	2017G				
1773E	1808L	1843G	1878B	1913C	1948K	1983F	2018A				
1774F	1809G	1844H	1879C	1914D	1949F	1984N	2019B				
1775G	1810A	1845C	1880K	1915E	1950G	1985B	2020J				
1776H	1811B	1846D	1881F	1916M	1951A	1986C	2021E				
1777C	1812J	1847E	1882G	1917A	1952I	1987D	2022F				
1778D	1813E	1848M	1883A	1918B	1953D	1988L	2023G				
1779E	1814F	1849A	1884I	1919C	1954E	1989G	2024H				
1780M	1815G	1850B	1885D	1920K	1955F	1990A	2025C				
1781A	1816H	1851C	1886E	1921F	1956N	1991B	2026D				
1782B	1817C	1852K	1887F	1922G	1957B	1992J	2027E				
1783C	1818D	1853F	1888N	1923A	1958C	1993E	2028M				
1784K	1819E	1854G	1889B	1924I	1959D	1994F	2029A				
1785F	1820M	1855A	1890C	1925D	1960L	1995G	2030B				
1786G	1821A	1856I	1891D	1926E	1961G	1996H	2031C				
1787A	1822B	1857D	1892L	1927F	1962A	1997C	2032K				
1788I	1823C	1858E	1893G	1928N	1963B	1998D	2033F				

STEP 2

Find that letter below and note which number falls under the month you are looking for.

MONTHS												
JAN	FEB	MAR	APR	MAY	JUN	JUL	AUG	SEP	OCT	NOV	DEC	
A	1	4	4	7	2	5	7	3	6	1	4	6
B	2	5	5	1	3	6	1	4	7	2	5	7
C	3	6	6	2	4	7	2	5	1	3	6	1
D	4	7	7	3	5	1	3	6	2	4	7	2
E	5	1	1	4	6	2	4	7	3	5	1	3
F	6	2	2	5	7	3	5	1	4	6	2	4
G	7	3	3	6	1	4	6	2	5	7	3	5
H	1	4	5	1	3	6	1	4	7	2	5	7
I	2	5	6	2	4	7	2	5	1	3	6	1
J	3	6	7	3	5	1	3	6	2	4	7	2
K	4	7	1	4	6	2	4	7	3	5	1	3
L	5	1	2	5	7	3	5	1	4	6	2	4
M	6	2	3	6	1	4	6	2	5	7	3	5
N	7	3	4	7	2	5	7	3	6	1	4	6

STEP 3

Use the calendar that bears the number you found in Step 2.

CALENDARS											
1	2	3	4	5	6	7					
Monday		1									
Tuesday		2	1								
Wednesday		3	2	1							
Thursday		4	3	2	1						
Friday		5	4	3	2	1					
Saturday		6	5	4	3	2	1				
SUNDAY		7	6	5	4	3	2	1			
Monday		8	7	6	5	4	3	2	1		
Tuesday		9	8	7	6	5	4	3	2	1	
Wednesday		10	9	8	7	6	5	4	3	2	1
Thursday		11	10	9	8	7	6	5	4	3	2
Friday		12	11	10	9	8	7	6	5	4	3
Saturday		13	12	11	10	9	8	7	6	5	4
SUNDAY		14	13	12	11	10	9	8	7	6	5
Monday		15	14	13	12	11	10	9	8	7	6
Tuesday		16	15	14	13	12	11	10	9	8	7
Wednesday		17	16	15	14	13	12	11	10	9	8
Thursday		18	17	16	15	14	13	12	11	10	9
Friday		19	18	17	16	15	14	13	12	11	10
Saturday		20	19	18	17	16	15	14	13	12	11
SUNDAY		21	20	19	18	17	16	15	14	13	12
Monday		22	21	20	19	18	17	16	15	14	13
Tuesday		23	22	21	20	19	18	17	16	15	14
Wednesday		24	23	22	21	20	19	18	17	16	15
Thursday		25	24	23	22	21	20	19	18	17	16
Friday		26	25	24	23	22	21	20	19	18	17
Saturday		27	26	25	24	23	22	21	20	19	18
SUNDAY		28	27	26	25	24	23	22	21	20	19
Monday		29	28	27	26	25	24	23	22	21	20
Tuesday		30	29	28	27	26	25	24	23	22	21
Wednesday		31	30	29	28	27	26	25	24	23	22
Thursday		31	30	29	28	27	26	25	24	23	22
Friday		31	30	29	28	27	26	25	24	23	22
Saturday		31	30	29	28	27	26	25	24	23	22
SUNDAY		31	30	29	28	27	26	25	24	23	22
Monday		29	28	27	26	25	24	23	22	21	20
Tuesday		30	29	28	27	26	25	24	23	22	21

1.19. PROBLEM SET

For clarity, the alphabet associated with the years 1755 through 2033, is repeated below.

1755	C	1756	K	1757	F	1758	G	1759	A	1760	I	1761	D	1762	E	1763	F
1764	N	1765	B	1766	C	1767	D	1768	L	1769	G	1770	A	1771	B	1772	J
1773	E	1774	F	1775	G	1776	H	1777	C	1778	D	1779	E	1780	M	1781	A
1782	B	1783	C	1784	K	1785	F	1786	G	1787	A	1788	I	1789	D	1790	E
1791	F	1792	N	1793	B	1794	C	1795	D	1796	K	1797	G	1798	A	1799	B
1800	C	1801	D	1802	E	1803	F	1804	N	1805	B	1806	C	1807	D	1808	L
1809	G	1810	A	1811	B	1812	J	1813	E	1814	F	1815	G	1816	H	1817	C
1818	D	1819	E	1820	M	1821	A	1822	B	1823	C	1824	K	1825	F	1826	G
1827	A	1828	I	1829	D	1830	E	1831	F	1832	N	1833	B	1834	C	1835	D
1836	L	1837	G	1838	A	1839	B	1840	J	1841	E	1842	F	1843	G	1844	H
1845	C	1846	D	1847	E	1848	M	1849	A	1850	B	1851	C	1852	K	1853	F
1854	G	1855	A	1856	I	1857	D	1858	E	1859	F	1860	N	1861	B	1862	C
1863	D	1864	L	1865	G	1866	A	1867	B	1868	J	1869	E	1870	F	1871	G
1872	H	1873	C	1874	D	1875	E	1876	M	1877	A	1878	B	1879	C	1880	K
1881	F	1882	G	1883	A	1884	I	1885	D	1886	E	1887	F	1888	N	1889	B
1890	C	1891	D	1892	L	1893	G	1894	A	1895	B	1896	J	1897	E	1898	F
1899	G	1900	A	1901	B	1902	C	1903	D	1904	L	1905	G	1906	A	1907	B
1908	J	1909	E	1910	F	1911	G	1912	H	1913	C	1914	D	1915	E	1916	M
1917	A	1918	B	1919	C	1920	K	1921	F	1922	G	1923	A	1924	I	1925	D
1926	E	1927	F	1928	N	1929	B	1930	C	1931	D	1932	L	1933	G	1934	A
1935	B	1936	J	1937	E	1938	F	1939	G	1940	H	1941	C	1942	D	1943	E
1944	M	1945	A	1946	B	1947	C	1948	K	1949	F	1950	G	1951	A	1952	I
1953	D	1954	E	1955	F	1956	N	1957	B	1958	C	1959	D	1960	L	1961	G
1962	A	1963	B	1964	J	1965	E	1966	F	1967	G	1968	H	1969	C	1970	D
1971	E	1972	M	1973	A	1974	B	1975	C	1976	K	1977	F	1978	G	1979	A
1980	I	1981	D	1982	E	1983	F	1984	N	1985	B	1986	C	1987	D	1988	L
1989	G	1990	A	1991	B	1992	J	1993	E	1994	F	1995	G	1996	H	1997	C
1998	D	1999	E	2000	M	2001	A	2002	B	2003	C	2004	K	2005	F	2006	G
2007	A	2008	I	2009	D	2010	E	2011	F	2012	N	2013	B	2014	C	2015	D
2016	L	2017	G	2018	A	2019	B	2020	J	2021	E	2022	F	2023	G	2024	H
2025	C	2026	D	2027	E	2028	M	2029	A	2030	B	2031	C	2032	K	2033	F

CHAPTER 1. INTRODUCTION

Problem 1.19.14. Write a *class* called **ninetynine** for printing a poem as described in Fig 1.38. You must write three files **ninetynine.h** and **ninetynine.cpp** and test your class using the file **ninetyninetest.cpp**. Also include screen shot of the program output as a pdf file.

lyrics for: "Ninety Nine Bottles of Beer on the Wall"

ninety nine bottles of beer on the wall,
ninety nine bottles of beer,
Take one down, pass it around,
ninety eight bottles of beer on the wall,

ninety eight bottles of beer on the wall,
ninety eight bottles of beer,
Take one down, pass it around,
ninety seven bottles of beer on the wall,

↓

ninety one bottles of beer on the wall,
ninety one bottles of beer,
Take one down, pass it around,
ninety bottles of beer on the wall,

ninety bottles of beer on the wall,
ninety bottles of beer,
Take one down, pass it around,
eighty nine bottles of beer on the wall,

↓

two bottles of beer on the wall,
two bottles of beer,
Take one down, pass it around,
one bottle of beer on the wall,

Plural

one bottle of beer on the wall,
one bottle of beer,
Take one down, pass it around,
zero bottle of beer on the wall,

Singular

**Your program should not use
ninety nine different output statements!**

**Your program should print the number
of bottles in English, not as a number.**

email 3 files:

ninetynine.h
ninetynine.cpp
ninetyninetest.cpp

Figure 1.38: Automatically print the lyrics of a famous song

VASUDEVAMURTHY

Chapter 2

C Strings

2.1 Introduction

2.2 Character manipulation

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: character.h  
4 -----*/  
5  
6 /*-----  
7 This file has character class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef character_H  
14 #define character_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of character class  
20 -----*/  
21 class character{  
22 public:  
23     static bool isdigit(const char s);  
24     static bool isalpha(const char s);  
25     static bool isalnum(const char s);  
26     static bool is_ascii(const char s);  
27     static bool isupper(const char s);  
28     static bool islower(const char s);  
29     static char toupper(char s);  
30     static char tolower(char s);  
31 private:  
32 };  
33  
34  
35 #endif  
36 //EOF  
37  
38
```

CHAPTER 2. C STRINGS

<code>isalnum</code>	test for alphanumeric character
<code>isalpha</code>	test for alphabetic character
<code>isblank</code>	test for blank character (new in C99)
<code>iscntrl</code>	test for control character
<code>isdigit</code>	test for digit. Not locale-specific.
<code>isgraph</code>	test for graphic character, excluding the space character.
<code>islower</code>	test for lowercase character
<code>isprint</code>	test for printable character , including the space character.
<code>ispunct</code>	test for punctuation character
<code>isspace</code>	test for any whitespace character
<code>isupper</code>	test for uppercase character
<code>isxdigit</code>	test for hexadecimal digit. Not locale-specific.
<code>tolower</code>	convert character to lowercase
<code>toupper</code>	convert character to uppercase

ctype.h

```

int isalnum(int c);
    isalpha(c) or isdigit(c)
int isalpha(int c);
    isupper(c) or islower(c)
int iscntrl(int c);
    is control character. In ASCII, control characters are 0x00 (NUL) to 0x1F (US), and 0x7F (DEL)
int isdigit(int c);
    is decimal digit
int isgraph(int c);
    is printing character other than space
int islower(int c);
    is lower-case letter
int isprint(int c);
    is printing character (including space). In ASCII, printing characters are 0x20 (' ') to 0x7E ('~')
int ispunct(int c);
    is printing character other than space, letter, digit
int isspace(int c);
    is space, formfeed, newline, carriage return, tab, vertical tab
int isupper(int c);
    is upper-case letter
int isxdigit(int c);
    is hexadecimal digit
int tolower(int c);
    return lower-case equivalent
int toupper(int c);
    return upper-case equivalent

```

Figure 2.1: Contents of ctype.h

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: character.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has enumeration class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "character.h"  
14  
15 /*-----  
16 Definition of routines of character class  
17 -----*/  
18  
19 /*-----  
20 Check if s is a digit  
21 -----*/  
22 bool character::isdigit(const char s) {  
23     return (s >= '0' && s <= '9') ? true : false ;  
24 }  
25  
26 /*-----  
27 Check if s is a character  
28 -----*/  
29 bool character::isalpha(const char s) {  
30     return ((s >= 'a' && s <= 'z') || (s >= 'A' && s <= 'Z')) ? true : false ;  
31 }  
32  
33 /*-----  
34 Check if character is a digit or a alphabetic character  
35 -----*/  
36 bool character::isalnum(const char s) {  
37     return (isdigit(s) || isalpha(s)) ? true : false ;  
38 }  
39  
40 /*-----  
41 Check if character s is ascii  
42 -----*/  
43 bool character::is_ascii(const char s) {  
44     unsigned short u = s ;  
45     return (u >= 0 && u <=127) ? true : false ;  
46 }  
47  
48 /*-----  
49 Check if character is uppercase letter  
50 -----*/  
51 bool character::isupper(const char s) {  
52     return (s >= 'A' && s <= 'Z') ? true : false ;  
53 }  
54  
55 /*-----  
56 Check if character is lowercase letter  
57 -----*/  
58 bool character::islower(const char s) {  
59     return (s >= 'a' && s <= 'z') ? true : false ;  
60 }  
61  
62 /*-----  
63 convert s to uppercase and return upper case equivalent  
64 -----*/  
65 char character::toupper(char s) {  
66     if (islower(s)) {
```

```
67     s = (s - 'a') + 'A' ;
68 }
69 return s ;
70 }
71
72 /*-----*
73 convert s to lowercase and return lower case equivalent
74 -----*/
75 char character::tolower(char s) {
76     if (isupper(s)) {
77         s = (s - 'A') + 'a' ;
78     }
79     return s ;
80 }
81
82 //EOF
83
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: charaktertest.cpp  
4  
5 On linux:  
6 g++ character.cpp charaktertest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test character object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "character.h"  
19  
20 /*-----  
21 2. ch d = 5 is NOT an alphabet  
22 3. ch d = 5 is an alpha numeric character  
23 4. ch P = P is a upper case character  
24 5. ch u = u is NOT a upper case character  
25 6. ch u = u is a lower case character  
26 8. ch u = U After converting to upper U  
27 9. ch u = U After converting to upper U  
28 P = P  
29 10.ch P = p After converting to lower p  
30 11.ch P = p After converting to lower p  
31 ha ha it is ascii  
32 -----*/  
33 void test1() {  
34     char d = '5' ;  
35     char P = 'P' ;  
36     char u = 'u' ;  
37  
38     if (!(character::isalpha(u))) {  
39         cout << "1. ch u is an alphabet\n" ;  
40     }  
41     if (!(character::isalpha(d))) {  
42         cout << "2. ch d = " << d << " is NOT an alphabet\n" ;  
43     }  
44     if (character::isalnum(d)) {  
45         cout << "3. ch d = " << d << " is an alpha numeric character\n" ;  
46     }  
47  
48     if (character::isupper(P)) {  
49         cout << "4. ch P = " << P << " is a upper case character\n" ;  
50     }  
51     if (!(character::isupper(u))) {  
52         cout << "5. ch u = " << u << " is NOT a upper case character\n" ;  
53     }  
54  
55     if (character::islower(u)) {  
56         cout << "6. ch u = " << u << " is a lower case character\n" ;  
57     }  
58     if (!(character::islower(u))) {  
59         cout << "7. ch u = " << u << " is NOT a lower case character\n" ;  
60     }  
61  
62     cout << "8. ch u = " << u << " After converting to upper " << (u = character::toupper(u)) << endl ;  
63     cout << "9. ch u = " << u << " After converting to upper " << (u = character::toupper(u)) << endl ;  
64     cout << "P = " << P << endl ;  
65     cout << "10.ch P = " << P << " After converting to lower " << (P = character::tolower(P)) << endl ;  
66     cout << "11.ch P = " << P << " After converting to lower " << (P = character::tolower(P)) << endl ;
```

```
67  if (character::is_ascii(u)) {  
68      cout << "ha ha it is ascii\n";  
69  }  
70 }  
71 /*-----  
73 test bed  
74 -----*/  
75 void testbed() {  
76     test1();  
77 }  
78 /*-----  
80 main  
81 -----*/  
82 int main() {  
83     testbed();  
84     return 0;  
85 }  
86 //EOF  
88  
89
```

2.3 C Strings

```
1  /*-----  
2 Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3 file: carray.h  
4 -----*/  
5  
6 /*-----  
7 This file has carray class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef carray_H  
14 #define carray_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of carray class  
20 -----*/  
21 class carray{  
22 public:  
23     //length  
24     static int strlen(const char* s) ;  
25     static int intlen(int i) ;  
26  
27     //Conversion  
28     static int atoi(const char* s) ;  
29     static void itoa(int i, char* s) ;  
30  
31     //compare  
32     static int strcmp(const char* s1, const char* s2) ;  
33     static int strncmp(const char* s1, const char* s2,int n) ;  
34     static int strcasecmp(const char* s1, const char* s2) ;  
35     static int strcasencmp(const char* s1, const char* s2,int n) ;  
36  
37     //copy  
38     static void strcpy(char* dest, const char* src) ;  
39     static void strncpy(char* dest, const char* src,int n) ;  
40  
41     //concatenation  
42     static void strcat(char* dest, const char* src) ;  
43     static void strncat(char* dest, const char* src,int n) ;  
44     //find  
45     static const char* strstr(const char* s, const char* find) ;  
46 private:  
47  
48 };  
49
```

```
1  /*
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3  file: carray.cpp
4  */
5
6  /*
7  This file has enumeration class definition
8  */
9
10 /*
11 All includes here
12 */
13 #include "carray.h"
14
15 /*
16 Definition of routines of carray class
17 */
18
19 /*
20 Give length of the string excluding NULL
21 01234
22 s = jag\0 -- U require 4 memory location to store
23 return 3 ;
24 length does *NOT* include '\0'
25 */
26 int carray::strlen(const char* s) {
27     int l = 0 ;
28     while (*s) {
29         l++ ;
30         s++ ;
31     }
32     return l ;
33 }
34
35 /*
36 4 -> 1
37 -89 -> 3
38
39 */
40 int carray::intlen(int i) {
41     int l = 0 ;
42     cout << "WRITE THE CODE\n" ;
43     return l ;
44 }
45
46 /*
47 "73" -> 73
48 "-4" -> -4
49 "7a" -> 0
```

```
50  "--> 0
51 When atoi encounters a string with non numerical sequence,
52 it returns zero (0).
53 -----
54 int carray::atoi(const char* s) {
55     int l = 0 ;
56     bool minus = false ;
57     if (s) {
58         if (*s == '-') {
59             minus = true ;
60             s++ ;
61         }else {
62             //do not increment s
63         }
64     }
65     if (s) { //the input can be only s = "-"
66         do {
67             if (*s >= '0' && *s <= '9') {
68                 l = l * 10 + (*s - '0') ;
69                 s++ ;
70             }else {
71                 return 0 ;
72             }
73         }while (*s) ;
74         return (minus) ? -l : l ;
75     }
76 }
77
78 /**
79 Converts
80 79 to "79\0"
81 8a to "\0"
82 -78 to "-78\0"
83 It is assumed that user gives s array which is >= length of i + 1
84 */
85 void carray::itoa(int i, char* s) {
86     cout << "WRITE THE CODE\n" ;
87 }
88
89 /**
90 Compares the two strings s1 and s2,
91 case sensitive: a and A are different.
92
93 0 : (s1 == s2)    abc = abc
94 - : (s1 < s2)    ABC < abc
95 + : (s1 > s2)    abc > ABC
96 */
97 int carray::strcmp(const char* s1, const char* s2) {
98     while (s1 && s2) {
```

C:\work\software\course\objects\carray\carray.cpp

```

99     if (*s1 == *s2) {
100         if (*s1 == '\0') { //that means *s1 = \0 && *s2 = \0
101             return 0 ;
102         }
103         s1++ ;
104         s2++ ;
105     }else {
106         return (*s1 - *s2) ;
107     }
108 }
109 return 0 ;
110 }
111
112 /*-----
113 Compares first n character of s1 with s2
114 case sensitive: a and A are different.
115
116 0 : (s1 == s2)
117 - : (s1 < s2)
118 + : (s1 > s2)
119 -----*/
120 int carray::strncmp(const char* s1, const char* s2, int n) {
121     cout << "WRITE THE CODE\n" ;
122
123     return 0 ;
124 }
125
126 /*-----
127 Compares the two strings s1 and s2,
128 ignoring the case of the characters.
129
130 0 : (s1 == s2)  abc = ABC
131 - : (s1 < s2)   ab < xb
132 + : (s1 > s2)   xb > ab
133 -----*/
134 int carray::strcasecmp(const char* s1, const char* s2) {
135     while (s1 && s2) {
136         char a = *s1 ;
137         a = tolower(a) ;
138         char b = *s2 ;
139         b = tolower(b) ;
140         if (a == b) {
141             if (a == '\0') { //that means *s1 = \0 && *s2 = \0
142                 return 0 ;
143             }
144             s1++ ;
145             s2++ ;
146         }else {
147             return (a - b) ;

```

C:\work\software\course\objects\carray\carray.cpp

```
148     }
149 }
150     return 0 ;
151 }
152
153 /*-----
154 Compares first n character of s1 with s2
155 ignoring the case of the characters.
156
157 0 : (s1 == s2)
158 - : (s1 < s2)
159 + : (s1 > s2)
160 -----*/
161 int carray::strcasncmp(const char* s1, const char* s2, int n) {
162     cout << "WRITE THE CODE\n" ;
163
164     return 0 ;
165 }
166
167 /*-----
168 src = "jag"
169 dest = "ucsc" ;
170
171 dest = "jag"
172
173 To avoid overflows, the size of the array pointed by destination
174 shall be long enough to contain the same C string as source
175 (including the terminating null character), and
176 should not overlap in memory with source.
177 -----*/
178 void carray::strcpy(char* dest, const char* src) {
179     int i = 0 ;
180     char ch = 'a' ;
181     do {
182         ch = src[i] ;
183         dest[i] = ch ;
184         ++i ;
185     }while (ch) ;
186 }
187
188 /*-----
189 same as strcpy, but first n characters of src is copied to dest
190 -----*/
191 void carray::strncpy(char* dest, const char* src, int n) {
192     cout << "WRITE THE CODE\n" ;
193
194 }
195
196 /*-----
```

C:\work\software\course\objects\carray\carraytest.cpp

```
1  /*
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3  file: carraytest.cpp
4
5  On linux:
6  g++ carray.cpp carraytest.cpp
7  valgrind a.out
8
9  */
10 /*
11 This file test carray object
12 */
13 /*
14 All includes here
15 */
16 /*
17 */
18 #include "carray.h"
19
20 /*
21 */
22 /*
23 void test1() {
24     cout << "1. length(\"jag\") = " << carray::strlen("jag") << endl ;
25     cout << "2. length(\"jag\\\") = " << carray::strlen("jag\\") << endl ;
26     cout << "3. length(\"\") = " << carray::strlen("") << endl ;
27     cout << "4. atoi(\"8602\") = " << carray::atoi("8602") << endl ;
28     cout << "5. atoi(\"86z2\") = " << carray::atoi("86z2") << endl ;
29     cout << "6. atoi(\"-8052\") = " << carray::atoi("-8052") << endl ;
30     cout << "7. atoi(\"-\") = " << carray::atoi("-") << endl ;
31     cout << "8. atoi(\"-7\") = " << carray::atoi("-7") << endl ;
32     cout << "9. atoi(\"7-6\") = " << carray::atoi("7-6") << endl ;
33     cout << "10. strcmp(abc,abc) = " << carray::strcmp("abc","abc") << endl ;
34     cout << "11. strcmp(ABC,abc) = " << carray::strcmp("ABC","abc") << endl ;
35     cout << "12. strcmp(abc,ABC) = " << carray::strcmp("abc","ABC") << endl ;
36     cout << "13. strcmp(\"\",\"\") = " << carray::strcmp("", "") << endl ;
37     cout << "14. strcmp(\"\",ABC) = " << carray::strcmp("", "ABC") << endl ;
38     cout << "15. strcmp(ABC,\"\") = " << carray::strcmp("ABC", "") << endl ;
39     cout << "16. strcasecmp(abc,ABC) = " << carray::strcasecmp("abc","ABC") << endl ;
40     cout << "17. strcmp(ab,xb) = " << carray::strcasecmp("ab","xb") << endl ;
41     cout << "18. strcmp(xb,ab) = " << carray::strcasecmp("xb","ab") << endl ;
42
43
44     /* Write test program to test all other functions */
45 }
46
47 /*
48 test bed
```

2.4 Problem set

Problem 2.4.1. Fill the code in the program `carray.cpp`. Write test programs in `carraytest.cpp` routine to test your code.

Problem 2.4.2. This is an assignment problem for my 3rd grade daughter at Tom Matsumoto Elementary School, San Jose, CA.

Problem: The cost of each letter is given in the table below.
Find a SINGLE LEGEAL WORD (that must be in a dictionary) such that the total cost of the word is EXACTLY 25\$.

Example:

```
begin 5.17
beginner 7.42
beginners 7.57
```

The cost of each alphabet is as follows:

a = 1.00\$
b = 0.01\$
c = 0.02\$
d = 0.03\$
e = 2.00\$
f = 0.04\$
g = 0.05\$
h = 0.06\$
i = 3.00\$
j = 0.07\$
k = 0.08\$
l = 0.09\$
m = 0.10\$
n = 0.11\$
o = 4.00\$
p = 0.12\$
q = 0.13\$
r = 0.14\$
s = 0.15\$
t = 0.16\$
u = 5.00\$
v = 0.17\$

2.4. PROBLEM SET

w = 0.18\$

x = 0.19\$

y = 6.00\$

z = 0.20\$

Write a *class* called **TomMatsumoto** that can print words that satisfies the above requirements.

Chapter 3

Concept of reference

3.1 Introduction

3.1. INTRODUCTION

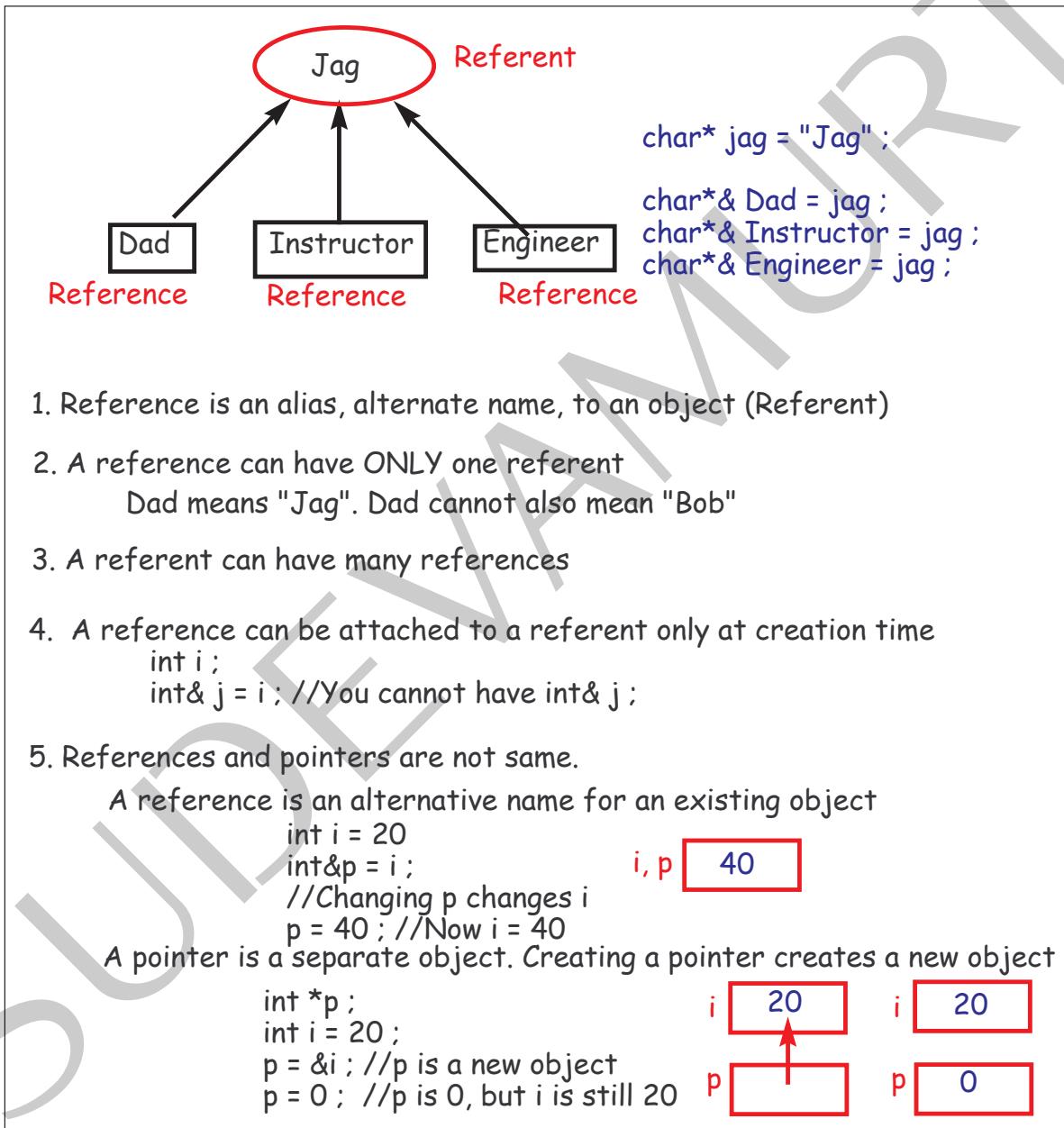


Figure 3.1: Alias and pointers

3.2 Concept of swapping using reference

3.2. CONCEPT OF SWAPPING USING REFERENCE

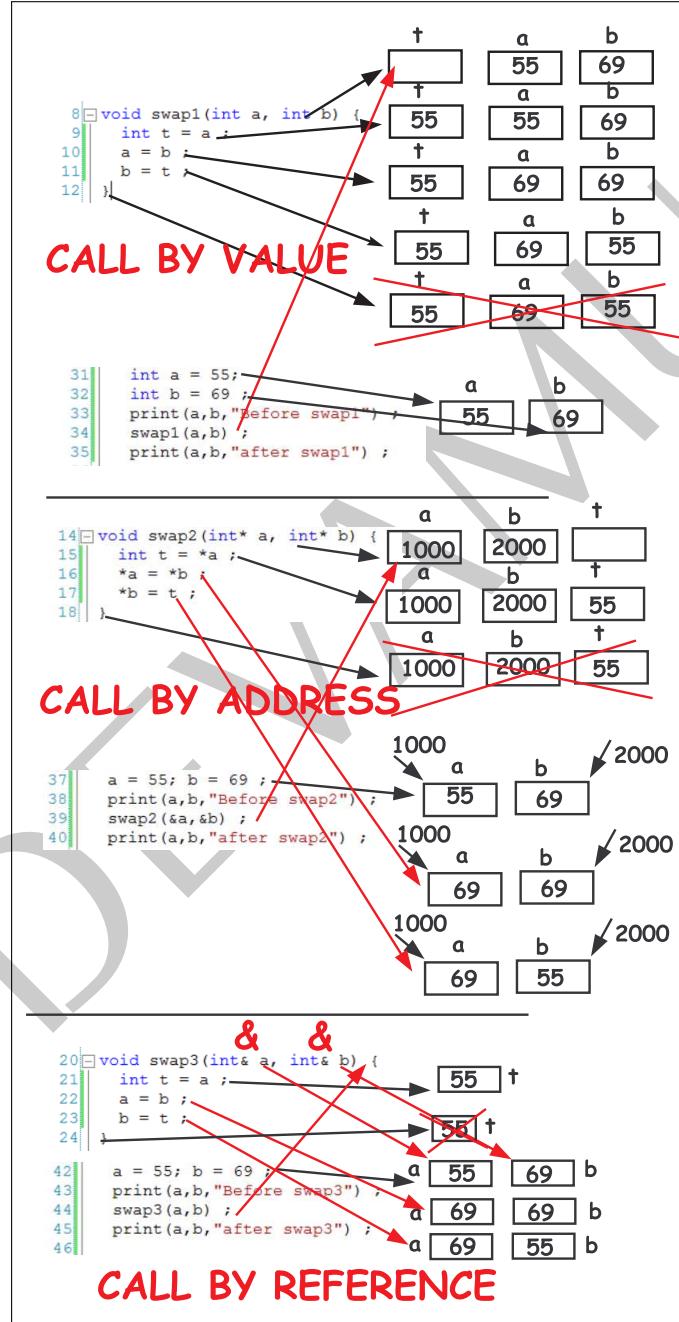


Figure 3.2: Concept of call by value, address and reference

3.3 FAQ on references

https://www.hasustorm.com/books/English/Addison.Wesley.Cpp.FAQs.2nd.Edition.internal.eBook-LiB.chm/0201309831_ch11lev1sec4.html

FAQ 11.04 What happens when a value is assigned to a reference?

The reference remains bound to the same referent, and the value of the referent is changed.

```
void f(int& x, int* y, int z){  
    x = 5; //main()'s i changed to 5  
    *y = 6; //main()'s i changed to 6  
    z = 7; //no change to main()'s i  
}  
  
int main()  
{  
    int i = 4;  
    f(i, &i, i);  
}
```

Figure 3.3: FAQ 1

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: reference.h  
4 -----*/  
5  
6 /*-----  
7 This file has reference class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef reference_H  
14 #define reference_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of reference class  
20 -----*/  
21 class reference{  
22 public:  
23     void test_swap();  
24     void test_faq();  
25     void test_return_by_reference();  
26  
27 private:  
28     void _print(const int a, const int b, const char *s);  
29     void _call_by_value(int a, int b);  
30     void _call_by_address(int* a, int* b);  
31     void _call_by_reference(int& a, int& b);  
32     void _f(int& x, int* y, int z);  
33     void _faq1();  
34     void _faq2();  
35     void _faq4();  
36     int& _f2();  
37  
38  
39 };  
40  
41  
42 #endif  
43 //EOF  
44  
45  
46
```

CHAPTER 3. CONCEPT OF REFERENCE

https://www.hasustorm.com/books/English/Addison.Wesley.Cpp.FAQs.2nd.Edition.internal.eBook-LiB.chm/0201309831_ch11lev1sec8.html

FAQ 11.08 Can a reference be made to refer to a different referent?

No, it can't.

Unlike a pointer, once a reference is bound to an object, it cannot be made to refer to a different object. The alias cannot be separated from the referent.

For example, the last line of the following example changes i to 6; it does not make the reference k refer to j. Throughout its short life, k will always refer to i.

```
int main()
{
    int i = 5;
    int j = 6;
    int& k = i; <- 1
                k = j; <- 2
}
```

- (1) Bind k so it is an alias for i
- (2) Change i to 6—does NOT bind k to j

Figure 3.4: FAQ 2

https://www.hasustorm.com/books/English/Addison.Wesley.Cpp.FAQs.2nd.Edition.internal.eBook-LiB.chm/0201309831_ch11lev1sec11.html

FAQ 11.11 When are pointers needed?

References are usually preferred over pointers when aliases are needed for existing objects, making them useful in parameter lists and as return values.

Pointers are required when it might be necessary to change the binding to a different referent or to refer to a nonobject (a NULL pointer). Pointers often show up as local variables and member objects.

Figure 3.5: FAQ 3

3.3. FAQ ON REFERENCES

https://www.hasustorm.com/books/English/Addison.Wesley.Cpp.FAQs.2nd.Edition.internal.eBook-LiB.chm/0201309831_ch11lev1sec10.html

FAQ 11.10 Aren't references just pointers in disguise?

No, they are not.

It is important to realize that references and pointers are quite different. A pointer should be thought of as a separate object with its own distinct set of operations (*p, p->blah, and so on). So creating a pointer creates a new object.

In contrast, creating a reference does not create a new object; it merely creates an alternative name for an existing object. Furthermore the operations and semantics for the reference are defined by the referent; references do not have operations of their own.

In the following example, notice that assigning 0 to the reference j is very different than assigning 0 to the pointer p (the 0 pointer is the same as the NULL pointer).

```
int main()
{
    int i = 5;
    int& j = i;           <-- 1
    int* p = &i;          <-- 2

    j = 67;              <-- 3
    p = 0;               <-- 4
}
```

(1) j is an alias for i
(2) p is a new object, not an alias
(3) Changes i. At this point both i and j are 67
(4) Changes p; does not affect i

Figure 3.6: FAQ 4

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: reference.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has reference class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "reference.h"  
14  
15 /*-----  
16 Definition of routines of reference class  
17 -----*/  
18  
19 /*-----  
20  
21 -----*/  
22 void reference::_print(const int a, const int b, const char *s) {  
23     cout << s << " a = " << a << " b = " << b << endl ;  
24 }  
25  
26 /*-----  
27  
28 -----*/  
29 void reference::_call_by_value(int a, int b) {  
30     int t = a ;  
31     a = b ;  
32     b = t ;  
33 }  
34  
35 /*-----  
36  
37 -----*/  
38 void reference::_call_by_address(int* a, int* b) {  
39     int t = *a ;  
40     *a = *b ;  
41     *b = t ;  
42 }  
43  
44 /*-----  
45  
46 -----*/  
47 void reference::_call_by_reference(int& a, int& b) {  
48     int t = a ;  
49     a = b ;  
50     b = t ;  
51 }  
52  
53 /*-----  
54 Before call_by_value a = 55 b = 69  
55 after call_by_value a = 55 b = 69  
56 Before _call_by_address a = 55 b = 69  
57 after _call_by_address a = 69 b = 55  
58 Before _call_by_reference a = 55 b = 69  
59 after _call_by_reference a = 69 b = 55  
60 -----*/  
61 void reference::test_swap() {  
62     int a = 55;  
63     int b = 69 ;  
64     _print(a,b,"Before call_by_value") ;  
65     _call_by_value(a,b) ;  
66     _print(a,b,"after call_by_value") ;
```

```
67
68     a = 55; b = 69 ;
69     _print(a,b,"Before _call_by_address") ;
70     _call_by_address(&a,&b) ;
71     _print(a,b,"after _call_by_address") ;
72
73     a = 55; b = 69 ;
74     _print(a,b,"Before _call_by_reference") ;
75     _call_by_reference(a,b) ;
76     _print(a,b,"after _call_by_reference") ;
77 }
78
79 /*-----*/
80
81 -----*/
82 void reference::_f(int& x, int* y, int z) {
83     x = 5;
84     *y = 6;
85     z = 7;
86 }
87
88 /*-----*/
89 value of i before calling _f = 4
90 value of i after calling _f = 6
91 -----*/
92 void reference::_faq1() {
93     int i = 4;
94     cout << "value of i before calling _f = " << i << endl ;
95     _f(i, &i, i);
96     cout << "value of i after calling _f = " << i << endl ;
97 }
98
99 /*-----*/
100 Before k = j: i = 5 j = 6 k = 5
101 After k = j: i = 6 j = 6 k = 6
102 -----*/
103 void reference::_faq2() {
104     int i = 5;
105     int j = 6;
106     int& k = i;
107     cout << "Before k = j: " << "i = " << i << " j = " << j << " k = " << k << endl ;
108     k = j;
109     cout << "After k = j: " << "i = " << i << " j = " << j << " k = " << k << endl ;
110 }
111
112 /*-----*/
113 Before j = 67, p = NULL: i = 5 j = 5 p = 0014E9B8
114 After j = 67, p = NULL: i = 67 j = 67 p = 00000000
115 -----*/
116 void reference::_faq4() {
117     int i = 5;
118     int& j = i;
119     int* p = &i;
120     cout << "Before j = 67, p = NULL: " << "i = " << i << " j = " << j << " p = " << p << endl ;
121     j = 67 ;
122     p = NULL ;
123     cout << "After j = 67, p = NULL: " << "i = " << i << " j = " << j << " p = " << p << endl ;
124 }
125
126 /*-----*/
127 Can you return by reference
128 -----*/
129 int& reference::_f2() {
130     int z = 897868;
131     cout << "z = " << z << endl ;
132     return z ;
```

```
133 }
134
135 /*-----
136 z = 897868
137
138 ==21355== Use of uninitialized value of size 8
139 ==21355== at 0x3251C793FB: ??? (in /usr/lib64/libstdc++.so.6.0.8)
140 ==21355== by 0x3251C895D4: std::ostreambuf_iterator<char, std::char_traits<ch
141 May not work
142 -----*/
143 void reference::test_return_by_reference() {
144     int& a = _f2();
145     cout << "a = " << a << endl;
146 }
147
148 /*-----
149
150 -----*/
151 void reference::test_faq(){
152     _faq1();
153     _faq2();
154     _faq4();
155 }
156
157
158 //EOF
159
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: referencetest.cpp  
4  
5 On linux:  
6 g++ reference.cpp referencetest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test reference object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "reference.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     reference a ;  
25     a.test_swap() ;  
26     a.test_faq() ;  
27     a.test_return_by_reference();  
28 }  
29  
30 /*-----  
31 main  
32 -----*/  
33 int main() {  
34     testbed() ;  
35     return 0 ;  
36 }  
37  
38 //EOF  
39
```

3.4 Passing char* by value, address and reference

```
//filename: C:\work\software\objects\hanging\charstarref.cpp
#include "../util/util.h"

const char* a[] = {"MIT", "UCB"};

void f(const char* d) {
    d = a[1];
    cout << "d = " << d << endl;
}

void f(const char** d) {
    *d = a[1];
    cout << "d = " << *d << endl;
}

void f1(const char*& d) {
    d = a[1];
    cout << "d = " << d << endl;
}

int main() {
    const char* c = a[0];
    cout << "c1 = " << c << endl;
    f(c);
    cout << "c2 = " << c << endl;
    cout << "-----\n";
    cout << "c3 = " << c << endl;
    f(&c);
    cout << "c4 = " << c << endl;
    cout << "-----\n";
    c = a[0];
    cout << "c5 = " << c << endl;
    f1(c);
    cout << "c6 = " << c << endl;
    cout << "-----\n";
    return 0;
}
```

Figure 3.7: passing char* by value, address and reference

3.4. PASSING CHAR* BY VALUE, ADDRESS AND REFERENCE

VASUDEVAMURTHY

Chapter 4

Dynamic memory allocation

4.1 Introduction

4.2 Static and dynamic memory allocation

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: dynamic.h  
4 -----*/  
5  
6 /*-----  
7 This file has dynamic class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef dynamic_H  
14 #define dynamic_H  
15  
16 #include "../util/util.h"  
17 #include "../carray/carray.h" //for strcpy  
18  
19 /*-----  
20 Declaration of dynamic class  
21 -----*/  
22 class dynamic{  
23 public:  
24     void test_all() ;  
25 private:  
26     void _test_int();  
27     void _test_char();  
28     void _test_int_array(int size);  
29     void _test_char_array(int size);  
30     void _test_obj(int size);  
31 };  
32  
33  
34 #endif  
35 //EOF  
36  
37  
38
```

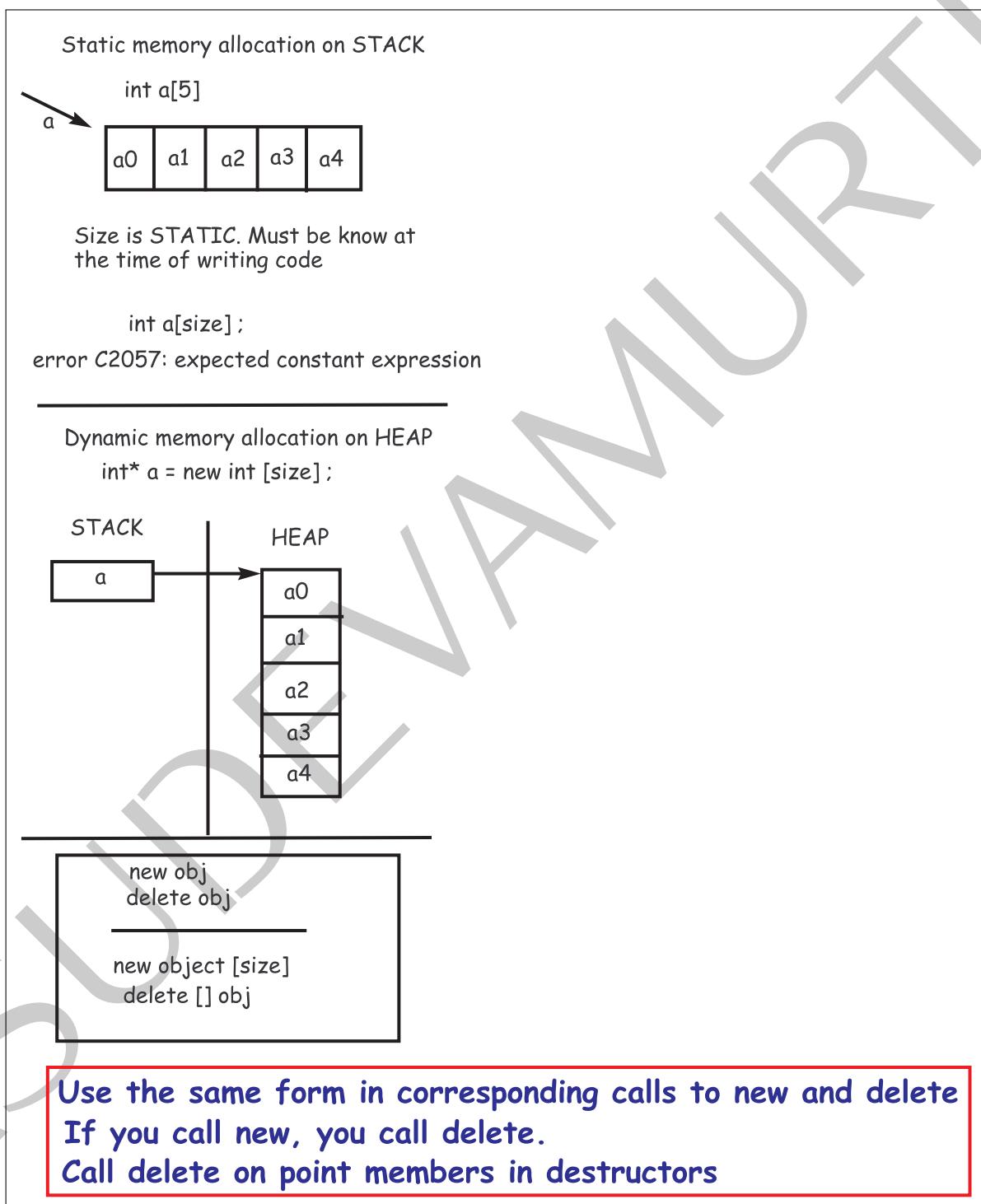


Figure 4.1: Static and dynamic memory allocation

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: dynamic.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has dynamic class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "dynamic.h"  
14  
15 /*-----  
16 Definition of routines of dynamic class  
17 -----*/  
18  
19 /*-----  
20 address of p = 007C7160 and the content of p is = 27  
21 -----*/  
22 void dynamic::_test_int() {  
23     int* p = new int(27);  
24     cout << "address of p = " << p << " and the content of p is = " << *p << endl;  
25     delete p;  
26 }  
27  
28 /*-----  
29 content of p is = H  
30 -----*/  
31 void dynamic::_test_char() {  
32     char* p = new char('H');  
33     cout << "content of p is = " << *p << endl;  
34     delete p;  
35 }  
36  
37 /*-----  
38 p[0] = 1 p[1] = 2 p[2] = 3 p[3] = 4 p[4] = 5  
39 p = abcd  
40 -----*/  
41 void dynamic::_test_int_array(int size) {  
42     int* p = new int [size];  
43     for (int i = 0; i < size; i++) {  
44         p[i] = i + 1;  
45     }  
46     for (int i = 0; i < size; i++) {  
47         cout << " p[" << i << "] = " << p[i];  
48     }  
49     cout << endl;  
50     delete [] p;  
51 }  
52  
53 /*-----  
54 p = abcd  
55 -----*/  
56 void dynamic::_test_char_array(int size) {  
57     char* p = new char [size];  
58     carray::strcpy(p,"abcd");  
59     cout << "p = " << p << endl;  
60     delete [] p;  
61 }  
62  
63 /*-----  
64 Student id = Z567 Student age = 25  
65 -----*/  
66 void dynamic::_test_obj(int size) {
```

```
67 class student{
68 public:
69     int age ;
70     char *id ;
71 };
72 student* a = new student ;
73 a->age = 25 ;
74 a->id = new char[size] ;
75 carray::strcpy(a->id,"Z567") ;
76 cout << "Student id = " << a->id << " Student age = " << a->age << endl ;
77 delete [] a->id ;
78 delete a ;
79 }
80
81 /*-----
82 test all
83 -----*/
84 void dynamic::test_all(){
85     _test_int() ;
86     _test_char();
87     _test_int_array(5);
88     _test_char_array(5);
89     _test_obj(5) ;
90 }
91
92
93 //EOF
94
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: dynamictest.cpp  
4  
5 On linux:  
6 g++ ..\carray\carray.cpp dynamic.cpp dynamictest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test dynamic object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "dynamic.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     dynamic a ;  
25     a.test_all() ;  
26 }  
27  
28 /*-----  
29 main  
30 -----*/  
31 int main() {  
32     testbed() ;  
33     return 0 ;  
34 }  
35  
36 //EOF  
37
```

4.3 Allocating two dimensional array

4.3.1 Static allocation

4.3. ALLOCATING TWO DIMENSIONAL ARRAY

STATIC TWO DIMENSIONAL ARRAY

```
const int Row = 3 ;
const int Col = 2 ;
int x[Row][Col] = { {0,1},{10,11},{20,21} } ;
```

Initializing static array

	Col 0	Col 1
Row 0	x[0,0] 0	x[0,1] 1
Row 1	x[1,0] 10	x[1,1] 11
Row 2	x[2,0] 20	x[2,1] 21

X

Internally, a two dimensional array is just a collection of 1-dimensional array

Column must be known to the compiler. CANNOT BE VARIABLE

Because, Internally, a two dimensional array is just a collection of 1-dimensional array

```
static void print_array(int a[][2] , int R, int C) {
    for (int r = 0 ; r < R; r++) {
        for (int c = 0; c < C; c++) {
            cout << a[r][c] << " " ;
        }
        cout << endl ;
    }
}
```

Accessing individual elements

Figure 4.2: Static allocation of two dimensional array

4.3.2 Dynamic allocation

DYNAMIC TWO DIMENSIONAL ARRAY

```

int Row = 3 ;
const int Col = 2 ;

int (*x)[C] = new int[Row][Col];
    
```

Dynamic MUST BE KNOWN AT COMPILED

x is a two dimensional integer array having column 'Col'

Note that contents of x will be garbage after new
deletion of x is done using: delete [] x ;

	Col 0	Col 1
Row 0	x[0,0]	x[0,1]
Row 1	x[1,0]	x[1,1]
Row 2	x[2,0]	x[2,1]

X

You cannot write a program that takes number of rows and columns from user and then allocate dynamically a two dimensional array

Figure 4.3: Dynamic allocation of two dimensional array

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: two2darray.h  
4 -----*/  
5  
6 /*-----  
7 This file has two2darray class declaration  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #ifndef two2darray_H  
14 #define two2darray_H  
15  
16 #include "../util/util.h"  
17  
18 /*-----  
19 Declaration of two2darray class  
20 -----*/  
21 class two2darray{  
22 public:  
23     void test_static_2d();  
24     void test_dynamic_2d(int R);  
25 private:  
26     void _print_array(const char*s, int a[][2] , int R, int C);  
27     void _add_contents_by(int a[][2], int n, int R, int C);  
28     void _fill(int a[][2]);  
29 };  
30  
31  
32 #endif  
33  
34 //EOF  
35
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: two2darray.cpp  
4 -----*/  
5  
6 /*-----  
7 This file has two2darray class definition  
8 -----*/  
9  
10 /*-----  
11 All includes here  
12 -----*/  
13 #include "two2darray.h"  
14  
15 /*-----  
16 Definition of routines of two2darray class  
17 -----*/  
18  
19 /*-----  
20 print two dimensional array  
21 -----*/  
22 void two2darray::_print_array(const char*s, int a[][2] , int R, int C) {  
23     cout << s << endl ;  
24     for (int r = 0 ; r < R; r++) {  
25         for (int c = 0; c < C; c++) {  
26             cout << a[r][c] << " " ;  
27         }  
28         cout << endl ;  
29     }  
30 }  
31  
32 /*-----  
33 add contents by n  
34 -----*/  
35 void two2darray::_add_contents_by(int a[][2], int n, int R, int C) {  
36     for (int r = 0 ; r < R; r++) {  
37         for (int c = 0; c < C; c++) {  
38             a[r][c] += n ;  
39         }  
40     }  
41 }  
42  
43 /*-----  
44 fill the dynamic array by predefined array  
45 -----*/  
46 void two2darray::_fill(int a[][2]) {  
47     const int R = 3 ;  
48     const int C = 2 ;  
49     int x[R][C] = { {0,1},{10,11},{20,21} } ;  
50  
51     for (int r = 0 ; r < R; r++) {  
52         for (int c = 0; c < C; c++) {  
53             a[r][c] = x[r][c] ;  
54         }  
55     }  
56 }  
57  
58 /*-----  
59 Test static two dimensional array  
60 c0 c1  
61 row0: 00 01  
62 row1: 10 11  
63 row2: 20 21  
64 -----*/  
65 void two2darray::test_static_2d() {  
66     const int R = 3 ;
```

```
67 const int C = 2 ;
68 int x[R][C] = { {0,1},{10,11},{20,21} } ;
69
70 _print_array("1. before", x,R,C) ;
71 _add_contents_by(x,5,R,C) ;
72 _print_array("2.After adding 5", x,R,C) ;
73 }
74
75
76 /*-----
77 Test static two dimensional array
78 c0 c1
79 row0: 00 01
80 row1: 10 11
81 row2: 20 21
82 -----*/
83 void two2darray::test_dynamic_2d(int R) {
84     const int C = 2 ; //must be constant and known at the time of compilation
85
86     int (*x)[C] = new int[R][C];
87     _fill(x) ;
88     _print_array("1. before", x,R,C) ;
89     _add_contents_by(x,5,R,C) ;
90     _print_array("2.After adding 5", x,R,C) ;
91     delete [] x ;
92 }
93
94 //EOF
95
96
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: two2darraytest.cpp  
4  
5 On linux:  
6 g++ two2darray.cpp two2darraytest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test two2darray object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "two2darray.h"  
19  
20 /*-----  
21 test bed  
22 -----*/  
23 void testbed() {  
24     two2darray a ;  
25     a.test_static_2d() ;  
26     a.test_dynamic_2d(3) ;  
27 }  
28  
29 /*-----  
30 main  
31 -----*/  
32 int main() {  
33     testbed() ;  
34     return 0 ;  
35 }  
36  
37 //EOF  
38
```

4.4. PROBLEM SET

4.4 Problem set

Problem 4.4.1. Write a *class* called **sieve** for generating **N prime numbers** as described in Fig 4.4. You must write two files **sieve.h** and **sieve.cpp** and test your class using the provided file **sievetest.cpp**. Your program must work for all the cases provided in **sievetest.cpp**. You cannot change anything in the file **sievetest.cpp**. **email** only **sieve.h** and **sieve.cpp**. Also include screen shot of the program output as a pdf file.

Finding all prime numbers between 2 to N-1

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself

If N = 30, all numbers that are prime between 1 and 30 are 2 3 5 7 11 13 17 19 23 29

SCHOOL METHOD

```
bool is_prime(const int n) {
    for (int i = 2; i < n; ++i) {
        if (n % i == 0){
            return false;
        }
    }
    return true;
}
```

The above method can be invoked from 2 to N-1, to get all prime numbers between 2 to N-1

INEFFICIENT
FIX IT

Sieve of Eratosthenes

To find all the prime numbers less than or equal to 15, proceed as follows.

First generate a list of integers from 2 to 15:
2 3 4 5 6 7 8 9 10 11 12 13 14 15

First number in the list is 2; cross out every 2nd number in the list after it (by counting up in increments of 2), i.e., all the multiples of 2:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15

Next number in the list after 2 is 3; cross out every 3rd number in the list after it (by counting up in increments of 3), i.e., all the multiples of 3:
2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ 10 ~~11~~ ~~12~~ 13 ~~14~~ 15

Next number not yet crossed out in the list after 3 is 5; cross out every 5th number in the list after it (by counting up in increments of 5), i.e., all the multiples of 5:
2 3 5 7 11 13

NOTHING CAN BE CROSSED. ALGORITHMS HALT

The numbers left not crossed out in the list at this point are all the prime numbers below 15

1. Implement school method which takes N and return array of prime numbers
2. Implement SIEVE method which takes N and return array of prime numbers
3. BOTH ARRAY MUST BE EQUIVALENT
4. sievetest.cpp must pass. **CANNOT BE MODIFIED**
5. Write your code in **sieve.h** and **sieve.cpp** and e-mail only these two files.
You need to write: school_method, sieve_of_eratosthenes, print and verify routines.
6. YOU CANNOT USE **sqrt** function in your code
7. YOU CANNOT USE ANY STL vectors in your code.

Figure 4.4: Finding all prime numbers

```
1 /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 file: sievetest.cpp  
4  
5 On linux:  
6 g++ sieve.cpp sievetest.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test sieve object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "sieve.h"  
19  
20 /*-----  
21 Compute prime up to n (all numbers that cannot divide n except 1 and n)  
22  
23 if n = 30  
24 prime numbers are: 2 3 5 7 11 13 17 19 23 29  
25 -----*/  
26 void prime(int n) {  
27     sieve s;  
28     int as = 0;  
29     int* a = s.school_method(n, &as);  
30     s.print("School Method", as, a);  
31     int ss = 0;  
32     int* b = s.sieve_of_eratosthenes(n, &ss);  
33     s.print("Sieve Method", ss, b);  
34     assert(s.verify(as, a, ss, b));  
35     delete [] a;  
36     delete [] b;  
37 }  
38  
39 /*-----  
40 test bed  
41 -----*/  
42 void testbed() {  
43     prime(30);  
44     prime(1000);  
45     prime(10000);  
46 }  
47  
48 /*-----  
49 main  
50 -----*/  
51 int main() {  
52     testbed();  
53     return 0;  
54 }  
55  
56 //EOF  
57  
58
```

4.4. PROBLEM SET

COXETER MAGIC SQUARE

Start with 1 in the middle of the top row; then go up and left, assigning numbers in increasing order to empty squares; if you fall off the square imagine the same square as tiling the plane and continue; if a square is occupied, move down instead and continue.

Works for only odd number

6	1	8
7	5	3
2	9	4

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Implement class `magicsquare`
and test for various odd values of n

1. Make sure the sum is same in all directions

e-mail `magicsquare.h`
`magicsquare.cpp`
`magicsquaretest.cpp`
must use only `../util/util.h`

Figure 4.5: Odd Magic square

Problem 4.4.2.

Initialize the chess board as shown in the figure.

Write a display routine that prints the board configuration

SHOW SOLUTION AS BOTH STATIC AND DYNAMIC ALLOCATION

K white king
Q white queen
R white rook
B white bishop
N white knight
P white pawn
- empty square

k black king
q black queen
r black rook
b black bishop
n black knight
p black pawn

r	n	b	q	k	b	n	r
p							
-							
-							
-							
-							
P							
R	N	B	Q	K	B	N	R

Figure 4.6: A chess board

Problem 4.4.3.

4.4. PROBLEM SET

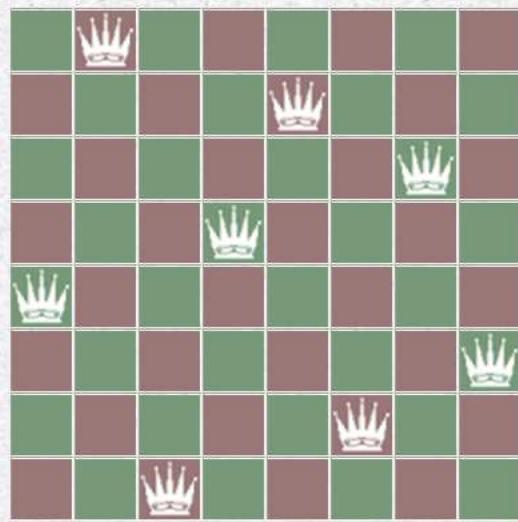
Problem 4.4.4.

The eight queens puzzle is the problem of placing eight chess queens on an 8×8 chessboard so that no two queens attack each other.

Thus, a solution requires that no two queens share the same row, column, or diagonal

The N Queens Problem

$N = 8$



$N = 4$

For a 1×1 board, there is one trivial solution:



$N=1$

For 2×2 and 3×3 boards, there are no solutions.

Output solution

-1-----
-----1---
-1-----
1-----
-----1---
-----1

The eight queens puzzle has 92 distinct solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 unique (or fundamental) solutions.

Figure 4.7: N Queens Puzzle

4.4. PROBLEM SET

VASUDEVAMURTHY

Chapter 5

Function overloading and default function arguments

5.1 Introduction

5.2 Function overloading

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: overload.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 void f(int i){  
10 cout << "In void f(int i) " << i << endl ;  
11 }  
12  
13 void f(bool i){  
14 cout << "In void f(bool i) " << i << endl ;  
15 }  
16  
17 void f(long i){  
18 cout << "In void f(long i) " << i << endl ;  
19 }  
20  
21 void f(float i){  
22 cout << "In void f(float i) " << i << endl ;  
23 }  
24  
25 void f(double i){  
26 cout << "In void f(double i) " << i << endl ;  
27 }  
28 void f(char i){  
29 cout << "In void f(char i) " << i << endl ;  
30 }  
31  
32 int main(){  
33 f(10);  
34 int i = 2147483647;  
35 f(i);  
36 long l = 2147483647;  
37 f(l);  
38 f(true);  
39 f(false);  
40 double d = 1.7e308;  
41 f(d);  
42 float f1 = float(5.67);  
43 f(f1);  
44 f('a');  
45 return 0;  
46 }  
47  
48 /*  
49  
50 In void f(int i) 10  
51 In void f(int i) 2147483647  
52 In void f(long i) 2147483647  
53 In void f(bool i) 1  
54 In void f(bool i) 0  
55 In void f(double i) 1.7e+308
```

5.3 Default function arguments

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: defaultfunc.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 void f1(const char *n, int m){  
10 cout << "f1: Name " << n << " Marks " << m << endl ;  
11 }  
12  
13 void f2(const char *n, int m, int b){  
14 cout << "f2: Name " << n << " Marks " << m + b << endl ;  
15 }  
16  
17 void f3(const char *n, int m, int b, char g){  
18 cout << "f3: Name " << n << " Marks " << m + b << " Grade " << g << endl ;  
19 }  
20  
21 void f(const char *n, int m, int b = 0 , char g = ' '){  
22 cout << "f: Name " << n << " Marks " << m + b ;  
23 if (g != ' '){  
24 cout << " Grade " << g ;  
25 }  
26 cout << endl ;  
27 }  
28  
29 int main(){  
30 f1("John Smith",78);  
31 f2("John Smith",78,10);  
32 f3("John Smith",78,10,'B');  
33 f("John Smith",78);  
34 f("John Smith",78,10);  
35 f("John Smith",78,10,'B');  
36 return 0 ;  
37 }  
38  
39 /*  
40 f1: Name John Smith Marks 78  
41 f2: Name John Smith Marks 88  
42 f3: Name John Smith Marks 88 Grade B  
43 f: Name John Smith Marks 78  
44 f: Name John Smith Marks 88  
45 f: Name John Smith Marks 88 Grade B  
46 */  
47
```

Chapter 6

C++ class

6.1 Introduction

6.2 Need for *class*

```
13 void code_without_classes() {
14     int a_age ;
15     char a_name[10] ;
16     bool a_is_male ;
17
18     int b_age ;
19     char b_name[10] ;
20     bool b_is_male ;
21
22     a_age = 17 ;
23     strcpy(a_name,"john") ;
24     a_is_male = true ;
25
26     b_age = 21 ;
27     strcpy(b_name,"debbie") ;
28     b_is_male = false ;
29
30     cout << a_name << ";Age = " << a_age << ";Sex = " << ((a_is_male)? "male" : "female") << endl ;
31     cout << b_name << ";Age = " << b_age << ";Sex = " << ((b_is_male)? "male" : "female") << endl ;
32 }
```

Figure 6.1: Need for a class

```
1 /*-----
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename:c1.cpp
4 -----*/
5
6 #include <iostream>
7 using namespace std;
8
9 /*-----
10 john;Age = 17;Sex = male
11 debbie;Age = 21;Sex = female
12 -----*/
13 void code_without_classes() {
14     int a_age ;
15     char a_name[10] ;
16     bool a_is_male ;
17
18     int b_age ;
19     char b_name[10] ;
20     bool b_is_male ;
21
22     a_age = 17 ;
23     strcpy(a_name,"john") ;
24     a_is_male = true ;
25
26     b_age = 21 ;
27     strcpy(b_name,"debbie") ;
28     b_is_male = false ;
29
30     cout << a_name << ";Age = " << a_age << ";Sex = " << ((a_is_male)? "male" : "female") << endl ;
31     cout << b_name << ";Age = " << b_age << ";Sex = " << ((b_is_male)? "male" : "female") << endl ;
32 }
33
34 /*-----
35 john;Age = 17;Sex = male
36 debbie;Age = 21;Sex = female
37 -----*/
38 void code_with_classes() {
39     class group {
40     public:
41         int age ;
42         char name[10] ;
43         bool is_male ;
44         void print() {
45             cout << name << ";Age = " << age << ";Sex = " << ((is_male)? "male" : "female") << endl ;
46         }
47     };
48
49     group a ;
50     a.age = 17 ;
51     strcpy(a.name,"john") ;
52     a.is_male = true ;
53     a.print() ;
54
55     group b ;
```

```
56     b.age = 21 ;
57     strcpy(b.name,"debbie") ;
58     b.is_male = false ;
59     b.print() ;
60 }
61
62 /*-----*
63
64 -----*/
65 int main() {
66     code_without_classes();
67     code_with_classes();
68     return 0 ;
69 }
```

6.2. NEED FOR CLASS

```

38 void code_with_classes() {
39     class group {
40     public:
41         int age ;
42         char name[10] ;
43         bool is_male ;
44         void print() {
45             cout << name << ";Age = " << age << ";Sex = " << ((is_male)? "male" : "female") << endl ;
46         }
47     };
48
49     group a ;
50     a.age = 17 ;
51     strcpy(a.name,"john") ;
52     a.is_male = true ;
53     a.print() ;
54
55     group b ;
56     b.age = 21 ;
57     strcpy(b.name,"debbie") ;
58     b.is_male = false ;
59     b.print() ;
60 }
```

Collection of

- 1) different types of data items
- 2) Functions that operates on this data

Class: templates/cookie cutter/abstractions
No memory space

Encapsulation

Encapsulation: Bringing together data and functions into a single structure scope.

Advantage:

1. Objects no longer needs to be dependent on "outside" or "global functions" to alter its state or behaviour.
2. Objects receives messages (function calls) and act on these messages via its functions

Figure 6.2: Class

```

38 void code_with_classes() {
39     class group {
40     public:
41         int age ;
42         char name[10] ;
43         bool is_male ;
44         void print() {
45             cout << name << ";Age = " << age << ";Sex = " << ((is_male)? "male" : "female") << endl ;
46         }
47     };
48
49     group a ;
50     a.age = 17 ;
51     strcpy(a.name,"john") ;
52     a.is_male = true ;
53     a.print() ;
54
55     group b ;
56     b.age = 21 ;
57     strcpy(b.name,"debbie") ;
58     b.is_male = false ;
59     b.print() ;
60 }
```

Class: templates/cookie cutter/abstractions
No memory space

objects: Instances of the class
A cookie from a cookie cutter

Objects differ from each other only in the value of the data that they hold, but all of them has the same data fields of that class

Figure 6.3: Object

6.3 class and objects

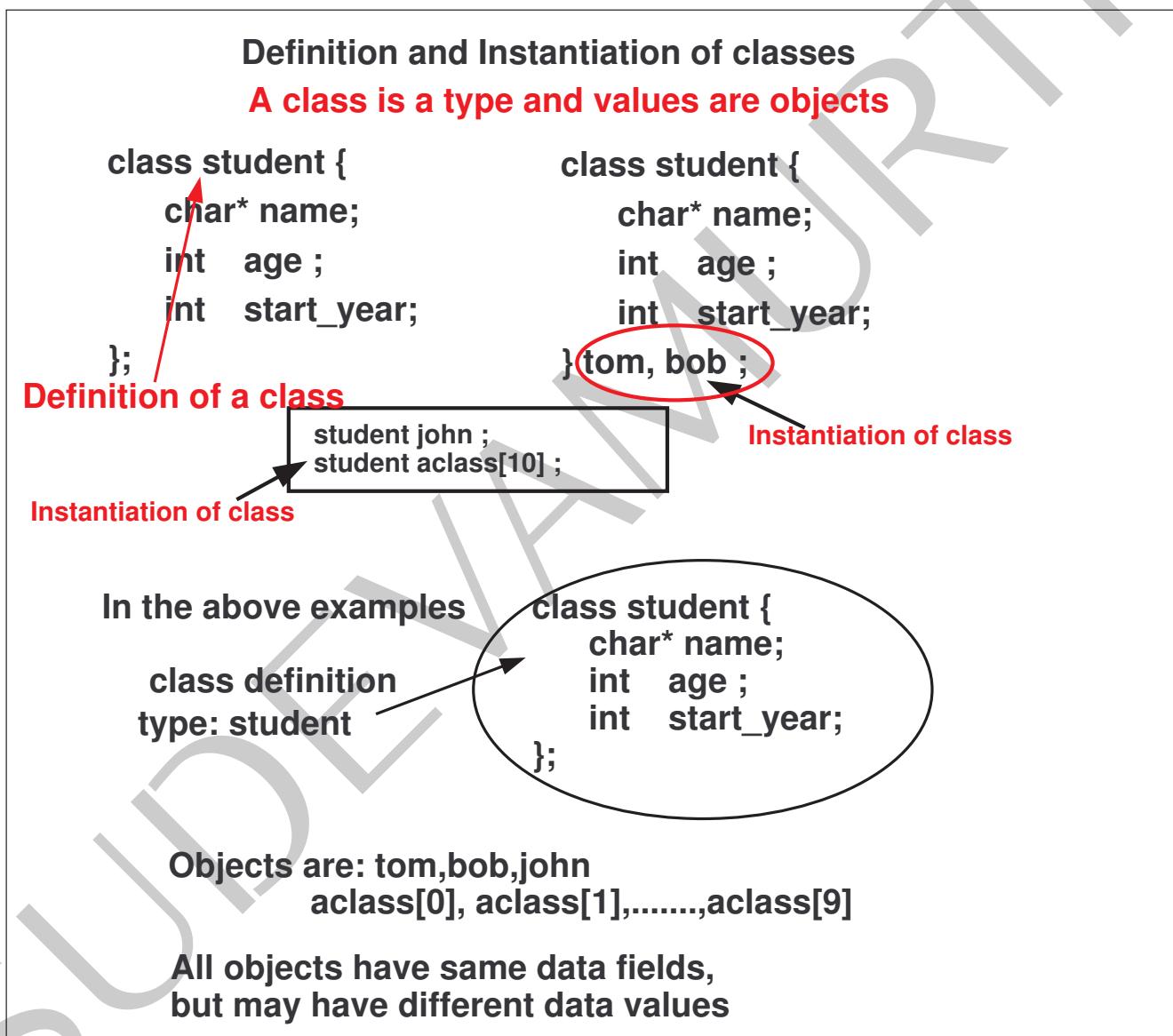


Figure 6.4: Class and objects

6.4 Accessing class members using dot and arrow operator

```
1 /*-----
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename:c2.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 Definition of a class
11 -----
12 class group_public {
13 public:
14     int age ;
15     char name[10];
16     bool is_male ;
17     void print() ;
18 };
19
20 /*
21 Definition of a class
22 -----
23 class group_private {
24     int age ;
25     char name[10];
26     bool is_male ;
27     void print() ;
28 };
29
30 /*
31 Accessing is using dot and arrow operators
32 -----
33 static void accessing_public_elements_of_a_class() {
34     group_public tom ;
35     tom.age = 7 ;
36     tom.is_male = true ;
37
38     group_public* tom_ptr = &tom ;
39     tom_ptr->age = 8 ;
40     strcpy(tom_ptr->name,"TOM") ;
41 }
42
43 /*
44 Cannot accessing private members of the class
45 -----
46 static void accessing_private_elements_of_a_class() {
47     group_private tom ;
48     //tom.age = 7 ;
49     // error C2248: 'group_private::age' : cannot access private member declared in class 'group_private'
50     //tom.is_male = true ;
51     // error C2248: 'group_private::is_male' : cannot access private member declared in class 'group_private'
52 }
53
54 /*
55
```

```
56 -----*/
57 int main() {
58     accessing_public_elements_of_a_class();
59     accessing_private_elements_of_a_class();
60     return 0 ;
61 }
62
```

6.4. ACCESSING CLASS MEMBERS USING DOT AND ARROW OPERATOR

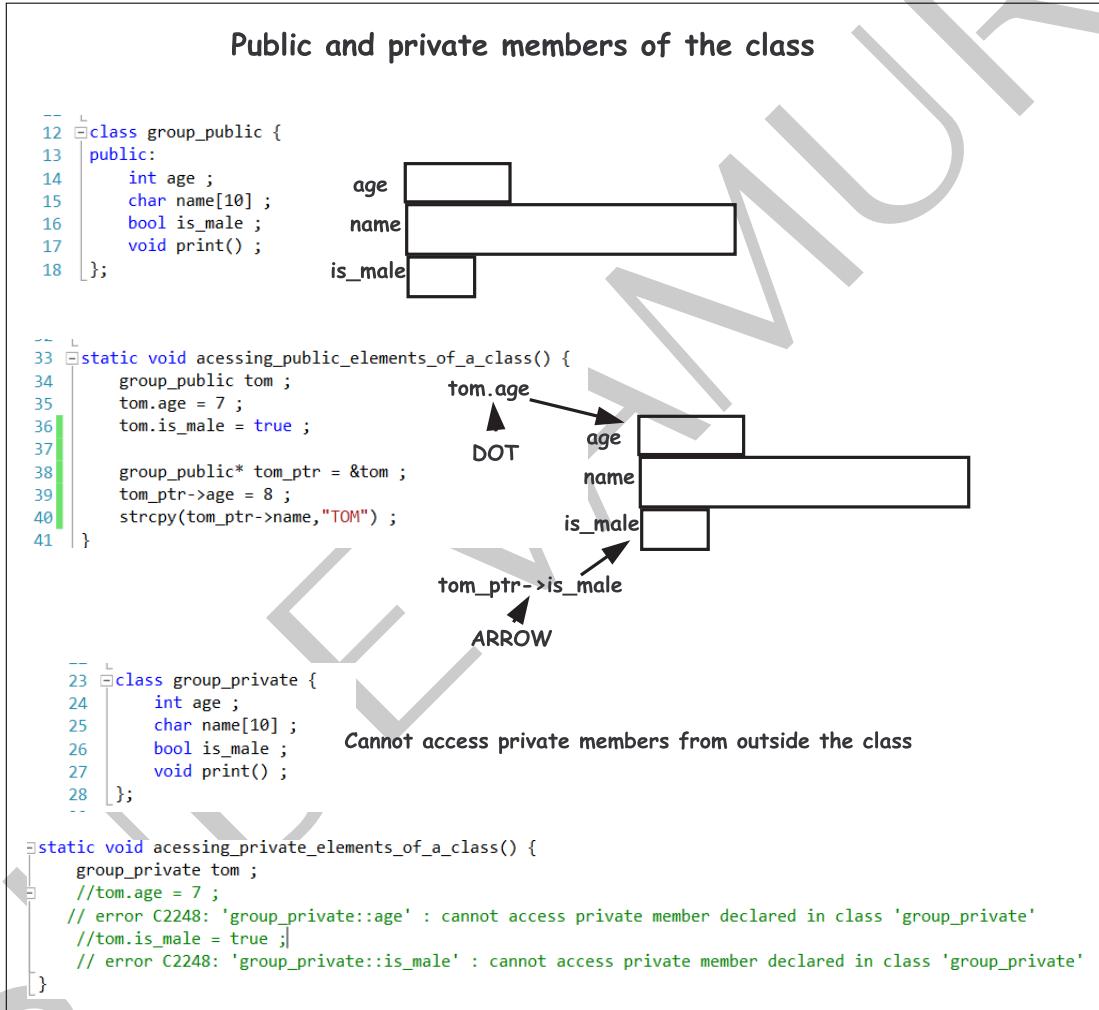


Figure 6.5: Public and private members of the class

6.5 Accessing private class members

Accessing private members of the class

```

12 class group_private {
13     private:
14         int _age ;
15         char _name[10] ;
16         bool _is_male ;
17
18     public:
19         void set_age(int a) {_age = a ; }
20         int age() const { return _age ; }
21         void set_male() {_is_male = true ; }
22         void set_female() {_is_male = false ; }
23         bool is_male() const { return _is_male ; }
24     };
25
26 static void accessing_private_elements_of_a_class() {
27     group_private tom ;
28     //tom.age = 7 ;
29     // error C2248: 'group_private::age' : cannot access private member declared in class 'group_private'
30     //tom.is_male = true ;
31     // error C2248: 'group_private::is_male' : cannot access private member declared in class 'group_private'
32     tom.set_age(7) ;
33     tom.set_male() ;
34     cout << "Tom " << tom.age() << " " ;
35     cout << (tom.is_male() ? "male" : "female") << endl ;
36 }
```

DATA HIDING

We can change the way we store name

Changes data. **Mutator functions**

Inspectors. Only read data and does not modify . **Const Accessor function**

Accessing private members of the class using public interface

Principle of data hiding:
Ensures that users of the class are not able to manipulate the data members of the class directly

Figure 6.6: Accessing private class members and concept of data hiding

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c3.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 Definition of a class  
11 -----*/  
12 class group_private {  
13 private:  
14     int _age ;  
15     char _name[10] ;  
16     bool _is_male ;  
17  
18 public:  
19     void set_age(int a) {_age = a ; }  
20     int age() const { return _age ; }  
21     void set_male() {_is_male = true ; }  
22     void set_female() {_is_male = false ; }  
23     bool is_male() const { return _is_male ; }  
24 };  
25  
26 /*-----  
27 Accessing private members using public interface. Data hiding  
28 -----*/  
29 static void accessing_private_elements_of_a_class() {  
30     group_private tom ;  
31     //tom.age = 7 ;  
32     // error C2248: 'group_private::age' : cannot access private member declared in class 'group_private'  
33     //tom.is_male = true ;  
34     // error C2248: 'group_private::is_male' : cannot access private member declared in class 'group_private'  
35     tom.set_age(7) ;  
36     tom.set_male() ;  
37     cout << "Tom " << tom.age() << " " ;  
38     cout << (tom.is_male() ? "male" : "female") << endl ;  
39 }  
40  
41 /*-----  
42 -----*/  
43  
44 int main()  
45 {  
46     accessing_private_elements_of_a_class();  
47     return 0 ;  
48 }
```

6.6 Separating interface and implementation

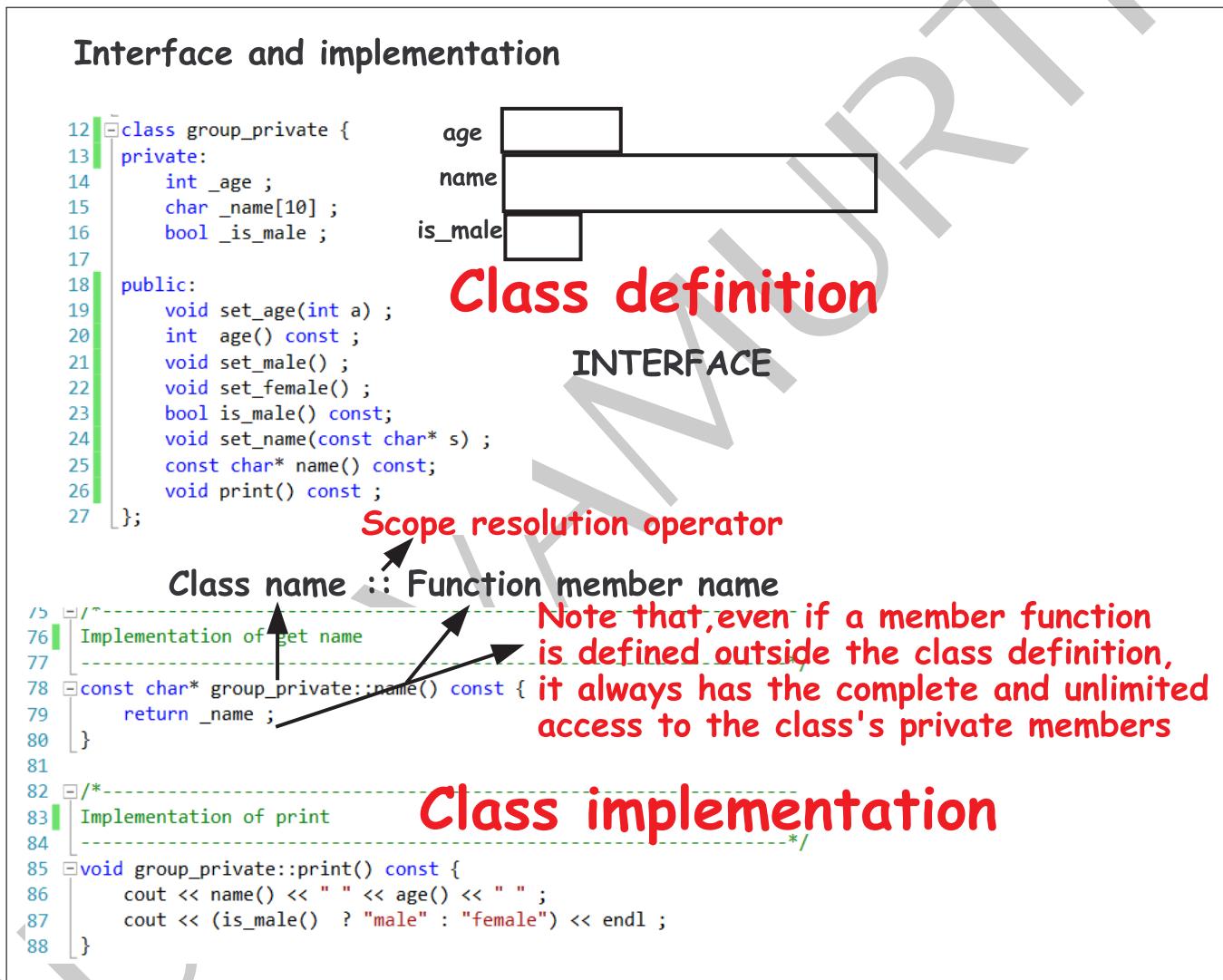


Figure 6.7: Interface and implementation

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c4.h  
4 -----*/  
5 #ifndef C4_H  
6 #define C4_H  
7  
8 /*-----  
9 Definition of the class - INTERFACE  
10 -----*/  
11 class group_private {  
12 private:  
13     int _age;  
14     char _name[10];  
15     bool _is_male;  
16  
17 public:  
18     void set_age(int a);  
19     int age() const;  
20     void set_male();  
21     void set_female();  
22     bool is_male() const;  
23     void set_name(const char* s);  
24     const char* name() const;  
25     void print() const;  
26 };  
27  
28 #endif  
29  
30
```

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c4.cpp  
4 -----*/  
5  
6 #include "c4.h"  
7  
8 #include <iostream>  
9 using namespace std;  
10  
11 /*-----  
12  IMPLEMENTATION of the class  
13 -----*/  
14  
15 /*-----  
16 Implementation of set age  
17 -----*/  
18 void group_private::set_age(int a){  
19     _age = a;  
20 }  
21  
22 /*-----  
23 Implementation of age  
24 -----*/  
25 int group_private::age() const {  
26     return _age;  
27 }  
28  
29 /*-----  
30 Implementation of set male  
31 -----*/  
32 void group_private::set_male(){  
33     _is_male = true;  
34 }  
35  
36 /*-----  
37 Implementation of set female  
38 -----*/  
39 void group_private::set_female(){  
40     _is_male = false;  
41 }  
42  
43 /*-----  
44 Implementation of is_male  
45 -----*/  
46 bool group_private::is_male() const {  
47     return _is_male;  
48 }  
49  
50 /*-----  
51 Implementation of set name  
52 -----*/  
53 void group_private::set_name(const char* s){  
54     strcpy(_name,s);  
55 }
```

```
56
57 -----
58 Implementation of get name
59 -----
60 const char* group_private::name() const {
61     return _name;
62 }
63
64 -----
65 Implementation of print
66 -----
67 void group_private::print() const {
68     cout << name() << " " << age() << " ";
69     cout << (is_male() ? "male" : "female") << endl;
70 }
71
72
```

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c4_main.cpp  
4  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 -----*/  
7  
8 #include "c4.h"  
9  
10 /*-----  
11 TOM JR 7 male  
12 -----*/  
13 static void accessing_private_elements_of_a_class()  
14 {  
15     group_private tom;  
16     //tom.age = 7;  
17     // error C2248: 'group_private::age' : cannot access private member declared in class 'group_private'  
18     //tom.is_male = true;  
19     // error C2248: 'group_private::is_male' : cannot access private member declared in class 'group_private'  
20     tom.set_name("TOM JR");  
21     tom.set_age(7);  
22     tom.set_male();  
23     tom.print();  
24 }  
25 /*-----  
26 -----*/  
27  
28 int main(){  
29     accessing_private_elements_of_a_class();  
30     return 0;  
31 }  
32
```

6.7 Allocating class objects on stack and heaps, phase 1

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c5.h  
4 -----*/  
5  
6 #ifndef C5_H  
7 #define C5_H  
8  
9 /*-----  
10 Definition of the class  
11 -----*/  
12 class student {  
13 private:  
14     int _age ;  
15     char* _name;  
16     bool _is_male ;  
17  
18 public:  
19     void print() const ;  
20     void init(const char* name,int age,bool m=true);  
21     void fini() {delete [] _name;}  
22 };  
23  
24 #endif  
25
```

6.7. ALLOCATING CLASS OBJECTS ON STACK AND HEAPS, PHASE 1

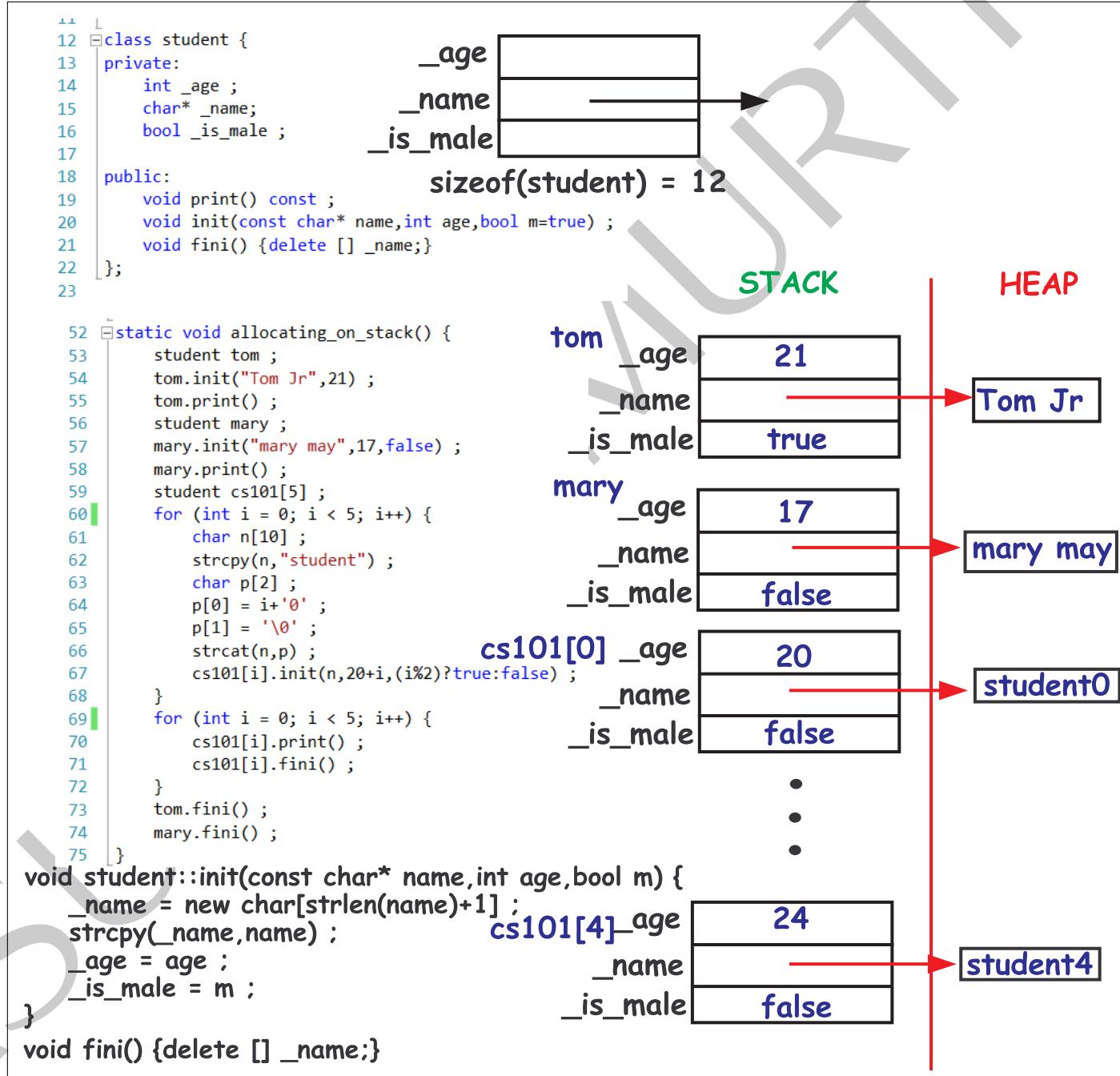


Figure 6.8: Allocating objects and arrays of objects on stack

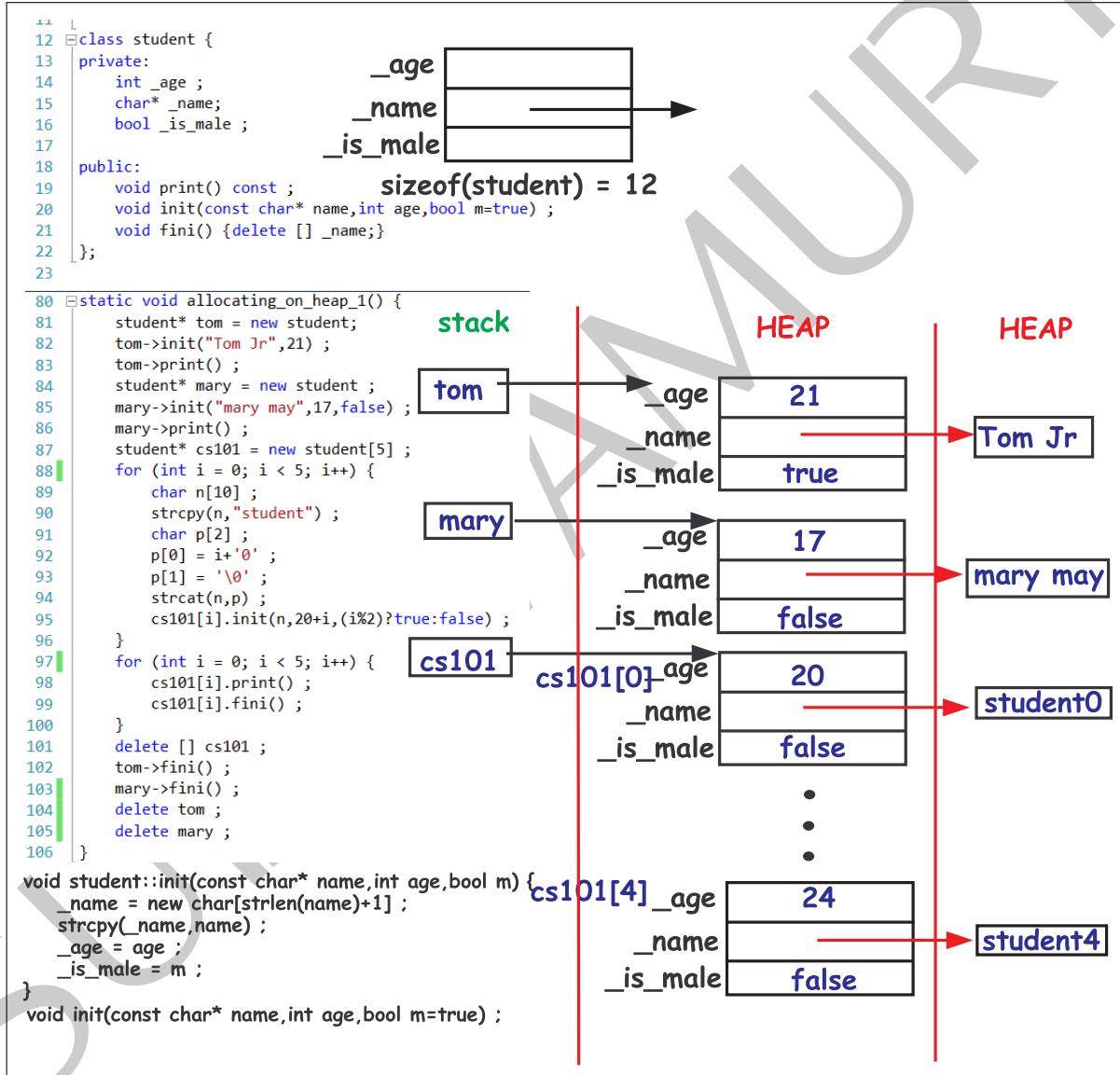


Figure 6.9: Allocating objects and arrays of objects on heap

6.7. ALLOCATING CLASS OBJECTS ON STACK AND HEAPS, PHASE 1

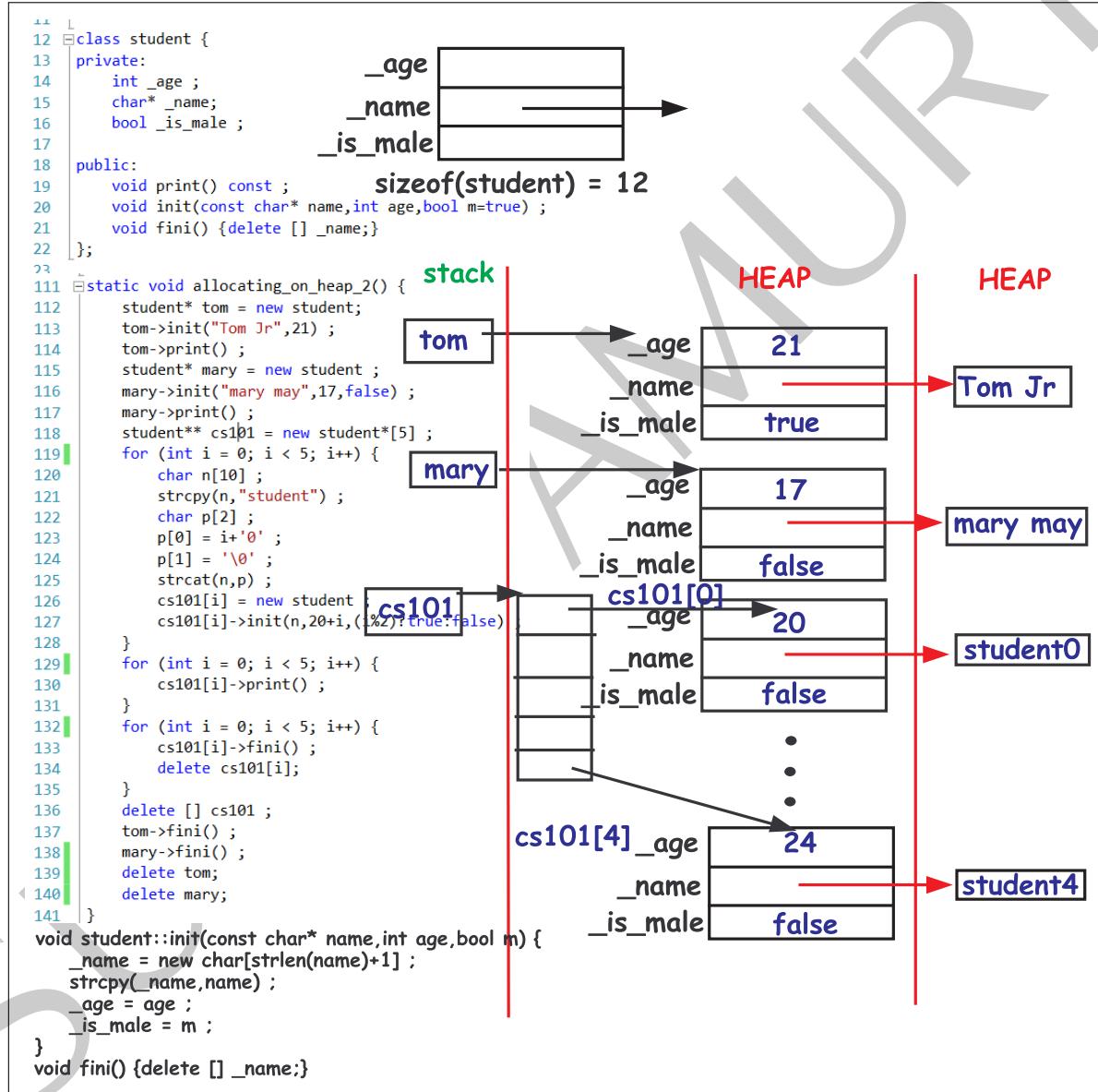


Figure 6.10: Allocating objects and arrays of objects on heap

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c5.cpp  
4 -----*/  
5 #include "c5.h"  
6  
7 #include <iostream>  
8 using namespace std;  
9  
10 /*-----  
11 Implementation of print  
12 -----*/  
13 void student::print() const {  
14     cout << _name << " " << _age << " "  
15     cout << (_is_male ? "male" : "female") << endl;  
16 }  
17  
18 /*-----  
19 Implementation init  
20 -----*/  
21 void student::init(const char* name,int age,bool m){  
22     _name = new char[strlen(name)+1];  
23     strcpy(_name,name);  
24     _age = age;  
25     _is_male = m;  
26 }  
27
```

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c5_main.cpp  
4  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 -----*/  
7 #include "c5.h"  
8  
9 #include <iostream>  
10 using namespace std;  
11  
12 /*-----  
13 size of the class student = 12  
14 Tom Jr 21 male  
15 mary may 17 female  
16 student0 20 female  
17 student1 21 male  
18 student2 22 female  
19 student3 23 male  
20 student4 24 female  
21 -----*/  
22 static void allocating_on_stack() {  
23     student tom ;  
24     tom.init("Tom Jr",21);  
25     tom.print();  
26     student mary ;  
27     mary.init("mary may",17,false);  
28     mary.print();  
29     student cs101[5];  
30     for (int i = 0; i < 5; i++) {  
31         char n[10];  
32         strcpy(n,"student");  
33         char p[2];  
34         p[0] = i+'0' ;  
35         p[1] = '\0' ;  
36         strcat(n,p);  
37         cs101[i].init(n,20+i,(i%2)?true:false);  
38     }  
39     for (int i = 0; i < 5; i++) {  
40         cs101[i].print();  
41         cs101[i].fini();  
42     }  
43     tom.fini();  
44     mary.fini();  
45 }  
46  
47 /*-----  
48 -----*/  
49 static void allocating_on_heap_1() {  
50     student* tom = new student;  
51     tom->init("Tom Jr",21);  
52     tom->print();  
53     student* mary = new student ;  
54     mary->init("mary may",17,false);
```

```
56     mary->print();
57     student* cs101 = new student[5];
58     for (int i = 0; i < 5; i++) {
59         char n[10];
60         strcpy(n,"student");
61         char p[2];
62         p[0] = i+'0' ;
63         p[1] = '\0';
64         strcat(n,p);
65         cs101[i].init(n,20+i,(i%2)?true:false);
66     }
67     for (int i = 0; i < 5; i++) {
68         cs101[i].print();
69         cs101[i].fini();
70     }
71     delete [] cs101;
72     tom->fini();
73     mary->fini();
74     delete tom;
75     delete mary;
76 }
77
78 /*-----
79
80 -----*/
81 static void allocating_on_heap_2() {
82     student* tom = new student;
83     tom->init("Tom Jr",21);
84     tom->print();
85     student* mary = new student ;
86     mary->init("mary may",17,false);
87     mary->print();
88     student** cs101 = new student*[5];
89     for (int i = 0; i < 5; i++) {
90         char n[10];
91         strcpy(n,"student");
92         char p[2];
93         p[0] = i+'0' ;
94         p[1] = '\0';
95         strcat(n,p);
96         cs101[i] = new student ;
97         cs101[i]->init(n,20+i,(i%2)?true:false);
98     }
99     for (int i = 0; i < 5; i++) {
100        cs101[i]->print();
101    }
102    for (int i = 0; i < 5; i++) {
103        cs101[i]->fini();
104        delete cs101[i];
105    }
106    delete [] cs101;
107    tom->fini();
108    mary->fini();
109    delete tom;
110    delete mary;
```

```
111 }
112
113 /*-----*
114 -----*/
115 int main() {
116     cout << "size of the class student = " << sizeof(student) << endl ;
117     allocating_on_stack();
118     allocating_on_heap_1();
119     allocating_on_heap_2();
120     return 0 ;
121 }
122
123
```

6.8 Shallow copying of objects, phase 1

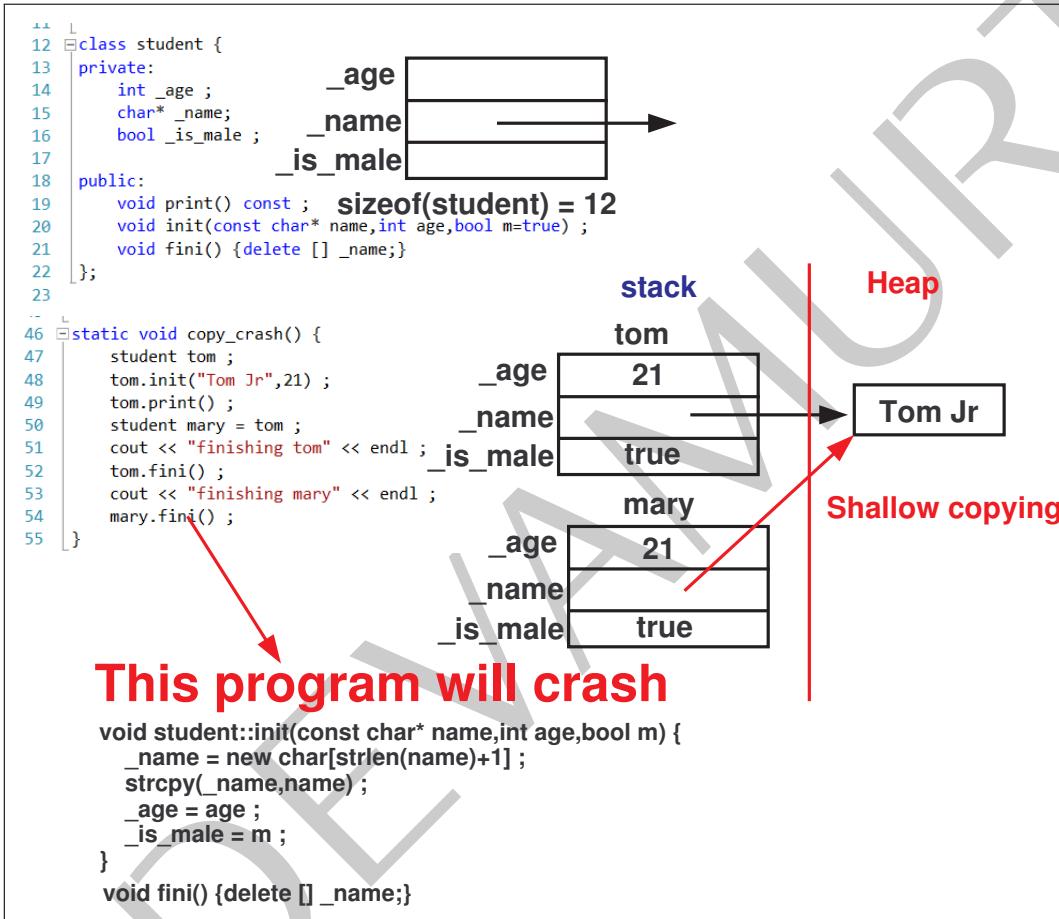


Figure 6.11: Copying objects that has data on the heap

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c6.cpp  
4  
5     FUM: Freeing unallocated memory  
6     Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
7     THIS PROGRAM WILL CRASH  
8 -----*/  
9  
10 #include <iostream>  
11 using namespace std;  
12  
13 /*-----  
14 Definition of the class  
15 -----*/  
16 class student {  
17 private:  
18     int _age ;  
19     char* _name;  
20     bool _is_male ;  
21  
22 public:  
23     void print() const ;  
24     void init(const char* name,int age,bool m=true);  
25     void fini() {delete [] _name;}  
26 };  
27  
28 /*-----  
29 Implementation of print  
30 -----*/  
31 void student::print() const {  
32     cout << _name << " " << _age << " "  
33     cout << (_is_male ? "male" : "female") << endl ;  
34 }  
35  
36 /*-----  
37 Implementation init  
38 -----*/  
39 void student::init(const char* name,int age,bool m){  
40     _name = new char[strlen(name)+1];  
41     strcpy(_name,name);  
42     _age = age ;  
43     _is_male = m ;  
44 }  
45  
46 /*-----  
47 -----*/  
48  
49 static void copy_crash(){  
50     student tom ;  
51     tom.init("Tom Jr",21);  
52     tom.print();  
53     student mary = tom ;  
54     cout << "finishing tom" << endl ;  
55     tom.fini();
```

```
56 cout << "finishing mary" << endl ;
57 mary.fini();
58 }
59
60 /*-----
61 THIS PROGRAM WILL CRASH
62 -----*/
63 int main(){
64 cout << "size of the class student = " << sizeof(student) << endl ;
65 copy_crash();
66 return 0;
67 }
68
```

6.9. CREATING TEMPORARY OBJECTS ON THE STACK, PHASE1

6.9 Creating temporary objects on the stack, phase1

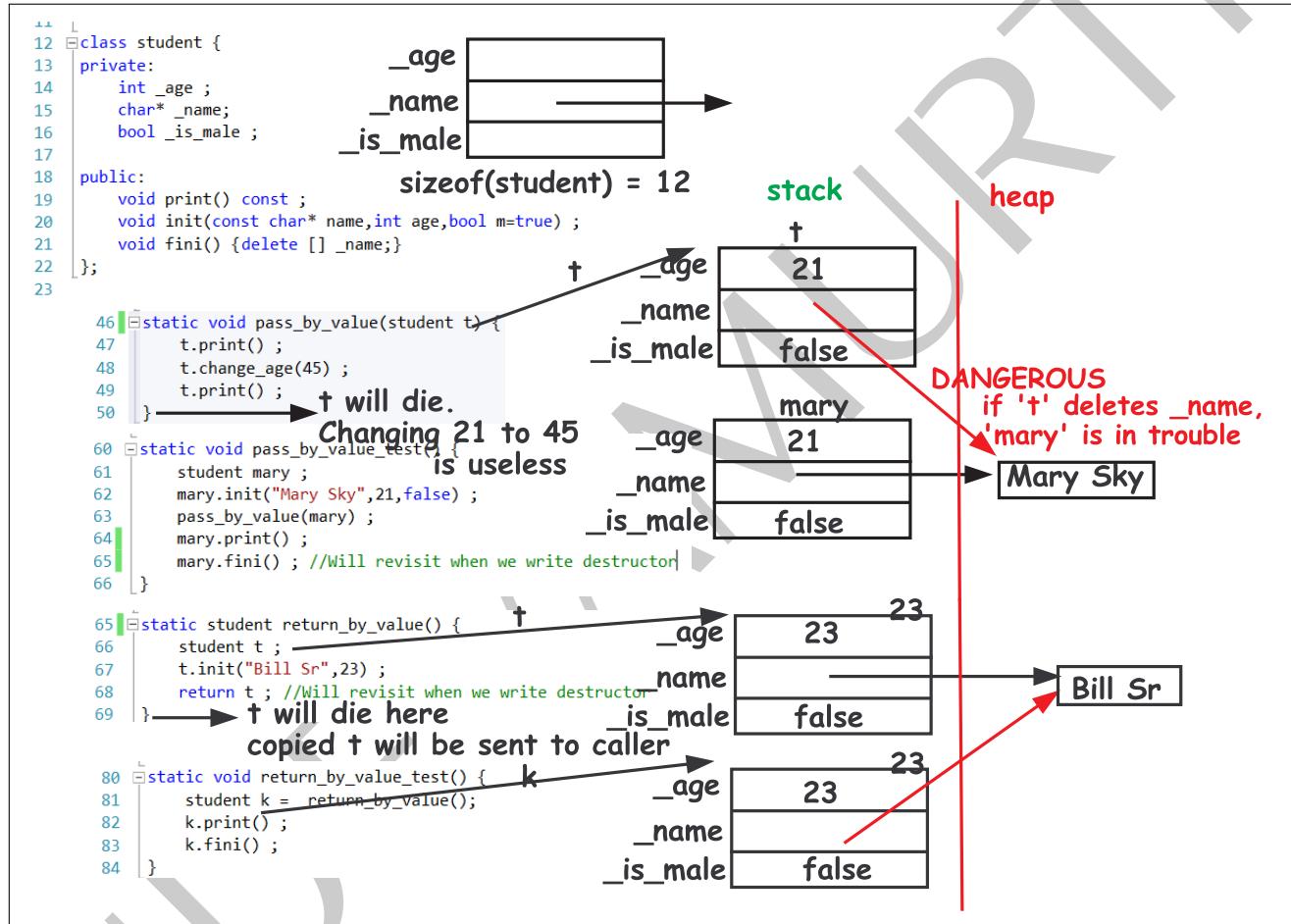


Figure 6.12: Creation of temporary objects due to pass by value and return by value

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c7.cpp  
4  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 Program exited with status code 0.  
7 -----*/  
8  
9 #include <iostream>  
10 using namespace std;  
11  
12 /*-----  
13 Definition of the class  
14 -----*/  
15 class student {  
16 private:  
17     int _age ;  
18     char* _name;  
19     bool _is_male ;  
20  
21 public:  
22     void change_age(int i) {_age = i;}  
23     void print() const ;  
24     void init(const char* name,int age,bool m=true);  
25     void fini() {delete [] _name;}  
26 };  
27  
28 /*-----  
29 Implementation of print  
30 -----*/  
31 void student::print() const {  
32     cout << _name << " " << _age << " "  
33     cout << (_is_male ? "male" : "female") << endl ;  
34 }  
35  
36 /*-----  
37 Implementation init  
38 -----*/  
39 void student::init(const char* name,int age,bool m){  
40     _name = new char[strlen(name)+1];  
41     strcpy(_name,name);  
42     _age = age ;  
43     _is_male = m ;  
44 }  
45  
46 /*-----  
47 -----*/  
48  
49 static void pass_by_value(student t){  
50     t.print();  
51     t.change_age(45);  
52     t.print();  
53 }  
54  
55 /*-----
```

```
56 Mary Sky 21 female
57 Mary Sky 45 female
58 Mary Sky 21 female
59 -----
60 static void pass_by_value_test() {
61     student mary;
62     mary.init("Mary Sky",21,false);
63     pass_by_value(mary);
64     mary.print();
65     mary.fini(); //Will revisit when we write destructor
66 }
67
68 /**
69
70 */
71 static student return_by_value() {
72     student t;
73     t.init("Bill Sr",23);
74     return t; //Will revisit when we write destructor
75 }
76
77 /**
78 Bill Sr 23 male
79 */
80 static void return_by_value_test() {
81     student k = return_by_value();
82     k.print();
83     k.fini();
84 }
85
86 /**
87
88 */
89 int main() {
90     pass_by_value_test();
91     return_by_value_test();
92     return 0;
93 }
94
```

6.10 inline function

An inline function is one whose code replaces the assembly language "CALL" that would be normally generated

All inline functions must be in class header files (*.h)

The compiler must have access to the source code inorder to do code substitution. That's why inline code must be in the same file

```
void set_age(int a) {_age = a ; }  
int age() const {return _age ;}
```

Implicit inlining

```
45 inline bool group_private::is_male() const {  
46     return _is_male ;  
47 }
```

Explicit inlining

If the inline request is NOT honored by compiler, the generated code will be larger, but the linker will NOT produce a duplicate definition error

Figure 6.13: Inlining of function

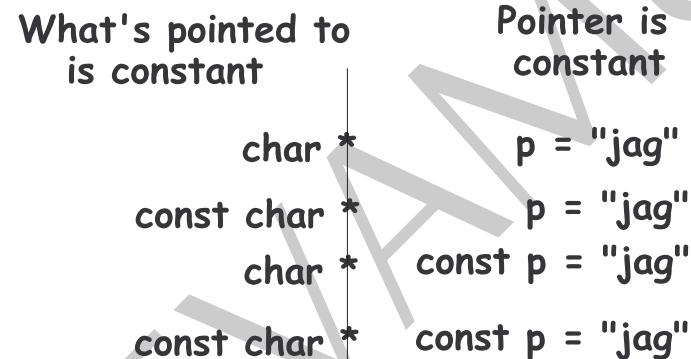
```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c8.h  
4 -----*/  
5  
6 #ifndef C8_H  
7 #define C8_H  
8 /*-----  
9 Definition of a class  
10 -----*/  
11 class group_private {  
12 private:  
13     int _age ;  
14     char _name[10] ;  
15     bool _is_male ;  
16  
17 public:  
18     void set_age(int a) {_age = a ; }  
19     int age() const {return _age ;}  
20     void set_male();  
21     void set_female();  
22     bool is_male() const ;  
23     void set_name(const char* s);  
24     const char* name() const ;  
25     void print() const ;  
26 };  
27  
28 /*-----  
29 Implementation of set male  
30 -----*/  
31 inline void group_private::set_male() {  
32     _is_male = true ;  
33 }  
34  
35 /*-----  
36 Implementation of set female  
37 -----*/  
38 inline void group_private::set_female() {  
39     _is_male = false ;  
40 }  
41  
42 /*-----  
43 Implementation of is_male  
44 -----*/  
45 inline bool group_private::is_male() const {  
46     return _is_male ;  
47 }  
48  
49 /*-----  
50 Implementation of get name  
51 -----*/  
52 inline const char* group_private::name() const {  
53     return _name ;  
54 }  
55 #endif
```

```
1 /*-----
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename:c8.cpp
4 -----
5 #include "c8.h"
6
7 #include <iostream>
8 using namespace std;
9
10 /*
11 Implementation of set name
12 -----
13 void group_private::set_name(const char* s){
14     strcpy(_name,s);
15 }
16
17 /*
18 Implementation of print
19 -----
20 void group_private::print() const {
21     cout << name() << " " << age() << " ";
22     cout << (is_male() ? "male" : "female") << endl ;
23 }
24
```

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c8_main.cpp  
4  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 Program exited with status code 0.  
7  
8 -----*/  
9 #include "c8.h"  
10  
11 /*-----  
12 TOM JR 7 male  
13 -----*/  
14 static void accessing_private_elements_of_a_class()  
15 {  
16     group_private tom;  
17     //tom.age = 7;  
18     // error C2248: 'group_private::age' : cannot access private member declared in class 'group_private'  
19     //tom.is_male = true;  
20     tom.set_name("TOM JR");  
21     tom.set_age(7);  
22     tom.set_male();  
23     tom.print();  
24 }  
25  
26 /*-----  
27 -----*/  
28 /*-----*/  
29 int main(){  
30     accessing_private_elements_of_a_class();  
31     return 0;  
32 }  
33
```

6.11 Understanding const

```
char* p = "jag"; //Non constant pointer, Non constant data  
const char* p = "jag" ; //Non constant pointer, constant data  
char* const p = "jag" ; //constant pointer, non constant data  
const char* const p = "jag" ; //constant pointer, constant data
```



Use const whenever possible

Figure 6.14: Use **const** whenever possible

```
1 -----
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: const.cpp
4 -----*/
5
6 #include <iostream>
7 using namespace std;
8
9 -----
10 integer test
11 -----*/
12 static void int_test() {
13     int x = 4 ;
14     x = 10 ;
15     //const int y = 2 ;
16     //y = 89 ;
17     //FAILS: error C3892: 'y' : you cannot assign to a variable that is const
18 }
19
20 -----
21 Generic print routine
22 -----*/
23 static void print(const char* title, const char* s) {
24     cout << title << endl ;
25     int l = strlen(s) ;
26     for (int i = 0; i < l; i++) {
27         cout << s[i] ;
28     }
29     cout << endl ;
30 }
31
32 -----
33 Non constant pointer Non constant data
34 -----*/
35 static void ncp_ncd() {
36     char* p = new char[4];
37     strcpy(p,"NPU");
38     print("ncp_ncd BEFORE: ", p);
39     p[0] = 'L';
40     print("ncp_ncd AFTER: ", p);
41     delete [] p;
42 }
43
44 -----
45 Non constant pointer Constant data
46 -----*/
47 static void ncp_cd() {
48     const char* p = "MIT";
49     cout << "p = " << p << endl ;
50     //p[0] = 'N';
51     //error C3892: 'p' : you cannot assign to a variable that is const
52     const char* q = "UCSC";
53     p = q;
54 }
```

```
56 /*-----  
57 Constant pointer Non constant data  
58 -----*/  
59 static void cp_ncd(){  
60     char* const p = new char[4];  
61     strcpy(p,"NPU");  
62     print("cp_ncd BEFORE: ", p);  
63     p[0] = 'L';  
64     print("cp_ncd AFTER: ", p);  
65  
66     char* const p1 = new char[4];  
67     strcpy(p1,"MIT");  
68     print("cp_ncd BEFORE2: ", p1);  
69 //p = p1;  
70 // error C3892: 'p' : you cannot assign to a variable that is const  
71     delete [] p;  
72     delete [] p1;  
73 }  
74  
75 /*-----  
76 Constant pointer Constant data  
77 -----*/  
78 static void cp_cd(){  
79     const char* const p = "MIT";  
80 //p[0] = 'K';  
81 //error C3892: 'p' : you cannot assign to a variable that is const  
82  
83     const char* q = "UCSC";  
84 //p = q;  
85 //error C3892: 'p' : you cannot assign to a variable that is const  
86 }  
87  
88 /*-----  
89 Main  
90 -----*/  
91 int main(){  
92     int _test();  
93     cout << "_____" << endl;  
94     ncp_ncd();  
95     cout << "_____" << endl;  
96     ncp_cd();  
97     cout << "_____" << endl;  
98     cp_ncd();  
99     cout << "_____" << endl;  
100    cp_cd();  
101    cout << "_____" << endl;  
102    return 0;  
103 }  
104 }
```

6.12. NEED FOR CONSTRUCTORS

6.12 Need for constructors

```
1 /*-----
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: constdest_c_way.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 class book
11 -----
12 class book {
13 public:
14     void init(const char *s, int c = 0, bool x = false);
15     void finish();
16     void print() const;
17
18 private:
19     char* _name;
20     int _cost;
21     bool _in;
22 };
23
24 /*
25 init function
26 -----
27 void book::init(const char *s, int c, bool x) {
28     cout << "In book init " << s << endl;
29     _cost = c;
30     _in = x;
31     int l = strlen(s) + 1;
32     this->_name = new char[l];
33     //you can write without this as: _name = new char[l];
34     strcpy(_name,s);
35 }
36
37 /*
38 finish function
39 -----
40 void book::finish() {
41     cout << "In book finish " << _name << endl;
42     delete [] this->_name;
43     //Without this: delete [] _name;
44 }
45
46 /*
47 print
48 -----
49 void book::print() const {
50     cout << "Name " << _name << " Cost " << _cost;
51     (_in) ? cout << " In " : cout << " Out ";
52     cout << endl;
53 }
54
55 */
```

```
56 main
57
58 In book int algorithm
59 In book int C++
60 In book int VLSI design
61 Name algorithm Cost 0 Out
62 Name C++ Cost 79 Out
63 Name VLSI design Cost 168 In
64 In book int A book on Java
65 Name A book on Java Cost 289 In
66 In book finish algorithm
67 In book finish C++
68 In book finish VLSI design
69 In book finish A book on Java
70 -----*/
71 int main() {
72     book b1;
73     b1.init("algorithm");
74     book b2;
75     b2.init("C++",79);
76     book b3;
77     b3.init("VLSI design", 168,true);
78     b1.print();
79     b2.print();
80     b3.print();
81     book* b4 = new book;
82     b4->init("A book on Java",289,1) ;
83     b4->print();
84
85     b1.finish();
86     b2.finish();
87     b3.finish();
88     b4->finish();
89     delete b4 ;
90     return 0 ;
91 }
92
```

```
1 /*-----
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: constdest.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 class book {
10 public:
11     book(const char *s, int c = 0, bool x = false);
12     ~book();
13     void print() const;
14
15 private:
16     char* _name;
17     int _cost;
18     bool _in;
19 };
20
21 /*-----
22 Constructor
23 -----
24 book::book(const char *s, int c, bool x)
25     : _name(NULL), _cost(c), _in(x){
26     cout << "In book Constructor " << s << endl;
27     int l = strlen(s) + 1;
28     this->_name = new char[l];
29     //you can write without this as: _name = new char[l];
30     strcpy(_name,s);
31 }
32
33 /*-----
34 Destructor
35 -----
36 book::~book() {
37     cout << "In book Destructor " << _name << endl;
38     delete [] this->_name;
39     //Without this: delete [] _name ;
40 }
41
42 /*-----
43 print
44 -----
45 void book::print() const {
46     cout << "Name " << _name << " Cost " << _cost ;
47     (_in) ? cout << " In " : cout << " Out ";
48     cout << endl ;
49 }
50
51 /*-----
52 main
53
54 In book Constructor algorithm
55 In book Constructor C++
```

```
56 In book Constructor VLSI design
57 Name algorithm Cost 0 Out
58 Name C++ Cost 79 Out
59 Name VLSI design Cost 168 In
60 In book Constructor A book on Java
61 Name A book on Java Cost 289 In
62 In book Destructor A book on Java
63 -----*/
64 int main() {
65     book b1("algorithm");
66     book b2("C++",79);
67     book b3("VLSI design", 168,true);
68     b1.print();
69     b2.print();
70     b3.print();
71     book* b4 = new book("A book on Java",289,1) ;
72     b4->print();
73     delete b4;
74     return 0;
75 }
76
```

Constructors

How do you know a constructor?

```
class obj {  
    obj(<input parameters, may be null>)  
};
```

1. Name of the constructor **MUST BE** name of the class
2. Constructor does **NOT** return any thing (**not even void**)
3. If you don't write a constructor, compiler will give one which does nothing as follows:

```
class obj {  
    obj() {}  
}
```
4. There can be many constructors, same name but different parameters.
5. It cannot have the qualifiers like static or const
6. Constructors can be private

When constructors are called ?

A constructor is automatically called when objects are created

Case 1: obj p ;

Case 2: obj *p = new obj(<input parameters>) ;

Creation of pointers, reference and primitive types does not call constructors

Figure 6.15: Constructors

Destructor

How do you know a destructor?

```
class obj {
    ~obj() :
}
```

1. The name of the destructor **MUST BE** name of the class preceded by ~
2. There are no inputs to destructor
3. Destructor does **NOT** return anything
4. If you don't write a destructor, compiler will give one with nothing inside
- 5. Destructor cannot be private**

C++ guarantees that destructor is called when the object dies

When destructor is called?

Case 1: When an object goes out of scope

```
{
    obj p ;
    <may be some code here>
}
```

Case 2: When you call delete

```
{
    obj *p = new(obj) ;
    delete p ; ←
}
```

Destructor is NOT called when a pointers, reference or primitive type dies

Figure 6.16: Destructor

6.13 Creating temporary objects on the stack, phase 2

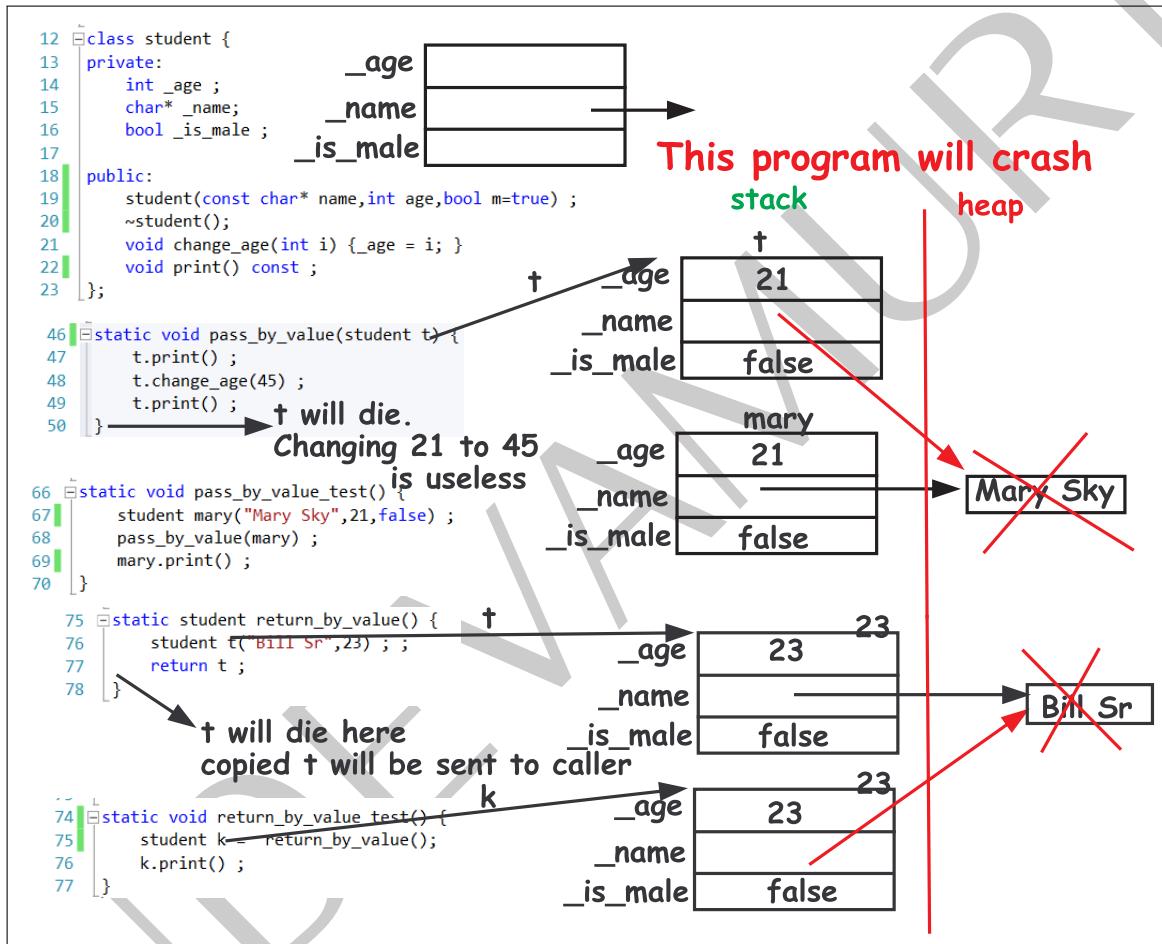


Figure 6.17: Creation of temporary objects due to pass by value and return by value

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:c7m.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 Declartion of a class - INTERFACE  
11 -----*/  
12 class student {  
13 private:  
14     int _age ;  
15     char* _name;  
16     bool _is_male ;  
17  
18 public:  
19     student(const char* name,int age,bool m=true) ;  
20     ~student();  
21     void change_age(int i) {_age = i; }  
22     void print() const ;  
23 };  
24  
25 /*-----  
26 Implementation of print  
27 -----*/  
28 void student::print() const {  
29     cout << _name << " " << _age << " "  
30     cout << (_is_male ? "male" : "female") << endl ;  
31 }  
32  
33 /*-----  
34 Implementation of constructor  
35 -----*/  
36 student::student(const char* name,int age,bool m):_name(0),_age(age),_is_male(m) {  
37     cout << "In student constructor " << name << ":" << age << ":" << boolalpha << m << endl ;  
38     _name = new char[strlen(name)+1] ;  
39     strcpy(_name,name) ;  
40 }  
41  
42 /*-----  
43 Implementation of distructor  
44 -----*/  
45 student::~student() {  
46     cout << "In student disstructor " << _name << ":" << _age << ":" << boolalpha << _is_male << endl ;  
47     delete [] _name;  
48 }  
49  
50 /*-----  
51 -----*/  
52 static void pass_by_value(student t) {  
53     t.print() ;  
54     t.change_age(45) ;  
55     t.print() ;  
56 }  
57 }  
58  
59 /*-----  
60 Mary Sky 21 female  
61 Mary Sky 45 female  
62 Mary Sky 21 female  
63 -----*/  
64 static void pass_by_value_test() {  
65     student mary("Mary_Sky",21,false) ;  
66     pass_by_value(mary) ;
```

```
67     mary.print() ;
68 }
69
70 /*-----
71 -----
72 static student return_by_value() {
73     student t("Bill Sr",23) ;
74     return t ;
75 }
76 */
77 /*
78 Bill Sr 23 male
79 -----
80 -----*/
81 static void return_by_value_test() {
82     student k = return_by_value();
83     k.print() ;
84 }
85
86 /*
87 THIS PROGRAM WILL CRASH
88 -----*/
89 int main() {
90     pass_by_value_test();
91     return_by_value_test() ;
92     return 0 ;
93 }
94
```

6.14. SHALLOW COPYING OF OBJECTS, PHASE 2

6.14 Shallow copying of objects, phase 2

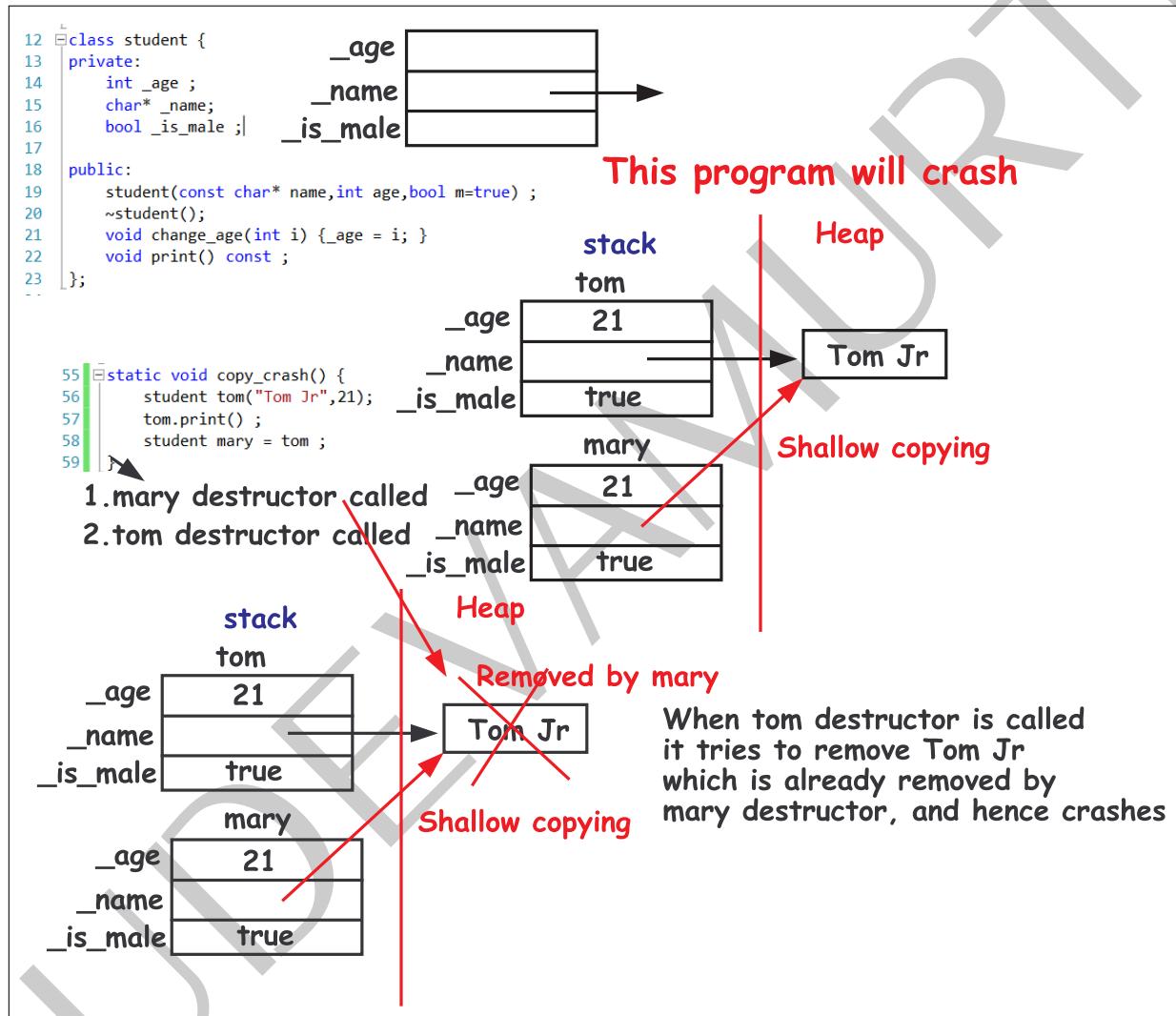


Figure 6.18: Copying objects that has data on the heap

```
1 /*-----
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename:c6m.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 Declaration of a class - INTERFACE
11 -----
12 class student {
13 private:
14     int _age ;
15     char* _name;
16     bool _is_male ;
17
18 public:
19     student(const char* name,int age,bool m=true);
20     ~student();
21     void change_age(int i) {_age = i; }
22     void print() const ;
23 };
24
25 /*
26 Implementation of print
27 -----
28 void student::print() const {
29     cout << _name << " " << _age << " ";
30     cout << (_is_male ? "male" : "female") << endl ;
31 }
32
33 /*
34 Implementation of constructor
35 -----
36 student::student(const char* name,int age,bool m) {
37     cout << "In student constructor " << name << ":" << age << ":" << boolalpha << m << endl ;
38     _name = new char[strlen(name)+1];
39     strcpy(_name,name);
40     _age = age ;
41     _is_male = m ;
42 }
43
44 /*
45 Implementation of distructor
46 -----
47 student::~student(){
48     cout << "In student disstructor " << _name << ":" << _age << ":" << boolalpha << _is_male << endl ;
49     delete [] _name;
50 }
51
52 /*
53 -----
54 static void copy_crash() {
```

```
56 student tom("Tom Jr",21);
57 tom.print();
58 student mary = tom;
59 }
60
61 /*-----
62 THIS PROGRAM WILL CRASH
63 -----*/
64 int main() {
65 copy_crash();
66 return 0;
67 }
68
```

6.15 Copy constructor and equal operator

6.15. COPY CONSTRUCTOR AND EQUAL OPERATOR

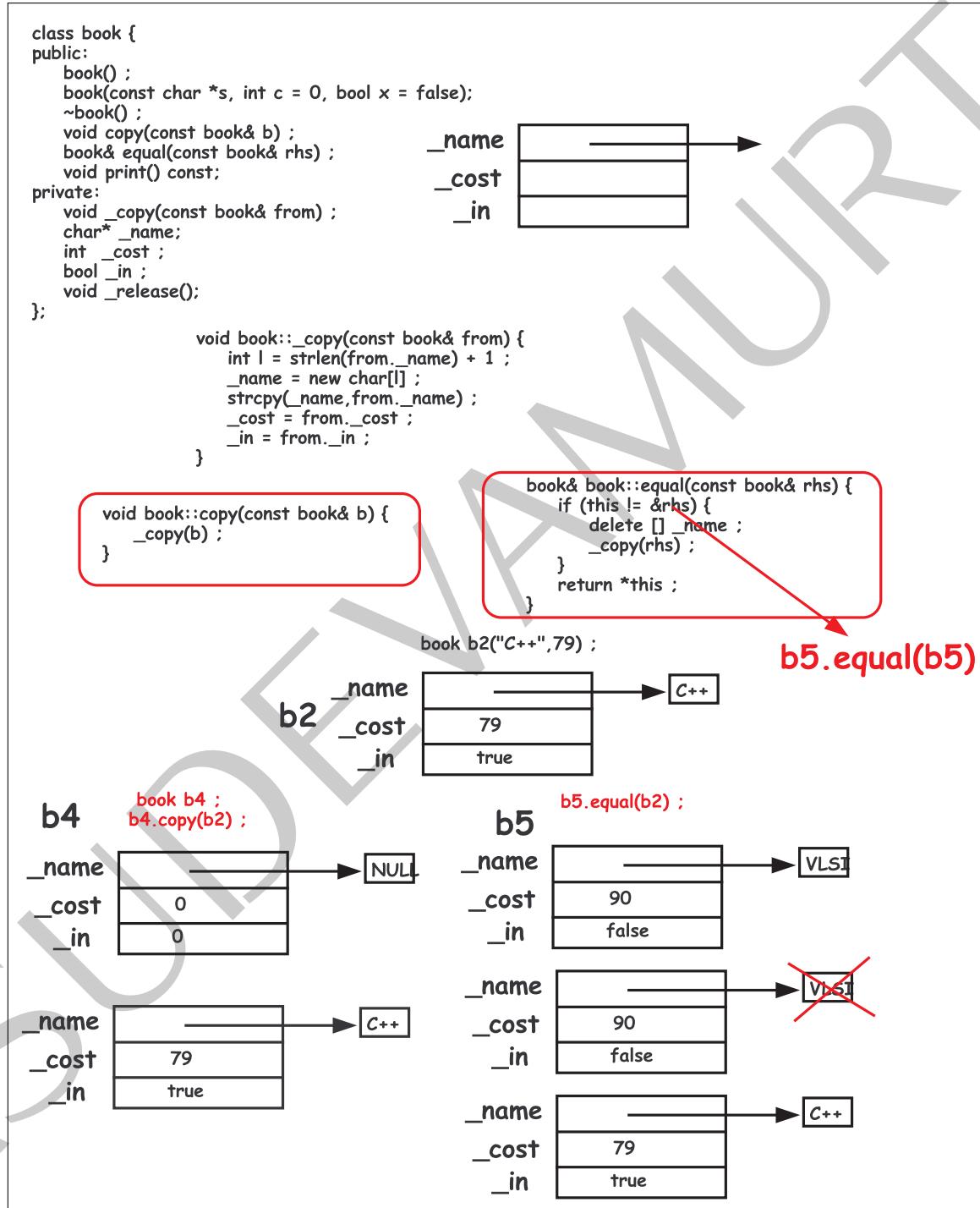


Figure 6.19: Deep copying of objects

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevarmurthy  
3 Filename: copyeq_c_way.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 class book {  
10 public:  
11     book();  
12     book(const char *s, int c = 0, bool x = false);  
13     ~book();  
14     void copy(const book& b);  
15     book& equal(const book& rhs);  
16     void print() const;  
17 private:  
18     void _copy(const book& from);  
19     char* _name;  
20     int _cost;  
21     bool _in;  
22     void _release();  
23 };  
24  
25 /*-----  
26 Constructor  
27 -----*/  
28 book::book(): _name(NULL),_cost(0),_in(0){  
29     cout << "book::book(): " << endl;  
30 }  
31  
32 /*-----  
33 Constructor  
34 -----*/  
35 book::book(const char *s, int c, bool x)  
36     : _name(NULL),_cost(c),_in(x){  
37     cout << "book::book(const char *s, int c, bool x): " << s << endl;  
38     int l = strlen(s) + 1;  
39     _name = new char[l];  
40     strcpy(_name,s);  
41 }  
42  
43 /*-----  
44 Release heap memory of this class  
45 -----*/  
46 void book::_release(){  
47     delete [] _name;  
48 }  
49  
50 /*-----  
51 Destructor  
52 -----*/  
53 book::~book(){  
54     cout << "In book Destructor " << _name << endl;  
55     _release();
```

```
56 }
57
58 /*-----
59 Helper: copy function
60 -----
61 void book::_copy(const book& from) {
62     int l = strlen(from._name) + 1;
63     _name = new char[l];
64     strcpy(_name,from._name);
65     _cost = from._cost;
66     _in = from._in;
67 }
68
69 /*-----
70 copy
71 -----
72 void book::copy(const book& b){
73     cout << "In copy: " << b._name << endl ;
74     _copy(b);
75 }
76
77 /*-----
78 equal
79 -----
80 book& book::equal(const book& rhs) {
81     cout << "In equal: " << rhs._name << endl ;
82     if (this != &rhs) {
83         _release();
84         _copy(rhs);
85     }
86     return *this;
87 }
88
89 /*-----
90 print
91 -----
92 void book::print() const {
93     cout << "Name " << _name << " Cost " << _cost ;
94     (_in)? cout << " In " : cout << " Out ";
95     cout << endl ;
96 }
97
98 /*-----
99 main
100 -----
101 int main() {
102     book b1("algorithm");
103     book b2("C++",79);
104     book b3("VLSI design", 168,true);
105     book b4 :
106     b4.copy(b2);
107     book b5 ;
108     b5.copy(b1);
109     b1.print();
110    b2.print();
```

```
111 b3.print();
112 b4.print();
113 b5.print();
114 b1.equal(b3);
115 b2.equal(b4);
116 b1.print();
117 b2.print();
118 b3.print();
119 b4.print();
120 return 0;
121 }
122 /*
123 output of the above program
124 -----
125 */
126 book::book(const char *s, int c, bool x): algorithm
127 book::book(const char *s, int c, bool x): C++
128 book::book(const char *s, int c, bool x): VLSI design
129 book::book():
130 In copy: C++
131 book::book():
132 In copy: algorithm
133 Name algorithm Cost 0 Out
134 Name C++ Cost 79 Out
135 Name VLSI design Cost 168 In
136 Name C++ Cost 79 Out
137 Name algorithm Cost 0 Out
138 In equal: VLSI design
139 In equal: C++
140 Name VLSI design Cost 168 In
141 Name C++ Cost 79 Out
142 Name VLSI design Cost 168 In
143 Name C++ Cost 79 Out
144 In book Destructor algorithm
145 In book Destructor C++
146 In book Destructor VLSI design
147 In book Destructor C++
148 In book Destructor VLSI design
149
150 */
151
```

6.15. COPY CONSTRUCTOR AND EQUAL OPERATOR

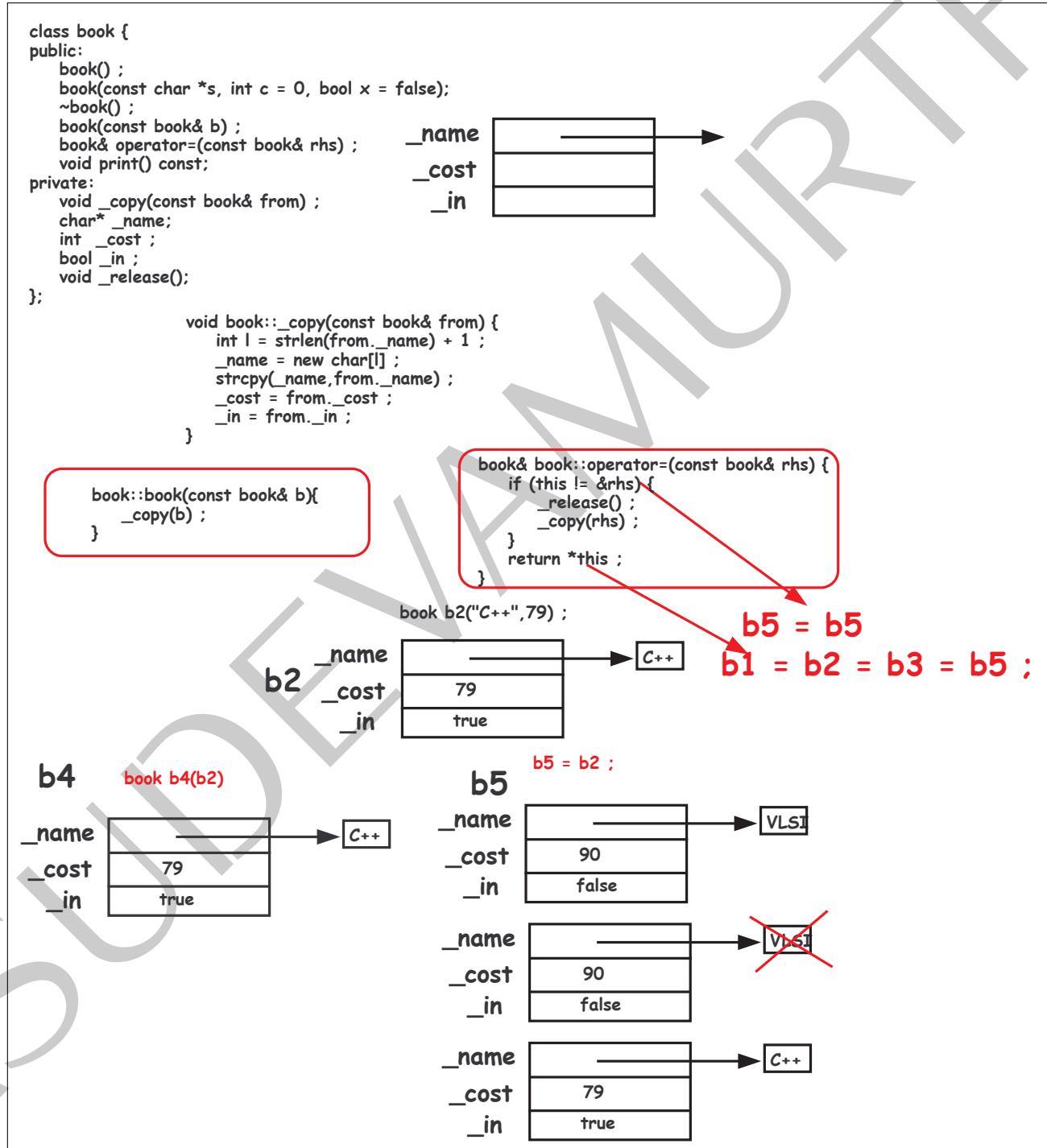


Figure 6.20: Deep copying of objects using copy constructor and equal operator

```
1 /*-
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy
3 Filename: copyeq.cpp
4 */
5
6 #include <iostream>
7 using namespace std;
8
9 class book {
10 public:
11     book(const char *s, int c = 0, bool x = false);
12     ~book();
13     book(const book& b);
14     book& operator=(const book& rhs);
15     void print() const;
16 private:
17     void _copy(const book& from);
18     char* _name;
19     int _cost;
20     bool _in;
21     void _release();
22 };
23
24 /*-
25 Constructor
26 */
27 book::book(const char *s, int c, bool x)
28 : _name(NULL),_cost(c),_in(x){
29     cout << "In book Constructor " << s << endl ;
30     int l = strlen(s) + 1 ;
31     _name = new char[l];
32     strcpy(_name,s);
33 }
34
35 /*-
36 Release heap memory of this class
37 */
38 void book::_release() {
39     delete [] _name;
40 }
41
42 /*-
43 Destructor
44 */
45 book::~book() {
46     cout << "In book Destructor " << _name << endl ;
47     _release();
48 }
49
50 /*-
51 Helper: copy function
52 */
53 void book::_copy(const book& from) {
54     int l = strlen(from._name) + 1 ;
55     _name = new char[l];
```

```
56 strcpy(_name,from._name);
57 _cost = from._cost;
58 _in = from._in;
59 }
60
61 /*-
62 Copy constructor
63 -----
64 book::book(const book& b){
65 cout << "In copy Constructor " << b._name << endl ;
66 _copy(b);
67 }
68
69 /*-
70 equal operator
71 -----
72 book& book::operator=(const book& rhs) {
73 cout << "In equal operator " << rhs._name << endl ;
74 if (this != &rhs) {
75 _release();
76 _copy(rhs);
77 }
78 return *this;
79 }
80
81 /*-
82 print
83 -----
84 void book::print() const {
85 cout << "Name " << _name << " Cost " << _cost ;
86 (_in)? cout << " In " : cout << " Out ";
87 cout << endl ;
88 }
89
90 /*-
91 main
92 -----
93 int main(){
94 book b1("algorithm");
95 book b2("C++",79);
96 book b3("VLSI design", 168,true);
97 book b4(b2);
98 book b5 = b1;
99 b1.print();
100 b2.print();
101 b3.print();
102 b4.print();
103 b5.print();
104 b1 = b3;
105 b2 = b4;
106 b1.print();
107 b2.print();
108 b3.print();
109 b4.print();
110 return 0;
```

```
111 }
112 /*-
113 output of the above program
114 -----
115 /*
116 In book Constructor algorithm
117 In book Constructor C++
118 In book Constructor VLSI design
119 In copy Constructor C++
120 In copy Constructor algorithm
121 Name algorithm Cost 0 Out
122 Name C++ Cost 79 Out
123 Name VLSI design Cost 168 In
124 Name C++ Cost 79 Out
125 Name algorithm Cost 0 Out
126 In equal operator VLSI design
127 In equal operator C++
128 Name VLSI design Cost 168 In
129 Name C++ Cost 79 Out
130 Name VLSI design Cost 168 In
131 Name C++ Cost 79 Out
132 In book Destructor algorithm
133 In book Destructor C++
134 In book Destructor VLSI design
135 In book Destructor C++
136 In book Destructor VLSI design
137
138 */
139
```

6.15. COPY CONSTRUCTOR AND EQUAL OPERATOR

Copy Constructor

How do you know a Copy constructor?

```
class obj {  
    obj(const obj& b)  
};
```

1. Name of the copy constructor **MUST BE** name of the class
2. Copy constructor does **NOT** return any thing.
3. Note that the object from which we copy is a constant object passed by reference
4. If you don't write a copy constructor, objects will be bit wised copied.

obj q(p)

without copy constructor

But we want like this:

Define a copy constructor and an assignment operator for classes with dynamically allocated memory

Figure 6.21: Copy constructor

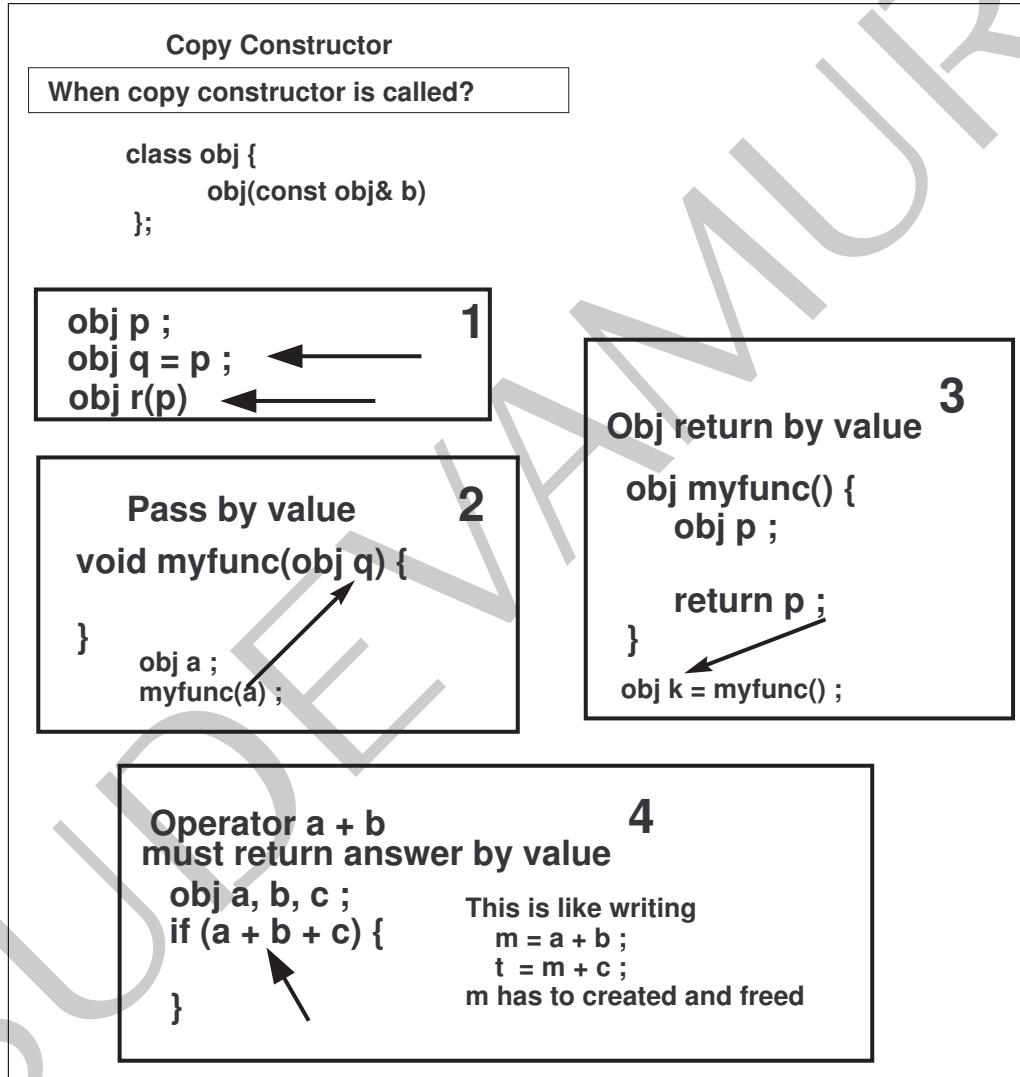


Figure 6.22: When copy constructor is called?

6.15. COPY CONSTRUCTOR AND EQUAL OPERATOR

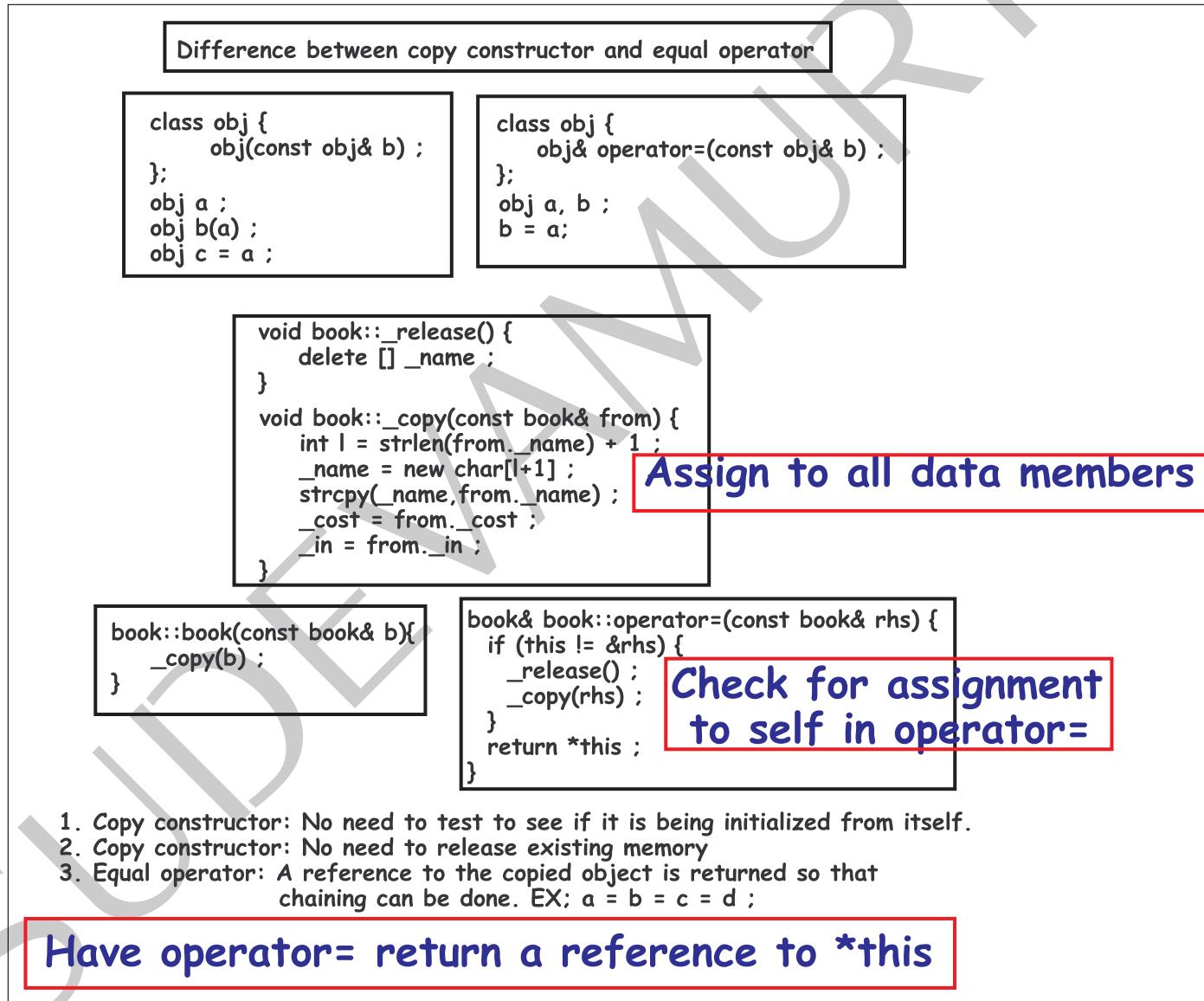


Figure 6.23: Difference between copy constructor and equal operator

6.15.1 Fraction class

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: c9.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 -----*/  
12 class fraction {  
13 public:  
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {  
15         cout << "In fraction constructor " << "(" << _numerator << "/" << _denominator << ")\\n" ;  
16     }  
17     ~fraction() {  
18         cout << "In fraction distructor " << "(" << _numerator << "/" << _denominator << ")\\n" ;  
19         _numerator = 999999; _denominator=12345;  
20     }  
21     fraction mult(const fraction& b) {  
22         return fraction((this->_numerator * b._numerator),(this->_denominator * b._denominator)) ;  
23     }  
24     fraction mult1(const fraction& b) {  
25         fraction x(this->_numerator * b._numerator, this->_denominator * b._denominator) ;  
26         return x ;  
27     }  
28     friend ostream& operator<< (ostream& o, const fraction& a) {  
29         cout << a._numerator << "/" << a._denominator;  
30         return o ;  
31     }  
32     fraction(const fraction& a) {  
33         cout << "In copy constructor " << "(" << a._numerator << "/" << a._denominator << ")\\n" ;  
34         _numerator = a._numerator;  
35         _denominator = a._denominator;  
36     }  
37     fraction& operator=(const fraction& rhs) {  
38         cout << "In Equal operator " << "(" << rhs._numerator << "/" << rhs._denominator << ")\\n" ;  
39         if (this != &rhs) {  
40             _numerator = rhs._numerator;  
41             _denominator = rhs._denominator;  
42         }  
43         return *this ;  
44     }  
45     void change(int x, int y) {  
46         _numerator = x ;  
47         _denominator = y ;  
48     }  
49 private:  
50     int _numerator ;  
51     int _denominator;  
52 };  
53  
54 /*-----  
55 To show:  
56 creation of a calls copy constructor. a is a local object  
57 Also, destructor is called on a when function ends -- local object deleted  
58 -----*/  
59 static void pass_by_value(fraction a) {  
60     //a is a local object. Constructor is called  
61     a.change(100,200) ;  
62     cout << "a in pass_by_value = " << a << endl ;  
63     //a is a local object. destructor is called and dies  
64     //change is never reflected to the external world  
65 }  
66
```

```
67 /*-----  
68 To show NO copy constructor is called. a is same as called object  
69 Also, destructor NOT called when function ends  
70 -----*/  
71 static void pass_by_reference(fraction& a) {  
72     //a is reference object. No constructor is called  
73     a.change(100,200) ;  
74     cout << "a pass_by_reference = " << a << endl ;  
75     //a is reference. No destructor is called  
76 }  
77  
78 /*-----  
79 To show:  
80 creation of t calls constructor -- local object  
81 Copy constructor is called on t and copied object is returned to the caller  
82 destructor is called on t at the end of function -- local object  
83 -----*/  
84 static fraction return_by_value(const fraction& a) {  
85     fraction t ; //local object  
86     t.change(100,200) ;  
87     cout << "t in return_by_value = " << t << endl ;  
88     return t ; //copy constructor called and copied object returned. local object t dies  
89 }  
90  
91 /*-----  
92 You are returning t to the caller which is dead  
93 warning C4172: returning address of local variable or temporary  
94 -----*/  
95 static fraction& dangerous_return_by_alias(const fraction& a) {  
96     fraction t ; //local object  
97     t.change(100,200) ;  
98     cout << "t in dangerous_return_by_alias = " << t << endl ;  
99     return t ; //local object dies and you are returning reference to the dead object  
100 }  
101  
102 /*-----  
103 -----*/  
104 int main() {  
105     fraction x(3,8) ;  
106     cout << "x = " << x << endl ;  
107     fraction y(x) ;  
108     cout << "y = " << y << endl ;  
109     fraction z = x ;  
110     cout << "z = " << z << endl ;  
111     z = y ;  
112     {  
113         fraction yx = y.mult(x) ;  
114         cout << "y * x = " << yx << endl ;  
115     }  
116     {  
117         fraction yx = y.mult1(x) ;  
118         cout << "y * x = " << yx << endl ;  
119     }  
120     {  
121         cout << "x = " << x << endl ;  
122         pass_by_value(x) ;  
123         cout << "x = " << x << endl ;  
124     }  
125     {  
126         cout << "x = " << x << endl ;  
127         pass_by_reference(x) ;  
128         cout << "x = " << x << endl ;  
129     }  
130     {  
131 }
```

```
133     cout << "y = " << y << endl ;
134     fraction p = return_by_value(y) ;
135     cout << "p = " << p << endl ;
136
137 }
138 {
139     cout << "y = " << y << endl ;
140     fraction p = dangerous_return_by_alias(y) ;
141     cout << "p = " << p << endl ;
142
143     cout << "y = " << y << endl ;
144     fraction& q = dangerous_return_by_alias(y) ;
145     cout << "q = " << q << endl ;
146 }
147 return 0 ;
148 }
149 }
```

6.16 Private constructors

PRIVATE CONSTRUCTOR

```

class point {
private:
    double _x;
    double _y;
    static bool display;
    point::point(double x, double y) :_x(x), _y(y) {
        if (display) {
            cout << "In private constructor " << x << " " << y << endl;
        }
    }
public:
    static point rectangular(double x, double y) {
        return point(x, y);
    }
    static point polar(double r, double theta) {
        //x = r*cos(theta);
        //y = r*sin(theta);
        double c = cos(theta * 3.14159265 / 180.00);
        double s = sin(theta * 3.14159265 / 180.00);
        return point(r*c, r*s);
    }
    friend ostream& operator<<(ostream& o, const point& p) {
        o << "x = " << p._x << " y = " << p._y;
        return o;
    }
};

int main() {
    //point p; //'point' : no appropriate default constructor
    point p1 = point::rectangular(12, 5);
    cout << "rectangular(12, 5) = " << p1 << endl;
    point p2 = point::polar(13, 22.6);
    cout << "polar(13, 22.6) = " << p2 << endl;
    return 0;
}

```

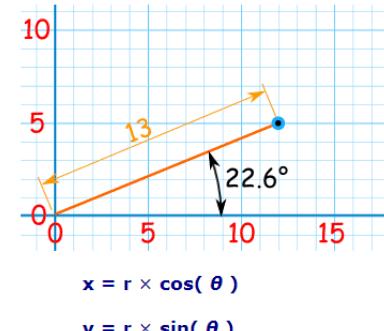
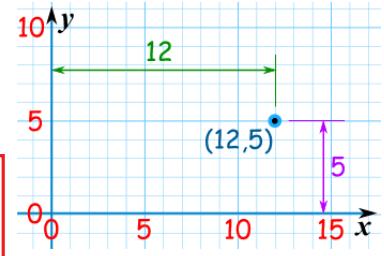


Figure 6.24: Need for private constructor

```
1 /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 privateconstructor.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7 #include <math.h>  
8  
9 /*-----  
10 Definition of the class  
11 -----*/  
12 class point {  
13 private:  
14     double _x;  
15     double _y;  
16     static bool _display;  
17     point::point(double x, double y) :_x(x), _y(y) {  
18         if (_display()) {  
19             cout << "In private constructor " << x << " " << y << endl;  
20         }  
21     }  
22 public:  
23     static void setDisplay(bool x) {  
24         _display = x;  
25     }  
26     static bool display() {  
27         return _display;  
28     }  
29     static point rectangular(double x, double y) {  
30         return point(x, y);  
31     }  
32     static point polar(double r, double theta) {  
33         //x = r*cos(theta) ;  
34         //y = r*sin(theta) ;  
35         double c = cos(theta * 3.14159265 / 180.00);  
36         double s = sin(theta * 3.14159265 / 180.00);  
37         return point(r*c, r*s);  
38     }  
39     friend ostream& operator<<(ostream& o, const point& p) {  
40         o << "x = " << p._x << " y = " << p._y;  
41         return o;  
42     }  
43 };  
44 /*-----  
45 Definition of the static variable  
46 -----*/  
47 /*-----*/  
48 bool point::_display = true;  
49 /*-----*/  
50 /*-----*/  
51 test  
52 http://www.mathsisfun.com/polar-cartesian-coordinates.html  
53 /*-----*/  
54 int main() {  
55     //point p; // 'point' : no appropriate default constructor  
56     point::setDisplay(true);  
57     point p1 = point::rectangular(12,5);  
58     cout << "rectangular(12,5) = " << p1 << endl;  
59     point p2 = point::polar(13,22.6);  
60     cout << "polar(13,22.6) = " << p2 << endl;  
61     return 0;  
62 }  
63  
64  
65
```

6.17 Constructor initialization list

6.17.1 Initialization versus assignment

The diagram consists of five code snippets arranged in a grid-like layout:

- Top Left:** Class `one`. It has a constructor `one(int x, int y)` that initializes member variables `_x` and `_y`. A note says: "Object value is assigned. Inefficient, but OK".
- Top Right:** Class `two`. It has a constructor `two(int x, int y):_x(x),_y(y){}` that initializes member variables `_x` and `_y`. A note says: "Object value is initialized".
- Middle Left:** Class `three`. It has a constructor `three(int x, int y)` that attempts to assign values to `_x` and `_y`. Red notes say: "Constant _x, Cannot be assigned" and "WILL NOT COMPILE".
- Middle Right:** Class `three_reference`. It has a constructor `three_reference(int x, int y)` that attempts to assign values to `_x` and `_y`. Red notes say: "Reference _x. Cannot be assigned" and "WILL NOT COMPILE".
- Bottom Left:** Class `four`. It has a constructor `four(int x, int y):_x(x),_y(y){}` that initializes member variables `_x` and `_y`.
- Bottom Right:** Class `four_reference`. It has a constructor `four_reference(int x, int y):_x(x),_y(y){}` that initializes member variables `_x` and `_y`.

A central blue note at the bottom of the grid states: "Initialization is mandatory for const and reference". Below this, a red box contains the blue text: "Prefer initialization to assignment in constructors".

Figure 6.25: Initialization versus assignment

6.17.2 Order of initialization

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: initialization.cpp  
4 -----*/  
5 #include <iostream>  
6 using namespace std;  
7  
8 /*-----  
9 When object one is created data is assigned, NOT initialized  
10 -----*/  
11 class one {  
12 public:  
13     one(int x, int y){  
14         _x = x;  
15         _y = y;  
16     }  
17  
18 private:  
19     int _x;  
20     int _y;  
21 };  
22  
23 /*-----  
24 When object two is created, data is initialized  
25 -----*/  
26 class two {  
27 public:  
28     two(int x, int y):_x(x),_y(y){}  
29 private:  
30     int _x;  
31     int _y;  
32 };  
33  
34 #if 0  
35 /*-----  
36 When object three is created data is assigned, NOT initialized  
37 -----*/  
38 WILL NOT compile  
39 error C2758: 'three::_x' :  
40 must be initialized in constructor base/member initializer list  
41 -----*/  
42 class three {  
43 public:  
44     three(int x, int y){  
45         _x = x;  
46         _y = y;  
47     }  
48 private:  
49     const int _x;  
50     int _y;  
51 };  
52 #endif  
53  
54 #if 0  
55 /*-----
```

```
56 When object three is created data is assigned, NOT initialized
57
58 WILL NOT compile
59 error C2758: 'three_reference::_x' : must be initialized
60 in constructor base/member initializer list
61 -----
62 class three_reference {
63 public:
64     three_reference(int x, int y) {
65         _x = x;
66         _y = y;
67     }
68 private:
69     int& _x;
70     int _y;
71 };
72 #endif
73
74 /*-
75 When object four is created, data is initialized
76 -----
77 class four {
78 public:
79     four(int x, int y):_x(x),_y(y){}
80 private:
81     const int _x;
82     int _y;
83 };
84
85 class four_reference {
86 public:
87     four_reference(int x, int y):_x(x),_y(y){}
88 private:
89     int& _x;
90     int _y;
91 };
92
93 /*-
94 _size is guaranteed to be initialized before _s
95 -----
96 class five {
97 public:
98     five(char *t):_size(strlen(t)),_s(new char[_size+1]) {
99         strcpy(_s,t);
100        cout << "string = " << _s << endl;
101    }
102 private:
103     int _size;
104     char* _s;
105 };
106
107 /*-
108 _s is guaranteed to be initialized before _size
109 What _size it uses ???
110 -----
```

```
111 class six {
112 public:
113     six(char *t):_size(strlen(t)), _s(new char[_size+1]) {
114         strcpy(_s,t);
115         cout << "string = " << _s << endl;
116     }
117 private:
118     char* _s;
119     int _size;
120 };
121
122 int main() {
123     one u1(30,40);
124     two u2(50,60);
125     //three u3(67,99);
126     //three_reference ur(8,89);
127     four u4(6,8);
128     four_reference u4r(6,8);
129     five u5("Jagadeesh");
130     six u6("ucsc extension at Santa Clara");
131 }
132
133
```

```

98 class five {
99 public:
100 five(char *t): _size(strlen(t)), _s(new char[_size+1]) {
101     strcpy(_s,t);
102     cout << "string = " << _s << endl;
103 }
104 private:
105     int _size;
106     char* _s;
107 };

```

Order of member initialization

1 2

1 Variable declaration order in class
2

Initialization order is tied to the order of the class layout

_size is guaranteed to be initialized before _s


```

113 class six {
114 public:
115 six(char *t): _size(strlen(t)), _s(new char[_size+1]) {
116     strcpy(_s,t);
117     cout << "string = " << _s << endl;
118 }
119 private:
120     char* _s;
121     int _size;
122 };

```

If _s is initialized first, what _size it uses?

List members in an initialization list in the order in which they are declared

Figure 6.26: Order of initialization

6.18. STATIC MEMBER OF A CLASS

6.18 Static member of a class

6.19 Size of a class

6.20 Need for friend

6.20.1 friend function

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: f1.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 -----*/  
12 class fraction {  
13 public:  
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d){  
15         cout << "In fraction constructor\n" ;  
16     }  
17     ~fraction() { cout << "In fraction distructor\n" ;}  
18     int numerator() const { return _numerator ;}  
19     int denominator() const { return _denominator ;}  
20  
21 private:  
22     int _numerator ;  
23     int _denominator;  
24     fraction(const fraction& a);  
25     fraction& operator=(const fraction& rhs);  
26 };  
27  
28 /*-----  
29 89  
30 In fraction constructor  
31 3/8  
32 In fraction distructor  
33 -----*/  
34 int main() {  
35     int y = 89 ;  
36     cout << y << endl ;  
37  
38     fraction x(3,8);  
39     //cout << x << endl ; -- cout does not know fraction. It knows basic types only  
40     //no operator found which takes a right-hand operand of type 'fraction'  
41     //cout << x._numerator;// -- _numerator' : cannot access private member  
42     cout << x.numerator();  
43     cout << "/" ;  
44     cout << x.denominator() ;  
45     cout << "\n" ;  
46     return 0 ;  
47 }  
48
```

Operator << works on basic types

```

9  /*
10 class fraction:
11 -----
12 class fraction {
13 public:
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
15         cout << "In fraction constructor\n" ;
16     }
17     ~fraction() { cout << "In fraction distructor\n" ;}
18     int numerator() const { return _numerator ;}
19     int denominator() const { return _denominator ;}
20
21 private:
22     int _numerator ;
23     int _denominator;
24     fraction(const fraction& a);
25     fraction& operator=(const fraction& rhs);
26 };
27
int y = 89 ;
cout << y << endl ;           1. cout is an object
                                2. cout has an overloaded operator
                                   cout << (basic_type)
                                   cout.operator<<(basic_type)

```

fraction x(3,8) ; This won't work because we are calling
cout << x << endl ; cout << (fraction)
cout does not know fraction

```

39 //cout << x << endl ; -- cout does not know fraction. It knows basic types only
40 //no operator found which takes a right-hand operand of type 'fraction'
41 //cout << x._numerator ;// -- _numerator' : cannot access private member
42 cout << x.numerator() ;
43 cout << "/" ;
44 cout << x.denominator() ;
45 cout << "\n" ;

```

We need to take basic types from fraction and give to cout

Figure 6.27: Using cout basic types to print

Consolidating fraction print

```

12 class fraction {
13 public:
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
15         cout << "In fraction constructor\n" ;
16     }
17     ~fraction() { cout << "In fraction destructor\n" ;}
18     void print(ostream& o) const {
19         o << _numerator ;
20         o << "/" ;
21         o << _denominator ;
22         o << "\n" ;
23     }

```

**fraction x(3,8) ;
 x.print(cout) ;**

It works, but different than basic types

```

int y = 98 ;
cout << y ;

```

**fraction x(3,8) ;
 x.print(cout) ;**

Not consistent

Figure 6.28: Printing fraction object

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: f2.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 -----*/  
12 class fraction {  
13 public:  
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {  
15         cout << "In fraction constructor\n" ;  
16     }  
17     ~fraction() { cout << "In fraction distructor\n" ;}  
18     void print(ostream& o) const {  
19         o << _numerator ;  
20         o << "/" ;  
21         o << _denominator ;  
22         o << "\n" ;  
23     }  
24  
25 private:  
26     int _numerator ;  
27     int _denominator ;  
28     fraction(const fraction& a);  
29     fraction& operator=(const fraction& rhs);  
30 };  
31  
32 /*-----  
33 89  
34 In fraction constructor  
35 3/8  
36 In fraction distructor  
37 -----*/  
38 int main() {  
39     int y = 89 ;  
40     cout << y << endl ;  
41     fraction x(3,8);  
42     x.print(cout);  
43     return 0 ;  
44 }  
45
```

Need for friend function

```

class fraction {
public:
    fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
        cout << "In fraction constructor\n";
    }
    ~fraction() { cout << "In fraction destructor\n"; }
    void print1(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
    friend void print(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
    //Exactly as print. print name is replaced by operator<<
    friend void operator<<(ostream& o, fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
};

instead of name: print
use: operator<<
private:
    int _numerator;
    int _denominator;
    fraction(const fraction& a);
    fraction& operator=(const fraction& rhs);
};

```

fraction x(3,8)

print1(cout,x) //What is print1? fails
x.print1(cout,x) : //Works ugly
print(cout,x) : //Works because of friend
//print is a NON member function
// Can be called WITHOUT an obj.print
//Must have obj as parameter in print function

Now everything works like basic types

```

int y = 89 ;
cout << y << endl ;

fraction x(3,8) ;
//print1(cout,x) ; //error C3861: 'print1': identifier not found
x.print1(cout,x) ; //Works, but ugly. User need to know print1
print(cout,x) ; // Works without invoking from x
                //But one of the parameter must be x
                //Still user should know function name "print"
operator<<(cout,x) ; //infix notation
cout << x ; //postfix

```

Figure 6.29: Friend function

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: f3.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 -----*/  
12 class fraction {  
13 public:  
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {  
15         cout << "In fraction constructor\n" ;  
16     }  
17     ~fraction() { cout << "In fraction distructor\n" ;}  
18     void print1(ostream& o) {  
19         o << _numerator << "/" << _denominator << endl ;  
20     }  
21     void print1(ostream& o, const fraction& a) {  
22         o << a._numerator << "/" << a._denominator << endl ;  
23     }  
24     friend void print(ostream& o, const fraction& a) {  
25         o << a._numerator << "/" << a._denominator << endl ;  
26     }  
27     //Exactly as print. print name is replaced by operator<<  
28     friend void operator<<(ostream& o, fraction& a) {  
29         o << a._numerator << "/" << a._denominator << endl ;  
30     }  
31  
32 private:  
33     int _numerator ;  
34     int _denominator;  
35     fraction(const fraction& a);  
36     fraction& operator=(const fraction& rhs);  
37 };  
38  
39 /*-----  
40 -----*/  
41 int main() {  
42     int a = 89 ;  
43     cout << "a = " << a << endl ;  
44     int b = 11 ;  
45     cout << "b = " << b << endl ;  
46     fraction x(3,8) ;  
47     x.print1(cout) ; // Works. Needs to invoke with x  
48     x.print1(cout,x) ; //Works. Needs to invoke with x  
49     //print1(cout,x) ; //error C3861: 'print1': identifier not found  
50     print(cout,x) ; // invoke without x. One of the parameter must be x. User should know print  
51     operator<<(cout,x) ; //infix notation  
52     cout << x ; //postfix  
53     return 0 ;  
54 }  
55 }  
56 }
```

Need for friend function

```

class fraction {
public:
    fraction(int n = 0, int d = 1) : _numerator(n), _denominator(d) {
        cout << "In fraction constructor\n";
    }
    ~fraction() { cout << "In fraction destructor\n"; }
    void print1(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
    friend void print(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator << endl;
    }
    friend ostream& operator<<(ostream& o, const fraction& a) {
        o << a._numerator << "/" << a._denominator;
        return o;
    }
};

```

fraction x(3,8)

`x.print1(cout,x); //What is print1? fails`
`x.print1(cout,x); //Works ugly`
`print(cout,x); //Works because of friend`
`//print is a NON member function`
`// Can be called WITHOUT an obj.print`
`//Must have obj as parameter in print function`

Non member function

Chaining

Must have fraction as a parameter

`cout << x;`
`operator<<(cout,x);`

Note that we are invoking this function by cout(like print above)
and NOT by x. x is a parameter to cout function

Now everything works like basic types

```
int y = 98;
cout << y;
```

```
fraction x(3,8);
fraction z(88,99);
cout << "x = " << x << " z = " << z << endl;
```

Figure 6.30: Friend function with chaining of cout

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: f4.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 -----*/  
12 class fraction {  
13 public:  
14     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d){  
15         cout << "In fraction constructor\n" ;  
16     }  
17     ~fraction() { cout << "In fraction distructor\n" ;}  
18     void print1(ostream& o, const fraction& a){  
19         o << a._numerator << "/" << a._denominator << endl ;  
20     }  
21     friend void print(ostream& o, const fraction& a){  
22         o << a._numerator << "/" << a._denominator << endl ;  
23     }  
24  
25     friend ostream& operator<< (ostream& o, const fraction& a){  
26         o << a._numerator << "/" << a._denominator;  
27         return o ;  
28     }  
29  
30 private:  
31     int _numerator ;  
32     int _denominator;  
33     fraction(const fraction& a);  
34     fraction& operator=(const fraction& rhs);  
35 };  
36  
37 /*-----  
38 -----*/  
39  
40 int main(){  
41     int y = 89 ;  
42     cout << y << endl ;  
43  
44     fraction x(3,8) ;  
45     //print1(cout,x) ; //error C3861: 'print1': identifier not found  
46     x.print1(cout,x) ;  
47     print(cout,x) ;  
48     cout << x << endl ;  
49     fraction z(88,99) ;  
50     cout << "x = " << x << " z = " << z << endl ;  
51     return 0 ;  
52 }  
53
```

Friend function

A non member function of the class

A function declared as friend in a class so that it has the same access as the class' members without having to be within the scope of the class.

Declaration

```
class cost {
    friend ostream& operator<<(ostream& o, const cost& c);
private:
    int _base;
};
```

Definition

```
ostream& operator<<(ostream& o, const cost& c){
    o << " base = " << c._base << endl;
    return o;
}
```

Usage

```
cost c;
cout << c;
operator<<(cout, c);
```

Note that this function is invoked by instance of cout and not by an instance of the user-defined class, cost

That means the function << must be declared as a binary friend of the class cost

1. The keyword friend is placed only in the function declaration of the friend function and not in the function definition.
2. A friend function, even though it is not a member function, would have the rights to access the private members of the class.
3. The function can be invoked without the use of an object
4. Friends are non-members hence do not get "this" pointer

Figure 6.31: Friend function

6.20. NEED FOR FRIEND

6.20.2 friend class

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevarurthy  
3 Filename: friendclass.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 class patient ;  
10  
11 class data {  
12     int _age ;  
13     int _sugar ;  
14     int _bp ;  
15     friend class patient ; // Try commenting this line  
16 };  
17  
18 class patient {  
19 public:  
20     patient(int a, int s = 0, int b = 0);  
21     ~patient();  
22     void print(const char* p) const ;  
23  
24 private:  
25     data _d;  
26 };  
27  
28 /*-----  
29 Constructor  
30 -----*/  
31 patient::patient(int a, int s, int b){  
32     //Without friend class patient ;  
33     //error C2248: 'data::_sugar' : cannot access private member declared in class 'data'  
34     _d._age = a ;  
35     _d._sugar = s ;  
36     _d._bp = b ;  
37     cout << "In patient constructor\n";  
38 }  
39  
40 /*-----  
41 Destructor  
42 -----*/  
43 patient::~patient(){  
44     cout << "In patient Destructor \n";  
45 }  
46  
47 /*-----  
48 print  
49 -----*/  
50 void patient::print(const char *s) const {  
51     cout << s << " age = " << _d._age << " sugar = " << _d._sugar << " bp = " << _d._bp << endl ;  
52 }  
53  
54 /*-----  
55 main
```

```
56 -----*/
57 int main() {
58     patient b1(25,90,118);
59     b1.print("b1");
60     return 0;
61 }
62 /*-----*
63 output of the above program
64 -----*/
65 /*
66 In patient constructor
67 b1 age = 25 sugar = 90 bp = 118
68 In patient Destructor
69
70 */
71
```

6.21 Type conversions

Primitive type to primitive type

```
static void prim_2_prim() {
{
    int a = 20 ;
    float f = a ; //conversion from 'int' to 'float', possible loss of data
    double d = 89.98 ;
    int ad = d ; //conversion from 'double' to 'int', possible loss of data
}
{
    int a = 20 ;
    float f = float(a) ; prim_type a = prim_type(other_prim_type b)
    double d = 89.98 ;
    int ad = int(d) ;
}
```

Figure 6.32: Primitive to primitive type

6.21. TYPE CONVERSIONS

Primitive type to UDT

Happens through constructor

```
class Int {  
public:  
    Int(int x = 0): _d(x) {}  
private:  
    int _d ;  
}  
  
static void prim_2_udt() {  
    int x = 89 ;  
    Int I = Int(x) ;  
}
```

Figure 6.33: Primitive to UDT

UDT to primitive type

Happens through

1. Member function
2. Conversion function

```

class Int {
public:
    Int(int x = 0) : _d(x) {cout << "In Int constructor " << x << endl ; }
    ~Int() {cout << "In Int destructor " << _d << endl ; }
    int getx() const {cout << "in getx()" << endl ; return _d ; }
    //conversion operator
    operator int() const {cout << "in int()" << endl ; return _d ; }
    operator float() const {cout << "in float()" << endl ; return float(_d) ; }
    operator const char*() const {cout << "in const char*()" << endl ; return "HAHA" ; }
}

static void udt2prim() {
    Int I = Int(89) ;
    int j = I.getx() ;
    cout << "j = " << j << endl ;
    int k = int(I) ;
    cout << "k = " << k << endl ;
    float f = float(I) ;
    cout << "f = " << f << endl ;
    const char* s = I ;
    cout << "s = " << s << endl ;
}

```

operator int() const { }

1. Name of the function must be operator followed by the type into which conversion is required
2. Must be a member function, not a friend
3. NO explicit return type. Function name itself is type
4. NO input argument to the function
5. Must return a value of the same type
6. Preferably a const function

In Int constructor 89
in getx()
j = 89
in int()
k = 89
in float()
f = 89
in const char*()
s = HAHA
In Int destructor 89

Figure 6.34: UDT to primitive

6.21. TYPE CONVERSIONS

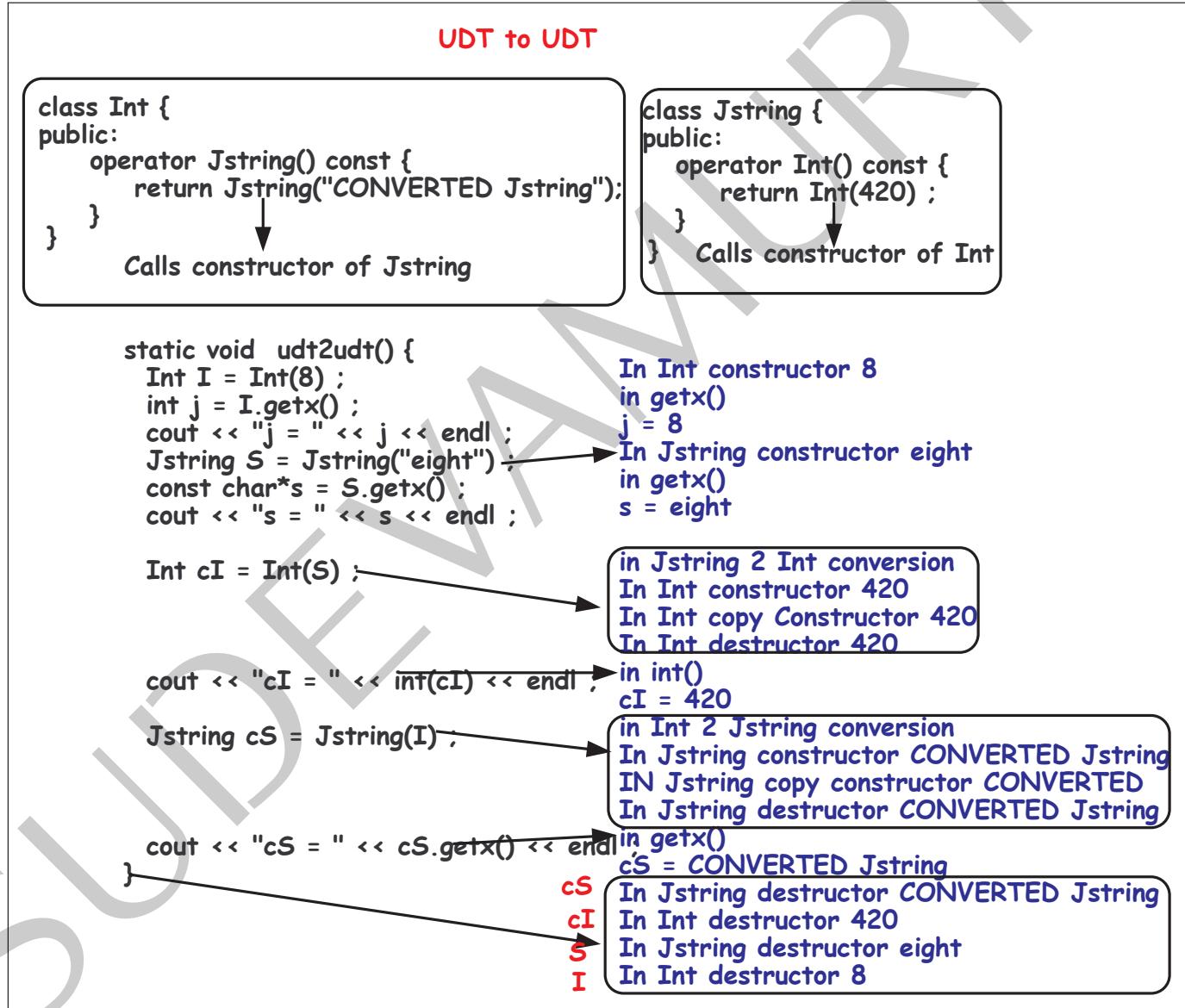


Figure 6.35: UDT to UDT

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename:type.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 forward declarations  
11 -----*/  
12 //class Int ;  
13 class Jstring;  
14  
15 /*-----  
16 Int class  
17 -----*/  
18 class Int {  
19 public:  
20     Int(int x = 0):_d(x) {cout << "In Int constructor " << x << endl ; }  
21     ~Int() { cout << "In Int destructor " << _d << endl ; }  
22     Int(const Int& r) { cout << "IN Int copy constructor " << r._d << endl ; _d = r._d ; }  
23     Int& operator=(const Int& r) { cout << "IN Int equal operator " << r._d << endl ; _d = r._d ; return *this ; }  
24     int getx() const {cout << "in getx() " << endl ;return _d ; }  
25     //conversion operator  
26     operator int() const {cout << "in int() " << endl ; return _d ; }  
27     operator float() const {cout << "in float() " << endl ; return float(_d) ; }  
28     operator const char*() const {cout << "in const char*() " << endl ; return "HAHA" ; }  
29  
30     //Conversion operator that takes Int and returns Jstring  
31     //operator Jstring() const {cout << "in Int 2 Jstring conversion " << endl ;return Jstring("CONVERTED Jstring");}  
32     //You cannot do above: error C2027: use of undefined type  
33     operator Jstring() const;  
34     private:  
35         int _d ;  
36     };  
37  
38 /*-----  
39 Jstring class  
40 -----*/  
41 class Jstring {  
42 public:  
43     Jstring(const char* x = 0):_d(x) {cout << "In Jstring constructor " << x << endl ; }  
44     ~Jstring() { cout << "In Jstring destructor " << _d << endl ; }  
45     Jstring(const Jstring& r) { cout << "IN Jstring copy constructor " << r._d << endl ; _d = r._d ; }  
46     Jstring& operator=(const Jstring& r) { cout << "IN Jstring equal operator " << r._d << endl ; _d = r._d ; return *this ; }  
47     const char* getx() const {cout << "in getx() " << endl ;return _d ; }  
48     //Conversion operator that takes Jstring and return Int  
49     operator Int() const { cout << "in Jstring 2 Int conversion " << endl ;return Int(420) ; }  
50     private:  
51         const char* _d ;  
52     };  
53  
54 /*-----  
55 Must be implemented after class definitions.  
56 To overcome: error C2027: use of undefined type  
57 -----*/  
58 Int::operator Jstring() const {cout << "in Int 2 Jstring conversion " << endl ;return Jstring("CONVERTED Jstring");}  
59  
60 /*-----  
61 Use C++ style casting:  
62 type(variable)
```

```
63     -----*/
64 static void prim_2_prim() {
65 {
66     int a = 20 ;
67     float f = a ; //conversion from 'int' to 'float', possible loss of data
68     double d = 89.98 ; //conversion from 'double' to 'int', possible loss of data
69     int ad = d ;
70 }
71 {
72     int a = 20 ;
73     float f = float(a) ;
74     double d = 89.98 ;
75     int ad = int(d) ;
76 }
77 }
78 */
79 /*-----*
80 Use constructor
81
82 In Int constructor 89
83 In Int destructor 89
84 -----*/
85 static void prim_2_udt() {
86     int x = 89 ;
87     Int I = Int(x) ;
88 }
89
90 */
91
92 -----*/
93 static void udt2prim() {
94     Int I = Int(89) ;
95     int j = I.getx() ;
96     cout << "j = " << j << endl ;
97     int k = int(I) ; //explicit
98     cout << "k = " << k << endl ;
99     k = I ; //Compiler will figure out
100    cout << "k = " << k << endl ;
101    float f = float(I) ; //explicit
102    cout << "f = " << f << endl ;
103    f = I ; //Compiler will figure out
104    const char* s = I ;
105    cout << "s = " << s << endl ;
106 }
107
108 */
109 In Int constructor 8
110 in getx()
111 j = 8
112 In Jstring constructor eight
113 in getx()
114 s = eight
115 in Jstring 2 Int conversion
116 In Int constructor 420
117 IN Int copy constructor 420
118 In Int destructor 420
119 cI = in int()
120 420
121 in Int 2 Jstring conversion
122 In Jstring constructor CONVERTED Jstring
123 IN Jstring copy constructor CONVERTED Jstring
124 In Jstring destructor CONVERTED Jstring
125 cS = in getx()
126 CONVERTED Jstring
127 In Jstring destructor CONVERTED Jstring
128 In Int destructor 420
```

```
129 In Jstring destructor eight
130 In Int destructor 8
131 -----
132 static void udt2udt() {
133     Int I = Int(8) ;
134     int j = I.getx() ;
135     cout << "j = " << j << endl ;
136     Jstring S = Jstring("eight") ;
137     const char*s = S.getx() ;
138     cout << "s = " << s << endl ;
139     Int cI = Int(S) ;
140     cout << "cI = " << int(cI) << endl ;
141     Jstring cS = Jstring(I) ;
142     cout << "cS = " << cS.getx() << endl ;
143 }
144
145 /*
146
147 */
148 int main() {
149     prim_2_prim();
150     prim_2_udt();
151     udt2prim();
152     udt2udt() ;
153     return 0 ;
154 }
```

6.22. USAGE OF CLASS STRING

6.22 Usage of class string

6.23 Object composition

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: composition.cpp  
4 -----*/  
5  
6  
7 /*-----  
8 All includes here  
9 -----*/  
10 #include <iostream>  
11 #include <fstream>  
12  
13 #include <iomanip> // std::setprecision  
14 using namespace std;  
15  
16 #ifdef WIN32  
17 #include <cassert>  
18 #include <ctime>  
19 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector  
20 #else  
21 #include <assert.h>  
22 #include <time.h>  
23 #endif  
24  
25 /*-----  
26 Basic defines here  
27 -----*/  
28 #define _NOCOPYOREQUAL_(X) X(const X& a);X& operator=(const X& a);  
29 static const int SHOW = 1;  
30  
31 /*-----  
32 mstring class  
33 -----*/  
34 class mstring{  
35 public:  
36     mstring(const char* s = 0):_s(0) {  
37         if (SHOW) {  
38             cout << "in mstring constructor ";  
39             if (s) {  
40                 cout << s ;  
41             }  
42             cout << endl ;  
43         }  
44         _allocate(s) ;  
45     }  
46     ~mstring() {  
47         if (SHOW) {  
48             cout << "in mstring distructor ";  
49             if (_s) {  
50                 cout << _s ;  
51             }  
52             cout << endl ;  
53         }  
54         _free();  
55     }  
56     mstring(const mstring& rhs) {  
57         if (SHOW) {  
58             cout << "in mstring copy constructor " << rhs << endl ;  
59         }  
60         _allocate(rhs._s) ;  
61     }  
62     mstring& operator=(const mstring& rhs) {  
63         if (SHOW) {  
64             cout << "in mstring equal constructor " << rhs << endl ;  
65         }  
66         if (this != &rhs) {
```

```
67     _free() ;
68     _allocate(rhs._s) ;
69 }
70 return *this;
71 }
72 friend ostream& operator<<(ostream& o, const mstring& rhs) {
73     if (rhs._s) {
74         o << rhs._s ;
75     }
76     return o ;
77 }
78 operator bool() const {return _s ? true: false;}
79 private:
80 //data
81 char* _s ;
82 //functions
83 void _allocate(const char* s) {_s = 0 ; if (s) {_s = new char[strlen(s)+1] ; strcpy(_s,s) ;}}
84 void _free() {delete [] _s ; _s = 0; }
85 };
86
87 /-----
88 Ssn class
89 -----
90 class ssn{
91 public:
92     ssn(const char* s, const char* n = 0 , const char* c = 0):_ssn(s),_name(n),_country(c){
93         if (SHOW) {
94             cout << "in ssn constructor " << endl ;
95         }
96     }
97     ~ssn() {
98         if (SHOW) {
99             cout << "in ssn disstructor " << endl ;
100        }
101    }
102 //COPY AND EQUAL OPERATOR NOT WRITTEN AS THIS CLASS HAS NO HEAP OBJECTS ON ITS OWN
103
104 friend ostream& operator<<(ostream& o, const ssn& r) {
105     //Why I am NOT doing this???
106     //o << r._name << " " << r._ssn << " " << r._country;
107     if (r._name) {
108         o << r._name << " " ;
109     }
110     if (r._ssn) {
111         o << r._ssn << " " ;
112     }
113     o << r._country; //I never put space for the last
114     return o ;
115 }
116 }
117 private:
118 //data
119 mstring _ssn ;
120 mstring _name ; //SSN/Pan etc
121 mstring _country ; //Country issued
122 };
123
124 /-----
125 name class
126 -----
127
128 class name{
129 public:
130     name(const char* f, const char* l = 0 , const char* m = 0, const char* t = 0):_fname(f),_lname(l),
131     _mname(m),_title(t){
132         if (SHOW) {
```

```
132     cout << "in name constructor " << endl ;
133 }
134 ~name() {
135     if (SHOW) {
136         cout << "in name disstructor " << endl ;
137     }
138 }
139 };
140
141 //COPY AND EQUAL OPERATOR NOT WRITTEN AS THIS CLASS HAS NO HEAP OBJECTS ON ITS OWN
142
143 friend ostream& operator<<(ostream& o, const name& r) {
144     //Why I am NOT doing this???
145     //o << r._title << " " << r._fname << " " << r._mname << " " << r._lname;
146     if (r._title) {
147         o << r._title << " ";
148     }
149     if (r._fname) {
150         o << r._fname << " ";
151     }
152     if (r._mname) {
153         o << r._mname << " ";
154     }
155     o << r._lname; //I never put space for the last
156     return o ;
157 }
158
159 private:
160     mstring _fname ;
161     mstring _lname ;
162     mstring _mname ;
163     mstring _title ;
164 };
165
166 /*
167 date of birth class
168 */
169 class dob{
170 public:
171     dob(const int m , const int d, const int y):_month(m),_day(d),_year(y){
172         if (SHOW) {
173             cout << "in dob constructor " << endl ;
174         }
175     }
176     ~dob(){
177         if (SHOW) {
178             cout << "in dob distructor " << endl ;
179         }
180     }
181
182 //COPY AND EQUAL OPERATOR NOT WRITTEN AS THIS CLASS HAS NO HEAP OBJECTS ON ITS OWN
183
184 friend ostream& operator<<(ostream& o, const dob& r) {
185     o << r._month << " " << r._day << " " << r._year;
186     return o ;
187 }
188
189 private:
190     //data
191     unsigned int _month ;
192     unsigned int _day ;
193     unsigned int _year ;
194 };
195
196
197 /*-----
```

```
198 person class
199 -----*/
200 class person{
201 public:
202     person(const ssn& s, const name& n, const dob& d, const mstring& p, bool l = true):_ssn(s),_name(n),
203     _dob(d),_current_place_of_residence(p),_isalive(l) {
204         if (SHOW) {
205             cout << "in person constructor " << endl ;
206         }
207     }
208     ~person() {
209         if (SHOW) {
210             cout << "in person disstructor " << endl ;
211         }
212     }
213 //COPY AND EQUAL OPERATOR NOT WRITTEN AS THIS CLASS HAS NO HEAP OBJECTS ON ITS OWN
214
215     friend ostream& operator<<(ostream& o, const person& r) {
216         o << r._name << " " << r._ssn << " " << r._dob << " " << r._current_place_of_residence ;
217         if (!r._isalive) {
218             o << " Dead " ;
219         }
220         return o ;
221     }
222 private:
223     //data
224     ssn _ssn ;
225     name _name ;
226     dob _dob ;
227     mstring _current_place_of_residence ;
228     bool _isalive ;
229 };
230
231 /*-----*/
232 graduate class
233 -----*/
234 class graduate{
235 public:
236     graduate(const person& p, const mstring& m, int y):_p(p),_degree_awarded(m),_year(y){
237         if (SHOW) {
238             cout << "in graduate constructor " << endl ;
239         }
240     }
241     ~graduate() {
242         if (SHOW) {
243             cout << "in graduate disstructor " << endl ;
244         }
245     }
246
247 //COPY AND EQUAL OPERATOR NOT WRITTEN AS THIS CLASS HAS NO HEAP OBJECTS ON ITS OWN
248
249     friend ostream& operator<<(ostream& o, const graduate& r) {
250         o << r._p << " " << r._degree_awarded << " " << r._year ;
251         return o ;
252     }
253 private:
254     person _p ;
255     mstring _degree_awarded ;
256     int _year ;
257 };
258
259 /*-----*/
260 graduate class
261 -----*/
262 class list_of_graduates{
```

```
263 public:
264     list_of_graduates(int maxsize = 50):_maxsize(maxsize),_size(0),_list(0) {
265         if (SHOW) {
266             cout << "in list_of_graduates constructor " << endl ;
267         }
268         _list = new graduate* [_maxsize] ;
269     }
270     ~list_of_graduates() {
271         if (SHOW) {
272             cout << "in list_of_graduates disstructor " << endl ;
273         }
274         for (int i = 0; i < _size; ++i) {
275             delete _list[i] ;
276         }
277         delete [] _list;
278         _list = 0;
279         _size = 0;
280     }
281     void append(const graduate& g) {
282         assert(_size < _maxsize) ;
283         graduate* t = new graduate(g) ;
284         _list[_size++] = t ;
285     }
286     friend ostream& operator<<(ostream& o, const list_of_graduates& r) {
287         for (int i = 0; i < r._size; ++i) {
288             o << *(r._list[i]) << endl ;
289         }
290         return o ;
291     }
292 private:
293     int _maxsize ;
294     int _size ;
295     graduate** _list ;
296     //DO NOT ALLOW TO COPY THIS OBJECT
297     _NOCOPYOREQUAL_(list_of_graduates) ;
298 };
299
300 /-----
301 test bench
302 -----
303 void test1() {
304     ssn s1("657897","pan","India") ;
305     cout << s1 << endl ;
306     ssn s2("65-89-90","ssn","us") ;
307     cout << s2 << endl ;
308
309     ssn s3(s1) ;
310     cout << s3 << endl ;
311     s2 = s3 ;
312     cout << s2 << endl ;
313     cout << s3 << endl ;
314
315     name n1("John") ;
316     cout << n1 << endl ;
317     name n2("John", "Smith") ;
318     cout << n2 << endl ;
319     name n3("John", "Smith", "Senior") ;
320     cout << n3 << endl ;
321     name n4("John", "Smith", "Senior", "Dr") ;
322     cout << n4 << endl ;
323 }
324
325 /-----
326 test bench
327 -----
328 void test2() {
```

```
329 list_of_graduates l(67) ;
330 {
331     ssn s("656-67-896","SSN","USA") ;
332     name n("John", "Smith",0,"DR") ;
333     dob d(8,14,1967) ;
334     person p(s,n,d,"Singapore",false) ;
335     graduate g(p,"Civil Engineering",2002) ;
336     l.append(g) ;
337 }
338 {
339     ssn s("JP9870","PAN","India") ;
340     name n("Ram", "Rao","Subba") ;
341     dob d(2,14,1977) ;
342     person p(s,n,d,"USA") ;
343     graduate g(p,"Doctor in Medicine",2020) ;
344     l.append(g) ;
345 }
346 cout << l << endl ;
347 }
348
349 /*-----
350 Main
351 -----*/
352 int main() {
353     test1() ;
354     test2() ;
355     return 0 ;
356 }
357
358 //EOF
359
360
```

6.24 Returning member variables of a class

Returning member variable of the class

Question:

What is the significance of const return type from a function? Similar to:

```
const int func1();
```

Answer:

The const in this position guarantees that the caller of the function doesn't change the returned value.

Note that the use of const in a function's return value is more useful when the function returns a reference or a pointer to, say an object's data member.

It's less useful when return-by-value is used (as in example above) because the caller gets a local copy of the variable.

Therefore, there's not much point in limiting the use of the caller's own copy

A function that returns its results by value should generally avoid const in the return type [FAQ 14.11]

Figure 6.36: Return by value should NOT use const

```
1 /*-----
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename:t.cpp
4 -----
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 -----
11 -----
12 class salary {
13 public:
14     salary(int s):_salary(s) {
15         cout << "In salary constructor\n";
16     }
17     ~salary() {
18         cout << "In salary destructor\n";
19     }
20     salary(const salary& s) {
21         cout << "In salary copy constructor\n";
22         _salary = s._salary;
23     }
24     salary& operator=(const salary& s) {
25         cout << "In salary equal operator\n";
26         _salary = s._salary;
27         return *this;
28     }
29     friend ostream& operator<<(ostream& o, const salary& s) {
30         o << " Salary = " << s._salary << " ";
31         return o;
32     }
33     void change_salary(int x) {
34         _salary = x;
35     }
36 private:
37     int _salary;
38 };
39 */
40 -----
41 -----
42 class employee {
43 public:
44     employee(int s, int id):_id(id),_s(s) {
45         cout << "In employee constructor\n";
46     }
47     ~employee() {
48         cout << "In employee destructor\n";
49     }
50     employee(const employee& e):_s(e._s),_id(e._id){
51         cout << "In employee copy constructor\n";
52     }
53     employee& operator=(const employee& e) {
54         _s = e._s;
```

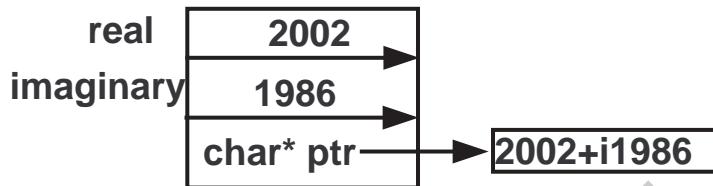
```
56     _id = e._id ;
57     cout << "In employee equal operator\n" ;
58     return *this ;
59 }
60 friend ostream& operator<<(ostream& o, const employee& e) {
61     o << "employee id = " << e._id << e._s ;
62     return o ;
63 }
64 salary get_salary() const{
65     return _s ;
66 }
67 void change_salary(int x){
68     _s.change_salary(x) ;
69 }
70
71 private:
72     salary _s ;
73     int _id ;
74 };
75
76 /*-----
77 In salary constructor
78 In employee constructor
79 employee id = 7 Salary = 1000
80 In salary copy constructor
81 In salary destructor
82 employee id = 7 Salary = 1000
83 In employee destructor
84 In salary destructor
85 -----*/
86 int main(){
87     employee jim(1000,7) ;
88     cout << jim << endl ;
89     (jim.get_salary()).change_salary(2000) ;
90     //Returned object is temp. Changing to 2000 will not change jim salary
91     cout << jim << endl ;
92     return 0 ;
93 }
94
```

6.25. PROBLEM SET

6.25 Problem set

Problem 6.25.1. Implement a `complex1` class, as shown in the figure 6.37.

Implementing complex1 without C++ constructors and destructors



1. create a class called `complex1` that has
 1. Real number, $x \geq 0$
 2. Imaginary number, $y \geq 0$
 3. construct a name $x+iy$ and store the pointer.
2. You must create 3 files: `complex1.h`, `complex1.cpp` `complex1test.cpp`
3. You must use only `util.h` in `complex1.h`
4. You should be able to construct `complex1` objects using 3 ways


```
complex1 h ;
h.init() ; //In that case x = 0, y = 0
h.init(5) ; //In that case x = 5, y = 0 ;
h.init(5,8) ; //In that case x = 5 and y = 8
```
5. Provide `print`, `init` and `fini` function as shown in `c5.h`
6. Test `complex1` as shown in `c5_main.cpp`. You should allocate objects on stack and heap as shown in `c5_main.cpp`
7. print `complex1test.cpp`. Draw by hand, adjacent to each procedure, the allocation of objects as shown in figure 6.8, 6.9 and 6.10
8. e-mail all the 3 files (scan `complex1test.cpp` after hand drawing) to TA

Figure 6.37: Complex class(Phase 1)

6.25. PROBLEM SET

Problem 6.25.2. Implement a **integer matrix class**, as shown in the figure 6.38.
You are given the test program in **intmatrix1test.cpp**.

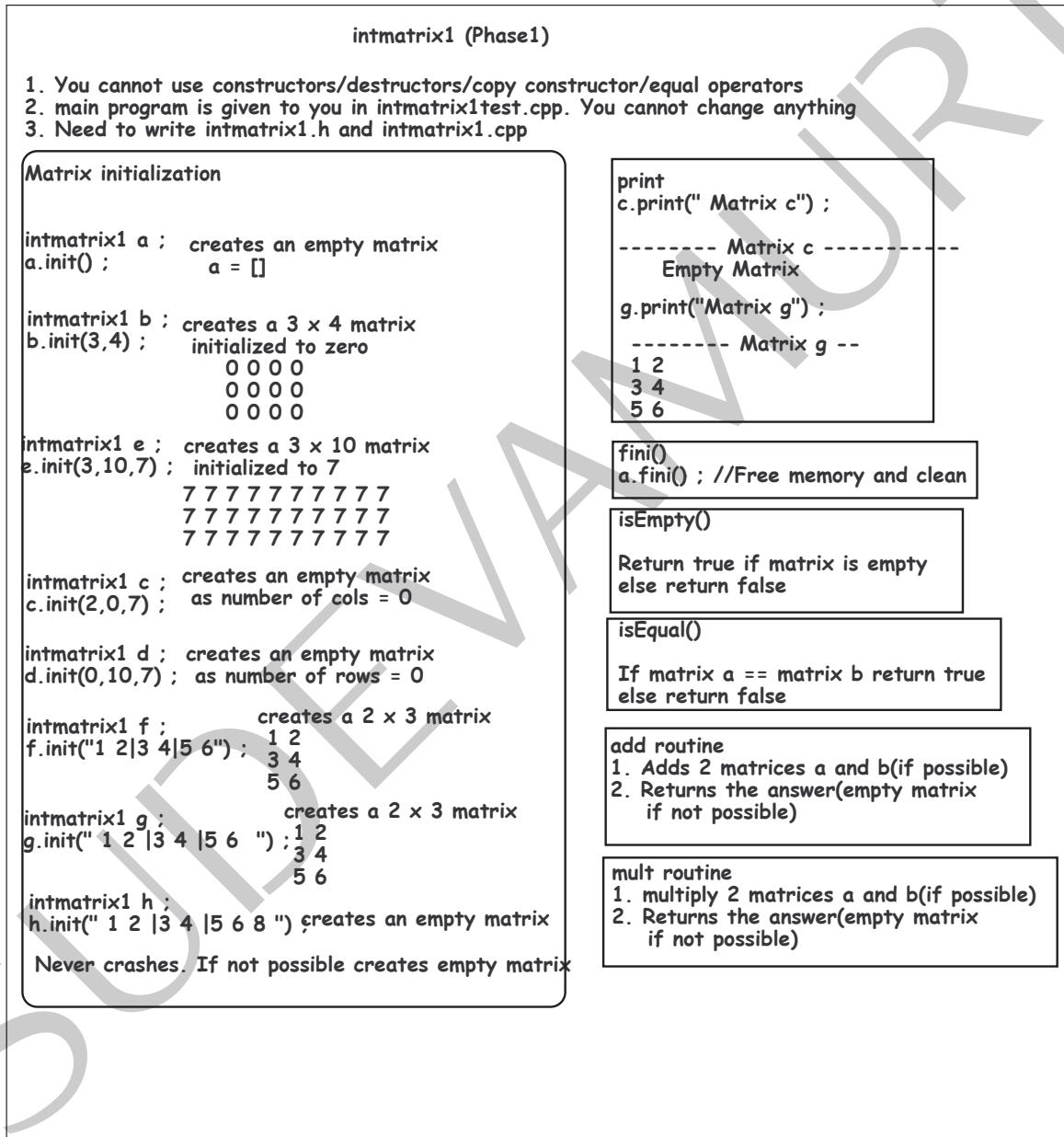


Figure 6.38: Integer matrix class(Phase 1)

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: intmatrix1test.cpp  
4  
5 On linux:  
6 g++ intmatrix1.cpp intmatrix1test.cpp  
7 valgrind a.out  
8  
9 -----*/  
10  
11 /*-----  
12 This file test intmatrix1 object  
13 -----*/  
14  
15 /*-----  
16 All includes here  
17 -----*/  
18 #include "intmatrix1.h"  
19  
20 /*-----  
21 test init and fini  
22 -----*/  
23 void test_init_fini() {  
24     intmatrix1 a ;  
25     a.init() ;  
26     a.print("Matrix a") ;  
27  
28     intmatrix1 b ;  
29     b.init(3,4) ;  
30     b.print("Matrix b") ;  
31  
32     intmatrix1 c ;  
33     c.init(2,0,7) ;  
34     c.print("Matrix c") ;  
35  
36     intmatrix1 d ;  
37     d.init(0,10,7) ;  
38     d.print("Matrix d") ;  
39  
40     intmatrix1 e ;  
41     e.init(3,10,7) ;  
42     e.print("Matrix e") ;  
43  
44  
45     intmatrix1 f ;  
46     f.init("1 2|3 4|5 6") ;  
47     f.print("Matrix e") ;  
48  
49     intmatrix1 g ;  
50     g.init(" 1 2 |3 4 |5 6  ") ;  
51     g.print("Matrix g") ;  
52     assert(f isEqual(g)) ;  
53     assert(g isEqual(f)) ;  
54  
55     intmatrix1 h ;  
56     h.init(" 1 2 |3 4 |5 6 8 ") ;  
57     h.print("Matrix h") ;  
58     assert(h.isEmpty()) ;  
59     assert(!f isEqual(h)) ;  
60  
61     a.fini() ;  
62     b.fini() ;  
63     c.fini() ;  
64     d.fini() ;  
65     e.fini() ;  
66     f.fini() ;
```

```
67     g.fini() ;
68 }
69
70 /*-----
71 test add1
72 -----*/
73 void test_add1(const char* as, const char* bs, const char* anss) {
74     intmatrix1 a ;
75     a.init(as) ;
76     a.print("Matrix a") ;
77     intmatrix1 b ;
78     b.init(bs) ;
79     b.print("Matrix b") ;
80     intmatrix1 s = a.add(b) ;
81     s.print("matrix s") ;
82     intmatrix1 ans;
83     ans.init(anss) ;
84     ans.print("matrix expected ans") ;
85     assert(s.isEqual(ans)) ;
86     assert(ans.isEqual(s)) ;
87     a.fini() ;
88     b.fini() ;
89     s.fini() ;
90     ans.fini() ;
91 }
92
93 /*-----
94 test add
95 -----*/
96 void test_add() {
97     test_add1("7 9 11|13 15 17 "," 6 8 10| 12 14 16 ","13 17 21 | 25 29 33") ;
98     test_add1("1 2 3|4 5 6 ","1 2 ","") ;
99 }
100
101 /*-----
102 test mult1
103 -----*/
104 void test_mult1(const char* as, const char* bs, const char* anss) {
105     intmatrix1 a ;
106     a.init(as) ;
107     a.print("Matrix a") ;
108     intmatrix1 b ;
109     b.init(bs) ;
110     b.print("Matrix b") ;
111     intmatrix1 s = a.mult(b) ;
112     s.print("matrix s") ;
113     intmatrix1 ans;
114     ans.init(anss) ;
115     ans.print("matrix expected ans") ;
116     assert(s.isEqual(ans)) ;
117     assert(ans.isEqual(s)) ;
118     a.fini() ;
119     b.fini() ;
120     s.fini() ;
121     ans.fini() ;
122     cout <<"-----\n" ;
123 }
124
125 /*-----
126 test mult
127 -----*/
128 void test_mult() {
129     test_mult1("1 2 3 "," 2 1 3 | 3 3 2| 4 1 2 ","20 10 13") ;
130     test_mult1("3 4 2 ","13 9 7 15|8 7 4 6| 6 4 0 3 ","83 63 37 75") ;
131     test_mult1("3 ","5 2 11|9 4 14","15 6 33|27 12 |42") ;
132     const char* a = "3 9 0 2 2 9 5 2|0 2 2 1 9 6 6 8|7 5 6 1 4 9 8 9|3 3 2 9 2 1 7 4|1 9 0 1 2 9 5 2|4 2
```

```
0 3 7 3 9 1|5 9 0 6 6 7 8 2|9 3 4 6 8 4 9 1" ;
133 const char* b = "6 1 6 0 8 3 0 0|6 8 9 0 6 6 7 2|4 8 2 0 5 4 6 7|2 4 4 2 2 6 9 8|4 8 2 2 4 6 4 1|1 5 ↵
5 6 4 7 5 5|7 4 6 5 0 6 5 3|2 3 7 0 1 3 8 5";
134 const char* s = "132 170 200 87 128 186 175 106|122 186 166 86 92 182 195 123|197 235 267 104 179 ↵
243 253 178|128 140 164 63 86 162 194 140|118 164 184 85 110 174 166 98|138 142 144 83 91 162 ↵
137 82|187 222 244 106 160 244 232 141|201 212 210 97 171 230 204 142";
135 test_mult1(a,b,s) ;
136 test_mult1("7 3|2 5 | 6 8| 9 0","8 14 0 3 1|7 11 5 91 3|8 4 19 5 57","","") ;
137 }
138
139 /*-----
140 test bed
141 -----*/
142 void testbed() {
143     test_init_fini();
144     test_add() ;
145     test_mult() ;
146 }
147
148 /*-----
149 main
150 -----*/
151 int main() {
152     testbed() ;
153     return 0 ;
154 }
155
156 //EOF
157
158
159
```

CHAPTER 6. C++ CLASS

Problem 6.25.3. Implement a **telephone_number** class, as shown in the figure 6.39.

6.25. PROBLEM SET

<http://www.csgnetwork.com/phonenumcvtrev.html>

INPUT : 1-800-4-XILINX
OUTPUT:1-800-4-945469

class telephone_number

1. The class should store
 1. The original phone number given by user and cannot be changed
 2. The converted phone numbers in terms of numbers. Convert only characters from {a ..z/A..Z}. All others should be left as is.
- That means numbers of characters before and conversions should be same
3. You must use `char*`, not the STL string
4. You must be able to copy and equal objects.
5. The print routine should print both the original and converted string

You should write: `telephone_number.h`,
`telephone_number.cpp`
`telephone_number_test.cpp`

Test the program for various numbers. Also copy objects

Figure 6.39: telephone_number class

CHAPTER 6. C++ CLASS

Problem 6.25.4. Implement a **date** class using Uspensky and Heaslet Algorithm, as shown in the figure 6.40.

6.25. PROBLEM SET

<http://mathforum.org/dr.math/faq/faq.calendar.html>

Uspensky and Heaslet Algorithm

The computations used to derive the formula require the assumption that the year is greater than or equal to 1600. But we also need to be sure that the date involved refers to the Gregorian calendar which was invented in 1582 but not adopted by some countries until many years later. The formula might not work for dates earlier than 1753.

$$t = (\text{date} + \text{floor}(2.6*m - 0.2) + y + \text{floor}(y/4) + \text{floor}(c/4) - 2*c) ; \\ \text{day} = (t \% 7)$$

c = Century

y = Year

Note that $c*100+y$ is the whole year

date = Date

m = month (starts from march)

In this algorithm, month starts from March
march=1, april=2, may=3,...dec=10, jan=11, feb=12

Because of this rule, January and February are always counted as the 11th and 12th months of the previous year.

Aug 4 1914

$$\text{date} = 4 \\ m = 8-2 = 6$$

$$\begin{array}{ccc} 1914 & & \\ \swarrow & \searrow & \\ c=19 & & y=14 \end{array}$$

$$t = 2 \\ \text{day} = (2 \% 7) = 2 = \text{TUE}$$

Jan 29 2064

$$\text{date} = 29 \\ m = 11$$

$$\begin{array}{ccc} 2064 & & \\ \swarrow & \searrow & \\ c=20 & y=64 & \\ & \downarrow & \\ & \text{Jan is previous year} & \\ t = 100 & y = 63 & \\ \text{day} = (100 \% 7) = 2 = \text{TUE} & & \end{array}$$

Mar 12 2011

$$\begin{array}{l} \text{date} = 12 \\ m = 1 \\ C = 20 \\ y = 11 \end{array} 20*100+11=2011$$

$$t = -8; \\ \text{day} = (-8 \% 7) = 6 = \text{SAT}$$

day will be between 0 .. 6, which is encoded as follows:

0 = sunday 1 = monday 2 = tuesday 3 = wednesday
4 = thursday 5 = friday 6 = saturday

Once we have found t , we divide it by 7 and take the remainder.

Note that if the result for t is negative, care must be taken in calculating the proper remainder.

Suppose $t = -17$. When we divide by 7, we have to follow the same rules as for the greatest integer function; namely we find the greatest multiple of 7 less than -17, so the remainder will be positive (or zero). -21 is the greatest multiple of 7 less than -17, so the remainder is 4 since $-21 + 4 = -17$.

Alternatively, we can say that -7 goes into -17 twice, making -14 and leaving a remainder of -3, then add 7 since the remainder is negative, so $-3 + 7$ is again a remainder of 4.

$\text{floor}(x) = \text{greatest integer}$
 $\text{floor}(5.9) \leq \text{less than or equal to } x$
 $\text{floor}(5.2) \text{ is } 5$
 $\text{floor}(47) \text{ is } 47$
 $\text{floor}(-7.3) \text{ is } -8$
 $\text{floor}(-2) \text{ is } -2$

We need to find day such that $\text{day} \geq 0 \ \& \ \text{day} \leq 6$

$$-(7 * y) + \text{day} = -t$$

$$\begin{array}{ll} t=-17 & -(7 * 3) + 4 = -17 \quad y \text{ is multiple of 7 and day}=4 \\ t=-8 & -(7 * 2) + 6 = -8 \quad y \text{ is multiple of 2 and day}=6 \end{array}$$

Figure 6.40: Computing day from date

CHAPTER 6. C++ CLASS

Problem 6.25.5. Design a program that simulates the movement of an elevator as show in the figure 6.41 and 6.42

6.25. PROBLEM SET

Simulation of the movement of an elevator

1. User should be able to specify the number of floors at the time of creation.
Otherwise, assume that the building has 6 floors.
2. The elevators starts at floor 1.
3. Assuming the number of floors = 6, for the first time,
flash the message as follows:

Constructing an elevator that has 6 floors

You are In floor 1
Enter a floor number between 1 to 6
Enter 0 to close door or enter 7 to quit

Enter:



4. At this point, user will enter the floor number.
Your program must not crash for wrong inputs.
Keep on getting data, until the user has presses 0 or 7 (number of floor +1)
A typical data entry is as follows:

Enter:jdhfj
Enter:90-
Enter:-8
Enter:8
Enter:5
Will take to 5
Enter:3
Will
take to 3
Enter:5
Will take to 5

5. When user enters 0, the door of the elevator will get closed.
It will go up or down depending on the floors pressed. It will remain in
the same floor, if no movement is required.

Continuing, the above example, the output is as follows:

You are in floor 1. The door is closing....
Going up.....
Now you are in floor 3. The door is opening....

You are In floor 3
Will take to 5
Enter a floor number between 1 to 6
Enter 0 to close door or enter 7 to quit

Enter:
Keep on getting data until 0 or 7(Number of floors +1)

6. If user press 7(number of floors + 1), get out of the program and destroy all the resources.
Output the following:

Enter:7

FIRE FIRE! All get out of the elevator. Elevator will be destroyed
Elevator Destroyed

Figure 6.41: Simulating an elevator

Implementing elevator movements

Must use the following files:
 1. elevator.h
 2. elevator.cpp
 3. Must use the elevatortest.cpp given

You will write 3 class definitions in elevator.h

```
class scanner{
  //NO DATA
public:
  void prompt_and_get(const char *title
    , unsigned int& i);
};
```

```
class button{
public:
  //WRITE YOUR CODE HERE
private:
  bool _b ;
};
```

```
class elevator{
public:
  elevator(int n = 6) :
    ~elevator() :
    //NO OTHER FUNCTIONS HERE
private:
  //WRITE ALL THE PRIVATE MEMBERS U NEED
  //TRY TO make scanner object static

  //WRITE ALL THE PRIVATE FUNCTIONS U NEED

  //YOU CANNOT REMOVE BELOW LINE
  _NOCOPYOREQUAL_(elevator) ;
};
```

```
elevatortest.cpp must have only these lines
#include "elevator.h"
void testbed() {
  elevator a;
  elevator b(100) ;
}
int main() {
  test_bed();
  return 0 ;
}
```

email: elevator.h and elevator.cpp.
 capture the output of the program as a comment in elevator.cpp

Figure 6.42: Implementing an elevator

6.25. PROBLEM SET

Problem 6.25.6. Play game at http://www.nbc.com/Deal_or_No_Deal/game/flash.shtml and implement the game



Figure 6.43: Deal or no deal game at http://www.nbc.com/Deal_or_No_Deal/game/flash.shtml :

Chapter 7

Operator overloading

7.1 Introduction

**7.2 Overloading an operator as a member function or
as a friend**

7.2. OVERLOADING AN OPERATOR AS A MEMBER FUNCTION OR AS A FRIEND

function as a member function (WITHOUT Friend)

```
14 class fraction {
15 public:
16     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
17         cout << "In fraction constructor\n" ;
18     }
19     ~fraction() { cout << "In fraction destructor\n" ;}
20     fraction mult(const fraction& b) {
21         return fraction((this->_numerator * b._numerator),(this->_denominator * b._denominator)) ;
22     }
23 }
```

fraction x(3,8);
fraction y(1,2);

x.mult(y); //OK
y.mult(x); //OK
x.mult(5); //OK
5.mult(x); //NOT OK
fraction(5).mult(x) ; //OK

Where is this?

Figure 7.1: Function as a member function

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: friend1func.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 mult is NOT a friend  
12 you can never do: 5 * fraction  
13 -----*/  
14 class fraction {  
15 public:  
16     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {  
17         cout << "In fraction constructor\n" ;  
18     }  
19     ~fraction() {  
20         cout << "In fraction distructor\n" ;  
21     }  
22     fraction mult(const fraction& b) {  
23         // Construct an object  
24         // Call a copy constructor  
25         // Destroy the object  
26         // All the above are just done by just constructing the object to the caller  
27         return fraction((this->_numerator * b._numerator),(this->_denominator * b._denominator)) ;  
28     }  
29     friend ostream& operator<< (ostream& o, const fraction& a) {  
30         o <<a. _numerator << "/" << a._denominator;  
31         return o ;  
32     }  
33 private:  
34     int _numerator ;  
35     int _denominator;  
36     fraction(const fraction& a);  
37     fraction& operator=(const fraction& rhs);  
38 };  
39  
40 /*-----  
41 In fraction constructor  
42 x = 3/8  
43 In fraction constructor  
44 y = 1/2  
45 -----1-----  
46 In fraction constructor  
47 x * y = 3/16  
48 In fraction distructor  
49 -----2-----  
50 In fraction constructor  
51 In fraction constructor  
52 x * 5 = 15/8  
53 In fraction distructor  
54 In fraction distructor  
55 -----3-----  
56 -----4-----  
57 In fraction constructor  
58 In fraction constructor  
59 5 * x = 15/8  
60 In fraction distructor  
61 In fraction distructor  
62 -----5-----  
63 In fraction distructor  
64 In fraction distructor  
65  
66 -----*/
```

```
67 int main() {
68     fraction x(3,8) ;
69     cout << "x = " << x << endl ;
70     fraction y(1,2) ;
71     cout << "y = " << y << endl ;
72     cout << "-----1-----" << endl ;
73     {
74         cout << " x * y = " << x.mult(y) << endl ;
75     }
76     cout << "-----2-----" << endl ;
77     {
78         cout << " x * 5 = " << x.mult(5) << endl ;
79     }
80     cout << "-----3-----" << endl ;
81     {
82         //cout << " 5 * x = " << (5.mult(x)) << endl ;
83     }
84     cout << "-----4-----" << endl ;
85     {
86         cout << " 5 * x = " << fraction(5).mult(x) << endl ;
87     }
88 }
89 cout << "-----5-----" << endl ;
90 return 0 ;
91 }
92
93 //EOF
94
```

function as a NON member function (Friend)

```

14 class fraction {
15 public:
16     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
17         cout << "In fraction constructor\n" ;
18     }
19     ~fraction() { cout << "In fraction distructor\n" ;}
20     friend fraction mult(const fraction& a, const fraction& b) {
21         return fraction((a._numerator * b._numerator),(a._denominator * b._denominator)) ;
22     }

```

`fraction x(3,8);
fraction y(1,2);`

There is NO
this.

mult is a global
function

`mult(x,y); //OK
mult(y,x); //OK
mult(x,5); //OK
mult(5,x); //OK`

Figure 7.2: Function as a non member friend function

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: friend2func.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 mult is friend  
12 you can do: 5 * fraction  
13 -----*/  
14 class fraction {  
15 public:  
16     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {  
17         cout << "In fraction constructor\n" ;  
18     }  
19     ~fraction() { cout << "In fraction distructor\n" ;}  
20     friend fraction mult(const fraction& a, const fraction& b) {  
21         // Construct an object  
22         // Call a copy constructor  
23         // Destroy the object  
24         // All the above are just done by just constructing the object to the caller  
25         return fraction((a._numerator * b._numerator),(a._denominator * b._denominator)) ;  
26     }  
27     friend ostream& operator<< (ostream& o, const fraction& a) {  
28         o << a._numerator << "/" << a._denominator;  
29         return o ;  
30     }  
31 private:  
32     int _numerator ;  
33     int _denominator;  
34     fraction(const fraction& a);  
35     fraction& operator=(const fraction& rhs);  
36 };  
37  
38 /*-----  
39 In fraction constructor  
40 x = 3/8  
41 In fraction constructor  
42 y = 1/2  
43 -----1-----  
44 In fraction constructor  
45 x * y = 3/16  
46 In fraction distructor  
47 -----2-----  
48 In fraction constructor  
49 In fraction constructor  
50 x * 5 = 15/8  
51 In fraction distructor  
52 In fraction distructor  
53 -----3-----  
54 In fraction constructor  
55 In fraction constructor  
56 5 * x = 15/8  
57 In fraction distructor  
58 In fraction distructor  
59 -----4-----  
60 In fraction distructor  
61 In fraction distructor  
62 -----*/  
63 int main() {  
64     fraction x(3,8) ;  
65     cout << "x = " << x << endl ;  
66     fraction y(1,2) ;
```

```
67 cout << "y = " << y << endl ;
68 cout << "-----1-----" << endl ;
69 {
70     cout << " x * y = " << mult(x,y) << endl ;
71 }
72 cout << "-----2-----" << endl ;
73 {
74     cout << " x * 5 = " << mult(x,5) << endl ;
75 }
76 cout << "-----3-----" << endl ;
77 {
78     cout << " 5 * x = " << mult(5,x) << endl ;
79 }
80 cout << "-----4-----" << endl ;
81 return 0 ;
82 }
83 //EOF
84
85
```

7.2. OVERLOADING AN OPERATOR AS A MEMBER FUNCTION OR AS A FRIEND

**Overloading * as a 1. member function
2. friend**

```
class fraction {
public:
    fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {
        cout << "In fraction constructor\n" ;
    }
    ~fraction() { cout << "In fraction distructor\n" ;}
    friend ostream& operator<< (ostream& o, const fraction& a) {
        cout << a._numerator << "/" << a._denominator;
        return o ;
    }

private:
    int _numerator ;
    int _denominator;
    fraction(const fraction& a);
    fraction& operator=(const fraction& rhs);
};
```

```
class fraction {
public
    fraction operator* (const fraction& b) {  
        return fraction((this->_numerator * b._numerator),(this->_denominator * b._denominator));  
    };
```

x * y
y * x
x * 5
5 * x
5 * x → **Cannot do**

NOTE that object has to be returned by value. It cannot be reference or a pointer

```
class fraction {
public
    friend fraction operator* (const fraction& a, const fraction& b) {  
        return fraction((a._numerator * b._numerator),(a._denominator * b._denominator));  
    };
```

x * y
y * x
x * 5
5 * x
5 * x → **Cannot do**

NOTE that object has to be returned by value. It cannot be reference or a pointer

Figure 7.3: Overloading operator *

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: friend1.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 operator* is NOT a friend  
12 you can never do: 5 * fraction  
13 -----*/  
14 class fraction {  
15 public:  
16     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {  
17         cout << "In fraction constructor\n";  
18     }  
19     ~fraction() { cout << "In fraction distructor\n";}  
20     fraction operator* (const fraction& b) {  
21         return fraction((this->_numerator * b._numerator),(this->_denominator * b._denominator));  
22     }  
23     friend ostream& operator<< (ostream& o, const fraction& a) {  
24         o << a._numerator << "/" << a._denominator;  
25         return o;  
26     }  
27  
28 private:  
29     int _numerator;  
30     int _denominator;  
31     fraction(const fraction& a);  
32     fraction& operator=(const fraction& rhs);  
33 };  
34  
35 /*-----  
36 In fraction constructor  
37 x = 3/8  
38 In fraction constructor  
39 y = 1/2  
40 -----1-----  
41 In fraction constructor  
42 x * y = 3/16  
43 In fraction distructor  
44 -----2-----  
45 In fraction constructor  
46 In fraction constructor  
47 x * 5 = 15/8  
48 In fraction distructor  
49 In fraction distructor  
50 -----3-----  
51 -----4-----  
52 In fraction constructor  
53 In fraction constructor  
54 In fraction constructor  
55 In fraction constructor
```

```
56     -----5-----
57 In fraction constructor
58 In fraction constructor
59 In fraction constructor
60 a * b * c * d = 8/8
61 In fraction distructor
62 In fraction distructor
63 In fraction distructor
64     -----6-----
65 In fraction distructor
66 In fraction distructor
67 In fraction distructor
68 In fraction distructor
69     -----7-----
70 -----8-----
71 In fraction distructor
72 In fraction distructor
73 -----*/
74 int main(){
75     fraction x(3,8);
76     cout << "x = " << x << endl;
77     fraction y(1,2);
78     cout << "y = " << y << endl;
79     cout << "-----1-----" << endl;
80 {
81     cout << " x * y = " << x * y << endl;
82 }
83 cout << "-----2-----" << endl;
84 {
85     cout << " x * 5 = " << x * 5 << endl;
86 }
87 cout << "-----3-----" << endl;
88 {
89 //cout << " 5 * x = " << 5 * x << endl;
90 // error C2677: binary '*' :
91 //no global operator found which takes type 'fraction' (or there is no acceptable conversion)
92 //Note that in this case, 5 is "this". "this" object is never promoted
93 }
94 cout << "-----4-----" << endl;
95 {
96     fraction a(1,2);
97     fraction b(1,4);
98     fraction c(2,1);
99     fraction d(4,1);
100    cout << "-----5-----" << endl;
101    cout << " a * b * c * d = " << a * b * c * d << endl;
102    cout << "-----6-----" << endl;
103 }
104 cout << "-----7-----" << endl;
105 cout << "-----8-----" << endl;
106 return 0;
107 }
108 }
```

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: friend2.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 class fraction:  
11 operator* is a friend  
12 you can do: 5 * fraction  
13 -----*/  
14 class fraction {  
15 public:  
16     fraction(int n = 0, int d = 1):_numerator(n),_denominator(d) {  
17         cout << "In fraction constructor\n";  
18     }  
19     ~fraction() { cout << "In fraction distructor\n";}  
20     friend fraction operator* (const fraction& a, const fraction& b) {  
21         return fraction((a._numerator * b._numerator),(a._denominator * b._denominator));  
22     }  
23     friend ostream& operator<< (ostream& o, const fraction& a) {  
24         o << a._numerator << "/" << a._denominator;  
25         return o ;  
26     }  
27  
28 private:  
29     int _numerator ;  
30     int _denominator;  
31     fraction(const fraction& a);  
32     fraction& operator=(const fraction& rhs);  
33 };  
34  
35 /*-----  
36 In fraction constructor  
37 x = 3/8  
38 In fraction constructor  
39 y = 1/2  
40 -----1-----  
41 In fraction constructor  
42 x * y = 3/16  
43 In fraction distructor  
44 -----2-----  
45 In fraction constructor  
46 In fraction constructor  
47 x * 5 = 15/8  
48 In fraction distructor  
49 In fraction distructor  
50 -----3-----  
51 In fraction constructor  
52 In fraction constructor  
53 5 * x = 15/8  
54 In fraction distructor  
55 In fraction distructor
```

```
56 -----4-----  
57 In fraction constructor  
58 In fraction constructor  
59 In fraction constructor  
60 In fraction constructor  
61 -----5-----  
62 In fraction constructor  
63 In fraction constructor  
64 In fraction constructor  
65 a * b * c * d = 8/8  
66 In fraction distructor  
67 In fraction distructor  
68 In fraction distructor  
69 -----6-----  
70 In fraction distructor  
71 In fraction distructor  
72 In fraction distructor  
73 In fraction distructor  
74 -----7-----  
75 -----8-----  
76 In fraction distructor  
77 In fraction distructor  
78 -----*/  
79 int main(){  
80     fraction x(3,8);  
81     cout << "x = " << x << endl;  
82     fraction y(1,2);  
83     cout << "y = " << y << endl;  
84     cout << "-----1-----" << endl;  
85     {  
86         cout << " x * y = " << x * y << endl;  
87     }  
88     cout << "-----2-----" << endl;  
89     {  
90         cout << " x * 5 = " << x * 5 << endl;  
91     }  
92     cout << "-----3-----" << endl;  
93     {  
94         cout << " 5 * x = " << 5 * x << endl;  
95     }  
96     cout << "-----4-----" << endl;  
97     {  
98         fraction a(1,2);  
99         fraction b(1,4);  
100        fraction c(2,1);  
101        fraction d(4,1);  
102        cout << "-----5-----" << endl;  
103        cout << " a * b * c * d = " << a * b * c * d << endl;  
104        cout << "-----6-----" << endl;  
105    }  
106    cout << "-----7-----" << endl;  
107    cout << "-----8-----" << endl;  
108    return 0;  
109 }  
110 }
```

7.3 Operators supported by C++ on basic types

7.3. OPERATORS SUPPORTED BY C++ ON BASIC TYPES

Arithmetic operators			Comparison operators/Relational operators		
Operator name	Syntax	Overloadable	Operator name	Syntax	Overloadable
Basic assignment	<code>a = b</code>	Yes	Equal to	<code>a == b</code>	Yes
Addition	<code>a + b</code>	Yes	Not equal to	<code>a != b</code>	Yes
Subtraction	<code>a - b</code>	Yes	Greater than	<code>a > b</code>	Yes
Unary plus (integer promotion)	<code>+a</code>	Yes	Less than	<code>a < b</code>	Yes
Unary minus (additive inverse)	<code>-a</code>	Yes	Greater than or equal to	<code>a >= b</code>	Yes
Multiplication	<code>a * b</code>	Yes	Less than or equal to	<code>a <= b</code>	Yes
Division	<code>a / b</code>	Yes			
Modulo (remainder)	<code>a % b</code>	Yes			
Increment	Prefix	<code>++a</code>			
	Suffix	<code>a++</code>			
Decrement	Prefix	<code>--a</code>			
	Suffix	<code>a--</code>			

Logical operators		
Operator name	Syntax	Overloadable
Logical negation (NOT)	<code>!a</code>	Yes
Logical AND	<code>a && b</code>	Yes
Logical OR	<code>a b</code>	Yes

Bitwise operators		
Operator name	Syntax	Overloadable
Bitwise NOT	<code>~a</code>	Yes
Bitwise AND	<code>a & b</code>	Yes
Bitwise OR	<code>a b</code>	Yes
Bitwise XOR	<code>a ^ b</code>	Yes
Bitwise left shift ^[note 1]	<code>a << b</code>	Yes
Bitwise right shift ^[note 1]	<code>a >> b</code>	Yes

Compound-assignment operators		
Operator name	Syntax	Overloadable
Addition assignment	<code>a += b</code>	Yes
Subtraction assignment	<code>a -= b</code>	Yes
Multiplication assignment	<code>a *= b</code>	Yes
Division assignment	<code>a /= b</code>	Yes
Modulo assignment	<code>a %= b</code>	Yes
Bitwise AND assignment	<code>a &= b</code>	Yes
Bitwise OR assignment	<code>a = b</code>	Yes
Bitwise XOR assignment	<code>a ^= b</code>	Yes
Bitwise left shift assignment	<code>a <<= b</code>	Yes
Bitwise right shift assignment	<code>a >>= b</code>	Yes

Figure 7.4: C++ operators on basic types

Member and pointer operators		
Operator name	Syntax	Overloadable
Array subscript	<code>a [b]</code>	Yes
Indirection ("variable pointed to by a")	<code>*a</code>	Yes
Reference ("address of a")	<code>&a</code>	Yes
Member <i>b</i> of object pointed to by <i>a</i>	<code>a->b</code>	Yes
Member <i>b</i> of object <i>a</i>	<code>a.b</code>	No
Member pointed to by <i>b</i> of object pointed to by <i>a</i> ,	<code>a->*b</code>	Yes
Member pointed to by <i>b</i> of object <i>a</i>	<code>a.*b</code>	No

Other operators		
Operator name	Syntax	Overloadable
Function call See Function object .	<code>a(a1, a2)</code>	Yes
Comma	<code>a, b</code>	Yes
Ternary conditional	<code>a ? b : c</code>	No
Scope resolution	<code>a::b</code>	No
Size-of	<code>sizeof(a) sizeof(type)</code>	No
Type identification	<code>typeid(a) typeid(type)</code>	No
Cast	<code>(type) a</code>	Yes
Allocate storage	<code>new type</code>	Yes
Allocate storage (array)	<code>new type[n]</code>	Yes
Deallocate storage	<code>delete a</code>	Yes
Deallocate storage (array)	<code>delete[] a</code>	Yes

Figure 7.5: C++ operators on basic types(contd)

7.4. OPERATORS THAT CANNOT BE OVERLOADED

7.4 Operators that cannot be overloaded

Operators that cannot be overloaded

. . * :: ?:

`new delete sizeof typeid
static_cast dynamic_cast
const_cast reinterpret_cast`

Operators that can be overloaded

`operator new
operator delete
operator new[]
operator delete[]`

`+ - * / % ^ & | ~
!= == < > += -= *= /= %=
^= &= |= << >> >>= <<= == !=
<= >= && || ++ -- , ->* ->
() []`

Never overload `&&`, `||`, or `,`

Figure 7.6: Operators that should not be overloaded

7.5 Operators precedence in C++

CHAPTER 7. OPERATOR OVERLOADING

Precedence	Operator	Description	Overloadable	Associativity
2	::	scope resolution	no	left to right
	()	function call	yes	
	[]	array access	yes	
	->	member access	yes	
	.	member access	no	
	++ --	postfix	yes	
	dynamic_cast static_cast reinterpret_cast const_cast typeid	type conversion	no	
3	! not	logical negation	yes	right to left
	~ compl	bitwise negation (complement)	yes	
	++ --	prefix	yes	
	+ -	unary sign operations	yes	
	* &	indirection and reference	yes	
	sizeof	Size (of the type) of the operand in bytes	no	
	new new[] delete delete[] (type)	dynamic memory management	yes	
4	->*	member pointer selector	yes	left to right
	.*	member object selector	no	
5	* / %	arithmetic operations	yes	left to right
6	+ -	shift operations	yes	left to right
7	<< >>	relational operations	yes	left to right
8	< <= > >=	bitwise AND	yes	left to right
9	== != not_eq	bitwise XOR	yes	left to right
10	& bitand	bitwise OR	yes	left to right
11	^ xor	logical AND	yes	left to right
12	bitor	logical OR	yes	left to right
13	&& and	Ternary conditional (if-then-else)	no	right to left
14	or	assignment	yes	right to left
15	?:	Sequential evaluation operator	yes	left to right
16	= += -= *= /= %= &= ^= = <<= >>=			
17	,			

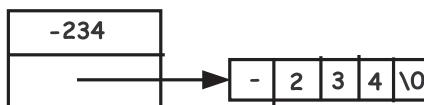
Figure 7.7: Operators precedence in C++

7.6 Int class

CHAPTER 7. OPERATOR OVERLOADING

```

class Int {
private:
    int _d;
    char* _s;
    static bool _display ;
};

Int x(-234);

class Int {
public:
    Int(int x = 0) ;
    ~Int() ;
    Int(const Int& x) ;
    Int& operator=(const Int& x) ;

    int num_digit() const { return strlen(_s) ; }
    const char& operator[](int index) const ;
    friend ostream& operator<<(ostream& o, const Int& x) ;
    static void set_verbose(bool x) {_display = x ; }

    Int operator+() const ;
    Int operator-() const;
    Int& operator++();
    Int operator++(int i) ;
    Int& operator--();
    Int operator--(int i) ;
    Int operator~() const ;
    bool operator!() const ;
    Int* operator&() ;

/* binary operators as member functions */
/* + operator */
    Int operator+(const Int& b) const;
    Int operator+(int b) const;

/* -= operator */
    Int& operator-=(const Int& b);
    Int& operator-=(int b);

/* <= operator */
    bool operator<=(const Int& b) const ;
    bool operator<=(int b) const ;

/* && operator */
    bool operator&&(const Int& b) const ;
    bool operator&&(int b) const ;
};

friend Int operator+(const Int& a) ;
friend Int operator-(const Int& a);
friend Int& operator++(Int& a) ;
friend Int operator++(Int& a,int i) ;
friend Int& operator--(Int& a) ;
friend Int operator--(Int& a, int i) ;
friend Int operator~(const Int& a) ;
friend bool operator!(const Int& a) ;
friend Int* operator&(Int& a) ;

/* binary operators as friend functions */
/* + operator */
friend Int operator+(const Int& a, const Int& b);
friend Int operator+(const Int& a, int b) ;
friend Int operator+(int a, const Int& b) ;

/* -= operator */
friend Int& operator-=(Int& a, const Int& b);
friend Int& operator-=(Int& a, int b);

/* <= operator */
friend bool operator<=(const Int& a, const Int& b) ;
friend bool operator<=(const Int& a, int b) ;
friend bool operator<=(int b, const Int& a);

/* && operator */
friend bool operator&&(const Int& a, const Int& b);
friend bool operator&&(const Int& a, int b) ;
friend bool operator&&(int b, const Int& a);

```

Figure 7.8: Int class

7.6. INT CLASS

7.6.1 Unary operators

```

class Int{
private
    int _x ;
public:
    Int(int x = 0):_x(x) {
        cout << "In constructor\n" ;
    }
    ~Int() {
        cout << "In disstructor\n" ;
        _x = 999999 ;
    }
    Int(const Int& r) {
        _x = r._x ;
        cout << "In copy constructor\n" ;
    }
    Int& operator=(const Int& r) {
        _x = r._x ;
        cout << "In equal operator\n" ;
        return *this ;
    }
    friend ostream& operator<<(ostream& o, const Int& r) {
        o << r._x ;
        return o ;
    }
}

Int& pre_increment() {
    ++_x;
    return *this;
}
Int post_increment() {
    Int t = *this;
    ++_x;
    return t;
}

Int& increment() {
    ++_x;
    return *this;
}
Int increment(int junk) {
    Int t = *this;
    ++_x;
    return t;
}

Int& operator++(){
    ++_x;
    return *this;
}
Int operator++(int j) {
    Int t = *this;
    ++_x;
    return t;
}

```

Int y(45);
Int z = y.pre_increment();
Int w = y.post_increment();

Int y(45);
Int z = y.increment();
Int w = y.increment(9);

Int y(45);
Int z = ++y ;
Int w = y++ ;

Figure 7.9: Overloading unary increment

```
1 #include <iostream>
2 using namespace std ;
3
4 class Int{
5 public:
6     Int(int x = 0):_x(x) {
7         cout << "In constructor\n" ;
8     }
9     ~Int() {
10        cout << "In disstructor\n" ;
11        _x = 999999 ;
12    }
13    Int(const Int& r) {
14        _x = r._x ;
15        cout << "In copy constructor\n" ;
16    }
17    Int& operator=(const Int& r) {
18        _x = r._x ;
19        cout << "In equal operator\n" ;
20        return *this ;
21    }
22    friend ostream& operator<<(ostream& o, const Int& r) {
23        o << r._x ;
24        return o ;
25    }
26
27    Int& pre_increment() {
28        ++_x;
29        return *this;
30    }
31
32    Int post_increment() {
33        Int t = *this;
34        ++_x;
35        return t;
36    }
37
38    Int& increment() {
39        ++_x;
40        return *this;
41    }
42
43    Int increment(int junk) {
44        Int t = *this;
45        ++_x;
46        return t;
47    }
48
49    Int& operator++(){
50        ++_x;
51        return *this;
52    }
53    Int operator++(int j) {
54        Int t = *this;
55        ++_x;
56        return t;
57    }
58
59 private:
60     int _x ;
61
62 };
63
64 void main() {
65 {
66     Int y(45);
```

```
67     cout << "y = " << y << endl;
68     Int z = y.pre_increment();
69     cout << "y = " << y << endl;
70     cout << "z = " << z << endl;
71
72     cout << "y = " << y << endl;
73     Int w = y.post_increment();
74     cout << "y = " << y << endl;
75     cout << "w = " << w << endl;
76 }
77 cout << "-----\n";
78 {
79     Int y(45);
80     cout << "y = " << y << endl;
81     Int z = y.increment();
82     cout << "y = " << y << endl;
83     cout << "z = " << z << endl;
84
85     cout << "y = " << y << endl;
86     Int w = y.increment(9);
87     cout << "y = " << y << endl;
88     cout << "w = " << w << endl;
89 }
90 cout << "-----\n";
91 {
92     Int y(45);
93     cout << "y = " << y << endl;
94     Int z = ++y;
95     cout << "y = " << y << endl;
96     cout << "z = " << z << endl;
97
98     cout << "y = " << y << endl;
99     Int w = y++;
100    cout << "y = " << y << endl;
101    cout << "w = " << w << endl;
102 }
103 cout << "-----\n";
104 }
```

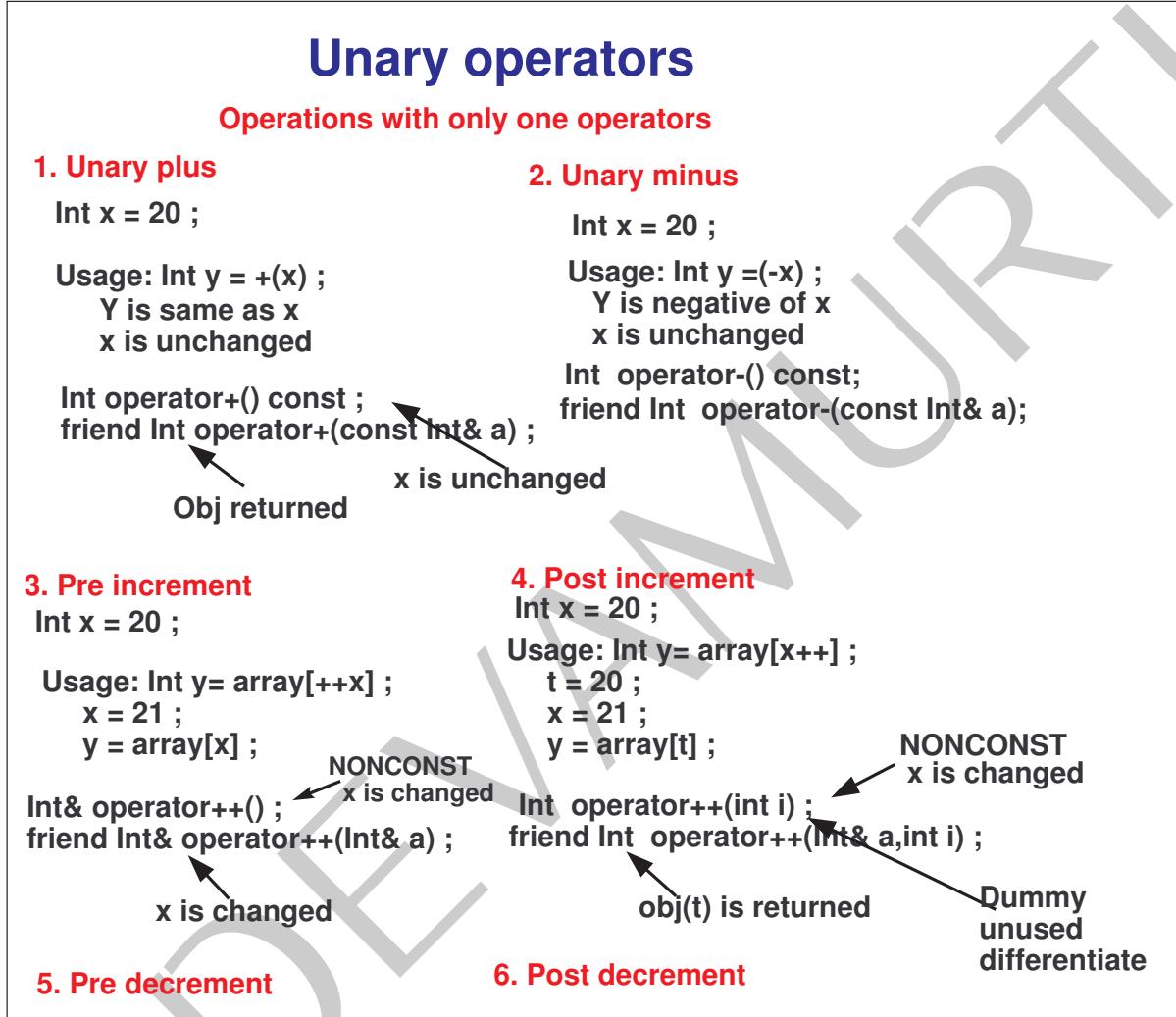


Figure 7.10: Overloading unary operators as member function

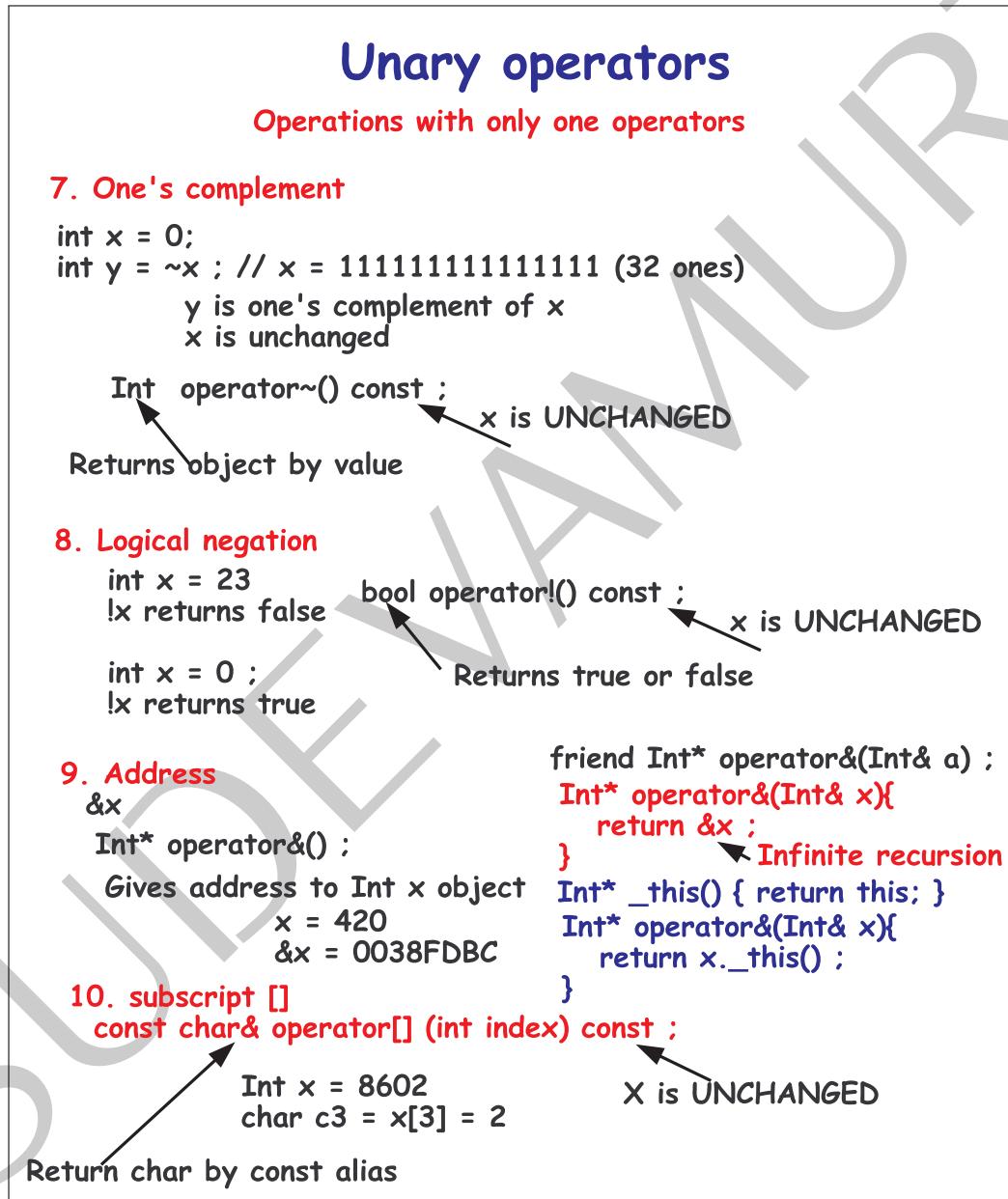


Figure 7.11: Overloading unary operators as member function

7.6.2 Binary operators

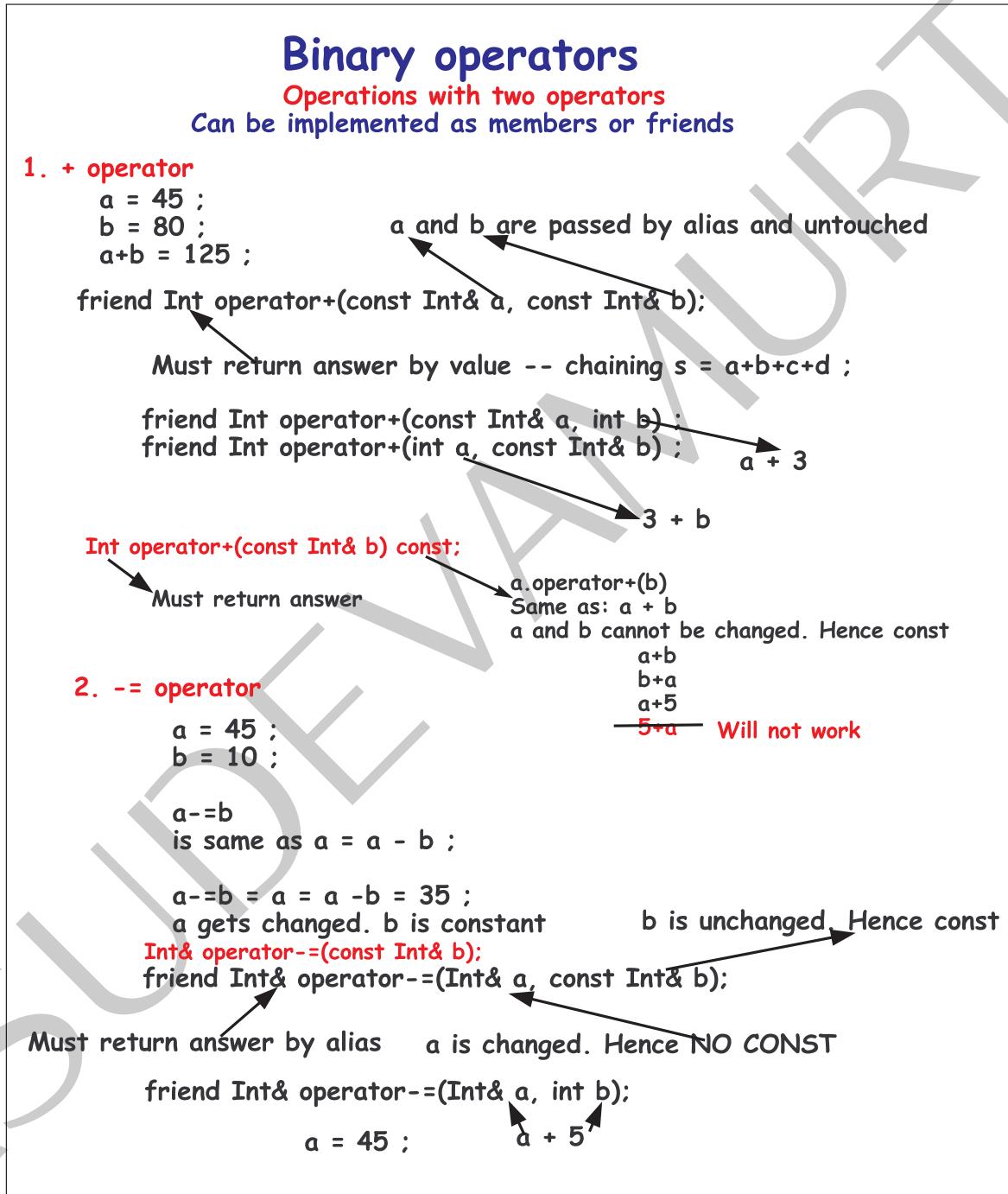


Figure 7.12: Overloading binary operators as friend

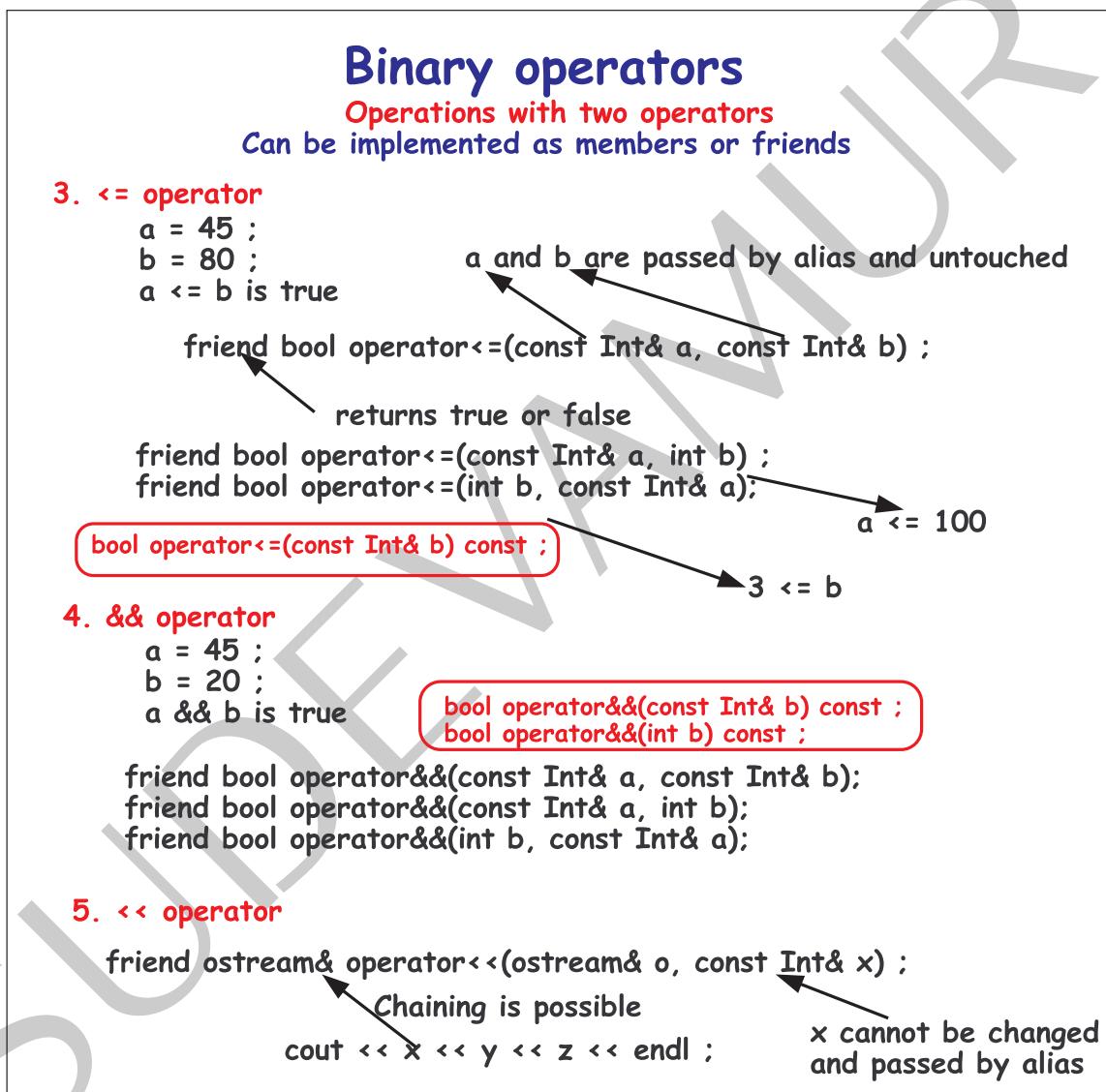


Figure 7.13: Overloading binary operators as friend

```
1 /*-----  
2 Copyright (c) 2015 Author: Jagadeesh Vasudevamurthy  
3 Filename: Int_main.cpp  
4  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 -----*/  
7 #include "Int.h"  
8  
9  
10 /*-----  
11 Place 1 of 2 you have to change  
12 -----*/  
13 //#define T int  
14 #define T Int  
15  
16 /*-----  
17  
18 -----*/  
19 static void P(const char* s) {  
20   cout << s << endl ;  
21 }  
22  
23 /*-----  
24 For exact match of x is Int  
25 -----*/  
26 static void P(const char* s, const Int& x) {  
27   cout << s << " = " << x << endl ;  
28 }  
29  
30 /*-----  
31 For exact match of x is bool  
32 -----*/  
33 static void P(const char* s, bool x) {  
34   cout << s << " = " << x << endl ;  
35 }  
36  
37 /*-----  
38 For exact match of x is int  
39 -----*/  
40 static void P(const char* s, int x) {  
41   cout << s << " = " << x << endl ;  
42 }  
43  
44 /*-----  
45 -----*/  
46  
47 static void test_unary() {  
48   P("unary + test") ;  
49   T x = 20 ;  
50   P("01. x",x) ;  
51   T yy = +x ;  
52   P("02. +x",x) ;  
53   P("02a: yy",yy) ;  
54   P("unary - test") ;  
55   P("03. x",x) ;  
56   T y = -x;  
57   P("04. x",x) ;  
58   P("04a. y",y) ;  
59   y = -20 ;  
60   P("05. y",y) ;  
61   P("06. -y",(-y)) ;  
62   P("pre increment ++x test") ;  
63   P("07. x",x) ;  
64   {  
65     T y = 0 ;  
66     P("08. y = ++x",y = ++x) ;
```

```
67     P("08A. y",y) ;
68 }
69 P("09. x",x) ;
70 P("post increment x++ test") ;
71 P("10. x",x) ;
72 {
73     T y = 0 ;
74     P("11. y = x++",y = x++) ;
75     P("11A. y",y) ;
76 }
77 P("12. x",x) ;
78 P("pre decrement --x test") ;
79 P("13. x",x) ;
80 P("14. --x",--x) ;
81 P("15. x",x) ;
82 P("post decrement x-- test") ;
83 P("16. x",x) ;
84 P("17. x--",x--) ;
85 P("18. x",x) ;
86 P("Complement ~x-- (one's complement) test") ;
87 P("19. x",x) ;
88 P("20. ~x",~x) ;
89 P("Logic negation !x test") ;
90 P("19. x",x) ;
91 P("20. !x",!x) ;
92 x = 0 ;
93 P("21. x",x) ;
94 P("22. !x",!x) ;
95 x = 420 ;
96 P("Address &x test") ;
97 P("23. x",x) ;
98 cout << "24. &x = " << &x << endl ;
99 P("Indirection *ptr test") ;
100 T* ptr = &x ;
101 cout << "25. *ptr = " << *ptr << endl ;
102 }
103
104 /-----
105 -----
106 -----*/
107 static void test_binary() {
108 P("Binary + test") ;
109 T x = 20 ;
110 T y = -10 ;
111 P("01. x",x) ;
112 P("02. y",y) ;
113 P("03. x+y",x+y) ;
114 P("04. x+5",x+5) ;
115 #ifndef AS_MEMBER
116 P("05. 5+x",5+x) ; //This will not work AS_MEMBER
117#endif
118
119 P("06. 5+8",5+8) ;
120
121 P("Binary - test") ;
122 P("Binary * test") ;
123 P("Binary / test") ;
124 P("Binary % test") ;
125
126 P("Binary & test") ;
127 P("Binary | test") ;
128 P("Binary ^ test") ;
129
130 P("Binary += test") ;
131 P("Binary -= test") ;
132 P("07. x",x) ;
```

```
133 P("08. y",y) ;
134 P("09. x-=y ", x-= y) ;
135 P("10. x",x) ;
136 P("11. y",y) ;
137 P("12. y-=x ", y-= x) ;
138 P("13. x",x) ;
139 P("14. y",y) ;
140 P("15. x-=5 ", x-= 5) ;
141 P("16. x",x) ;
142 P("17. y",y) ;
143 P("18. y-=3 ", y-= 3) ;
144
145 P("Binary *= test") ;
146 P("Binary /= test") ;
147 P("Binary %= test") ;
148
149 P("Binary &= test") ;
150 P("Binary |= test") ;
151 P("Binary ^= test") ;
152
153 P("Binary == test") ;
154 P("Binary != test") ;
155 P("Binary < test") ;
156 P("Binary > test") ;
157
158 P("Binary <= test") ;
159 x = 67 ;
160 y = 99 ;
161 P("19. x",x) ;
162 P("20. y",y) ;
163 P("21. x<=y ", x<=y) ;
164 P("22. x<=5 ", x<=5) ;
165 #ifndef AS_MEMBER
166 P("23. 5<=x ", 5<=x) ;//This will not work AS_MEMBER
167 #endif
168
169 P("Binary >= test") ;
170
171 P("Binary && test") ;
172 x = 67 ;
173 y = 99 ;
174 P("24. x",x) ;
175 P("25. y",y) ;
176 P("26. x&&y ", x&&y) ;
177 P("27. x&&67 ", x&&67) ;
178 #ifndef AS_MEMBER
179 P("23. 67&&x ", 66&&x) ;//This will not work AS_MEMBER
180 #endif
181
182 P("Binary || test") ;
183
184 }
185
186 /*****
187 ****
188 ****
189 static void test_special() {
190 cout << "-----test_special-----" << endl ;
191 Int x(8602) ;
192 cout << "x = " << x << " x[3] = " << x[3] -'0' << " x[100] = " << x[100] -'0' << endl ;
193 Int y(5) ;
194 Int z(10) ;
195 Int w = x + y + z + 77 + y ;
196 cout << "w = " << w << endl ;
197 }
```

```
199 /*-----  
200 -----*/  
201  
202 int main() {  
203     Int::set_verbose(true) ;  
204     test_unary() ;  
205     test_binary() ;  
206     test_special() ;  
207     return 0 ;  
208 }  
209
```

7.7 A set class

Definition

What is a set? Well, simply put, it's a **collection**.

Set of prime numbers: {2, 3, 5, 7, 11, 13, 17}
Positive multiples of 3 that are less than 10: {3, 6, 9}

iset64

1. Empty set $a = \{ \}$
2. The set we are implementing will have numbers between 0 to 63 only
3. In sets it does not matter what order the elements are in
 Example: {1,2,3,4} is the same set as {3,1,4,2}
4. Number of elements in the above set = 4
5. In our set minimum number of element is 0 - Empty set
 maximum number of element is 64
 and the elements will be between 0 to 63

1

Adding an element to set
 $a = \{1,2\}$
 $a + 5 = \{1,2,5\}$
 $a + \{10,63\} = \{0,1,2,5,63\}$

3

Union of sets
 (overload with +)



4

Intersection of sets
 (overload with *)

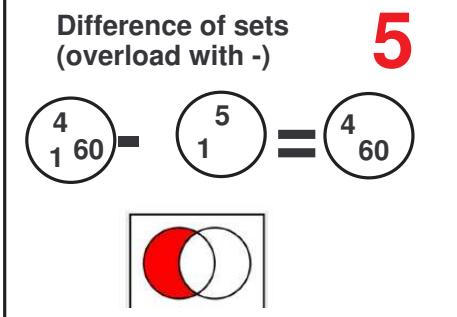


2

Removing an element
 $a = \{1,6,10\}$
 $a - 6 = \{1,10\}$

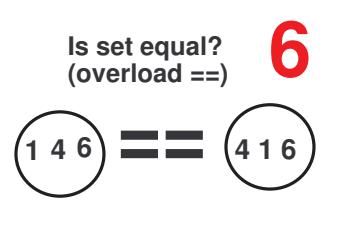
5

Difference of sets
 (overload with -)



6

Is set equal?
 (overload ==)



What to submit?

1. iset64test.cpp cannot be modified. All tests must pass
 2. Submit as a hardcopy
 1. iset64.h
 2. iset64.cpp
 3. Output as a pdf file
 4. A word doc that explains
 1. Data structure used
 2. Algorithms used for all the 6 methods above

Figure 7.14: Set

7.7. A SET CLASS

iset64

<p>++i Preincrement (overload with <code>++</code>)</p> $\begin{array}{c} 2 \\ 1 \end{array} \begin{array}{c} 63 \\ + + = \end{array} \begin{array}{c} 3 \\ 2 \end{array} \begin{array}{c} 0 \end{array}$ <p>i++ Postincrement</p>	<p>--i Predecrement (overload with <code>--</code>)</p> $\begin{array}{c} 2 \\ 0 \end{array} \begin{array}{c} 63 \\ - - = \end{array} \begin{array}{c} 1 \\ 62 \\ 63 \end{array}$ <p>i-- Postdecrement</p>	<p>~i complement (overload with <code>~</code>)</p> $\sim \begin{array}{c} 2 \\ 0 \end{array} \begin{array}{c} 63 \\ = \end{array} \begin{array}{c} 1 \\ 3 \end{array} \begin{array}{c} to \\ 62 \end{array}$
<p>10</p> <p>if (a) conversion operator (overload with <code>()</code>)</p>	<p>11</p> <p>if (!a) conversion operator (overload with <code>!</code>)</p>	<p>De Morgan laws</p> $\begin{array}{l} \overline{(a+b)} = \overline{a} \cdot \overline{b} \\ \overline{(a \cdot b)} = \overline{a} + \overline{b} \end{array}$

Figure 7.15: Set

```
1 /*-
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy
3 file: iset64test.cpp
4
5 On linux:
6 g++ iset64.cpp iset64test.cpp
7 valgrind a.out
8
9 -----*/
10
11 /*-
12 This file test iset64 object
13 -----*/
14
15 /*-
16 All includes here
17 -----*/
18 #include "iset64.h"
19
20 /*-
21 test a set
22 -----*/
23 void test_basic() {
24     iset64 a;
25     cout << "a = " << a << endl;
26     a = a + 5;
27     cout << "set a after adding 5 = " << a << endl;
28     a = a + 5;
29     cout << "set a after adding 5 = " << a << endl;
30     a += 63;
31     a += 0;
32     cout << "set a after adding 0 and 63 = " << a << endl;
33     int x[] = { 1, 3, 6 };
34     iset64 b(x, sizeof(x) / sizeof(int));
35     cout << "set b = " << b << endl;
36     b = b - 3;
37     cout << "set b after removing 3 = " << b << endl;
38     b = b - 3;
39     cout << "set b after removing 3 = " << b << endl;
40     b = b - 10;
41     cout << "set b after removing 10 = " << b << endl;
42     b = b - 6;
43     cout << "set b after removing 6 = " << b << endl;
44     b = b - 1;
45     cout << "set b after removing 1 = " << b << endl;
46     b = b + 10;
47     b = b + 2;
48     cout << "set b after adding {10,2} = " << b << endl;
49 }
50
51 /*-
52 test union
53 -----*/
54 void test_union() {
55     {
56         cout << "TESTING: iset64 operator+(const iset64& a, const iset64& b)" << endl;
57         iset64 a;
58         a += 1;
59         a += 2;
60         iset64 b;
61         b += 1;
62         b += 2;
63         b += 3;
64         cout << "Set a " << a << endl;
65         cout << "Set b " << b << endl;
66         iset64 c = a + b;
```

```
67     cout << "a + b = " << c << endl;
68 }
69 {
70     cout << "TESTING:iset64 operator+(const iset64& a, const int b)" << endl;
71     iset64 a;
72     a += 1;
73     a += 2;
74     cout << a << endl;
75     a = a + 1;
76     cout << "{1,2} + 1 = " << a << endl;
77     a += 1;
78     a += 2;
79     cout << a << endl;
80     a = a + 3;
81     cout << "{1,2} + 3 = " << a << endl;
82 }
83 {
84     cout << "TESTING:iset64 operator+(const int b, const iset64& a)" << endl;
85     iset64 a;
86     a += 1;
87     a += 2;
88     cout << "Set a " << a << endl;
89     a = 1 + a;
90     cout << " 1 + {1,2} = " << a << endl;
91     a += 1;
92     a += 2;
93     cout << "Set a " << a << endl;
94     a = 3 + a;
95     cout << " 3 + {1,2}  = " << a << endl;
96 }
97 {
98 {
99     cout << "TESTING:iset64& iset64::operator+=(const iset64& a)" << endl;
100    iset64 b;
101    b += 1;
102    b += 2;
103    iset64 a;
104    a += 1;
105    a += 3;
106    cout << "Set b " << b << endl;
107    cout << "Set a " << a << endl;
108    b += a;
109    cout << " {1,2} + {1,3}  = " << b << endl;
110 }
111 {
112     cout << "iset64& iset64::operator+=(const int b)" << endl;
113     iset64 a;
114     a += 1;
115     a += 2;
116     cout << "Set a " << a << endl;
117     a += 3;
118     cout << " {1,2} + 3  = " << a << endl;
119 }
120 {
121 //test chaining
122     iset64 a;
123     a += 1;
124     a += 2;
125     iset64 b;
126     b += 3;
127     b += 4;
128     iset64 c;
129     c += 7;
130     c += 8;
131     iset64 d = a + b + c + 5;
132     cout << "Set a " << a << endl;
```

```
133     cout << "Set b " << b << endl;
134     cout << "Set c " << c << endl;
135     cout << "Set d " << d << endl;
136 }
137 }
138 */
139 /*-----
140 test difference
141 -----*/
142 void test_difference() {
143 {
144     cout << "TESTING: iset64 operator-(const iset64& a, const iset64& b)" << endl;
145     iset64 a;
146     a += 1;
147     a += 2;
148     iset64 b;
149     b += 1;
150     b += 2;
151     iset64 c = a - b;
152     cout << "Set a " << a << endl;
153     cout << "Set b " << b << endl;
154     cout << "a - b = " << c << endl;
155 }
156 {
157     cout << "TESTING: iset64 operator-(const iset64& a, const iset64& b)" << endl;
158     iset64 a;
159     a += 1;
160     a += 5;
161     iset64 b;
162     b += 1;
163     b += 2;
164     b += 3;
165     iset64 c = a - b;
166     cout << "Set a " << a << endl;
167     cout << "Set b " << b << endl;
168     cout << "a - b = " << c << endl;
169 }
170 {
171     cout << "TESTING: iset64 operator-(const iset64& a, const int b)" << endl;
172     iset64 a;
173     a += 1;
174     a += 2;
175     cout << "Set a " << a << endl;
176     a = a - 3;
177     cout << "a - 3 = " << a << endl;
178 }
179 {
180     cout << "TESTING: iset64 operator-(const int b, const iset64& a)" << endl;
181     iset64 a;
182     a += 1;
183     a += 2;
184     cout << "Set a " << a << endl;
185     a = 3 - a;
186     cout << "3 - a = " << a << endl;
187 }
188 {
189     cout << "TESTING: iset64& iset64::operator+=(const iset64& a)" << endl;
190     iset64 a;
191     a += 1;
192     a += 3;
193     iset64 b;
194     b += 1;
195     b += 2;
```

```
199     cout << "Set a " << a << endl;
200     cout << "Set b " << b << endl;
201     b -= a;
202     cout << "b -= a = " << b << endl;
203 }
204
205 {
206     cout << "TESTING: iset64& iset64::operator-=(const int b)" << endl;
207     iset64 a;
208     a += 1;
209     a += 2;
210     cout << "Set a " << a << endl;
211     a -= 3;
212     cout << "a -= 3 = " << a << endl;
213 }
214
215 //test chaining
216 iset64 a;
217 a += 1;
218 a += 2;
219 iset64 b;
220 b += 2;
221 b += 4;
222 iset64 c;
223 c += 2;
224 c += 8;
225 iset64 d = a - b - c + 5;
226 cout << "Set a " << a << endl;
227 cout << "Set b " << b << endl;
228 cout << "Set c " << c << endl;
229 cout << "Set d " << d << endl;
230 }
231 }
232
233 /*-----*
234 test intersection
235 -----*/
236 void test_intersection() {
237 {
238     cout << "TESTING: iset64 operator*(const iset64& a, const iset64& b)" << endl;
239     iset64 a;
240     a += 1;
241     a += 2;
242     iset64 b;
243     b += 1;
244     b += 2;
245     b += 3;
246     cout << "Set a " << a << endl;
247     cout << "Set b " << b << endl;
248     iset64 c = a * b;
249     cout << "a * b = " << c << endl;
250 }
251 {
252     cout << "TESTING: iset64 operator*(const iset64& a, const int b)" << endl;
253     iset64 a;
254     a += 1;
255     a += 2;
256     cout << "Set a " << a << endl;
257     a = a * 1;
258     cout << "{1,2} * 1 = " << a << endl;
259     a += 1;
260     a += 2;
261     cout << "Set a " << a << endl;
262     a = a * 3;
263     cout << "{1,2} * 3 = " << a << endl;
264 }
```

```
265 {  
266     cout << "TESTING:iset64 operator*(const int b, const iset64& a)" << endl;  
267     iset64 a;  
268     a += 1;  
269     a += 2;  
270     cout << "Set a " << a << endl;  
271     a = 1 * a;  
272     cout << " 1 * {1,2} = " << a << endl;  
273     a += 1;  
274     a += 2;  
275     cout << "Set a " << a << endl;  
276     a = 3 * a;  
277     cout << " 3 * {1,2} = " << a << endl;  
278 }  
279  
280 {  
281     cout << "TESTING:iset64& iset64::operator*=(const iset64& a)" << endl;  
282     iset64 b;  
283     b += 1;  
284     b += 2;  
285     iset64 a;  
286     a += 1;  
287     a += 3;  
288     cout << "Set b " << b << endl;  
289     cout << "Set a " << a << endl;  
290     b *= a;  
291     cout << " {1,2} * {1,3} = " << b << endl;  
292 }  
293 {  
294     cout << "iset64& iset64::operator*=(const int b)" << endl;  
295     iset64 a;  
296     a += 1;  
297     a += 2;  
298     cout << "Set a " << a << endl;  
299     a *= 3;  
300     cout << " {1,2} * 3 = " << a << endl;  
301 }  
302 {  
303     //test chaining  
304     iset64 a;  
305     a += 1;  
306     a += 2;  
307     iset64 b;  
308     b += 2;  
309     b += 4;  
310     iset64 c;  
311     c += 2;  
312     c += 8;  
313     iset64 d = a * b * c + 5;  
314     cout << "Set a " << a << endl;  
315     cout << "Set b " << b << endl;  
316     cout << "Set c " << c << endl;  
317     cout << "Set d " << d << endl;  
318 }  
319 }  
320  
321 /*-----  
322 test equal  
323 -----*/  
324 void test_equal_not_equal() {  
325     cout << "TESTING: bool operator==(const iset64& a, const iset64& b)" << endl;  
326     iset64 a;  
327     a += 1;  
328     a += 2;
```

```
331     iset64 b;
332     b += 1;
333     b += 2;
334     cout << "Set a " << a << endl;
335     cout << "Set b " << b << endl;
336     cout << "a == b " << boolalpha << (a == b) << endl;
337     b -= 1;
338     cout << a;
339     cout << b;
340     cout << "a == b " << boolalpha << (a == b) << endl;
341 }
342 {
343     cout << "TESTING: bool operator!=(const iset64& a, const iset64& b)" << endl;
344     iset64 a;
345     a += 1;
346     a += 2;
347     iset64 b;
348     b += 1;
349     b += 2;
350     cout << "Set a " << a << endl;
351     cout << "Set b " << b << endl;
352     cout << "a != b " << boolalpha << (a != b) << endl;
353     b -= 1;
354     cout << "Set a " << a << endl;
355     cout << "Set b " << b << endl;
356     cout << "a != b " << boolalpha << (a != b) << endl;
357 }
358 }
359
360 /*-----
361 ++ and --
362 -----*/
363 void test_pre_post_inr_dec() {
364 {
365     int x[] = { 1, 2, 63 };
366     iset64 a(x, sizeof(x) / sizeof(int));
367     cout << "a = " << a << endl;
368     ++a;
369     cout << "+a = " << a << endl;
370     int y[] = { 2, 3, 0 };
371     iset64 b(y, sizeof(y) / sizeof(int));
372     assert(a == b);
373 }
374 {
375     int x[] = { 1, 2, 63 };
376     iset64 a(x, sizeof(x) / sizeof(int));
377     cout << "a = " << a << endl;
378     iset64 acopy(x, sizeof(x) / sizeof(int));
379     cout << "acopy = " << acopy << endl;
380     iset64 rhs = a++;
381     assert(rhs == acopy);
382     cout << "a++ = " << a << endl;
383     cout << "rhs = " << rhs << endl;
384     int y[] = { 2, 3, 0 };
385     iset64 b(y, sizeof(y) / sizeof(int));
386     assert(a == b);
387 }
388 {
389     int x[] = { 0, 2, 63 };
390     iset64 a(x, sizeof(x) / sizeof(int));
391     cout << "a = " << a << endl;
392     --a;
393     cout << "--a = " << a << endl;
394     int y[] = { 63, 1, 62 };
395     iset64 b(y, sizeof(y) / sizeof(int));
396     assert(a == b);
```

```
397 }
398 {
399     int x[] = { 0, 2, 63 };
400     iset64 a(x, sizeof(x) / sizeof(int));
401     cout << "a = " << a << endl;
402     iset64 acopy(x, sizeof(x) / sizeof(int));
403     cout << "acopy = " << acopy << endl;
404     iset64 rhs = a--;
405     assert(rhs == acopy);
406     cout << "a-- = " << a << endl;
407     cout << "rhs = " << rhs << endl;
408     int y[] = { 63, 1, 62 };
409     iset64 b(y, sizeof(y) / sizeof(int));
410     assert(a == b);
411 }
412 */
413
414 /*-----*
415 ~
416 Complement of a set.
417 The complement of A is the set of all element in the universal set U, but not in A.
418 a = {0,2,63}
419 x = ~a
420 {1,3,...,62}
421 -----*/
422 void test_complement() {
423 {
424     int x[] = { 0, 2, 63 };
425     iset64 a(x, sizeof(x) / sizeof(int));
426     cout << "a = " << a << endl;
427     iset64 nota = (~a);
428     cout << "~a = " << nota << endl;
429     iset64 ans;
430     ans += 1;
431     for (int i = 3; i < 63; ++i) {
432         ans += i;
433     }
434     cout << "ans = " << ans << endl;
435     assert(nota == ans);
436     ans = ~ans;
437     cout << "~ans = " << ans << endl;
438     assert(ans == a);
439 }
440 }
441
442 /*-----*
443 a = {0,2,63}
444 if (a) {
445 }
446 */
447 -----*/
448 void test_conversion_operator() {
449     int x[] = { 0, 2, 63 };
450     iset64 a(x, sizeof(x) / sizeof(int));
451     cout << "a = " << a << endl;
452     if (a) {
453         cout << "a exists\n";
454     } else {
455         cout << "a does not exists\n";
456     }
457     iset64 b;
458     cout << "b = " << b << endl;
459     if (b) {
460         cout << "b exists\n";
461     } else {
462         cout << "b does not exists\n";
```

```
463     }
464 }
465
466 /*-----
467 a = {0,2,63}
468 if (!a) {
469 }
470 */
471 -----*/
472 void test_not_operator() {
473     int x[] = { 0, 2, 63 };
474     iset64 a(x, sizeof(x) / sizeof(int));
475     cout << "a = " << a << endl;
476     if (!a) {
477         cout << "a does not exists\n";
478     } else {
479         cout << "a exists\n";
480     }
481     iset64 b;
482     cout << "b = " << b << endl;
483     if (!b) {
484         cout << "b does not exists\n";
485     } else {
486         cout << "b exists\n";
487     }
488 }
489
490 /*-----
491 (a+b)' = a'. b'
492 (a.b)' = a' + b'
493 -----*/
494 void test_demorgan_laws(const int x[], int lx, const int y[], int ly) {
495 {
496     iset64 a(x, lx);
497     cout << "a = " << a << endl;
498
499     iset64 b(y, ly);
500     cout << "b = " << b << endl;
501
502     iset64 aplusb = a + b;
503     cout << "aplusb = " << aplusb << endl;
504
505     iset64 aplusbbar = ~(aplusb);
506     cout << "aplusbbar = " << aplusbbar << endl;
507
508     iset64 abar = ~(a);
509     cout << "abar = " << abar << endl;
510
511     iset64 bbar = ~(b);
512     cout << "bbar = " << bbar << endl;
513
514     iset64 abarplusbbar = abar + bbar;
515     cout << "abarplusbbar = " << abarplusbbar << endl;
516
517     iset64 abardotbbar = abar * bbar;
518     cout << "abardotbbar = " << abardotbbar << endl;
519
520     iset64 adotb = a * b;
521     cout << "adotb = " << adotb << endl;
522
523     iset64 adotbbar = ~(adotb);
524     cout << "adotbbar = " << adotbbar << endl;
525
526     assert(aplusbbar == abardotbbar);
527     cout << "Demorgan law (a+b)' = a'. b' is proved\n";
528     assert(adotbbar == abarplusbbar);
```

```
529     cout << "Demorgan law (a.b)' = a' + b' is proved\n";
530 }
531 }
532 */
533 /*-
534 (a+b)' = a' . b'
535 (a.b)' = a' + b'
536 -----*/
537 void test_demorgan_laws() {
538 {
539     int x[] = { 4, 5, 6 };
540     int y[] = { 5, 6, 8 };
541     test_demorgan_laws(x, (sizeof(x) / sizeof(int)), y, (sizeof(y) / sizeof(int)));
542 }
543 {
544     int x[] = { 1,2,4,5 };
545     int y[] = { 2,3,5,6 };
546     test_demorgan_laws(x, (sizeof(x) / sizeof(int)), y, (sizeof(y) / sizeof(int)));
547 }
548
549 }
550
551 /*-
552 test bed
553 -----*/
554 void testbed() {
555     test_basic();
556     test_union();
557     test_difference();
558     test_intersection();
559     test_equal_not_equal();
560     test_pre_post_inr_dec();
561     test_complement();
562     test_conversion_operator();
563     test_not_operator();
564     test_demorgan_laws();
565 }
566
567 /*-
568 main
569 -----*/
570 int main() {
571     testbed();
572     return 0;
573 }
574
575 //EOF
576
577
578
```

7.8. A LONG NUMBER CLASS

7.8 A long number class

unsigned longnum class

Write a class called longnum which can hold arbitrarily long unsigned numbers.

The class should have all the constructors, destructors, copy constructor equal operator, <<, and + operator

1. The user should be able to use this class like built-in type int object
2. One of the function in that class must be factorial function

```
class longnum {
    friend longnum factorial(unsigned n) ;
    void multiply(unsigned n) ;
};
```

Idea

You need to implement only + operator
and this is trivial

factorial (100) should produce result like:

```
9332621544394415268169923885626670049071596826438162146859
29638952175999932299156089414639761565182862536979208272237582511852109168
6400000000000000000000000000000000000000000000000000000000000000
```

Note that you need NOT
have to implement * operator
multiply function can be
implemented as repeated +

factorial(5) = 1 * 2 * 3 * 4 * 5 = 120 Idea

Use this main program like this to test your class.

```
int main() {
    longnum a(789) ;
    cout << "a = " << a << endl ;
    longnum b("56789") ;
    cout << "b = " << b << endl ;
    longnum c("12345678901234567890123456789012345678901234567890") ;
    cout << "c = " << c << endl ;
    longnum ta(a) ;
    cout << "ta = " << ta << endl ;
    ta = b ;
    cout << "ta = " << ta << endl ;
    cout << "a + c = " << a + c << endl ;
    a.multiply(5) ;
    cout << "a * 5 = " << a << endl ;
    longnum f100 = factorial(100) ;
    cout << "Factorial of 100 = " << endl ;
    cout << f100 << endl ;
    longnum f1000 = factorial(1000) ;
    cout << "Factorial of 1000 = " << endl ;
    cout << f1000 << endl ;
    return 0 ;
}
```

Idea

```
longnum factorial(unsigned n) {
    longnum o(1) ;
    for(unsigned i = 2; i <= n ; i++) {
        o.multiply(i) ;
    }
    return o ;
}
```

```
a = 789
b = 56789
c = 123456789012
1234567890123456
ta = 789
ta = 56789
a = 789 a + 20 =
.....
Factorial of 100=
.....
```

Figure 7.16: A long number class

7.9 mstring class

mstring class behaves exactly like *stl string* class. *mstringtest.cpp* file is given to you. You need to implement all the functions required in the *mstringtest.cpp*. Then comment the line `#define mstring string` in *mstringtest.cpp*. Your program must pass all the `asserts` and there should be no `memory leaks`. You should write all the declarations of the class *mstring* in *mstring.h* and all definitions of the class *mstring* in *mstring.cpp*. You cannot alter anything in *mstringtest.cpp*, except commenting `#define mstring string`. Class *mstring* in the file *mstring.h* must use `char* _s` as the **ONLY private data member**. You cannot use any other data members. You are free to use any number of private and public functions. **email me only *mstring.h* and *mstring.cpp***


```
63     assert(gs.size() == s.size()) ;
64 }
65 {
66     string gs1("9332621544394415268169923885626670049071596826438162146859
67     29638952175999932299156089414639761565182862536979208272237582511852109168 640
68 ") ; //gold
69     string gs2(gs1) ;
70     mstring s1("9332621544394415268169923885626670049071596826438162146859
71     29638952175999932299156089414639761565182862536979208272237582511852109168 640
72 ") ;
73     mstring s2(s1) ;
74     assert(gs1.length() == s1.length()) ;
75     assert(gs2.size() == s2.size()) ;
76 }
77 {
78     string gs("UCSC EXTENSION",4) ; //gold
79     mstring s("UCSC EXTENSION",4) ;
80     cout << "gs = " << gs << endl ;
81     cout << "s = " << s << endl ;
82     assert_string_mstring_equal(gs,s) ;
83 }
84 {
85     string gs("UCSC EXTENSION",5,6) ; //gold
86     mstring s("UCSC EXTENSION",5,6) ;
87     cout << "gs = " << gs << endl ;
88     cout << "s = " << s << endl ;
89     assert_string_mstring_equal(gs,s) ;
90 }
91 }
92 */
93 /*-----
94 test element access
95 -----*/
96 void test_element_access() {
97 {
98     string gs("Return iterator to beginning (public member function )") ; //gold
99     mstring s("abcu78") ;
100    const char& gc = gs[3] ;
101    const char& c = s[3] ;
102    cout << "gc = " << gc << endl ;
103    assert(gc == c) ;
104    const char& gc1 = gs[0] ;
105    const char& c1 = s[0] ;
106    cout << "c1 = " << c1 << endl ;
107    assert(gc1 != c1) ;
108 }
109 }
110 */
111 /*-----
112 test_operator_plus_equal
113 -----*/
114 void test_operator_plus_equal() {
115 {
116     string gs1("Return") ; //gold
117     string gs2(" gold") ; //gold
118     gs1 += gs2 ;
119     cout << "gs1 = " << gs1 << endl ;
120     mstring s1("Return") ;
121     mstring s2(" gold") ;
122     s1 += s2 ;
123     assert_string_mstring_equal(gs1,s1) ;
124     assert_string_mstring_equal(gs2,s2) ;
```

```
125 }  
126 {  
127     string gs1("Return") ; //gold  
128     gs1 += " book" ;  
129     cout << "gs1 = " << gs1 << endl ;  
130     mstring s1("Return") ;  
131     s1 += " book" ;  
132     assert_string_mstring_equal(gs1,s1) ;  
133 }  
134 {  
135     string gs1("Return") ; //gold  
136     gs1 += 'Z' ;  
137     cout << "gs1 = " << gs1 << endl ;  
138     mstring s1("Return") ;  
139     s1 += 'Z' ;  
140     assert(gs1 == s1) ;  
141     s1 += 'Z' ;  
142     assert_string_mstring_not_equal(gs1,s1) ;  
143 }  
144 }  
145 /*-----  
146 operator plus  
147 -----*/  
148 void test_operator_plus() {  
149 {  
150     string gs1("Return") ; //gold  
151     string gs2(" gold") ; //gold  
152     string gs3 = gs1 + gs2 ;  
153     cout << "gs3 = " << gs3 << endl ;  
154  
155     mstring s1("Return") ;  
156     mstring s2(" gold") ;  
157     mstring s3 = s1 + s2 ;  
158  
159     assert_string_mstring_equal(gs3,s3) ;  
160 }  
161 }  
162 {  
163     string gs1("Return") ; //gold  
164     string gs3 = gs1 + "gold" ;  
165     cout << "gs3 = " << gs3 << endl ;  
166  
167     mstring s1("Return") ;  
168     mstring s3 = s1 + "gold" ;  
169  
170     assert_string_mstring_equal(gs3,s3) ;  
171 }  
172 }  
173 {  
174     string gs1("Return") ; //gold  
175     string gs3 = "BOOK " + gs1 ;  
176     cout << "gs3 = " << gs3 << endl ;  
177  
178     mstring s1("Return") ;  
179     mstring s3 = "BOOK " + s1 ;  
180  
181     assert_string_mstring_equal(gs3,s3) ;  
182 }  
183 #if 0  
184 {  
185     string gs3 = "BOOK " + " C++" ; //gold  
186     cout << "gs3 = " << gs3 << endl ;  
187  
188     mstring s3 = "BOOK " + " C++" ;  
189     assert_string_mstring_equal(gs3,s3) ;  
190 }
```

```
191 #endif
192 {
193     string gs1("Return") ; //gold
194     string gs3 = 'P' + gs1 ;
195     cout << "gs3 = " << gs3 << endl ;
196
197     mstring s1("Return") ;
198     mstring s3 = 'P' + s1 ;
199
200     assert_string_mstring_equal(gs3,s3) ;
201 }
202 {
203     string gs1("Return") ; //gold
204     string gs3 = gs1 + 'P';
205     cout << "gs3 = " << gs3 << endl ;
206
207     mstring s1("Return") ;
208     mstring s3 = s1 + 'P';
209
210     assert_string_mstring_equal(gs3,s3) ;
211 }
212 {
213     string gs1("Return") ; //gold
214     string gs2 = " Hell" ;
215     string gs3 = gs1 + gs2 + 'a' + gs2 + gs1 ;
216     cout << "gs3 = " << gs3 << endl ;
217
218     mstring s1("Return") ; //gold
219     mstring s2 = " Hell" ;
220     mstring s3 = s1 + s2 + 'a' + s2 + s1 ;
221     cout << "s3 = " << s3 << endl ;
222
223     assert_string_mstring_equal(gs3,s3) ;
224 }
225 {
226     string gs1("abc") ;
227     string gs2("xyz") ;
228     string gs3("mnp") ;
229     string gs4("zoo") ;
230     string gs5 = gs1 + gs2 + "FOO1"+gs3 + gs4 + gs1 + gs2 + "FOO2"+gs3;
231     cout << "gs5 = " << gs5 << endl ;
232
233     string s1("abc") ;
234     string s2("xyz") ;
235     string s3("mnp") ;
236     string s4("zoo") ;
237     string s5 = s1 + s2 + "FOO1"+s3 + s4 + s1 + s2 + "FOO2"+s3;
238     cout << "s5 = " << s5 << endl ;
239     assert_string_mstring_equal(gs5,s5) ;
240 }
241 }
242 */
243 -----
244 relational operators
245 -----
246 void test_equal_equal_operators() {
247 {
248     string gs1("Jag"); //gold
249     string gs2("Jag"); //gold
250     assert(gs1 == gs2) ;
251
252     mstring s1("Jag") ;
253     mstring s2("Jag") ;
254     assert(s1 == s2) ;
255 }
256 }
```

```
257     string gs1("Jag"); //gold
258     assert("Jag" == gs1) ;
259
260     mstring s1("Jag") ;
261     assert("Jag" == s1) ;
262 }
263 {
264     string gs1("Jag"); //gold
265     assert(gs1 == "Jag") ;
266
267     mstring s1("Jag") ;
268     assert(s1 == "Jag") ;
269 }
270 }
271
272 /*-----
273 relational operators
274 -----*/
275 void test_not_equal_operators() {
276 {
277     string gs1("Jag"); //gold
278     string gs2("jag"); //gold
279     assert(gs1 != gs2) ;
280
281     mstring s1("Jag") ;
282     mstring s2("jag") ;
283     assert(s1 != s2) ;
284 }
285 {
286     string gs1("Jag"); //gold
287     assert("jag" != gs1) ;
288
289     mstring s1("Jag") ;
290     assert("jag" != s1) ;
291 }
292 {
293     string gs1("Jag"); //gold
294     assert(gs1 != "jag") ;
295
296     mstring s1("Jag") ;
297     assert(s1 != "jag") ;
298 }
299 }
300
301 /*-----
302 relational operators
303 -----*/
304 void test_less_than_operators() {
305 {
306     string gs1("Ja"); //gold
307     string gs2("Jag"); //gold
308     assert(gs1 < gs2) ;
309
310     mstring s1("Ja") ;
311     mstring s2("jag") ;
312     assert(s1 < s2) ;
313 }
314 {
315     string gs1("Jag"); //gold
316     assert("Ja" < gs1) ;
317
318     mstring s1("Jag") ;
319     assert("Ja" < s1) ;
320 }
321 {
322     string gs1("Jag"); //gold
```

```
323     assert(gs1 < "ab") ;
324
325     mstring s1("Jag") ;
326     assert(s1 < "ab") ;
327 }
328 }
329
330 /*-----
331 relational operators
332 -----*/
333 void test_less_than_equal_operators() {
334 {
335     string gs1("Ja"); //gold
336     string gs2("Jag"); //gold
337     assert(gs1 <= gs2) ;
338
339     mstring s1("Ja") ;
340     mstring s2("jag") ;
341     assert(s1 <= s2) ;
342 }
343 {
344     string gs1("Jag"); //gold
345     assert("Ja" <= gs1) ;
346
347     mstring s1("Jag") ;
348     assert("Ja" <= s1) ;
349 }
350 {
351     string gs1("Jag"); //gold
352     assert(gs1 <= "ab") ;
353
354     mstring s1("Jag") ;
355     assert(s1 <= "ab") ;
356 }
357 }
358
359 /*-----
360 relational operators
361 -----*/
362 void test_greater_than_operators() {
363 {
364     string gs1("XYfg"); //gold
365     string gs2("Jag"); //gold
366     assert(gs1 > gs2) ;
367
368     mstring s1("XYfg") ;
369     mstring s2("Jag") ;
370     assert(s1 > s2) ;
371 }
372 {
373     string gs1("Jag"); //gold
374     assert("Ma" > gs1) ;
375
376     mstring s1("Jag") ;
377     assert("Ma" > s1) ;
378 }
379 {
380     string gs1("Jag"); //gold
381     assert(gs1 > "Aa") ;
382
383     mstring s1("Jag") ;
384     assert(s1 > "Aa") ;
385 }
386 }
387
388 */-----
```

```
389 relational operators
390 -----
391 void test_greater_equal_operators() {
392 {
393     string gs1("XYfg"); //gold
394     string gs2("Jag"); //gold
395     assert(gs1 >= gs2) ;
396
397     mstring s1("XYfg") ;
398     mstring s2("Jag") ;
399     assert(s1 >= s2) ;
400 }
401 {
402     string gs1("Jag"); //gold
403     assert("Ma" >= gs1) ;
404
405     mstring s1("Jag") ;
406     assert("Ma" >= s1) ;
407 }
408 {
409     string gs1("Jag"); //gold
410     assert(gs1 >= "Aa") ;
411
412     mstring s1("Jag") ;
413     assert(s1 >= "Aa") ;
414 }
415 }
416
417 -----
418 relational operators
419 -----
420 void test_swap() {
421 {
422     string gs1("XYfg"); //gold
423     string gs2("Jag"); //gold
424
425     cout << "gs1 = " << gs1 << endl ;
426     cout << "gs2 = " << gs2 << endl ;
427
428     swap(gs1,gs2) ;
429
430     cout << "gs1 = " << gs1 << endl ;
431     cout << "gs2 = " << gs2 << endl ;
432
433     string s1("XYfg");
434     string s2("Jag");
435     swap(s1,s2) ;
436     assert_string_mstring_equal(gs1,s1) ;
437     assert_string_mstring_equal(gs2,s2) ;
438 }
439 }
440
441
442 -----
443 test bed
444 -----
445 void testbed() {
446     test_constructors() ;
447     test_element_access() ;
448     test_operator_plus_equal() ;
449     test_operator_plus() ;
450     test_equal_equal_operators();
451     test_not_equal_operators();
452     test_less_than_operators();
453     test_less_than_equal_operators();
454     test_greater_than_operators();
```

```
455     test_greater_than_equal_operators();  
456     test_swap() ;  
457 }  
458  
459 /*-----  
460 main  
461 -----*/  
462 int main() {  
463     testbed() ;  
464     return 0 ;  
465 }  
466  
467 //EOF  
468
```

VASUDEVAMURTHY

Chapter 8

Inheritance

8.1 Introduction

8.2 Why Inheritance?

8.2. WHY INHERITANCE?

Why use inheritance?

```
class employee {  
    char* name ;  
    char* ssn ;  
};  
  
class part_time_employee: public employee {  
    int how_many_hours;  
    int pay_per_hour ;  
};
```

The diagram illustrates the inheritance relationship between the base class `employee` and the derived class `part_time_employee`. The base class `employee` is represented by a box containing `name` and `ssn`. The derived class `part_time_employee` is also represented by a box, which includes the inherited attributes `name` and `ssn`, and its own added attributes `how_many_hours` and `pay_per_hour`. The `how_many_hours` and `pay_per_hour` fields are highlighted with a red border.

1. Create a class (derived class) from an existing class (base class)
2. Do not reinvent the wheel
3. `part_time_employee` is-a `employee`. Create is-a relationship
4. Derived class is a specialization of a base (parent) class
5. If a derived class is derived from one base, single inheritance
6. If a derived class is derived from two or more base, multiple inheritance

Figure 8.1: Need for inheritance

8.3 Need for keyword: protected

Access privileges

```

class base {
private:
    int _x ;
public:
    int y ;
    int getx() const {return _x ; }

    base() {
        _x = 10, y = 20 ;
    }
};

class derived: public base {
private:
    int _d ;
public:
    void f() {
        //_x = 100 ; //cannot access private member declared in class 'base'
        int x = getx() ; //OK can call public function
        y = 200 ; //OK. Can access public data
        _d = 6 ;
    }
};

```

//b._x = 89 ; //Only class can access private members, not the instances
derived d ;
d.y = 300 ; //OK. Can access public data
//d._d = 10 ; //Only class can access private members, not the instances

Problem:

- 1) base class private members/data CANNOT be accessed by derived class
- 2) Only way to access is through public set/get functions or making EVERY PRIVATE members/data PUBLIC.

Violation of DATA HIDING PRINCIPLE

Figure 8.2: Problem of accessing private data/methods by derived class

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: in1.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 base class  
11 -----*/  
12 class base {  
13 private:  
14     int _x ;  
15 public:  
16     int y ;  
17     int getx() const {return _x ;}  
18  
19     base() {  
20         _x = 10, y = 20 ;  
21     }  
22 };  
23  
24 /*-----  
25 derived class  
26 -----*/  
27 class derived: public base {  
28 private:  
29     int _d ;  
30 public:  
31     void f() {  
32         //_x = 100 ; //cannot access private member declared in class 'base'  
33         int x = getx() ; //OK can call public function  
34         y = 200 ; //OK. Can access public data  
35         _d = 6 ;  
36     }  
37 };  
38  
39 /*-----  
40 -----*/  
41  
42 int main() {  
43     base b ;  
44     //b._x = 89 ;// Only class can access private members, not the instances  
45     derived d ;  
46     d.y = 300 ; //OK. Can access public data  
47     //d._d = 10 ; //Only class can access private members, not the instances  
48     return 0 ;  
49 }  
50
```

Access privileges

```

class base {
private:
    int _x ;
protected:
    int _z;
public:
    int y ;
    int getx() const {return _x ; }
    base() {
        _x = 10, y = 20, _z = 87 ;
    }
};

class derived: public base {
private:
    int _d ;
public:
    void f() {
        //_x = 100 ; //cannot access private member declared in class 'base'
        int x = getx() ; //OK can call public function
        y = 200 ; //OK. Can access public data
        _z = 7 ; //OK. Can access protected base class data
        _d = 6 ;
    }
};

Solution: 1) base class protected members/data CAN be
accessed by derived class
2) Only derived class can access data/members of
protected fields of base class. For all others classes
and instances, protected is same as private

```

NO Violation of DATA HIDING PRINCIPLE

Figure 8.3: Protected same as public, only for the derived class

8.3. NEED FOR KEYWORD: PROTECTED

Access privileges summary

```
class base {  
private:  
    int _x ;  
protected:  
    int _z;  
public:  
    int y ;  
    int getx() const {return _x ; }  
    base()  
    {  
        _x = 10, y = 20, _z = 87 ;  
    }  
};
```

```
class derived: public base {  
private:  
    int _d ;  
public:  
    void f()  
    {  
        //_x = 100 ;  
        int x = getx() ;  
        y = 200 ;  
        _z = 7 ;  
        _d = 6 ;  
    }  
};
```

1. base private: ONLY base class can access
NO derived class from base class
or instances of base class can access
private members
2. base protected: ONLY base class and derived class
can access private data/members
protected base data/members are like
public data/members for ONLY
derived class.
Instances of derived classes CANNOT access
protected data/members. For all others, except,
derived class, protected is same as private
3. base public: Any body can access any data/members

Figure 8.4: Summary

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: in2.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 base class  
11 -----*/  
12 class base {  
13 private:  
14     int _x ;  
15 protected:  
16     int _z ;  
17 public:  
18     int y ;  
19     int getx() const {return _x ;}  
20     base() {  
21         _x = 10, y = 20, _z = 87 ;  
22     }  
23 };  
24  
25 /*-----  
26 derived class  
27 -----*/  
28 class derived: public base {  
29 private:  
30     int _d ;  
31 public:  
32     void f(){  
33         //_x = 100 ; //cannot access private member declared in class 'base'  
34         int x = getx(); //OK can call public function  
35         y = 200 ; //OK. Can access public data  
36         _z = 7 ; //OK. Can access protected base class data  
37         _d = 6 ;  
38     }  
39 };  
40  
41 /*-----  
42 -----*/  
43  
44 int main(){  
45     base b ;  
46     //b._x = 89 ;// Only class can access private members, not the instances  
47     //b._z = 5; //protected is same as private to ALL except derived class  
48     derived d ;  
49     d.y = 300 ; //OK. Can access public data  
50     //d._d = 10 ; //Only class can access private members, not the instances  
51     //d._z = 5 ; //protected is same as private to ALL except derived class  
52     return 0 ;  
53 }  
54
```

8.4. OVERRIDING OR HIDING BASE CLASS MEMBER FUNCTIONS

8.4 Overriding or hiding base class member functions

Overriding(hiding) base class member functions

```

class base {
public:
    void print() const { cout << "I am base print\n" ; }
    void print1() const {cout << "I am base print1\n" ; }
};

class derived: public base {
public:
    void print() const { cout << "I am derived print\n" ; }
};

int main() {
    base b ;
    derived d ;
    b.print() ;
    d.print() ;
    d.base::print() ;
    b.print1() ;
    d.print1() ;
    d.base::print1() ;
    base& ab = b ;
    derived& ad = d ;
    ab.print() ;
    ad.print() ;
    ad.base::print() ;
    ab.print1() ;
    ad.print1() ;
    ad.base::print1() ;
    base* bp = &(b) ;
    derived* dp = &(d) ;
    bp->print() ;
    dp->print() ;
    dp->base::print() ;
    bp->print1() ;
    dp->print1() ;
    dp->base::print1() ;
    return 0 ;
}

```

I am base print
 I am derived print
 I am base print
 I am base print1
 I am base print1
 I am base print1

 I am base print
 I am derived print
 I am base print
 I am base print1
 I am base print1
 I am base print1

 I am base print
 I am derived print
 I am base print
 I am base print1
 I am base print1
 I am base print1

1. A member function declaration in a derived class hides/overrides all inherited functions with the same name.
2. `d.base::` Using `<base_name>::` can be used to bring hidden names of base class into derived class.

Figure 8.5: hiding and bring back base class member functions

```
1 /*-----  
2 Copyright (c) 2010 Author: Jagadeesh Vasudevamurthy  
3 Filename: in4.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 using namespace std;  
8  
9 /*-----  
10 base class  
11 -----*/  
12 class base {  
13 public:  
14     void print() const { cout << "I am base print\n" ; }  
15     void print1() const {cout << "I am base print1\n" ; }  
16 };  
17  
18 /*-----  
19 derived class  
20 -----*/  
21 class derived: public base {  
22 public:  
23     void print() const { cout << "I am derived print\n" ; }  
24 };  
25  
26 /*-----  
27 I am base print  
28 I am derived print  
29 I am base print  
30 I am base print1  
31 I am base print1  
32 I am base print1  
33 I am base print  
34 I am derived print  
35 I am base print  
36 I am base print1  
37 I am base print1  
38 I am base print1  
39 I am base print  
40 I am derived print  
41 I am base print  
42 I am base print1  
43 I am base print1  
44 I am base print1  
45 -----*/  
46 int main() {  
47     base b ;  
48     derived d ;  
49     b.print() ;  
50     d.print() ;  
51     d.base::print() ;  
52     b.print1() ;  
53     d.print1() ;  
54     d.base::print1() ;  
55     base& ab = b ;
```

```
56 derived& ad = d ;
57 ab.print() ;
58 ad.print() ;
59 ad.base::print() ;
60 ab.print1() ;
61 ad.print1() ;
62 ad.base::print1() ;
63 base* bp = &(b) ;
64 derived* dp = &(d) ;
65 bp->print() ;
66 dp->print() ;
67 dp->base::print() ;
68 bp->print1() ;
69 dp->print1() ;
70 dp->base::print1() ;
71 return 0 ;
72 }
73
```

8.5. WRITING CONSTRUCTOR FOR A DERIVED CLASS

8.5 Writing constructor for a derived class

Writing constructor for base and derived class

```
class exam {  
public:  
    exam(const string& s, int m = 0, int f = 0):  
        _sname(s), _midterm(m), _final(f) {  
            cout << "exam constructor " << s << endl;  
    }  
private:  
    string _sname;  
    int _midterm;  
    int _final;  
};  
  
class cs32exam : public exam {  
public:  
    cs32exam(const string& s,const string& e, int m = 0, int f = 0,int p = 0) :  
        exam(s,m,f), _examname(e), _project(p) {  
            cout << "cs32 exam constructor " << e << endl;  
    }  
private:  
    string _examname;  
    int _project;  
};
```

`cs32exam p("sushma","cs32",65.75.85) ;`

NOTE THIS CALL: `exam(s,m,f)`

If you comment `exam(s,m,f)`
error C2512: 'exam' : no appropriate default constructor available

There can be many constructors for a class. That's why you need to call constructors on your own.

If the base class has only default constructor, example `exam()`, then you need NOT have to call `exam()`, explicitly in the derived class. See animal and dog examples, in future slides.

1. Raw memory allocated for derived class
2. Base class constructor is called first
3. Derived class data members are then initialized

Figure 8.6: Writing constructor for a derived class

8.6 Writing destructor for a derived class

Writing destructor for base and derived class

```

class exam {
public:
    ~exam() {
        cout << "exam destructor " << _sname << endl ;
    }
}

class cs32exam : public exam {
public:
    ~cs32exam() {
        cout << "cs32exam destructor " << _examname << endl ;
        /* Code here to delete cs32exam data members only */
    }
}

```

When derived class destructor is called for an object
base class destructor is called automatically. You need not
have to call base class destructor explicitly, like
calling base class constructor in the derived class.

This is because there is only one destructor

```

{
    cs32exam s("sushma","cs32",99,100,97) ;
}

cs32exam destructor cs32
1. Kills derived class data members
2. Then kills base class data members using base class destructor(~exam)

```

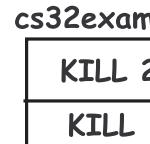


Figure 8.7: Writing destructor for a derived class

8.7. WRITING COPY CONSTRUCTOR AND EQUAL OPERATOR FOR A DERIVED CLASS IN A BAD WAY

8.7 Writing copy constructor and equal operator for a derived class in a BAD way

Why does the implicit copy constructor calls the base class copy constructor and the defined copy constructor doesn't?

```

class B {
public:
    B() :_b(100) {
        cout << "B constructor " << endl;
    }
    ~B() {
        cout << "B destructor " << endl;
    }
    B(const B& e) :_b(e._b){
        cout << "B copy constructor " << endl;
    }
    B& operator=(const B& e) {
        cout << "B = operator " << e._b << endl;
        if (&e != this) {
            _b = e._b;
        }
        return *this;
    }
    friend ostream& operator<<(ostream& o, const B& s) {
        o << s._b << " ";
        return o;
    }
    void setx(int x) {
        _b = x;
    }
private:
    int _b;
};

class D: public B {
public:
    D():_d(200) {
        cout << "D constructor " << endl;
    }
    ~D() {
        cout << "D destructor " << endl;
    }
    D(const D& e) :_d(e._d){
        cout << "D copy constructor " << endl;
    }
    D& operator=(const D& e) {
        cout << "D = operator " << e._d << endl;
        if (&e != this) {
            _d = e._d;
        }
        return *this;
    }
    friend ostream& operator<<(ostream& o, const D& s) {
        const B& b = s;
        o << b;
        o << s._d << " ";
        return o;
    }
private:
    int _d;
};

{-----1-----
cout << "----1----\n";
D d1;
cout << "----2----\n";
D d2(d1);
cout << "----3----\n";
cout << "d1 = " << d1 << endl;
cout << "d2 = " << d2 << endl;
d1.setx(-89);
cout << "d1 = " << d1 << endl;
cout << "----4----\n";
d2 = d1;
cout << "----5----\n";
cout << "d2 = " << d2 << endl;
}-----2-----
B constructor
D constructor
-----3-----
d1 = 100 200
d2 = 100 200
d1 = -89 200
-----4-----
D = operator 200 (DOES NOT CALL BASE =)
-----5-----
d2 = 100 200 (You want -89 200 ?)
D destructor
B destructor
D destructor
B destructor
-----
```

Figure 8.8: Writing copy constructor and equal operator for a derived class, BAD way

8.8 Writing copy constructor and equal operator for a derived class in a CORRECT way

```
1 /*-
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy
3 Filename: in8.cpp
4 -----*/
5
6 #include <iostream>
7 #include <cassert>
8 #include <string>
9 using namespace std;
10
11 #ifndef _WIN32
12 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector
13 #endif
14
15 /*
16 class B
17 -----*/
18 class B {
19 public:
20     B() :_b(100) {
21         cout << "B constructor " << endl;
22     }
23     ~B() {
24         cout << "B destructor " << endl;
25     }
26     B(const B& e) :_b(e._b){
27         cout << "B copy constructor " << endl;
28     }
29     B& operator=(const B& e) {
30         cout << "B = operator " << e._b << endl;
31         if (&e != this) {
32             _b = e._b;
33         }
34         return *this;
35     }
36     friend ostream& operator<<(ostream& o, const B& s) {
37         o << s._b << " ";
38         return o;
39     }
40     void setx(int x) {
41         _b = x;
42     }
43 private:
44     int _b;
45 };
46
47 /*
48 class D
49 -----*/
50 class D: public B {
51 public:
52     D():_d(200) {
53         cout << "D constructor " << endl;
54     }
55     ~D() {
56         cout << "D destructor " << endl;
57     }
58     D(const D& e) :_d(e._d){
59         cout << "D copy constructor " << endl;
60     }
61     D& operator=(const D& e) {
62         cout << "D = operator " << e._d << endl;
63         if (&e != this) {
64             _d = e._d;
65         }
66     }
}
```

```
67     return *this;
68 }
69 friend ostream& operator<<(ostream& o, const D& s) {
70     const B& b = s;
71     o << b;
72     o << s._d << " ";
73     return o;
74 }
75 private:
76     int _d;
77 };
78
79
80 /*-----
81 test bed
82 -----*/
83 int main() {
84 {
85     B b1;
86     B b2(b1);
87     b2 = b1;
88     cout << "b2 = " << b2 << endl;
89 }
90 {
91     cout << "----1----\n";
92     D d1;
93     cout << "----2----\n";
94     D d2(d1);
95     cout << "----3----\n";
96     cout << "d1 = " << d1 << endl;
97     cout << "d2 = " << d2 << endl;
98     d1.setx(-89);
99     cout << "d1 = " << d1 << endl;
100    cout << "----4----\n";
101    d2 = d1;
102    cout << "----5----\n";
103    cout << "d2 = " << d2 << endl;
104 }
105 return 0;
106 }
107 }
```

CHAPTER 8. INHERITANCE

Why does the implicit copy constructor calls the base class copy constructor and the defined copy constructor doesn't?

All base child constructors call the parent default constructor. This is how the standard is defined.
If you wanted the base class D to call B's copy constructor you have to explicitly ask for it

```
class B {
public:
    B() :_b(100) {
        cout << "B constructor " << endl;
    }
    ~B() {
        cout << "B destructor " << endl;
    }
    B(const B& e) :_b(e._b){
        cout << "B copy constructor " << endl;
    }
    B& operator=(const B& e) {
        cout << "B = operator " << e._b << endl;
        if (&e != this) {
            _b = e._b;
        }
        return *this;
    }
    friend ostream& operator<<(ostream& o, const B& s) {
        o << s._b << " ";
        return o;
    }
    void setx(int x) {
        _b = x;
    }
private:
    int _b;
};
```

```
class D: public B {
public:
    D():_d(200) {
        cout << "D constructor " << endl;
    }
    ~D() {
        cout << "D destructor " << endl;
    }
    D(const D& e) :B(e)/*NOTE 1*/,_d(e._d){
        cout << "D copy constructor " << endl;
    }
    D& operator=(const D& e) {
        cout << "D = operator " << e._d << endl;
        if (&e != this) {
            B::operator=(e); //NOTE 2
            _d = e._d;
        }
        return *this;
    }
    friend ostream& operator<<(ostream& o, const D& s) {
        const B& b = s;
        o << b;
        o << s._d << " ";
        return o;
    }
private:
    int _d;
};
```

```
{
    cout << "----1----\n";
    D d1;
    cout << "----2----\n";
    D d2(d1);
    cout << "----3----\n";
    cout << "d1 = " << d1 << endl;
    cout << "d2 = " << d2 << endl;
    d1.setx(-89);
    cout << "d1 = " << d1 << endl;
    cout << "----4----\n";
    d2 = d1;
    cout << "----5----\n";
    cout << "d2 = " << d2 << endl;
}
```

-----1-----
 B constructor
 D constructor
 -----2-----
 B copy constructor (CALLS BASE COPY)
 D copy constructor
 -----3-----
 d1 = 100 200
 d2 = 100 200
 d1 = -89 200
 -----4-----
 D = operator 200
 B = operator -89(CALLS BASE = operator)
 -----5-----
 d2 = -89 200 (CORRECT)

Figure 8.9: Writing copy constructor and equal operator for a derived class, CORRECT way

```
1 /*-
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy
3 Filename: in9.cpp
4 -----*/
5
6 #include <iostream>
7 #include <cassert>
8 #include <string>
9 using namespace std;
10
11 #ifndef _WIN32
12 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector
13 #endif
14
15 /*
16 class B
17 -----*/
18 class B {
19 public:
20     B() :_b(100) {
21         cout << "B constructor " << endl;
22     }
23     ~B() {
24         cout << "B destructor " << endl;
25     }
26     B(const B& e) :_b(e._b){
27         cout << "B copy constructor " << endl;
28     }
29     B& operator=(const B& e) {
30         cout << "B = operator " << e._b << endl;
31         if (&e != this) {
32             _b = e._b;
33         }
34         return *this;
35     }
36     friend ostream& operator<<(ostream& o, const B& s) {
37         o << s._b << " ";
38         return o;
39     }
40     void setx(int x) {
41         _b = x;
42     }
43 private:
44     int _b;
45 };
46
47 */
48 /*
49 class D
50 -----*/
51 class D: public B {
52 public:
53     D():_d(200) {
54         cout << "D constructor " << endl;
55     }
56     ~D() {
57         cout << "D destructor " << endl;
58     }
59     D(const D& e) :B(e)/*NOTE THIS CALL*/,_d(e._d){
60         cout << "D copy constructor " << endl;
61     }
62     D& operator=(const D& e) {
63         cout << "D = operator " << e._d << endl;
64         if (&e != this) {
65             B::operator=(e); //Note this call.base of e (RHS) is copied to base of this object(on LHS)
66             _d = e._d;
```

```
67     }
68     return *this;
69 }
70 friend ostream& operator<<(ostream& o, const D& s) {
71     const B& b = s;
72     o << b;
73     o << s._d << " ";
74     return o;
75 }
76 private:
77     int _d;
78 };
79
80
81 /*-----
82 test bed
83 -----*/
84 int main() {
85 {
86     B b1;
87     B b2(b1);
88     b2 = b1;
89     cout << "b2 = " << b2 << endl;
90 }
91 {
92     cout << "----1----\n";
93     D d1;
94     cout << "----2----\n";
95     D d2(d1);
96     cout << "----3----\n";
97     cout << "d1 = " << d1 << endl;
98     cout << "d2 = " << d2 << endl;
99     d1.setx(-89);
100    cout << "d1 = " << d1 << endl;
101    cout << "----4----\n";
102    d2 = d1;
103    cout << "----5----\n";
104    cout << "d2 = " << d2 << endl;
105 }
106 return 0;
107 }
108 }
```

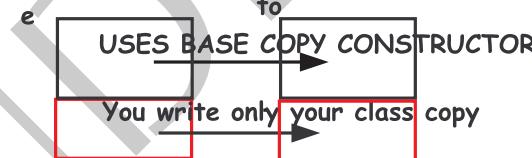
8.9. WRITING COPY CONSTRUCTOR FOR A DERIVED CLASS

8.9 Writing copy constructor for a derived class

Writing copy constructor for base and derived class

```
class exam {  
public:  
    exam(const exam& e):  
        _sname(e._sname), _midterm(e._midterm), _final(e._final) {  
            cout << "exam copy constructor " << e._sname << endl ;  
        }  
private:  
    string _sname ;  
    int _midterm ;  
    int _final ;  
};  
  
class cs32exam : public exam {  
public:  
    cs32exam(const cs32exam& e):  
        exam(e),_examname(e._examname),_project(e._project) {  
            cout << "cs32exam copy constructor " << e._examname << endl ;  
        }  
private:  
    string _examname ;  
    int _project ;  
};
```

NOTE THIS CALL: exam(e)



If you comment `exam(e)`
error C2512: 'exam' : no appropriate default constructor available

Figure 8.10: Writing copy constructor for a derived class

8.10 Writing equal operator for a derived class

Writing equal operator for base and derived class

```

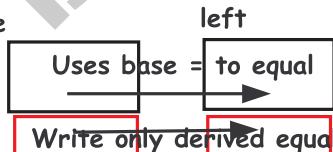
class exam {
public:
    exam& operator=(const exam& e) {
        cout << "exam = operator " << e._sname << endl;
        if (&e != this) {
            _sname = e._sname ;
            _midterm = e._midterm ;
            _final = e._final ;
        }
        return *this ;
    }
}

class cs32exam : public exam {
public:
    cs32exam& operator=(const cs32exam& e) {
        if (&e != this) {
            cout << "cs32exam = operator " << e._examname << endl ;
            exam::operator=(e) ; //Note this call
            //base of e (RHS) is copied to base of this object(on LHS)
            _examname = e._examname ;
            _project = e._project;
        }
        return *this ;
    }
}

```

Without `exam::operator=`
`cs32exam operator=` will be called.
INFINITE RECURSION

NOTE THIS CALL: `exam::operator=(e)`



if you comment `operator=(e)`, base of `e` will NOT be copied to base of `left`
Only derived portion of `e` will be copied to the left derived portion

Figure 8.11: Writing equal operator for a derived class

8.11. WRITING INSERTION OPERATOR FOR A DERIVED CLASS

8.11 Writing insertion operator for a derived class

Writing << operator for base and derived class

```
class exam {  
public:  
    friend ostream& operator<<(ostream& o, const exam& s) {  
        o << s._sname << " " << s._midterm << " " << s._final << " " ;  
        return o ;  
    }  
}  
  
class cs32exam : public exam {  
public:  
    friend ostream& operator<<(ostream& o, const cs32exam& s) {  
        const exam& e = s ;  
        o << e ;  
        o << s._examname << " " << s._project << " " ;  
        return o ;  
    }  
}
```

NOTE THIS CALL: o << e

Figure 8.12: Writing insertion operator for a derived class

8.12 Using base and derived class objects

1. Using base and derived class

```

void test1() {
    exam s("sushma",52,80) ; exam constructor sushma
    cout << s << endl ; sushma 52 80

    exam s1(s) ; exam copy constructor sushma
    cout << s1 << endl ; sushma 52 80

    exam s2("neema",89,78) ; exam constructor neema
    cout << s2 << endl ; neema 89 78

    s1 = s2 ;
    cout << s1 << endl ;
    s1.who_am_i() ;
}

```

exam = operator neema
 neema 89 78
 I am exam class neema

exam destructor neema (for s2)
 exam destructor neema (for s1)
 exam destructor sushma (for s)

Figure 8.13: Using only base

2. Using base and derived class

```

void test2() {
    cs32exam s("sushma","cs32",99,100,97) ;
    cout << s << endl ;

    cs32exam s1(s) ;
    cout << s1 << endl ;

    cs32exam s2("Bill","cs32",56,77,88) ;
    cout << s2 << endl ;

    s1 = s2 ;
    cout << s1 << endl ;

    s1.who_am_i() ;
    s1.exam::who_am_i() ;
}

```

exam constructor sushma
 cs32 exam constructor cs32
 sushma 99 100 cs32 97

exam copy constructor sushma
 cs32exam copy constructor cs32
 sushma 99 100 cs32 97

exam constructor Bill
 cs32 exam constructor cs32
 Bill 56 77 cs32 88

cs32exam = operator cs32
 exam = operator Bill
 Bill 56 77 cs32 88

I am cs32exam class cs32
 I am exam class Bill

cs32exam destructor cs32 (s2)
 exam destructor Bill (s2)
 cs32exam destructor cs32 (s1)
 exam destructor Bill (s1)
 cs32exam destructor cs32 (s)
 exam destructor sushma (s)

Figure 8.14: Using only derived

3. Using base and derived class

```
void test3() {
    exam* b = new exam("sushma",52,80);
    cout << *b << endl;

    cs32exam* d = new cs32exam
        ("sushma","cs32",99,100,97);
    cout << *d << endl;

    b->who_am_i();
    d->who_am_i();
    d->exam::who_am_i();

    delete b;
    delete d;
}
```

exam constructor sushma
sushma 52 80

exam constructor sushma
cs32 exam constructor cs32
sushma 99 100 cs32 97

I am exam class sushma
I am cs32exam class cs32
I am exam class sushma
exam destructor sushma

cs32exam destructor cs32
exam destructor sushma

Figure 8.15: Using derived objects on heap

4. Using base and derived class

```
void test4() {
    exam* b = new exam("sushma",52,80); exam constructor sushma
    cout << *b << endl; sushma 52 80

    exam* d = new cs32exam
        ("sushma","cs32",99,100,97); exam constructor sushma
                                         cs32 exam constructor cs32

    cout << *d << endl; /* static binding */ *sushma 99 100
                           (why it did not print cs32 and 97?)

    b->who_am_i();
    d->who_am_i();

    d->exam::who_am_i();

    delete b;
    delete d;
}
```

I am exam class sushma
I am exam class sushma
I am exam class sushma
exam destructor sushma
exam destructor sushma
(Why destructor of cs32exam is not called)

Problem of static binding. Pointer d is thought of
base exam class, although it is pointing to derived cs32exam class

Figure 8.16: Problems of static binding

5. Using base and derived class

```

void test5() {
    exam bi("sushma",52,80) ;
    cs32exam di("sushma","cs32",99,100,97)

    exam* b = &bi ;
    cout << *b << endl ;
    exam* d = &di ; /* static binding */
    cout << *d << endl ;

    b->who_am_i() ;
    d->who_am_i() ;
    d->exam::who_am_i() ;
}

```

exam constructor sushma
exam constructor sushma
cs32 exam constructor cs32

sushma 52 80
sushma 99 100

I am exam class sushma
I am exam class sushma
I am exam class sushma

cs32exam destructor cs32(di)
exam destructor sushma(di)
exam destructor sushma(bi)

Problem of static binding. Pointer d is thought of base exam class, although it is pointing to derived cs32exam class

Note that: when di went out of scope both cs32exam and exam destructor is called.

Figure 8.17: Problems of static binding

6. Using base and derived class

```

void test6() {
    exam bi("sushma",52,80) ;
    cs32exam di("sushma","cs32",99,100,97)

    exam& b = bi ;
    cout << b << endl ;
    exam& d = di ; /* static binding */
    cout << d << endl ;

    b.who_am_i() ;
    d.who_am_i() ;
    d.exam::who_am_i() ;
}

```

exam constructor sushma
exam constructor sushma
cs32 exam constructor cs32

sushma 52 80
sushma 99 100

I am exam class sushma
I am exam class sushma
I am exam class sushma

cs32exam destructor cs32(di)
exam destructor sushma(di)
exam destructor sushma(bi)

Problem of static binding. Alias object d is thought of base exam class, although it is an object of the derived cs32exam class

Note that: when di went out of scope both cs32exam and exam destructor is called.

Figure 8.18: Problems of static binding

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename: in7.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 #include <cassert>  
8 #include <string>  
9 using namespace std;  
10  
11 /*-----  
12 class exam  
13 -----*/  
14 class exam {  
15 public:  
16     exam(const string& s, int m = 0, int f = 0):_sname(s),_midterm(m),_final(f) {  
17         cout << "exam constructor " << s << endl ;  
18     }  
19     ~exam(){  
20         cout << "exam destructor " << _sname << endl ;  
21     }  
22     exam(const exam& e):_sname(e._sname), _midterm(e._midterm),_final(e._final) {  
23         cout << "exam copy constructor " << e._sname << endl ;  
24     }  
25     exam& operator=(const exam& e){  
26         cout << "exam = operator " << e._sname << endl;  
27         if (&e != this){  
28             _sname = e._sname ;  
29             _midterm = e._midterm ;  
30             _final = e._final ;  
31         }  
32         return *this ;  
33     }  
34     friend ostream& operator<<(ostream& o, const exam& s){  
35         o << s._sname << " " << s._midterm << " " << s._final << " "  
36         return o ;  
37     }  
38     void who_am_i() const {  
39         cout << "I am exam class " << _sname << endl ;  
40     }  
41 private:  
42     string _sname ;  
43     int _midterm ;  
44     int _final ;  
45 };  
46  
47 /*-----  
48 cs32 exam  
49 -----*/  
50 class cs32exam : public exam {  
51 public:  
52     cs32exam(const string& s,const string& e, int m = 0, int f = 0,int p = 0):  
53         exam(s,m,f)/*Note this call*/,_examname(e),_project(p){  
54         cout << "cs32 exam constructor " << e << endl ;  
55     }
```

```
56     ~cs32exam() {
57         cout << "cs32exam destructor " << _examname << endl ;
58     }
59     cs32exam(const cs32exam& e):exam(e) /*NOTE THIS CALL */,_examname(e._examname),_project(e._project) {
60         cout << "cs32exam copy constructor " << e._examname << endl ;
61     }
62     cs32exam& operator=(const cs32exam& e) {
63         if (&e != this) {
64             cout << "cs32exam = operator " << e._examname << endl ;
65             exam::operator=(e); //Note this call. base of e (RHS) is copied to base of this object(on LHS)
66             _examname = e._examname ;
67             _project = e._project;
68         }
69         return *this ;
70     }
71     friend ostream& operator<<(ostream& o, const cs32exam& s) {
72         const exam& e = s ;
73         o << e ;
74         o << s._examname << " " << s._project << " ";
75         return o ;
76     }
77     void who_am_i() const {
78         cout << "I am cs32exam class " << _examname << endl ;
79     }
80 private:
81     string _examname ;
82     int _project ;
83 };
84
85
86 /*****
87 -----test1-----
88 exam constructor sushma
89 sushma 52 80
90 exam copy constructor sushma
91 sushma 52 80
92 exam constructor neema
93 neema 89 78
94 exam = operator neema
95 neema 89 78
96 I am exam class neema
97 exam destructor neema
98 exam destructor neema
99 exam destructor sushma
100 ****/
101 void test1() {
102     cout << "-----test1-----\n" ;
103     exam s("sushma",52,80);
104     cout << s << endl ;
105     exam s1(s);
106     cout << s1 << endl ;
107     exam s2("neema",89,78);
108     cout << s2 << endl ;
109     s1 = s2 ;
110     cout << s1 << endl ;
```

```
111 s1.who_am_i();
112 }
113
114 /*-----
115 -----test2-----
116 exam constructor sushma
117 cs32 exam constructor cs32
118 sushma 99 100 cs32 97
119 exam copy constructor sushma
120 cs32exam copy constructor cs32
121 sushma 99 100 cs32 97
122 exam constructor Bill
123 cs32 exam constructor cs32
124 Bill 56 77 cs32 88
125 cs32exam = operator cs32
126 exam = operator Bill
127 Bill 56 77 cs32 88
128 I am cs32exam class cs32
129 I am exam class Bill
130 cs32exam destructor cs32
131 exam destructor Bill
132 cs32exam destructor cs32
133 exam destructor Bill
134 cs32exam destructor cs32
135 exam destructor sushma
136 -----*/
137 void test2() {
138     cout << "-----test2-----\n";
139     cs32exam s("sushma","cs32",99,100,97);
140     cout << s << endl ;
141     cs32exam s1(s);
142     cout << s1 << endl ;
143     cs32exam s2("Bill","cs32",56,77,88);
144     cout << s2 << endl ;
145     s1 = s2;
146     cout << s1 << endl ;
147     s1.who_am_i();
148     s1.exam::who_am_i();
149 }
150
151 /*-----
152 -----test3-----
153 exam constructor sushma
154 sushma 52 80
155 exam constructor sushma
156 cs32 exam constructor cs32
157 sushma 99 100 cs32 97
158 I am exam class sushma
159 I am cs32exam class cs32
160 I am exam class sushma
161 exam destructor sushma
162 cs32exam destructor cs32
163 exam destructor sushma
164 -----*/
165 void test3(){
```

```
166 cout << "-----test3-----\n";
167 exam* b = new exam("sushma",52,80);
168 cout << *b << endl;
169 cs32exam* d = new cs32exam("sushma","cs32",99,100,97);
170 cout << *d << endl;
171 b->who_am_i();
172 d->who_am_i();
173 d->exam::who_am_i();
174 delete b;
175 delete d;
176 }
177
178 /*-----test4-----
179 exam constructor sushma
180 sushma 52 80
181 exam constructor sushma
182 cs32 exam constructor cs32
183 sushma 99 100 <-----WHY
184 I am exam class sushma
185 I am exam class sushma <----- WHY
186 I am exam class sushma
187 I am exam class sushma
188 exam destructor sushma
189 exam destructor sushma <-----WHY
190 */
191 void test4(){
192 cout << "-----test4-----\n";
193 exam* b = new exam("sushma",52,80);
194 cout << *b << endl;
195 exam* d = new cs32exam("sushma","cs32",99,100,97);
196 cout << *d << endl; /* static binding */
197 b->who_am_i();
198 d->who_am_i();
199 d->exam::who_am_i();
200 delete b;
201 delete d;
202 }
203
204 /*-----test5-----
205 exam constructor sushma
206 sushma 52 80
207 exam constructor sushma
208 cs32 exam constructor cs32
209 sushma 99 100 <-----WHY
210 I am exam class sushma
211 I am exam class sushma <-----WHY
212 I am exam class sushma
213 I am exam class sushma
214 cs32exam destructor cs32
215 exam destructor sushma
216 exam destructor sushma
217 */
218 void test5(){
219 cout << "-----test5-----\n";
220 exam bi("sushma",52,80);
```

```
221 cs32exam di("sushma","cs32",99,100,97);
222 exam* b = &bi;
223 cout << *b << endl;
224 exam* d = &di; /* static binding */
225 cout << *d << endl;
226 b->who_am_i();
227 d->who_am_i();
228 d->exam::who_am_i();
229 }
230
231 /*-----test6-----
232 -----test6-----
233 exam constructor sushma
234 exam constructor sushma
235 cs32 exam constructor cs32
236 sushma 52 80
237 sushma 99 100 <-----WHY
238 I am exam class sushma
239 I am exam class sushma <-----WHY
240 I am exam class sushma
241 cs32exam destructor cs32
242 exam destructor sushma
243 exam destructor sushma
244 */
245 void test6(){
246 cout << "-----test6-----\n";
247 exam bi("sushma",52,80);
248 cs32exam di("sushma","cs32",99,100,97);
249 exam& b = bi;
250 cout << b << endl;
251 exam& d = di; /* static binding */
252 cout << d << endl;
253 b.who_am_i();
254 d.who_am_i();
255 d.exam::who_am_i();
256 }
257
258 /*-----main
259 main
260 */
261 int main(){
262 test1();
263 test2();
264 test3();
265 test4();
266 test5();
267 test6();
268 return 0;
269 }
270
```

8.13 Problems of static binding and need for polymorphism

8.13. PROBLEMS OF STATIC BINDING AND NEED FOR POLYMORPHISM

Problem of static binding and need for Polymorphism

```

class animal {
public:
    animal() {cout << "animal constructor\n";}
    void who_am_i() { cout << "I am an animal\n" ; }
    ~animal() {cout << "animal destructor\n";}
};

class dog:public animal {
public:
    dog() {cout << "dog constructor\n";}
    void who_am_i() { cout << "I am a dog\n" ; }
    ~dog() {cout << "dog destructor\n";}
};

class cat:public animal {
public:
    cat() {cout << "cat constructor\n";}
    void who_am_i() { cout << "I am a cat\n" ; }
    ~cat() {cout << "cat destructor\n";}
};

class lion:public animal {
public:
    lion() {cout << "lion constructor\n";}
    void who_am_i() { cout << "I am lion\n" ; }
    ~lion() {cout << "lion destructor\n";}
};

```

Static binding. Every object is thought as animal

```

void object_without_polymorphism() {
    dog d ;
    cat c ;
    lion n ;
    cat c1 ;
}

```

```

animal* a[4] ;
a[0] = &d ;
a[1] = &c ;
a[2] = &n ;
a[3] = &c1 ;
for (int i = 0; i < 4; i++) {
    a[i]->who_am_i();
}

```

Correct because objects are destroyed

```

I am an animal
I am an animal
I am an animal
I am an animal
cat destructor
animal destructor
lion destructor
animal destructor
cat destructor
animal destructor
dog destructor
animal destructor

```

Figure 8.19: Problems of static binding with objects

Problem of static binding and need for Polymorphism

```

class animal {
public:
    animal() {cout << "animal constructor\n";}
    void who_am_i() { cout << "I am an animal\n" ; }
    ~animal() {cout << "animal destructor\n";}
};

class dog:public animal {
public:
    dog() {cout << "dog constructor\n";}
    void who_am_i() { cout << "I am a dog\n" ; }
    ~dog() {cout << "dog destructor\n";}
};

class cat:public animal {
public:
    cat() {cout << "cat constructor\n";}
    void who_am_i() { cout << "I am a cat\n" ; }
    ~cat() {cout << "cat destructor\n";}
};

class lion:public animal {
public:
    lion() {cout << "lion constructor\n";}
    void who_am_i() { cout << "I am lion\n" ; }
    ~lion() {cout << "lion destructor\n";}
};

void pointer_without_polymorphism() {
    animal* a[4] ;
    a[0] = new dog() ;
    a[1] = new cat() ;
    a[2] = new lion() ;
    a[3] = new cat() ;
}

animal constructor
dog constructor
animal constructor
cat constructor
animal constructor
lion constructor
animal constructor
cat constructor

```

I am an animal
 animal destructor
 animal destructor
 animal destructor
 animal destructor

Static binding. Every object is thought as animal
 NOT} Correct because objects are thought as animals

Figure 8.20: Problems of static binding with pointers to objects

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevanurthy  
3 Filename: need_for_poly.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 #include <cassert>  
8 #include <string>  
9 using namespace std;  
10  
11 /*-----  
12      Polymorphism  
13 -----*/  
14 class animal {  
15 public:  
16     animal() {cout << "animal constructor\n";}  
17     void who_am_i() { cout << "I am an animal\n" ; }  
18     ~animal() {cout << "animal destructor\n";}  
19 };  
20  
21 class dog:public animal {  
22 public:  
23     dog() {cout << "dog constructor\n";}  
24     void who_am_i() { cout << "I am a dog\n" ; }  
25     ~dog() {cout << "dog destructor\n";}  
26 };  
27  
28 class cat:public animal {  
29 public:  
30     cat() {cout << "cat constructor\n";}  
31     void who_am_i() { cout << "I am a cat\n" ; }  
32     ~cat() {cout << "cat destructor\n";}  
33 };  
34  
35 class lion:public animal {  
36 public:  
37     lion() {cout << "lion constructor\n";}  
38     void who_am_i() { cout << "I am lion\n" ; }  
39     ~lion() {cout << "lion destructor\n";}  
40 };  
41  
42 /*-----  
43 animal constructor  
44 dog constructor  
45 animal constructor  
46 cat constructor  
47 animal constructor  
48 lion constructor  
49 animal constructor  
50 cat constructor  
51 I am an animal  
52 I am an animal  
53 I am an animal  
54 I am an animal  
55 cat destructor
```

```
56 animal destructor
57 lion destructor
58 animal destructor
59 cat destructor
60 animal destructor
61 dog destructor
62 animal destructor
63 -----
64 void object_without_polymorphism() {
65     dog d;
66     cat c;
67     lion n;
68     cat c1;
69     animal* a[4];
70     a[0] = &d;
71     a[1] = &c;
72     a[2] = &n;
73     a[3] = &c1;
74     for (int i = 0; i < 4; i++) {
75         a[i]->who_am_i();
76     }
77 }
78 /*
79 -----
80 animal constructor
81 dog constructor
82 animal constructor
83 cat constructor
84 animal constructor
85 lion constructor
86 animal constructor
87 cat constructor
88 I am an animal
89 I am an animal
90 I am an animal
91 I am an animal
92 animal destructor
93 animal destructor
94 animal destructor
95 animal destructor
96 -----
97 void pointer_without_polymorphism() {
98     animal* a[4];
99     a[0] = new dog();
100    a[1] = new cat();
101    a[2] = new lion();
102    a[3] = new cat();
103    for (int i = 0; i < 4; i++) {
104        a[i]->who_am_i();
105    }
106    for (int i = 0; i < 4; i++) {
107        delete a[i];
108    }
109 }
```

```
111 /*-----  
112     main  
113 -----*/  
114 void main() {  
115     object_without_polymorphism();  
116     pointer_without_polymorphism();  
117 }  
118  
119
```

8.14 Polymorphism through keyword virtual

Dynamic binding and Polymorphism

```

class animal {
public:
    animal() {cout << "animal constructor\n";}
    virtual void who_am_i() { cout << "I am an animal\n" ; }
    virtual ~animal() {cout << "animal destructor\n";}
};

class dog:public animal {
public:
    dog() {cout << "dog constructor\n";}
    void who_am_i() { cout << "I am a dog\n" ; }
    ~dog() {cout << "dog destructor\n";}
};

class cat:public animal {
public:
    cat() {cout << "cat constructor\n";}
    void who_am_i() { cout << "I am a cat\n" ; }
    ~cat() {cout << "cat destructor\n";}
};

class lion:public animal {
public:
    lion() {cout << "lion constructor\n";}
    void who_am_i() { cout << "I am lion\n" ; }
    ~lion() {cout << "lion destructor\n";}
};

```

```

void object_polymorphism() {
    dog d ;
    cat c ;
    lion n ;
    cat c1 ;

    animal* a[4] ;
    a[0] = &d ;
    a[1] = &c ;
    a[2] = &n ;
    a[3] = &c1 ;
    for (int i = 0; i < 4; i++) {
        a[i]->who_am_i() ;
    }
}

```

Dynamic binding. Every object is NOT thought as just animal. It exactly knows who it is

animal constructor
dog constructor
animal constructor
cat constructor
animal constructor
lion constructor
animal constructor
cat constructor

I am a dog
I am a cat
I am lion
I am a cat

cat destructor
animal destructor
lion destructor
animal destructor
cat destructor
animal destructor
dog destructor
animal destructor

Figure 8.21: Dynamic binding of objects

Dynamic binding and Polymorphism

```

class animal {
public:
    animal() {cout << "animal constructor\n";}
    virtual void who_am_i() { cout << "I am an animal\n" ; }
    virtual ~animal() {cout << "animal destructor\n";}
};

class dog:public animal {
public:
    dog() {cout << "dog constructor\n";}
    void who_am_i() { cout << "I am a dog\n" ; }
    ~dog() {cout << "dog destructor\n";}
};

class cat:public animal {
public:
    cat() {cout << "cat constructor\n";}
    void who_am_i() { cout << "I am a cat\n" ; }
    ~cat() {cout << "cat destructor\n";}
};

class lion:public animal {
public:
    lion() {cout << "lion constructor\n";}
    void who_am_i() { cout << "I am lion\n" ; }
    ~lion() {cout << "lion destructor\n";}
};

```

void pointer_polymorphism() {
 animal* a[4] ;
 a[0] = new dog() ; animal constructor
 a[1] = new cat() ; dog constructor
 a[2] = new lion() ; animal constructor
 a[3] = new cat() ; cat constructor
 for (int i = 0; i < 4; i++) {
 a[i]->who_am_i() ;
 I am a dog
 I am a cat
 I am lion
 I am a cat
 }
 for (int i = 0; i < 4; i++) {
 delete a[i] ;
 }
}

Dynamic binding. Every object is NOT thought as just animal. It exactly knows who it is

Note that both base and derived destructors are called because base destructor is virtual
 virtual ~animal() {cout << "animal destructor\n";}

DREAM **animal* a[4]** which stores dog, cats and lion
 Heterogenous container

Figure 8.22: Dynamic binding of pointers to objects

```
1 /*-----  
2 Copyright (c) 2009 Author: Jagadeesh Vasudevamurthy  
3 Filename: poly.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 #include <cassert>  
8 #include <string>  
9 using namespace std;  
10  
11 /*-----  
12      Polymorphism  
13 -----*/  
14 class animal {  
15 public:  
16     animal() {cout << "animal constructor\n";}  
17     virtual void who_am_i() { cout << "I am an animal\n" ; }  
18     virtual ~animal() {cout << "animal destructor\n";}  
19 };  
20  
21 class dog:public animal {  
22 public:  
23     dog() {cout << "dog constructor\n";}  
24     void who_am_i() { cout << "I am a dog\n" ; }  
25     ~dog() {cout << "dog destructor\n";}  
26 };  
27  
28 class cat:public animal {  
29 public:  
30     cat() {cout << "cat constructor\n";}  
31     void who_am_i() { cout << "I am a cat\n" ; }  
32     ~cat() {cout << "cat destructor\n";}  
33 };  
34  
35 class lion:public animal {  
36 public:  
37     lion() {cout << "lion constructor\n";}  
38     void who_am_i() { cout << "I am lion\n" ; }  
39     ~lion() {cout << "lion destructor\n";}  
40 };  
41  
42 /*-----  
43 -----*/  
44 void object_polymorphism(){  
45     dog d ;  
46     cat c ;  
47     lion n ;  
48     cat c1 ;  
49     animal* a[4] ;  
50     a[0] = &d ;  
51     a[1] = &c ;  
52     a[2] = &n ;  
53     a[3] = &c1 ;  
54     for (int i = 0; i < 4; i++) {
```

```
56     a[i]->who_am_i();
57 }
58 }
59
60 /*-----*
61 -----*/
62 void pointer_polymorphism() {
63     animal* a[4];
64     a[0] = new dog();
65     a[1] = new cat();
66     a[2] = new lion();
67     a[3] = new cat();
68     for (int i = 0; i < 4; i++) {
69         a[i]->who_am_i();
70     }
71     for (int i = 0; i < 4; i++) {
72         delete a[i];
73     }
74 }
75 }
76
77 /*-----*
78     main
79 -----*/
80 void main() {
81     object_polymorphism();
82     pointer_polymorphism();
83 }
84
```

8.15 Rewriting exam program using polymorphism

4. Problem of static binding corrected

1. virtual ~exam() { }
2. virtual ostream& print(ostream& o) const {}
3. virtual void who_am_i() const {}
4. virtual ostream& print(ostream& o) const {}

```
void test4() {
    exam* b = new exam("sushma", 52, 80);
    cout << *b << endl;
    exam* d = new cs32exam("sushma", "cs32", 99, 100, 97);
    cout << *d << endl; /* Dynamic binding */
    d->print(cout); cout << endl; /* dynamic binding */
    b->who_am_i();                                Dynamic binding
    d->who_am_i();                                Dynamic binding
    d->exam::who_am_i();                          Dynamic binding
    delete b;
    delete d;
}
```

Dynamic binding

exam constructor sushma
sushma 52 80
exam constructor sushma
cs32 exam constructor cs32
sushma 99 100 cs32 97
sushma 99 100 cs32 97
I am exam class sushma
I am cs32exam class cs32
I am exam class sushma
exam destructor sushma
cs32exam destructor cs32
exam destructor sushma

```
friend ostream& operator<<(ostream& o, const exam& s) {
    s.print(o);
    return o;
}
virtual ostream& print(ostream& o) const {
    o << _sname << " " << _midterm << " " << _final << " ";
    return o;
}
```

exam class

Because of virtual

```
friend ostream& operator<<(ostream& o, const cs32exam& s) {
    s.print(o);
}
virtual ostream& print(ostream& o) const {
    this->exam::print(o);
    o << _examname << " " << _project << " ";
    return o;
}
```

cs32exam class

Figure 8.23: Problems of static binding fixed

5. Problem of static binding corrected

```

1. virtual ~exam() { }
2. virtual ostream& print(ostream& o) const {}
3. virtual void who_am_i() const {}
4. virtual ostream& print(ostream& o) const {
    o << *this ;
    return o ;
}

void test5() {
    exam bi("sushma",52,80) ;
    cs32exam di("sushma","cs32",99,100,97) ;
    exam* b = &bi ;
    cout << *b << endl ;
    exam* d = &di ; /* Dynamic binding */
    cout << *d << endl ;
    d->print(cout) ; cout << endl ;
    b->who_am_i() ;
    d->who_am_i() ;
    d->exam::who_am_i() ;
}

```

exam constructor sushma
 exam constructor sushma
 cs32 exam constructor cs32
 sushma 52 80
 sushma 99 100 cs32 97
 sushma 99 100 cs32 97
 I am exam class sushma
 I am cs32exam class cs32
 I am exam class sushma
 cs32exam destructor cs32
 exam destructor sushma
 exam destructor sushma

Figure 8.24: Problems of static binding fixed

6. Problem of static binding corrected

```

1. virtual ~exam() { }
2. virtual ostream& print(ostream& o) const {}
3. virtual void who_am_i() const {}
4. virtual ostream& print(ostream& o) const {
    o << *this ;
    return o ;
}

void test6() {
    exam bi("sushma",52,80);
    cs32exam di("sushma","cs32",99,100,97);
    exam& b = bi;
    cout << b << endl;
    exam& d = di; /* dynamic binding */
    cout << d << endl;
    d.print(cout); cout << endl;
    b.who_am_i();
    d.who_am_i();
    d.exam::who_am_i();
}

```

exam constructor sushma
exam constructor sushma
cs32 exam constructor cs32
sushma 52 80
sushma 99 100 cs32 97
sushma 99 100 cs32 97
I am exam class sushma
I am cs32exam class cs32
I am exam class sushma
cs32exam destructor cs32
exam destructor sushma
exam destructor sushma

Dynamic binding

Figure 8.25: Problems of static binding fixed

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename: in7_corrected.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 #include <cassert>  
8 #include <string>  
9 using namespace std;  
10  
11 /*-----  
12 class exam  
13 -----*/  
14 class exam {  
15 public:  
16     exam(const string& s, int m = 0, int f = 0):_sname(s),_midterm(m),_final(f) {  
17         cout << "exam constructor " << s << endl ;  
18     }  
19     virtual ~exam() { cout << "exam destructor " << _sname << endl ; }  
20     exam(const exam& e):_sname(e._sname), _midterm(e._midterm),_final(e._final) {  
21         cout << "exam copy constructor " << e._sname << endl ;  
22     }  
23     exam& operator=(const exam& e) {  
24         cout << "exam = operator " << e._sname << endl;  
25         if (&e != this) {  
26             _sname = e._sname;  
27             _midterm = e._midterm;  
28             _final = e._final;  
29         }  
30         return *this;  
31     }  
32     friend ostream& operator<<(ostream& o, const exam& s) {  
33         s.print(o);  
34         return o;  
35     }  
36     virtual ostream& print(ostream& o) const {  
37         o << _sname << " " << _midterm << " " << _final << " "  
38         return o;  
39     }  
40     virtual void who_am_i() const {  
41         cout << "I am exam class " << _sname << endl ;  
42     }  
43     void pass_by_reference(exam*& e){  
44         e->who_am_i();  
45     }  
46     void pass_by_value(exam* e){  
47         e->who_am_i();  
48     }  
49 private:  
50     string _sname;  
51     int _midterm;  
52     int _final;  
53 };  
54  
55 /*-----
```

```
56 cs32 exam
57 -----
58 class cs32exam : public exam {
59 public:
60     cs32exam(const string& s,const string& e, int m = 0, int f = 0,int p = 0) :
61         exam(s,m,f)/*Note this call*/,_examname(e),_project(p) {
62     cout << "cs32 exam constructor " << e << endl ;
63 }
64 ~cs32exam() { cout << "cs32exam destructor " << _examname << endl ; }
65 cs32exam(const cs32exam& e):exam(e)/* NOTE */,_examname(e._examname),_project(e._project) {
66     cout << "cs32exam copy constructor " << e._examname << endl ;
67 }
68 cs32exam& operator=(const cs32exam& e) {
69     if (&e != this) {
70         cout << "cs32exam = operator " << e._examname << endl ;
71         exam::operator=(e); //Note this call. base of e is copied base of this e
72         _examname = e._examname ;
73         _project = e._project;
74     }
75     return *this ;
76 }
77 friend ostream& operator<<(ostream& o, const cs32exam& s) {
78     s.print(o);
79 }
80 virtual ostream& print(ostream& o) const {
81     this->exam::print(o);
82     o << _examname << " " << _project << " ";
83     return o ;
84 }
85 void who_am_i() const {
86     cout << "I am cs32exam class " << _examname << endl ;
87 }
88 private:
89     string _examname ;
90     int _project ;
91 };
92 /*
93 -----test4 OLD-----
94 exam constructor sushma
95 sushma 52 80
96 exam constructor sushma
97 cs32 exam constructor cs32
98 sushma 99 100 <-----WHY
99 I am exam class sushma
100 I am exam class sushma <----- WHY
101 I am exam class sushma
102 exam destructor sushma
103 exam destructor sushma <-----WHY
104 exam destructor sushma <-----WHY
105
106 -----test4 NOW -----
107 exam constructor sushma
108 sushma 52 80
109 exam constructor sushma
110 cs32 exam constructor cs32
```

```
111 sushma 99 100 cs32 97 <---- SOLVED
112 sushma 99 100 cs32 97 <---- SOLVED
113 I am exam class sushma
114 I am cs32exam class cs32 <----- SOLVED
115 I am exam class sushma
116 exam destructor sushma
117 cs32exam destructor cs32 <-----SOLVED
118 exam destructor sushma
119 -----
120 void test4(){
121     cout << "-----test4-----\n";
122     exam* b = new exam("sushma",52,80);
123     cout << *b << endl;
124     exam* d = new cs32exam("sushma","cs32",99,100,97);
125     cout << *d << endl; /* static binding */
126     d->print(cout); cout << endl; /* dynamic binding */
127     b->who_am_i();
128     d->who_am_i();
129     d->exam::who_am_i();
130     delete b;
131     delete d;
132 }
133 /*
134 -----test5 OLD -----
135 exam constructor sushma
136 exam constructor sushma
137 cs32 exam constructor cs32
138 sushma 52 80
139 sushma 99 100 <-----WHY
140 I am exam class sushma
141 I am exam class sushma <-----WHY
142 I am exam class sushma
143 cs32exam destructor cs32
144 exam destructor sushma
145 exam destructor sushma
146 exam destructor sushma
147
148 -----test5 NOW -----
149 exam constructor sushma
150 exam constructor sushma
151 cs32 exam constructor cs32
152 sushma 52 80
153 sushma 99 100 cs32 97 <--- SOLVED
154 sushma 99 100 cs32 97 <--- SOLVED
155 I am exam class sushma
156 I am cs32exam class cs32 <--- SOLVED
157 I am exam class sushma
158 cs32exam destructor cs32
159 exam destructor sushma
160 exam destructor sushma
161 -----
162 void test5(){
163     cout << "-----test5-----\n";
164     exam bi("sushma",52,80);
165     cs32exam di("sushma","cs32",99,100,97);
```

```
166 exam* b = &bi ;
167 cout << *b << endl ;
168 exam* d = &di ; /* Dynamic binding */
169 cout << *d << endl ;
170 d->print(cout) ; cout << endl ;
171 b->who_am_i() ;
172 d->who_am_i() ;
173 d->exam::who_am_i() ;
174 }
175
176 /*-----*
177 -----test6 OLD -----
178 exam constructor sushma
179 exam constructor sushma
180 cs32 exam constructor cs32
181 sushma 52 80
182 sushma 99 100 <-----WHY
183 I am exam class sushma
184 I am exam class sushma <-----WHY
185 I am exam class sushma
186 cs32exam destructor cs32
187 exam destructor sushma
188 exam destructor sushma
189
190 -----test6 NOW -----
191 exam constructor sushma
192 exam constructor sushma
193 cs32 exam constructor cs32
194 sushma 52 80
195 sushma 99 100 cs32 97 <--- SOLVED
196 sushma 99 100 cs32 97 <--- SOLVED
197 I am exam class sushma
198 I am cs32exam class cs32 <--- SOLVED
199 I am exam class sushma
200 cs32exam destructor cs32
201 exam destructor sushma
202 exam destructor sushma
203 -----*/
204 void test6() {
205 cout << "-----test6-----\n" ;
206 exam bi("sushma",52,80) ;
207 cs32exam di("sushma","cs32",99,100,97) ;
208 exam& b = bi ;
209 cout << b << endl ;
210 exam& d = di ; /* Dynamic binding */
211 cout << d << endl ;
212 d.print(cout) ; cout << endl ;
213 b.who_am_i() ;
214 d.who_am_i() ;
215 d.exam::who_am_i() ;
216 }
217
218 /*-
219 exam constructor sushma
220 sushma 52 80
```

```
221 exam constructor sushma
222 cs32 exam constructor cs32
223 sushma 99 100 cs32 97
224 sushma 99 100 cs32 97
225 I am cs32exam class cs32 -- Dynamic binding
226 I am cs32exam class cs32 -- Dynamic binding
227 I am exam class sushma -- Dynamic binding
228 -----
229 void test7() {
230     exam *b = new exam("sushma",52,80);
231     cout << *b << endl ;
232     exam *d = new cs32exam("sushma","cs32",99,100,97);
233     cout << *d << endl ;
234     d->print(cout); cout << endl ;
235     d->pass_by_reference(d);
236     d->pass_by_value(d);
237     b->pass_by_value(b);
238 }
239
240 /*
241 main
242 -----
243 int main() {
244     test4();
245     test5();
246     test6();
247     test7();
248     return 0;
249 }
250
```

8.16 Abstract class and concept of pure virtual function

Pure Virtual function and abstract class

```

class exam {
    virtual void compute_grade() const = 0 ;
}
class cs32exam : public exam {
    virtual void compute_grade() const {
        cout << "final grade is : " << midterm() + final() + _project << endl ;
    }
}

void test_pure_virtual_function() {
    //exam b = new exam("sushma") ; // 'exam' : cannot instantiate abstract class
    cs32exam* c = new cs32exam("jag","cs320",99,100,97) ;
    cout << *c << endl ;
    c->print(cout) ;cout << endl ;
    exam *d = new cs32exam("sushma","cs32",99,100,97) ;
    cout << *d << endl ;
    d->print(cout) ;cout << endl ;
    d->who_am_i() ;
    d->compute_grade() ;
    delete c ;
    delete d ;
}

```

exam constructor jag
 cs32 exam constructor cs320
 jag 99 100 cs320 97
 jag 99 100 cs320 97
 exam constructor sushma
 cs32 exam constructor cs32
 sushma 99 100
 sushma 99 100 cs32 97
I am cs32exam class cs32
final grade is : 296
 cs32exam destructor cs320
 exam destructor jag
 cs32exam destructor cs32
 exam destructor sushma

Figure 8.26: Abstract class and pure virtual function

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename: purevirtual.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 #include <cassert>  
8 #include <string>  
9 using namespace std;  
10  
11 /*-----  
12 class exam  
13 -----*/  
14 class exam {  
15 public:  
16     exam(const string& s, int m = 0, int f = 0):_sname(s),_midterm(m),_final(f){  
17         cout << "exam constructor " << s << endl ;  
18     }  
19     virtual ~exam() { cout << "exam destructor " << _sname << endl ;}  
20     exam(const exam& e):_sname(e._sname),_midterm(e._midterm),_final(e._final){  
21         cout << "exam copy constructor " << e._sname << endl ;  
22     }  
23     exam& operator=(const exam& e){  
24         cout << "exam = operator " << e._sname << endl;  
25         if (&e != this){  
26             _sname = e._sname;  
27             _midterm = e._midterm;  
28             _final = e._final;  
29         }  
30         return *this;  
31     }  
32     friend ostream& operator<<(ostream& o, const exam& s){  
33         o << s._sname << " " << s._midterm << " " << s._final << " "  
34         return o;  
35     }  
36     virtual ostream& print(ostream& o) const {  
37         o << *this;  
38         return o;  
39     }  
40     virtual void who_am_i() const {  
41         cout << "I am exam class " << _sname << endl ;  
42     }  
43     virtual void compute_grade() const = 0; //PURE VIRTUAL FUNCTION - ABSTRACT CLASS  
44     int midterm() const {return _midterm;}  
45     int final() const {return _final;}  
46  
47 private:  
48     string _sname;  
49     int _midterm;  
50     int _final;  
51 };  
52  
53 /*-----  
54 cs32 exam  
55 -----*/
```

```
56 class cs32exam : public exam {
57 public:
58     cs32exam(const string& s,const string& e, int m = 0, int f = 0,int p = 0) :
59         exam(s,m,f)/*Note this call*/,_examname(e),_project(p) {
60     cout << "cs32 exam constructor " << e << endl ;
61 }
62 ~cs32exam() { cout << "cs32exam destructor " << _examname << endl ; }
63 cs32exam(const cs32exam& e):exam(e)/* NOTE */,_examname(e._examname),_project(e._project) {
64     cout << "cs32exam copy constructor " << e._examname << endl ;
65 }
66 cs32exam& operator=(const cs32exam& e) {
67     if (&e != this) {
68         cout << "cs32exam = operator " << e._examname << endl ;
69         exam::operator=(e); //Note this call. base of e is copied base of this e
70         _examname = e._examname ;
71         _project = e._project;
72     }
73     return *this ;
74 }
75 friend ostream& operator<<(ostream& o, const cs32exam& s) {
76     const exam* e = &s ;
77     o << *e ;
78     o << s._examname << " " << s._project << " ";
79     return o ;
80 }
81 virtual ostream& print(ostream& o) const {
82     o << *this ;
83     return o ;
84 }
85 void who_am_i() const {
86     cout << "I am cs32exam class " << _examname << endl ;
87 }
88 virtual void compute_grade() const {
89     cout << "final grade is :" << midterm() + final() + _project << endl ;
90 }
91 private:
92     string _examname ;
93     int _project ;
94 };
95
96 /*-----
97 exam constructor jag
98 cs32 exam constructor cs320
99 jag 99 100 cs320 97
100 jag 99 100 cs320 97
101 exam constructor sushma
102 cs32 exam constructor cs32
103 sushma 99 100
104 sushma 99 100 cs32 97
105 I am cs32exam class cs32
106 final grade is : 296
107 cs32exam destructor cs320
108 exam destructor jag
109 cs32exam destructor cs32
110 exam destructor sushma
```

```
111 -----*/
112 void test_pure_virtual_function() {
113     //exam b = new exam("sushma"); //'exam' : cannot instantiate abstract class
114     cs32exam* c = new cs32exam("jag","cs320",99,100,97);
115     cout << *c << endl;
116     c->print(cout);cout << endl;
117     exam *d = new cs32exam("sushma","cs32",99,100,97);
118     cout << *d << endl;
119     d->print(cout);cout << endl;
120     d->who_am_i();
121     d->compute_grade();
122     delete c;
123     delete d;
124 }
125
126 /*-
127 main
128 -----*/
129 int main(){
130     test_pure_virtual_function();
131     return 0;
132 }
133
```

8.17. CASTING FROM DERIVED TO BASE AND BASE TO DERIVED

8.17 Casting from derived to base and base to derived

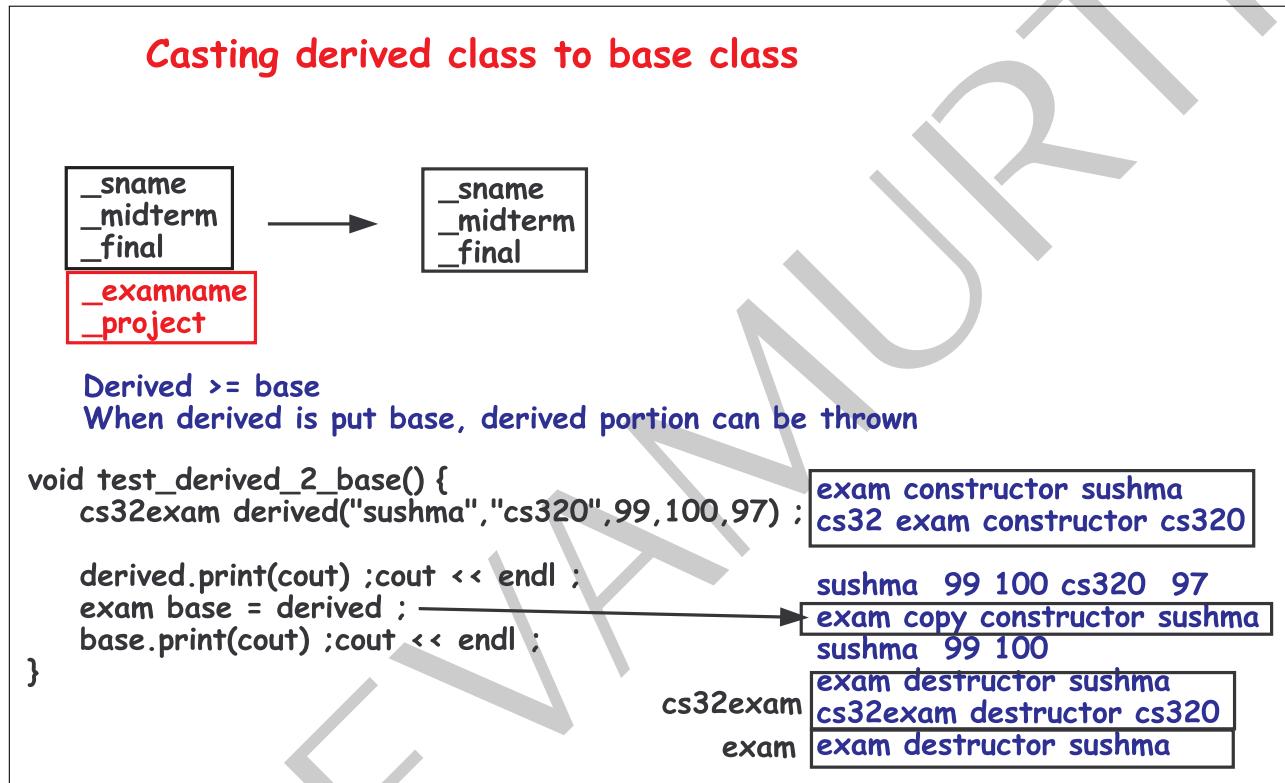


Figure 8.27: Casting from derived to base

Casting base class to derived class

```

class base {
public:
    string _sname;
    int _midterm;
    int _final;
};

class derived : public base {
public:
    string _examname;
    int _project;
};

```

base <= derived
When base is put to derived, what happens to extra space?

```

cs32exam derived = base ;
'initializing' : cannot convert from 'exam' to 'cs32exam'

```

```

cs32exam(const exam& e) :exam(e),_examname("hole"),_project(-1) {
    cout << "cs32exam conversion constructor " << "hole" << endl ;
}

```

1. This constructor must be written in derived class
2. base class DOES NOT know about derived class
3. This constructor MUST TAKE base class as a parameter
4. It should take care of HOLES

```

void test_base_2_derived() {
    exam base("sushma",99,100) ;
    base.print(cout);cout << endl ;
    cs32exam derived = base ;
    derived.print(cout);cout << endl ;
}

```

```

exam constructor sushma
sushma 99 100
exam copy constructor sushma
cs32exam conversion constructor hole
sushma 99 100 hole -1
cs32exam destructor hole
exam destructor sushma
exam destructor sushma

```

```

cs32exam(const exam& e) :exam(e),_examname("hole"),_project(-1) {
}

```

Must take base class Note the call: exam(e) HOLES fixed

Figure 8.28: Casting from base to derived

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename: casting.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 #include <cassert>  
8 #include <string>  
9 using namespace std;  
10  
11 /*-----  
12 class exam  
13 -----*/  
14 class exam {  
15 public:  
16     exam(const string& s, int m = 0, int f = 0):_sname(s),_midterm(m),_final(f) {  
17         cout << "exam constructor " << s << endl ;  
18     }  
19     virtual ~exam() { cout << "exam destructor " << _sname << endl ; }  
20     exam(const exam& e):_sname(e._sname),_midterm(e._midterm),_final(e._final) {  
21         cout << "exam copy constructor " << e._sname << endl ;  
22     }  
23     exam& operator=(const exam& e) {  
24         cout << "exam = operator " << e._sname << endl;  
25         if (&e != this){  
26             _sname = e._sname ;  
27             _midterm = e._midterm ;  
28             _final = e._final ;  
29         }  
30         return *this ;  
31     }  
32     friend ostream& operator<<(ostream& o, const exam& s) {  
33         o << s._sname << " " << s._midterm << " " << s._final << " " ;  
34         return o ;  
35     }  
36     virtual ostream& print(ostream& o) const {  
37         o << *this ;  
38         return o ;  
39     }  
40  
41 private:  
42     string _sname ;  
43     int _midterm ;  
44     int _final ;  
45 };  
46  
47 /*-----  
48 cs32 exam  
49 -----*/  
50 class cs32exam : public exam {  
51 public:  
52     cs32exam(const string& s,const string& e, int m = 0, int f = 0,int p = 0) :  
53         exam(s,m,f)/*Note this call*/,_examname(e),_project(p) {  
54         cout << "cs32 exam constructor " << e << endl ;  
55     }  
56 }
```

```
56 ~cs32exam() { cout << "cs32exam destructor " << _examname << endl ; }
57 cs32exam(const cs32exam& e):exam(e)/* NOTE */,_examname(e._examname),_project(e._project) {
58     cout << "cs32exam copy constructor " << e._examname << endl ;
59 }
60 cs32exam& operator=(const cs32exam& e) {
61     if (&e != this) {
62         cout << "cs32exam = operator " << e._examname << endl ;
63         exam::operator=(e); //Note this call. base of e is copied base of this e
64         _examname = e._examname ;
65         _project = e._project;
66     }
67     return *this ;
68 }
69 friend ostream& operator<<(ostream& o, const cs32exam& s) {
70     const exam* e = &s ;
71     o << *e ;
72     o << s._examname << " " << s._project << " ";
73     return o ;
74 }
75 virtual ostream& print(ostream& o) const {
76     o << *this ;
77     return o ;
78 }
79 //Constructor that takes a base class and generate derived class
80 cs32exam(const exam& e):exam(e),_examname("hole"),_project(-1) {
81     cout << "cs32exam conversion constructor " << "hole" << endl ;
82 }
83
84 private:
85     string _examname ;
86     int _project ;
87 } ;
88
89 /*-----
90 derived is >= base
91 So derived can always put to base
92 derived class portion is thrown away keeping only base portion
93
94 exam constructor sushma
95 cs32 exam constructor cs320
96 sushma 99 100 cs320 97
97 exam copy constructor sushma
98 sushma 99 100
99 exam destructor sushma
100 cs32exam destructor cs320
101 exam destructor sushma
102 -----*/
103 void test_derived_2_base() {
104     cs32exam derived("sushma","cs320",99,100,97) ;
105     derived.print(cout);cout << endl ;
106     exam base = derived ;
107     base.print(cout);cout << endl ;
108 }
109
110 /*-----
```

```
111 base <= derived
112 So when base is put to derived, what happens to the extra space? What data it will have?
113 'initializing' : cannot convert from 'exam' to 'cs32exam'
114 So you need to have a converter which takes base and convert to derived
115
116 exam constructor sushma
117 sushma 99 100
118 exam copy constructor sushma
119 cs32exam conversion constructor hole
120 sushma 99 100 hole -1
121 cs32exam destructor hole
122 exam destructor sushma
123 exam destructor sushma
124 -----
125 void test_base_2_derived() {
126     exam base("sushma",99,100);
127     base.print(cout);cout << endl;
128     cs32exam derived = base;
129     derived.print(cout);cout << endl;
130 }
131
132 /*
133 main
134 -----
135 int main() {
136     test_derived_2_base();
137     test_base_2_derived();
138     return 0;
139 }
140
```

8.18 Implementing polymorphism without using virtual

```
1 /*-
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy
3 Filename: implement_virtual.cpp
4 -----*/
5
6 #include <iostream>
7 #include <cassert>
8 #include <string>
9 using namespace std;
10
11 #ifdef WIN32
12 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector
13 #endif
14
15 /*-
16 Polymorphism using Virtual
17 -----*/
18 class animal {
19 public:
20     animal() {cout << "animal constructor\n";}
21     void where_am_i() { cout << "I am In Africa\n" ; } //NON VIRTUAL
22     virtual void who_am_i() { cout << "I am an animal\n" ; } //VIRTUAL
23     virtual ~animal() {cout << "animal destructor\n";} //VIRTUAL.
24 };
25
26
27 class lion:public animal {
28 public:
29     lion() {cout << "lion constructor\n";}
30     virtual void who_am_i() { cout << "I am a lion\n" ; }
31     ~lion() {cout << "lion destructor\n";}
32 };
33
34
35 /*-
36 -----*/
37
38 void test_virtual_polymorphism() {
39     animal* a[2] ;
40     a[0] = new animal;
41     a[1] = new lion ;
42     for (int i = 0; i < 2; i++) {
43         a[i]->where_am_i() ;
44         a[i]->who_am_i() ;
45     }
46     for (int i = 0; i < 2; i++) {
47         delete a[i] ;
48     }
49 }
50
51 /*-
52 Polymorphism WITHOUT using Virtual
53 -----*/
54 class ANIMAL;
55 typedef void(ANIMAL::*ANIMAL_PTR)(void) ;
56
57 class LION;
58 typedef void(LION::*LION_PTR)(void) ;
59
60
61 class ANIMAL {
62 public:
63     static ANIMAL_PTR animal_vtable ;
64     //Made static to show that vtable is ONLY one for each CLASS
65     //Even though this class is instantiated million times.
66     ANIMAL_PTR* vptr ; //Note global. Can access from main for easy coding
```

```
67
68     ANIMAL() {cout << "ANIMAL constructor\n"; vptr = &animal_vtable; }
69     void where_am_i() { cout << "I am In Africa\n" ; }
70     void who_am_i() { cout << "I am an ANIMAL\n" ; }
71     ~ANIMAL() {cout << "ANIMAL destructor\n";}
72 };
73
74 class LION:public ANIMAL {
75 public:
76     static LION_PTR lion_vtable ;
77     //Made static to show that vtable is ONLY one for each CLASS
78     //Even though this class is instantiated million times.
79     LION() {cout << "LION constructor\n"; vptr = (ANIMAL_PTR*) &lion_vtable;}
80     void who_am_i() { cout << "I am a LION\n" ; }
81     ~LION() {cout << "LION destructor\n";}
82 };
83
84 /*-----
85 Initialize all static now
86 PURPOSEFULLY ~distructor is NOT made virtual
87 -----*/
88 ANIMAL_PTR ANIMAL::animal_vtable = {&ANIMAL::who_am_i};
89 LION_PTR LION::lion_vtable = {&LION::who_am_i};
90
91 void test_non_virtual_polymorphism() {
92     ANIMAL* a[2] ;
93     a[0] = new ANIMAL ;
94     a[1] = new LION ;
95     for (int i = 0; i < 2; i++) {
96         a[i]->where_am_i() ; //Static binding
97         ANIMAL_PTR ptr = *(a[i]->vptr) ;
98         ((a[i])->*(ptr)) () ; //Dynamic binding
99     }
100    for (int i = 0; i < 2; i++) {
101        delete a[i] ;
102    }
103 }
104
105 /*-----
106 main
107
108 animal constructor
109 animal constructor
110 lion constructor
111 I am In Africa
112 I am an animal
113 I am In Africa
114 I am a lion
115 animal destructor
116 lion destructor
117 animal destructor
118 =====
119 ANIMAL constructor
120 ANIMAL constructor
121 LION constructor
122 I am In Africa
123 I am an ANIMAL
124 I am In Africa
125 I am a LION
126 ANIMAL destructor
127 <<----- BUG HERE ON PURPOSE
128 ANIMAL destructor
129
130
131 -----*/
132 void main() {
```

```
133     test_virtual_polymorphism() ;  
134     test_non_virtual_polymorphism();  
135 }  
136  
137  
138 //EOF  
139
```

Chapter 9

Namespaces

9.1 Introduction

9.2 What is the problem?

```
/*-----  
n1.h  
-----*/  
#ifndef N1_H  
#define N1_H  
  
/*-----  
Declaration of sqrt class  
-----*/  
class sqrt{  
public:  
    sqrt() {cout << "I am in sqrt of file n1.h" << endl;}  
  
private:  
};  
#endif  
  
/*-----  
n2.h  
-----*/  
#ifndef N2_H  
#define N2_H  
  
/*-----  
Declaration of sqrt class  
-----*/  
class sqrt{  
public:  
    sqrt() {cout << "I am in sqrt of file n2.h" << endl;}  
  
private:  
};  
#endif  
  
main.cpp  
#include "n1.h"  
#include "n2.h"  
main()  
{}
```

Figure 9.1: Problem

9.3. NEED FOR NAMESPACE

9.3 Need for namespace

```
/*-----  
n1.h-----*/  
#ifndef N1_H  
#define N1_H  
  
/*-----  
Declaration of sqrt class-----*/  
namespace N1 {  
    class sqrt{  
        public:  
            sqrt() {cout << "I am in sqrt of file n1.h" << endl;}  
  
        private:  
    };  
}  
#endif  
  
/*-----  
n2.h-----*/  
#ifndef N2_H  
#define N2_H  
  
/*-----  
Declaration of sqrt class-----*/  
namespace N2 {  
    class sqrt{  
        public:  
            sqrt() {cout << "I am in sqrt of file n2.h" << endl;}  
  
        private:  
    };  
}  
#endif  
  
main.cpp  
#include "n1.h"  
#include "n2.h"  
main()  
{  
    N1::sqrt a;  
    N2::sqrt b;  
}
```

Namespace provide a solution to the problem of duplicate names in the global space

Figure 9.2: Need for namespace

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: n1.h  
4 -----*/  
5  
6 /*-----  
7 All includes here  
8 -----*/  
9 #ifndef N1_H  
10 #define N1_H  
11  
12 #include <iostream>  
13 #include <cassert>  
14 #include <string>  
15 using namespace std;  
16  
17 #ifdef _WIN32  
18 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector  
19 #endif  
20  
21 namespace N1 {  
22     //A variable  
23     int x ;  
24     //A function  
25     void hello() {cout << "I am hello of namespace N1\n" ;}  
26     void f1() ;  
27     class arith{  
28         public:  
29             arith() {cout << "I am in class arith of file n1.h" << endl ;}  
30             void f2() ;  
31         private:  
32     };  
33 }  
35  
36 /*-----  
37 To show namespace functions can be written outside  
38 -----*/  
39 void N1::f1() {  
40     cout << "I am in N1::f1()\n" ;  
41 }  
42 void N1::arith::f2() {  
43     cout << "I am in N1::arith::f2()\n" ;  
44 }  
45  
46 /*-----  
47 To show namespace can be extended in same file or in different file  
48 -----*/  
49 namespace N1 {  
50     //A variable  
51     int x2 ;  
52 }  
53  
54 #endif  
55 //EOF  
56
```

```
1 /*-----  
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy  
3 file: n2.h  
4 -----*/  
5  
6 /*-----  
7 All includes here  
8 -----*/  
9 #ifndef N2_H  
10 #define N2_H  
11  
12 #include <iostream>  
13 #include <cassert>  
14 #include <string>  
15 using namespace std;  
16  
17 #ifdef _WIN32  
18 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector  
19 #endif  
20  
21 namespace N2 {  
22     //A variable  
23     int x ;  
24     //A function  
25     void hello() {cout << "I am hello of namespace N2\n"; }  
26     class arith{  
27         public:  
28             arith() {cout << "I am in class arith of file n2.h" << endl ;}  
29         private:  
30     };  
31 }  
32  
33  
34  
35 #endif  
36 //EOF  
37
```

```
1 /*-
2 Copyright (c) 2013 Author: Jagadeesh Vasudevamurthy
3 Filename: n1.cpp
4 -----*/
5 #include <iostream>
6 #include <cassert>
7 #include <string>
8 using namespace std;
9
10 #ifndef _WIN32
11 #include "vld.h" //Comment this line, if you have NOT installed Visual leak detector
12 #endif
13
14 /*-
15 All includes here
16 -----*/
17 #include "n1.h"
18 #include "n2.h"
19
20 /*-
21 Global variables
22 -----*/
23 int y = 80 ;
24
25 /*-
26 UNARY SCOPE RESOLUTION OPERATOR:
27 ::x = 9089 ;
28 It tells the compiler to look RHS variable(x) in GLOBAL SCOPE
29 BINARY SCOPE RESOLUTION OPERATOR
30 N1::x= 40 ;
31 Look for the RHS variable(x) in the NAMESPACE of the LHS(N1)
32 -----*/
33 void accessing_name_space_directly() {
34     //accessing variable
35     N1::x= 40 ; //BINARY SCOPE RESOLUTION OPERATOR
36     N2::x= 40 ;
37     //accessing function
38     N1::hello() ;
39     N2::hello() ;
40     //accessing class
41     N1::arith a;
42     N2::arith b ;
43
44     int y = 90 ;
45     cout << "Local y = " << y << endl ;
46
47     //Without :: you could not have accessed global y
48     ::y = 9089 ; //UNARY SCOPE RESOLUTION OPERATOR
49     cout << "global y = " << ::y << endl ;
50
51     N1::f1() ;
52     N1::x2 = 80 ;
53     a.f2() ;
54 }
55
56 /*-
57 -----*/
58 void accessing_name_space_by_using() {
59 {
60     using namespace N1;
61     //accessing variable
62     x= 40 ; //BINARY SCOPE RESOLUTION OPERATOR
63     N2::x= 80 ;
64     cout << "N1 x = " << x << endl ;
65     cout << "N2 x = " << N2::x << endl ;
66 }
```

```
67     //accessing function
68     hello() ;
69     N2::hello() ;
70     //accessing class
71     arith a;
72     N2::arith b ;
73     //accessing global
74     cout << "global y = " << y << endl ;
75     cout << "global y = " << ::y << endl ;
76 }
77 {
78     //Bring N2
79     using namespace N2;
80     //accessing variable
81     x= 40 ; //BINARY SCOPE RESOLUTION OPERATOR
82     N1::x= 80 ;
83     cout << "N2 x = " << x << endl ;
84     cout << "N1 x = " << N1::x << endl ;
85     //accessing function
86     hello() ;
87     N1::hello() ;
88     //accessing class
89     arith a;
90     N1::arith b ;
91     //accessing global
92     cout << "global y = " << y << endl ;
93     cout << "global y = " << ::y << endl ;
94 }
95 }
96
97 /*-
98 Global variables again
99 -----*/
100 int x = 80 ;
101 /*
102 /*
103 WILL NOT COMPILE
104
105 'x' : ambiguous symbol
106 could be 'n1.cpp(96) : int x' (GLOBAL)
107 or      n1.h(23) : int N1::x'
108 -----*/
109 #if 0
110 void ambiguous_situation() {
111     using namespace N1;
112     x= 40 ;
113     N2::x= 80 ;
114 }
115 #endif
116
117 void ambiguous_situation_fixed() {
118     using namespace N1;
119     ::x= 40 ;
120     N1::x = 9 ;
121     N2::x= 80 ;
122 }
123
124 namespace UCSC {
125     const char* name = "UCSC EXTENSION" ;
126 }
127 /*
128 -----*/
129 /*
130 void you_are_using_already() {
131     //If you comment using namespace std;
132     std::cout << "Hello World \n" ;
```

```
133 //If you uncomment using namespace std;
134 cout << "Hello World \n" ;
135 cout << UCSC::name << endl ;
136 //cout << name ; //'name' : undeclared identifier
137 using namespace UCSC;
138 cout << name << endl ;
139 }
140
141
142 /*-----
143 main
144 -----*/
145 void main() {
146   accessing_name_space_directly() ;
147   accessing_name_space_by_using() ;
148   ambiguous_situation_fixed();
149   you_are_using_already() ;
150 }
151
152
153 //EOF
154
155
```

9.3. NEED FOR NAMESPACE

VASUDEVAMURTHY

Chapter 10

Template

10.1 Introduction

10.2 Macros

```
1  /*
2  Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy
3  Filename: macro.cpp
4  */
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 int minimum
11 */
12 static void minimum(int i, int j) {
13     cout << " minimum(int i, int j) " << endl ;
14     if (i < j) {
15         cout << i << " is less than " << j << endl ;
16     }else {
17         cout << i << " is greater than or equal " << j << endl ;
18     }
19 }
20
21 /*
22 float minimum
23 */
24 static void minimum(double i, double j) {
25     cout << " minimum(double i, double j) " << endl ;
26     if (i < j) {
27         cout << i << " is less than " << j << endl ;
28     }else {
29         cout << i << " is greater than or equal " << j << endl ;
30     }
31 }
32
33 /*
34 ch minimum
35 */
36 static void minimum(char i, char j) {
37     cout << " minimum(char i, char j) " << endl ;
38     if (i < j) {
39         cout << i << " is less than " << j << endl ;
40     }else {
41         cout << i << " is greater than or equal " << j << endl ;
42     }
43 }
44
45 /*
46 string minimum
47 */
48 static void minimum(const char* i, const char* j) {
49     cout << " minimum(const char* i, const char* j) " << endl ;
```

```
50     int x = strcmp(i,j) ;
51     if (x < 0)
52         cout << i << " is less than " << j << endl ;
53     else if (x == 0) {
54         cout << i << " is equal to " << j << endl ;
55     }else {
56         cout << i << " is greater than " << j << endl ;
57     }
58 }
59
60 /*-----
61 define minimum
62 -----*/
63 #define tminimum(i,j) { \
64     cout << " tminimum(T i, T j) " << endl ; \
65     if (i < j) { \
66         cout << i << " is less than " << j << endl ; \
67     }else { \
68         cout << i << " is greater than or equal " << j << endl ; \
69     } \
70 } \
71
72 /*-----
73 define print_array
74 -----*/
75 #define print_array(s, a, size) { \
76     cout << s << " {" ; \
77     for (int i = 0; i < size; i++) { \
78         cout << a[i] ; \
79         if (i < size - 1) { \
80             cout << ", " ; \
81         } \
82     } \
83     cout << " } " << endl ; \
84 } \
85
86
87 /*-----
88 assignment
89 -----*/
90 static void assignment() {
91     int a[] = { 1, 2, 3, 4, 5 };
92     print_array("a is ",a,sizeof(a)/sizeof(int)) ;
93     double b[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 } ;
94     print_array("b is ",b,sizeof(b)/sizeof(double)) ;
95     char c[] = "HELLO";
96     print_array("c is ",c,sizeof(c)/sizeof(char)) ;
97 }
98 }
```

```
99
100 /*-----
101   minimum(int i, int j)
102   47 is less than 90
103   minimum(double i, double j
104   47.9 is less than 90.8
105   minimum(char i, char j)
106   a is less than b
107   minimum(const char* i, const char* j)
108   jag is equal to jag
109   minimum(const char* i, const char* j)
110   jaf is less than jag
111   minimum(const char* i, const char* j)
112   jah is greater than jag
113   tminimum(T i, T j)
114   47 is less than 90
115   tminimum(T i, T j)
116   47.9 is less than 90.8
117   tminimum(T i, T j)
118   a is less than b
119   tminimum(T i, T j)
120   jag is less than jag
121   tminimum(T i, T j)
122   jaf is less than jag
123   tminimum(T i, T j)
124   jah is less than jag
125   a is {1, 2, 3, 4, 5 }
126   b is {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 }
127   c is {H, E, L, L, O,   }
128
129 -----*/
130 int main() {
131   minimum(47,90) ;
132   minimum(47.9,90.8) ;
133   minimum('a','b') ;
134   minimum("jag","jag") ;
135   minimum("jaf","jag") ;
136   minimum("jah","jag") ;
137   tminimum(47,90) ;
138   tminimum(47.9,90.8) ;
139   tminimum('a','b') ;
140   tminimum("jag","jag") ;
141   tminimum("jaf","jag") ;
142   tminimum("jah","jag") ;
143   assignment();
144   return 0 ;
145 }
```

10.3 Function templates

```
1  /*
2  Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy
3  Filename: ftemp.cpp
4  */
5
6 #include <iostream>
7 using namespace std;
8
9 /*
10 int minimum
11 */
12 static void minimum(int i, int j) {
13     cout << " minimum(int i, int j) " << endl ;
14     if (i < j) {
15         cout << i << " is less than " << j << endl ;
16     }else {
17         cout << i << " is greater than or equal " << j << endl ;
18     }
19 }
20
21 /*
22 float minimum
23 */
24 static void minimum(double i, double j) {
25     cout << " minimum(double i, double j) " << endl ;
26     if (i < j) {
27         cout << i << " is less than " << j << endl ;
28     }else {
29         cout << i << " is greater than or equal " << j << endl ;
30     }
31 }
32
33 /*
34 ch minimum
35 */
36 static void minimum(char i, char j) {
37     cout << " minimum(char i, char j) " << endl ;
38     if (i < j) {
39         cout << i << " is less than " << j << endl ;
40     }else {
41         cout << i << " is greater than or equal " << j << endl ;
42     }
43 }
44
45 /*
46 string minimum
47 */
48 static void minimum(const char* i, const char* j) {
49     cout << " minimum(const char* i, const char* j) " << endl ;
```

```
c:\work\software\course\code\template\function\ftemp.cpp
50     int x = strcmp(i,j) ;
51     if (x < 0)
52         cout << i << " is less than " << j << endl ;
53     else if (x == 0) {
54         cout << i << " is equal to " << j << endl ;
55     }else {
56         cout << i << " is greater than " << j << endl ;
57     }
58 }
59
60 /*-----
61 template minimum
62 -----*/
63 template <typename T>
64 static void tminimum(T i, T j) {
65     cout << " tminimum(T i, T j) " << endl ;
66     if (i < j) {
67         cout << i << " is less than " << j << endl ;
68     }else {
69         cout << i << " is greater than or equal " << j << endl ;
70     }
71 }
72
73 /*-----
74 string minimum
75 -----*/
76 static void tminimum(const char* i, const char* j) {
77     cout << " tminimum(const char* i, const char* j) " << endl ;
78     int x = strcmp(i,j) ;
79     if (x < 0)
80         cout << i << " is less than " << j << endl ;
81     else if (x == 0) {
82         cout << i << " is equal to " << j << endl ;
83     }else {
84         cout << i << " is greater than " << j << endl ;
85     }
86 }
87
88 /*-----
89 template assignment
90 -----*/
91 template <typename T>
92 static void print_array(const char *s, const T& a, int size) {
93     cout << s << " {";
94     for (int i = 0; i < size; i++) {
95         cout << a[i] ;
96         if (i < size - 1) {
97             cout << ", " ;
98         }
99     }
100 }
```

```
c:\work\software\course\code\template\function\ftemp.cpp
 99    }
100   cout << " } " << endl ;
101 }
102
103 /*-----
104 assignment
105 -----*/
106 static void assignment() {
107   int a[] = { 1, 2, 3, 4, 5 };
108   print_array("a is ",a,sizeof(a)/sizeof(int)) ;
109   double b[] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
110   print_array("b is ",b,sizeof(b)/sizeof(double)) ;
111   char c[] = "HELLO";
112   print_array("c is ",c,sizeof(c)/sizeof(char)) ;
113 }
114
115 /*-----
116 47.9 is less than 90.8
117 minimum(char i, char j)
118 a is less than b
119 minimum(const char* i, const char* j)
120 jag is equal to jag
121 minimum(const char* i, const char* j)
122 jaf is less than jag
123 minimum(const char* i, const char* j)
124 jah is greater than jag
125 tminimum(T i, T j)
126 47 is less than 90
127 tminimum(T i, T j)
128 47.9 is less than 90.8
129 tminimum(T i, T j)
130 a is less than b
131 tminimum(const char* i, const char* j)
132 jag is equal to jag
133 tminimum(const char* i, const char* j)
134 jaf is less than jag
135 tminimum(const char* i, const char* j)
136 jah is greater than jag
137 a is {1, 2, 3, 4, 5 }
138 b is {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 }
139 c is {H, E, L, L, O,   }
140 -----*/
141 int main() {
142   minimum(47,90) ;
143   minimum(47.9,90.8) ;
144   minimum('a','b') ;
145   minimum("jag","jag") ;
146   minimum("jaf","jag") ;
147   minimum("jah","jag") ;
```

```
148     tminimum(47,90) ;
149     tminimum(47.9,90.8) ;
150     tminimum('a','b') ;
151     tminimum("jag","jag") ;
152     tminimum("jaf","jag") ;
153     tminimum("jah","jag") ;
154     assignment();
155     return 0 ;
156 }
157
158
```

10.4. CLASS TEMPLATES

10.4 Class templates

10.5 Integer stack implementation

```
1  /*-----*
2 Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy
3 Filename: istack.h
4 -----*/
5 #ifndef ISTACK_H
6 #define ISTACK_H
7
8 #include <iostream>
9 #include <cassert>
10 using namespace std;
11
12 /*-----*
13 class istack
14 -----*/
15 namespace vasudevamurthy {
16     class istack {
17     public:
18         static const int SIZE = 100;
19         istack(int n = SIZE) ;
20         ~istack() ;
21         istack(const istack& s) ;
22         istack& operator=(const istack& rhs) ;
23
24         int size() const {return _sp; }
25         int capacity() const {return _size ; }
26         bool isempty() const {return _sp ? false : true ; }
27         bool isfull() const {return (_sp < _size) ? false: true ;}
28         void push(const int& x) ;
29         int& top() const ;
30         void pop() ;
31         friend ostream& operator<<(ostream& o, const istack& s) ;
32         static void set_verbose(bool x) {_display = x ; }
33
34     private:
35         int* _array ;
36         int _size ;
37         int _sp ;
38         void _copy(const istack& s) ;
39         static bool _display ;
40
41     };
42 }
43
44#endif
45
46
```

```
1  /*-----  
2  Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy  
3  Filename: istack.cpp  
4  -----*/  
5  
6 #include "istack.h"  
7 #include <iostream>  
8 using namespace std;  
9 using namespace vasudevamurthy ;  
10  
11 /*-----  
12 static function declaration  
13 -----*/  
14 bool istack::_display = true ;  
15  
16 /*-----  
17 Constructor  
18 -----*/  
19 istack::istack(int n):_size(n),_sp(0),_array(nullptr) {  
20     if (_display) {  
21         cout << "istack constructor" << endl ;  
22     }  
23     _array = new int[n] ;  
24 }  
25  
26 /*-----  
27 destructor  
28 -----*/  
29 istack::~istack() {  
30     if (_display) {  
31         cout << "istack destructor" << endl ;  
32     }  
33     delete [] _array ;  
34     _size = 0 ;  
35     _sp = 0 ;  
36 }  
37  
38 /*-----  
39 _Copy  
40 -----*/  
41 void istack::_copy(const istack& s){  
42     _size = s._size ;  
43     _sp = s._sp ;  
44     _array = new int[_size] ;  
45     for (int i = 0; i < _sp ; i++) {  
46         _array[i] = s._array[i] ;  
47     }  
48 }  
49
```

```
50 /*-----  
51 Copy Constructor  
52 -----*/  
53 istack::istack(const istack& s){  
54     if (_display) {  
55         cout << "istack Copy constructor" << endl ;  
56     }  
57     _copy(s) ;  
58 }  
59  
60 /*-----  
61 equal Constructor  
62 -----*/  
63 istack& istack::operator=(const istack& s){  
64     if (_display) {  
65         cout << "istack equal operator" << endl ;  
66     }  
67     if (this != &s) {  
68         delete [] _array ;  
69         _copy(s) ;  
70     }  
71     return *this ;  
72 }  
73  
74 /*-----  
75 push  
76 -----*/  
77 void istack::push(const int& x){  
78     if (isfull()) {  
79         assert(0) ;  
80     }  
81     _array[_sp++] = x ;  
82 }  
83  
84 /*-----  
85 top  
86 -----*/  
87 int& istack::top() const{  
88     if (isempty()) {  
89         assert(0) ;  
90     }  
91     return _array[_sp-1] ;  
92 }  
93  
94 /*-----  
95 pop  
96 -----*/  
97 void istack::pop() {  
98     if (isempty()) {
```

```
c:\work\software\course\code\template\istack\istack.cpp
99     assert(0) ;
100 }
101     _sp-- ;
102 }
103
104 /*-----
105 Overloaded print operator
106 Because operator<< is global, we need to mention namespace
107 -----*/
108 namespace vasudevamurthy {
109     ostream& operator<<(ostream& o, const istack& s) {
110         for (int i = 0; i < s._sp; i++) {
111             cout << s._array[i] << " " ;
112         }
113         return o ;
114     }
115 }
116
```

```
1  /*-----*
2 Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy
3 Filename: istack_test.cpp
4
5 -----*/
6 #include "istack.h"
7 #include <iostream>
8 using namespace std;
9 using namespace vasudevamurthy ;
10
11 /*-----
12 istack constructor
13 s1 looks like this: 0 100 200 300 400
14 istack Copy constructor
15 s2 looks like this: 0 100 200 300 400
16 s2 and s1 looks like this:
17 0 100 200 300 400
18 0 100 200 300 400
19 top of s1 = 400
20 After poping 4 elements s1 looks like this:
21 0
22 s2 looks like this:
23 0 100 200 300 400
24 istack equal operator
25 After s2 = s1, s2 looks like this:
26 0
27 istack destructor
28 istack destructor
29 -----*/
30 static void test() {
31     istack s1 ;
32     for (int i = 0; i < 5; i++) {
33         s1.push(i * 100) ;
34     }
35     cout << "s1 looks like this: " ;
36     cout << s1 << endl ;
37     istack s2(s1) ;
38     cout << "s2 looks like this: " ;
39     cout << s2 << endl ;
40     cout << "s2 and s1 looks like this: " << endl ;
41     cout << s2 << endl << s1 << endl ;
42     cout << "top of s1 = " << s1.top() << endl ;
43     for (int i = 0 ; i < 4; i++) {
44         s1.pop() ;
45     }
46     cout << "After poping 4 elements s1 looks like this: " << endl ;
47     cout << s1 << endl ;
48     cout << "s2 looks like this: " << endl ;
49     cout << s2 << endl ;
```

```
c:\work\software\course\code\template\istack\istack_test.cpp
50   s2 = s1 ;
51   cout << "After s2 = s1, s2 looks like this: " << endl ;
52   cout << s2 << endl ;
53 }
54
55 /*-----*/
56
57 -----*/
58 int main() {
59   istack::set_verbose(true) ;
60   test() ;
61   return 0 ;
62 }
63
```

10.6 Generic stack implementation using `typedef`

c:\work\software\course\code\template\tstack\tstack.h

1

```
1  /*-----*
2 Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy
3 Filename: tstack.h
4 -----*/
5 #ifndef tstack_H
6 #define tstack_H
7
8 #include <iostream>
9 #include <cassert>
10 #include <string>
11 using namespace std;
12
13 /*-----*
14 Only place you have to change
15 -----*/
16 //#define TEST_INT
17 #define TEST_STRING
18 //#define TEST_UNSIGNED_LONG
19
20 /*-----*
21 All typedef here
22 -----*/
23 #ifdef TEST_INT
24 typedef int T;
25 #endif
26
27 #ifdef TEST_STRING
28 typedef string T;
29 #endif
30
31 #ifdef TEST_UNSIGNED_LONG
32 typedef unsigned long T;
33 #endif
34
35 /*-----*
36 class tstack
37 -----*/
38 namespace vasudevamurthy {
39     class tstack {
40     public:
41         static const int SIZE = 100;
42         tstack(int n = SIZE) ;
43         ~tstack() ;
44         tstack(const tstack& s) ;
45         tstack& operator=(const tstack& rhs) ;
46
47         int size() const {return _sp; }
48         int capacity() const {return _size ; }
49         bool isempty() const {return _sp ? false : true ; }
```

```
50     bool isfull() const {return (_sp < SIZE) ? false: true ;}
51     void push(const T& x) ;
52     T& top() const ;
53     void pop() ;
54     friend ostream& operator<<(ostream& o, const tstack& s) ;
55     static void set_verbose(bool x) {_display = x ; }
56
57 private:
58     T* _array ;
59     int _size ;
60     int _sp ;
61     void _copy(const tstack& s) ;
62     static bool _display ;
63 };
64 }
65
66 #endif
67
68
```

```
1  /*-----  
2  Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy  
3  Filename: tstack.cpp  
4  -----*/  
5  
6 #include "tstack.h"  
7 #include <iostream>  
8 using namespace std;  
9 using namespace vasudevamurthy ;  
10  
11 /*-----  
12 static function declaration  
13 -----*/  
14 bool tstack::_display = true ;  
15  
16 /*-----  
17 Constructor  
18 -----*/  
19 tstack::tstack(int n):_size(n),_sp(0),_array(nullptr) {  
20     if (_display) {  
21         cout << "tstack constructor" << endl ;  
22     }  
23     _array = new T[n] ;  
24 }  
25  
26 /*-----  
27 destructor  
28 -----*/  
29 tstack::~tstack() {  
30     if (_display) {  
31         cout << "tstack destructor" << endl ;  
32     }  
33     delete [] _array ;  
34     _size = 0 ;  
35     _sp = 0 ;  
36 }  
37  
38 /*-----  
39 _Copy  
40 -----*/  
41 void tstack::_copy(const tstack& s){  
42     _size = s._size ;  
43     _sp = s._sp ;  
44     _array = new T[_size] ;  
45     for (int i = 0; i < _sp ; i++) {  
46         _array[i] = s._array[i] ;  
47     }  
48 }  
49 }
```

```
50 /*-----*
51 Copy Constructor
52 -----*/
53 tstack::tstack(const tstack& s){
54     if (_display) {
55         cout << "tstack Copy constructor" << endl ;
56     }
57     _copy(s) ;
58 }
59
60 /*-----*
61 equal Constructor
62 -----*/
63 tstack& tstack::operator=(const tstack& s){
64     if (_display) {
65         cout << "tstack equal operator" << endl ;
66     }
67     if (this != &s) {
68         delete [] _array ;
69         _copy(s) ;
70     }
71     return *this ;
72 }
73
74 /*-----*
75 push
76 -----*/
77 void tstack::push(const T& x){
78     if (isfull()) {
79         assert(0) ;
80     }
81     _array[_sp++] = x ;
82 }
83
84 /*-----*
85 top
86 -----*/
87 T& tstack::top() const{
88     if (isempty()) {
89         assert(0) ;
90     }
91     return _array[_sp-1] ;
92 }
93
94 /*-----*
95 pop
96 -----*/
97 void tstack::pop() {
98     if (isfull()) {
```

```
c:\work\software\course\code\template\tstack\tstack.cpp
99     assert(0) ;
100 }
101     _sp-- ;
102 }
103
104 /*
105 Overloaded print operator
106 Because operator<< is global, we need to mention namespace
107 */
108 namespace vasudevamurthy {
109     ostream& operator<<(ostream& o, const tstack& s) {
110         for (int i = 0; i < s._sp; i++) {
111             cout << s._array[i] << " " ;
112         }
113         return o ;
114     }
115 }
116
```

```
1  /*-----*
2 Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy
3 Filename: tstack_main.cpp
4
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
6 -----*/
7 #include "tstack.h"
8 #include <iostream>
9 using namespace std;
10 using namespace vasudevamurthy ;
11
12 /*-----*
13
14 -----*/
15 #ifdef TEST_INT
16 static void test_int() {
17     tstack s1 ;
18     for (int i = 0; i < 5; i++) {
19         s1.push(i * 100) ;
20     }
21     cout << "s1 looks like this: " ;
22     cout << s1 << endl ;
23     tstack s2(s1) ;
24     cout << "s2 looks like this: " ;
25     cout << s2 << endl ;
26     cout << "s2 and s1 looks like this: " << endl ;
27     cout << s2 << endl << s1 << endl ;
28     cout << "top of s1 = " << s1.top() << endl ;
29     for (int i = 0 ; i < 4; i++) {
30         s1.pop() ;
31     }
32     cout << "After poping 4 elements s1 looks like this: " << endl ;
33     cout << s1 << endl ;
34     cout << "s2 looks like this: " << endl ;
35     cout << s2 << endl ;
36     s2 = s1 ;
37     cout << "After s2 = s1, s2 looks like this: " << endl ;
38     cout << s2 << endl ;
39 }
40#endif
41
42 /*-----*
43
44 -----*/
45 #ifdef TEST_STRING
46 static void test_string() {
47     tstack s1 ;
48     for (int i = 0; i < 5; i++) {
49         char a[2] ;
```

```
c:\work\software\course\code\template\tstack\tstack_test.cpp
50     a[0] = i + '0' ;
51     a[1] = '\0' ;
52     string s = "course " ;
53     s1.push(s + a) ;
54 }
55 cout << "s1 looks like this: " ;
56 cout << s1 << endl ;
57 tstack s2(s1) ;
58 cout << "s2 looks like this: " ;
59 cout << s2 << endl ;
60 cout << "s2 and s1 looks like this: " << endl ;
61 cout << s2 << endl << s1 << endl ;
62 cout << "top of s1 = " << s1.top() << endl ;
63 for (int i = 0 ; i < 4; i++) {
64     s1.pop() ;
65 }
66 cout << "After poping 4 elements s1 looks like this: " << endl ;
67 cout << s1 << endl ;
68 cout << "s2 looks like this: " << endl ;
69 cout << s2 << endl ;
70 s2 = s1 ;
71 cout << "After s2 = s1, s2 looks like this: " << endl ;
72 cout << s2 << endl ;
73 }
74 #endif
75
76 /*-----*/
77
78 -----*/
79 int main() {
80     tstack::set_verbose(true) ;
81 #ifdef TEST_INT
82     test_int() ;
83 #endif
84 #ifdef TEST_STRING
85     test_string() ;
86 #endif
87     return 0 ;
88 }
89
```

10.7 Stack implementation using inheritance

10.7.1 Problem of using objects and static binding

```

class B {
public:
    B() {cout << "in B constructor\n";}
    B(const B& t) {cout << "in B copy constructor\n";}
    virtual ~B() {cout << "in B destructor\n";}
    virtual void who_am_i() const {cout << " I am B\n";}
};

class D:public B {
public:
    D() {cout << "in D constructor\n";}
    ~D() {cout << "in D destructor\n";}
    D(const D& r) {cout << "in D copy constructor\n";}
    virtual void who_am_i() const {cout << " I am D\n";}
};

static void test_object() {
    B b;
    D d;
    b.who_am_i(); I am B
    d.who_am_i(); I am D
    B c = d;
    c.who_am_i(); I am B
}

static void test_value() {
    B b;
    D d;
    b.who_am_i();
    d.who_am_i();
    B* c = &d;
    c->who_am_i(); I am D
}

static void test_alias() {
    B b;
    D d;
    b.who_am_i();
    d.who_am_i();
    B& c = d;
    c.who_am_i(); I am D
}

```

Figure 10.1: Using instance as object, pointers and reference

```

c:\Documents and Settings\jvasudev\My Documents\Dropbox\c++\course\code\template\virtualstack\test.cpp 1

1 /*-
2 Copyright (c) 2011 Author: Jagadeesh Vasudevanurthy
3 Filename: test.cpp
4 -----*/
5 #include <iostream>
6 using namespace std;
7
8
9 /*-
10 Base class
11 -----*/
12 class B {
13 public:
14     B() {cout << "in B constructor\n";}
15     B(const B& t) {cout << "in B copy constructor\n";}
16     B& operator=(const B& r) { cout << "in B EQUAL operator \n";}
17     virtual ~B() {cout << "in B destructor\n";}
18     virtual void who_am_i() const {cout << " I am B\n"; }
19 };
20
21 /*-
22 Derived class
23 -----*/
24 class D:public B {
25 public:
26     D() {cout << "in D constructor\n";}
27     ~D() {cout << "in D destructor\n";}
28     D(const D& r) {cout << "in D copy constructor\n"; }
29     D& operator=(const D& r) { cout << "in D EQUAL operator \n"; return *this; }
30     virtual void who_am_i() const {cout << " I am D\n"; }
31 };
32
33 /*-
34 in B constructor
35 in B constructor
36 in D constructor
37 I am B
38 I am D
39 in B copy constructor
40 I am B
41 in B destructor
42 in D destructor
43 in B destructor
44 in B destructor
45 in B constructor
46 in B constructor
47 in D constructor
48 -----*/
49 static void test_object() {
50     B b ;
51     D d ;
52     b.who_am_i() ;
53     d.who_am_i() ;
54
55     /* problem */
56     B c = d ;
57     c.who_am_i() ;
58 }
59
60 /*-
61 in B constructor
62 in B constructor
63 in D constructor
64 I am B
65 I am D
66 I am D

```

```
c:\Documents and Settings\jvasudev\My Documents\Dropbox\c++\course\code\template\virtualstack\test.cpp_2

67 in D destructor
68 in B destructor
69 in B destructor
70 -----*/
71 static void test_value() {
72     B b ;
73     D d ;
74     b.who_am_i() ;
75     d.who_am_i() ;
76
77     /* problem resolved */
78     B* c = &d ;
79     c->who_am_i() ;
80 }
81
82 /*-
83 in B constructor
84 in B constructor
85 in D constructor
86 I am B
87 I am D
88 I am D
89 in D destructor
90 in B destructor
91 in B destructor
92 -----*/
93 static void test_alias() {
94     B b ;
95     D d ;
96     b.who_am_i() ;
97     d.who_am_i() ;
98
99     /* problem resolved */
100    B& c = d ;
101    c.who_am_i() ;
102 }
103
104 /*-
105 -----*/
106 -----*/
107 int main() {
108     test_object() ;
109     test_value() ;
110     test_alias() ;
111     return 0 ;
112 }
113
```

10.7. STACK IMPLEMENTATION USING INHERITENCE

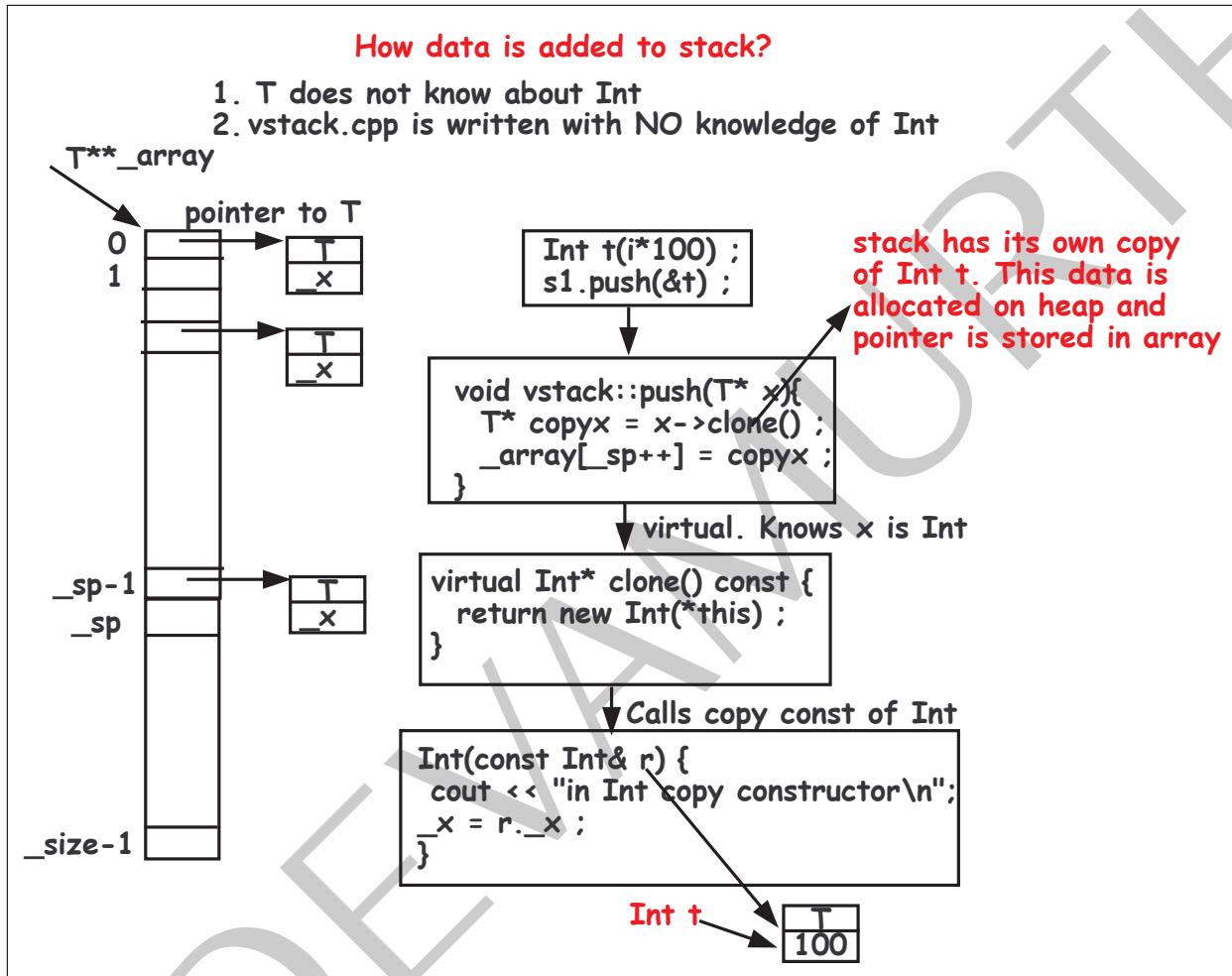


Figure 10.2: Adding derived data by using base base pointer

```

1 /*-----
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename: vstack.h
4 -----*/
5 #ifndef vstack_H
6 #define vstack_H
7
8 #include <iostream>
9 #include <cassert>
10 #include <string>
11 using namespace std;
12
13
14 /*-----
15 DATA is T
16 -----*/
17 class T {
18 public:
19     T() {cout << "in T constructor\n";}
20     T(const T& t) {cout << "in T copy constructor\n";}
21     T& operator=(const T& r) { cout << "in T EQUAL operator \n";}
22     virtual ~T() {cout << "in T destructor\n";}
23
24     virtual T* clone() const = 0 ; //Uses the copy constructor
25     virtual T* create() const = 0 ; //Uses the default constructor
26
27     //friend ostream& operator<<(ostream& o, const T& r) { return o; }
28     //Operator <<: It's not a member function, so it can't be made virtual. However, it could
29     //delegate to a virtual member function
30
31     virtual ostream& print(ostream& o) const = 0 ; //PURE VIRTUAL FUNCTION - ABSTRACT CLASS
32 };
33
34
35 /*-----
36 class vstack
37 -----*/
38 namespace vasudevamurthy {
39     class vstack {
40     public:
41         enum {SIZE = 100} ;
42         vstack(int n = SIZE) ;
43         ~vstack() ;
44         vstack(const vstack& s) ;
45         vstack& operator=(const vstack& rhs) ;
46
47         int size() const {return _sp; }
48         int capacity() const {return _size ; }
49         bool isempty() const {return _sp ? false : true ; }
50         bool isfull() const {return (_sp < SIZE) ? false: true ;}
51         void push(T* x) ;
52         T* top() const ;
53         void pop() ;
54         friend ostream& operator<<(ostream& o, const vstack& s) ;
55         static void set_verbose(bool x) {_display = x ; }
56
57     private:
58         T** _array ;
59         int _size ;
60         int _sp ;
61         void _copy(const vstack& s) ;
62         static bool _display ;
63         void _release() ;
64     };
65 }
66

```

10.7. STACK IMPLEMENTATION USING INHERITENCE

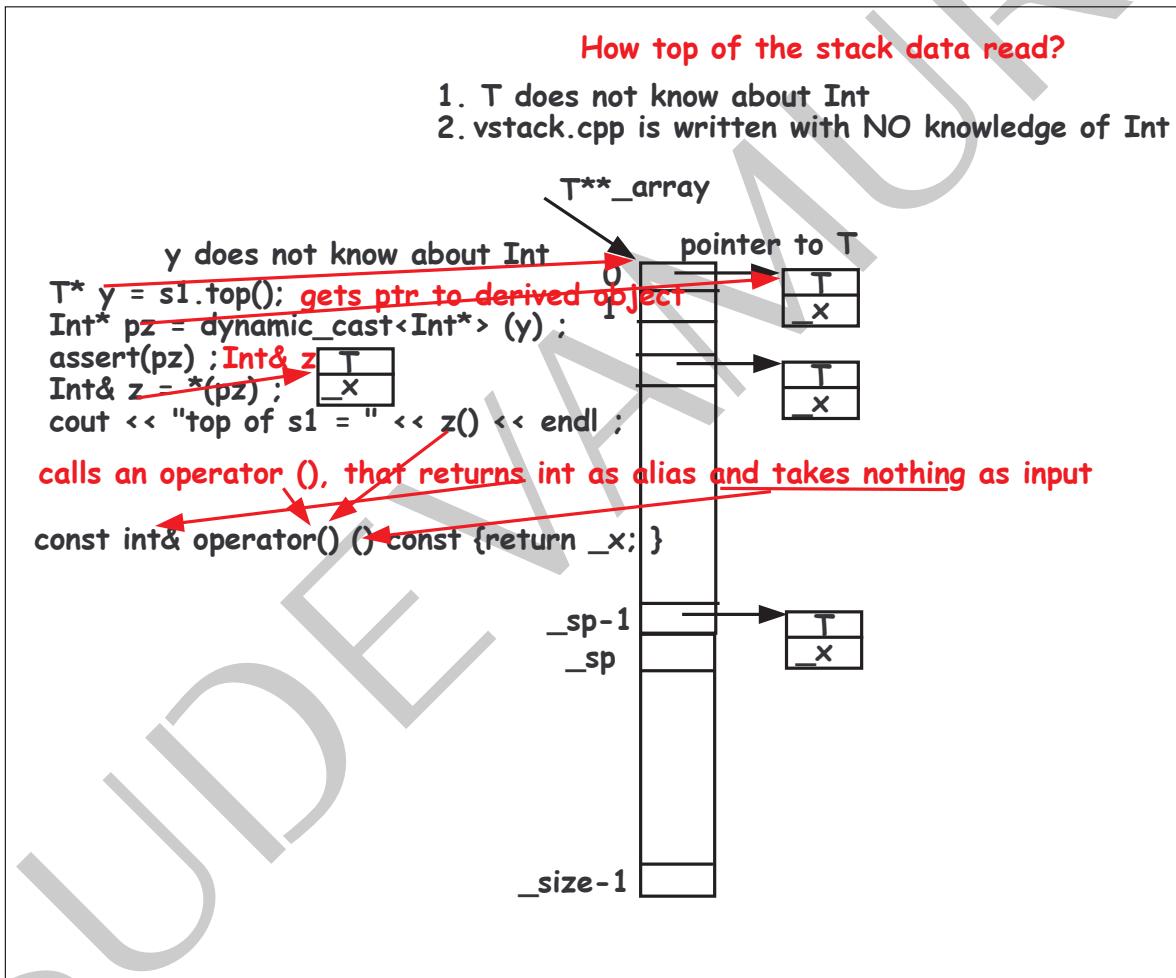


Figure 10.3: operation top

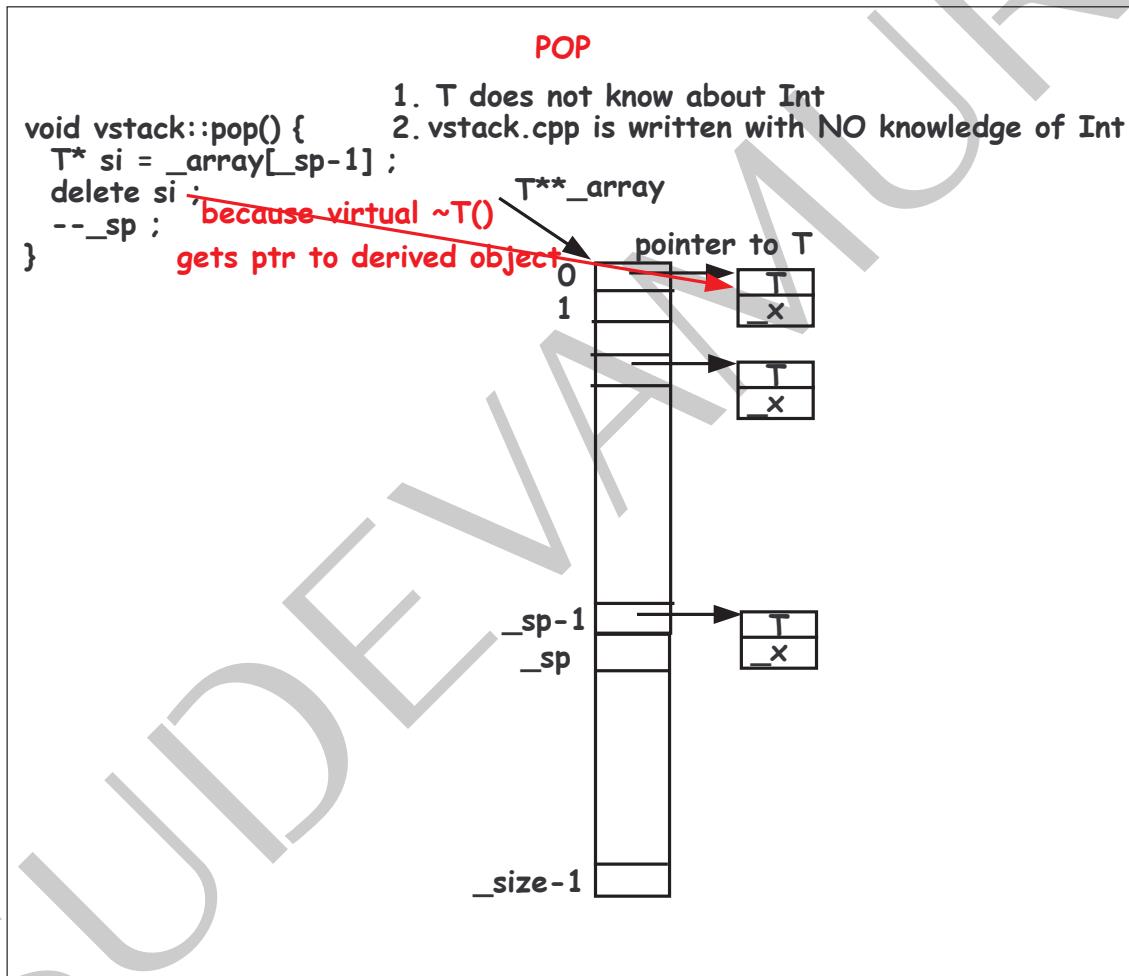


Figure 10.4: operation pop

10.7. STACK IMPLEMENTATION USING INHERITENCE

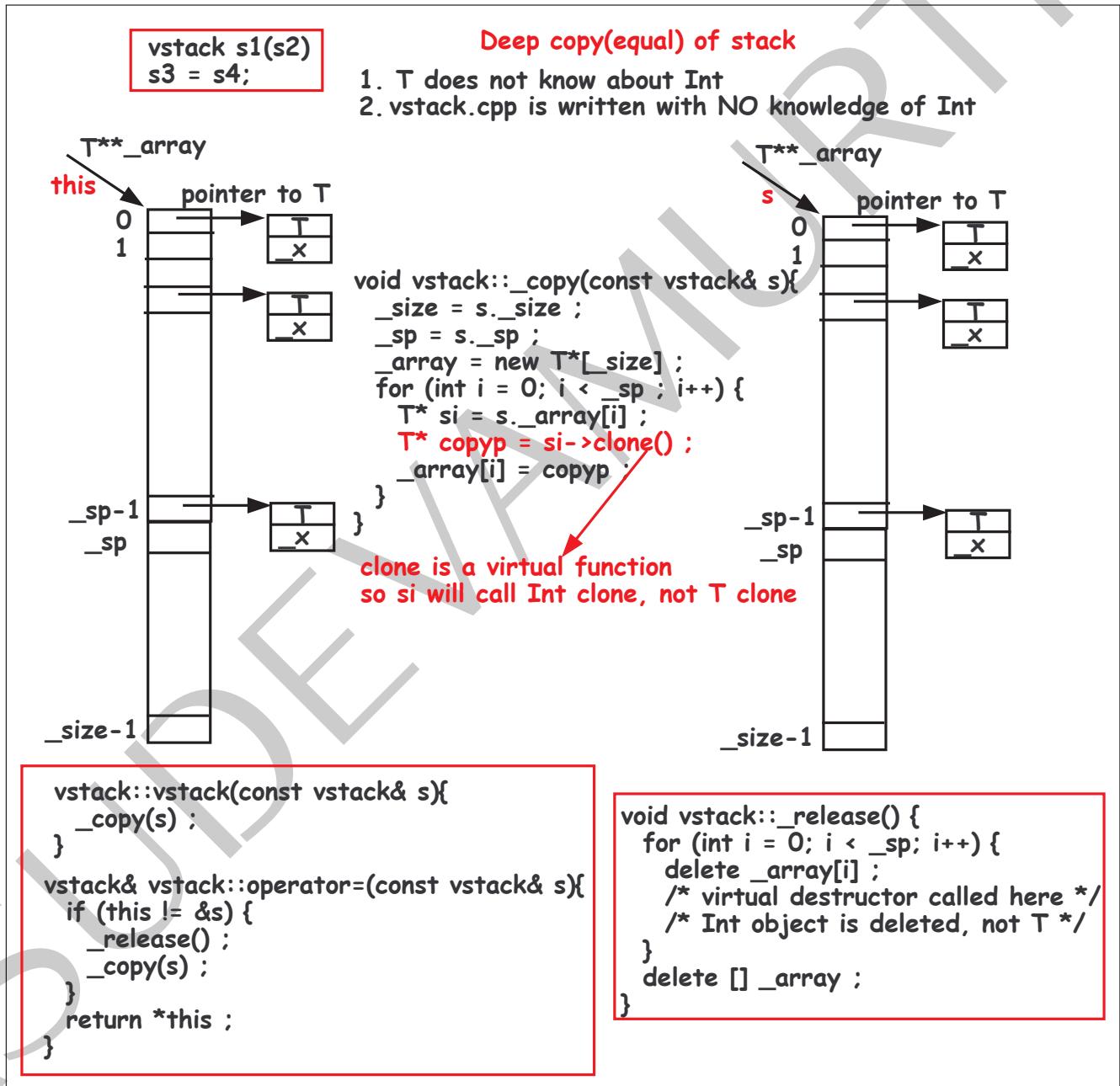


Figure 10.5: stack copy and equal operator

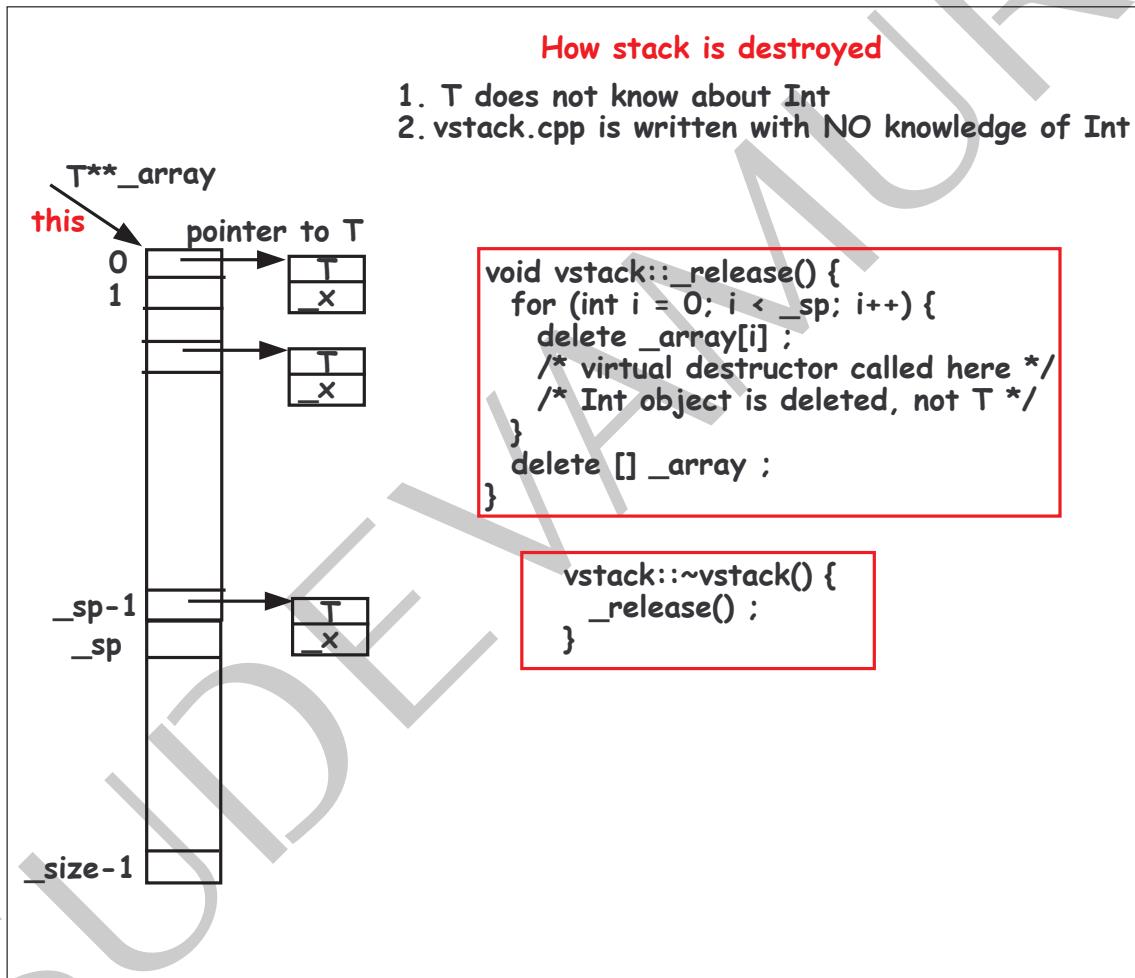


Figure 10.6: stack destructor

```

c:\Documents and Settings\jvasudev\My Documents\...\c++\course\code\template\virtualstack\vstack.cpp 1

1 /*-
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename: vstack.cpp
4 -----*/
5
6 #include "vstack.h"
7 #include <iostream>
8 using namespace std;
9 using namespace vasudevamurthy ;
10
11 /*-
12 static function declaration
13 -----*/
14 bool vstack::_display = true ;
15
16 /*-
17 Constructor
18 -----*/
19 vstack::vstack(int n):_size(n),_sp(0),_array(NULL) {
20     if (_display) {
21         cout << "vstack constructor" << endl ;
22     }
23     _array = new T*[n] ;
24 }
25
26 /*-
27 release
28 -----*/
29 void vstack::_release() {
30     for (int i = 0; i < _sp; i++) {
31         delete _array[i] ; /* virtual destructor called here */
32     }
33     delete [] _array ;
34 }
35
36 /*-
37 destructor
38 -----*/
39 vstack::~vstack() {
40     if (_display) {
41         cout << "vstack destructor" << endl ;
42     }
43     _release() ;
44     _size = 0 ;
45     _sp = 0 ;
46 }
47
48 /*-
49 _Copy
50 -----*/
51 void vstack::_copy(const vstack& s){
52     _size = s._size ;
53     _sp = s._sp ;
54     _array = new T*[_size] ;
55     for (int i = 0; i < _sp ; i++) {
56         T* si = s._array[i] ;
57         T* copyp = si->clone() ; /* virtual so that you copy derived obj */
58         _array[i] = copyp ;
59     }
60 }
61
62 /*-
63 Copy Constructor
64 -----*/
65 vstack::vstack(const vstack& s){
66     if (_display) {

```

```

c:\Documents and Settings\jvasudev\My Documents\...\c++\course\code\template\virtualstack\vstack.cpp 2

67     cout << "vstack Copy constructor" << endl ;
68 }
69 _copy(s) ;
70 }
71 */
72 /*-----*
73 equal Constructor
74 -----*/
75 vstack& vstack::operator=(const vstack& s){
76     if (_display) {
77         cout << "vstack equal operator" << endl ;
78     }
79     if (this != &s) {
80         _release() ;
81         _copy(s) ;
82     }
83     return *this ;
84 }
85 */
86 /*-----*
87 push
88 -----*/
89 void vstack::push(T* x){
90     if (isfull()) {
91         assert(0) ;
92     }
93     //PROBLEM. We don't know that x is a pointer to an object
94     //or x is a pointer to a pointer object
95     T* copyx = x->clone() ; /* virtual so that you copy derived obj */
96     _array[_sp++] = copyx ;
97 }
98 */
99 /*-----*
100 -----*/
101 /*-----*
102 T* vstack::top() const{
103     if (isempty()) {
104         assert(0) ;
105     }
106     return _array[_sp-1] ;
107 }
108 */
109 /*-----*
110 pop
111 -----*/
112 void vstack::pop() {
113     if (isfull()) {
114         assert(0) ;
115     }
116     T* si = _array[_sp-1] ;
117     delete si ; /* virtual so that you free derived obj */
118     --_sp ;
119 }
120 */
121 /*-----*
122 Overloaded print operator
123 Because operator<< is global, we need to mention namespace
124 -----*/
125 namespace vasudevamurthy {
126     ostream& operator<<(ostream& o, const vstack& s) {
127         for (int i = 0; i < s._sp; i++) {
128             T* x = s._array[i] ;
129             //Note that x will be derived object pointer here
130             x->print(o) ;
131             o << " " ;
132         }

```

```
133     return o ;  
134 }  
135 }  
136
```

```
1 /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename: vstack_main.cpp  
4  
5 Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6 Program exited with status code 0.  
7 -----*/  
8 #include "vstack.h"  
9 #include <iostream>  
10 using namespace std;  
11 using namespace vasudevamurthy ;  
12  
13 /*-----  
14 Class Int  
15 -----*/  
16 class Int:public T {  
17 public:  
18     Int(int x=0): _x(x) {cout << "in Int constructor\n";}  
19     ~Int() {cout << "in Int destructor\n";}  
20     Int(const Int& r) {cout << "in Int copy constructor\n";_x = r._x ; }  
21     Int& operator=(const Int& r) { cout << "in Int EQUAL operator \n";_x = r._x ; return *this; }  
22  
23     virtual Int* clone() const { return new Int(*this) ; } // You need a virtual destructor here  
24     virtual Int* create() const { return new Int(); }  
25  
26     ostream& print(ostream& o) const { o << *this; return o ; }  
27     friend ostream& operator<<(ostream& o, const Int& r) { cout << r._x; return o; }  
28     const int& operator() () const {return _x; } //calls a function (), that returns int and takes  
nothing as input  
29  
30 private:  
31     int _x ;  
32 };  
33  
34 /*-----  
35 -----*/  
36 static void test_int_by_value() {  
37     vstack s1 ;  
38     for (int i = 0; i < 5; i++) {  
39         Int t(i*100) ;  
40         s1.push(&t) ;  
41     }  
42     cout << "s1 looks like this: " ;  
43     cout << s1 << endl ;  
44  
45     vstack s2(s1) ;  
46     cout << "s2 looks like this: " ;  
47     cout << s2 << endl ;  
48     cout << "s2 and s1 looks like this: " << endl ;  
49     cout << s2 << endl << s1 << endl ;  
50     T* y = s1.top();  
51     //Int* z = y->clone() ;  
52     Int* pz = dynamic_cast<Int*> (y) ;  
53     assert(pz) ;  
54     Int& z = *(pz) ;  
55     cout << "top of s1 = " << z() << endl ;  
56     for (int i = 0 ; i < 4; i++) {  
57         s1.pop() ;  
58     }  
59     cout << "After popping 4 elements s1 looks like this: " << endl ;  
60     cout << s1 << endl ;  
61     cout << "s2 looks like this: " << endl ;  
62     cout << s2 << endl ;  
63     s2 = s1 ;  
64     cout << "After s2 = s1, s2 looks like this: " << endl ;
```

```

c:\Documents and Settings\jvasudev\My Documents\...\course\code\template\virtualstack\vstack_test.cpp 2

66   cout << s2 << endl ;
67 }
68
69 /*-----
70 -----
71 -----*/
72 static void test_int_by_address() {
73   vstack s1 ;
74   for (int i = 0; i < 5; i++) {
75     Int* t = new Int(i*100) ;
76     s1.push(t) ;
77     delete t ;
78   }
79   cout << "s1 looks like this: " ;
80   cout << s1 << endl ;
81
82   vstack s2(s1) ;
83   cout << "s2 looks like this: " ;
84   cout << s2 << endl ;
85   cout << "s2 and s1 looks like this: " << endl ;
86   cout << s2 << endl << s1 << endl ;
87   T* y = s1.top();
88   //Int* z = y->clone() ;
89   Int *pz = dynamic_cast<Int*> (y) ;
90   assert(pz) ;
91   Int& z = *(pz) ;
92   cout << "top of s1 = " << z() << endl ;
93   for (int i = 0 ; i < 4; i++) {
94     s1.pop() ;
95   }
96   cout << "After poping 4 elements s1 looks like this: " << endl ;
97   cout << s1 << endl ;
98   cout << "s2 looks like this: " << endl ;
99   cout << s2 << endl ;
100  s2 = s1 ;
101  cout << "After s2 = s1, s2 looks like this: " << endl ;
102  cout << s2 << endl ;
103 }
104
105 /*-----
106 Class String
107 -----*/
108 class String:public T {
109 public:
110   String() {cout << "in String Default constructor\n";}
111   String(string x): _x(x) {cout << "in String constructor\n";}
112   ~String() {cout << "in String destructor\n";}
113   String(const String& r) {cout << "in String copy constructor\n";_x = r._x ; }
114   String& operator=(const String& r) { cout << "in String EQUAL operator \n";_x = r._x ; return *this; }
115
116   virtual String* clone() const { return new String(*this) ;} // You need a virtual destructor here
117   virtual String* create() const { return new String(); }
118
119   ostream& print(ostream& o) const { o << *this; return o ; }
120   friend ostream& operator<<(ostream& o, const String& r) { cout << r._x; return o; }
121   const string& operator() () const {return _x; } //calls a function (), that returns int and takes
122   nothing as input
123 private:
124   string _x ;
125 };
126
127 /*-----
128 -----
129 -----*/

```

```
130 static void test_string() {
131     vstack s1 ;
132     for (int i = 0; i < 5; i++) {
133         char a[2] ;
134         a[0] = i + '0' ;
135         a[1] = '\0' ;
136         string s = "course " ;
137         string sa = s + a ;
138         String f(sa) ;
139         s1.push(&f) ;
140     }
141     cout << "s1 looks like this: " ;
142     cout << s1 << endl ;
143     vstack s2(s1) ;
144     cout << "s2 looks like this: " ;
145     cout << s2 << endl ;
146     cout << "s2 and s1 looks like this: " << endl ;
147     cout << s2 << endl << s1 << endl ;
148
149     T* y = s1.top();
150     //Int* z = y->clone() ;
151     String *pz = dynamic_cast<String*> (y) ;
152     assert(pz) ;
153     String& z = *(pz) ;
154     cout << "top of s1 = " << z() << endl ;
155     for (int i = 0 ; i < 4; i++) {
156         s1.pop() ;
157     }
158     cout << "After popping 4 elements s1 looks like this: " << endl ;
159     cout << s1 << endl ;
160     cout << "s2 looks like this: " << endl ;
161     cout << s2 << endl ;
162     s2 = s1 ;
163     cout << "After s2 = s1, s2 looks like this: " << endl ;
164     cout << s2 << endl ;
165 }
166
167
168 /*-----
169
170 -----*/
171 int main() {
172     vstack::set_verbose(true) ;
173     test_int_by_value() ;
174     test_int_by_address();
175     test_string() ;
176     return 0 ;
177 }
178 }
```

10.8 Stack implementation using template

Difference between templates and macros

1. Macro gets executed at compile time. Macros are expanded by the preprocessor, the compiler doesn't see them. They are simple text substitution. This is why macros can't be scoped by classes, namespaces, etc., because those haven't been parsed yet. The simplicity and flexibility of a macro is exactly what's called for some purposes.
2. There is no way for the compiler to verify that the macro parameters are of compatible types. The macro is expanded without any special type checking.
3. Template gets executed at run time. A template is a way to make functions independent of data-types. This cannot be accomplished using macros. E.g. a sorting function doesn't have to care whether it's sorting integers or letters since the same algorithm might apply anyway.
4. Template is known to the compiler and hence provides stronger type checking. A template is defined to expand as if it were text, so the compiler can then compile the result. Naturally this is more type safe than a macro.
5. Template is also being available for classes, but macros are just functions.

Figure 10.7: Templates versus macros

Inheritance VERSUS template

1. The inheritance approach is simply a 2-step process where in template it is a one-step process.
2. Inheritance first assigns something to a "type". The behavior is then "inherited" in a way by being a member of that type.
3. One disadvantage of being type-tied is that if typing {is-a} turns out to be the wrong modeling approach, then you have a lot of changing to do.

Figure 10.8: Inheritance versus template

Template has both declaration and implementation in one file. Why?

1. Template gets executed at run time.
That is the reason why, no code is generated when the compiler sees the templated function bodies.
2. Code is generated only when we declare an object with a specific template argument.
For this reason, declaration and implementation must be in the header files.
3. To create the code, C++ needs to know the function definitions.
Since C++ typically creates functions for a template class when it sees a declaration of an instance of that class, it must know the function definitions at that time.
Thus, they must appear in the *.h file.

Figure 10.9: Why all in one file?

```
1  /*-----  
2 Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy  
3 Filename: stack.h  
4 -----*/  
5 #ifndef stack_H  
6 #define stack_H  
7  
8 #include <iostream>  
9 #include <cassert>  
10 #include <string>  
11 using namespace std;  
12  
13 /*-----  
14 class stack  
15 -----*/  
16 namespace vasudevamurthy {  
17     template <typename T>  
18     class stack {  
19     public:  
20         static const int SIZE = 100 ;  
21         stack(int n = SIZE) ;  
22         ~stack() ;  
23         stack(const stack<T>& s) ;  
24         stack<T>& operator=(const stack<T>& rhs) ;  
25  
26         int size() const {return _sp; }  
27         int capacity() const {return _size ; }  
28         bool isempty() const {return _sp ? false : true ; }  
29         bool isfull() const {return (_sp < SIZE) ? false: true ; }  
30         void push(const T& x) ;  
31         T& top() const ;  
32         void pop() ;  
33         //Note the call below  
34         template <typename U> friend ostream& operator<<(ostream& o, const stack<U>& p  
            s) ;  
35         static void set_verbose(bool x) {_display = x ; }  
36  
37     private:  
38         T* _array ;  
39         int _size ;  
40         int _sp ;  
41         void _copy(const stack<T>& s) ;  
42         static bool _display ;  
43     };  
44 }  
45  
46 /*-----  
47 template implementations must be seen by compiler  
48 -----*/
```

c:\work\software\course\code\template\templateteststack\stack.h

```
49 #include "stack_impl.h"
50
51 #endif
52
53
```

2

```
1  /*-----  
2  Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy  
3  Filename: stack_impl.h  
4  -----*/  
5  using namespace vasudevamurthy ;  
6  
7  /*-----  
8  static function declaration  
9  -----*/  
10 template <typename T>  
11 bool stack<T>::_display = true ;  
12  
13 /*-----  
14 Constructor  
15 -----*/  
16 template <typename T>  
17 stack<T>::stack(int n):_size(n),_sp(0),_array(nullptr) {  
18     if (_display) {  
19         cout << "stack constructor" << endl ;  
20     }  
21     _array = new T[n] ;  
22 }  
23  
24 /*-----  
25 destructor  
26 -----*/  
27 template <typename T>  
28 stack<T>::~stack() {  
29     if (_display) {  
30         cout << "stack destructor" << endl ;  
31     }  
32     delete [] _array ;  
33     _size = 0 ;  
34     _sp = 0 ;  
35 }  
36  
37 /*-----  
38 _Copy  
39 -----*/  
40 template <typename T>  
41 void stack<T>::_copy(const stack<T>& s){  
42     _size = s._size ;  
43     _sp = s._sp ;  
44     _array = new T[_size] ;  
45     for (int i = 0; i < _sp ; i++) {  
46         _array[i] = s._array[i] ;  
47     }  
48 }  
49 }
```

```
50 /*-----  
51 Copy Constructor  
52 -----*/  
53 template <typename T>  
54 stack<T>::stack(const stack<T>& s){  
55     if (_display) {  
56         cout << "stack Copy constructor" << endl ;  
57     }  
58     _copy(s) ;  
59 }  
60  
61 /*-----  
62 equal Constructor  
63 -----*/  
64 template <typename T>  
65 stack<T>& stack<T>::operator=(const stack<T>& s){  
66     if (_display) {  
67         cout << "stack equal operator" << endl ;  
68     }  
69     if (this != &s) {  
70         delete [] _array ;  
71         _copy(s) ;  
72     }  
73     return *this ;  
74 }  
75  
76 /*-----  
77 push  
78 -----*/  
79 template <typename T>  
80 void stack<T>::push(const T& x){  
81     if (isfull()) {  
82         assert(0) ;  
83     }  
84     _array[_sp++] = x ;  
85 }  
86  
87 /*-----  
88 top  
89 -----*/  
90 template <typename T>  
91 T& stack<T>::top() const{  
92     if (isempty()) {  
93         assert(0) ;  
94     }  
95     return _array[_sp-1] ;  
96 }  
97  
98 /*-----
```

```
c:\work\software\course\code\template\templatestack\stack_impl.h
99  pop
100  -----
101 template <typename T>
102 void stack<T>::pop() {
103     if (isfull()) {
104         assert(0) ;
105     }
106     _sp-- ;
107 }
108
109 /**
110 Overloaded print operator
111 Because operator<< is global, we need to mention namespace
112 -----
113 namespace vasudevamurthy {
114     template <typename T>
115     ostream& operator<<(ostream& o, const stack<T>& s) {
116         for (int i = 0; i < s._sp; i++) {
117             cout << s._array[i] << " " ;
118         }
119         return o ;
120     }
121 }
122
```

```
1  /*-----  
2  Copyright (c) 2016 Author: Jagadeesh Vasudevamurthy  
3  Filename: Int.cpp stack_test.cpp  
4  
5  Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6  -----*/  
7  #include "stack.h"  
8  #include "Int.h"  
9  #include <iostream>  
10 using namespace std;  
11 using namespace vasudevamurthy ;  
12  
13 /*-----  
14 stack constructor  
15 s1 looks like this: 0 100 200 300 400  
16 stack Copy constructor  
17 s2 looks like this: 0 100 200 300 400  
18 s2 and s1 looks like:  
19 0 100 200 300 400  
20 0 100 200 300 400  
21 top of s1 = 400  
22 After poping 4 elements s1 looks like this:  
23 0  
24 s2 looks like this:  
25 0 100 200 300 400  
26 stack equal operator  
27 After s2 = s1, s2 looks like this:  
28 0  
29 stack destructor  
30 stack destructor  
31 -----*/  
32 static void test_int_by_value() {  
33     stack<int> s1 ;  
34     for (int i = 0; i < 5; i++) {  
35         s1.push(i * 100) ;  
36     }  
37     cout << "s1 looks like this: " ;  
38     cout << s1 << endl ;  
39     stack<int> s2(s1) ;  
40     cout << "s2 looks like this: " ;  
41     cout << s2 << endl ;  
42     cout << "s2 and s1 looks like this: " << endl ;  
43     cout << s2 << endl << s1 << endl ;  
44     cout << "top of s1 = " << s1.top() << endl ;  
45     for (int i = 0 ; i < 4; i++) {  
46         s1.pop() ;  
47     }  
48     cout << "After poping 4 elements s1 looks like this: " << endl ;  
49     cout << s1 << endl ;
```

```

...oftware\course\code\template\templateteststack\stack_test.cpp
50     cout << "s2 looks like this: " << endl ;
51     cout << s2 << endl ;
52     s2 = s1 ;
53     cout << "After s2 = s1, s2 looks like this: " << endl ;
54     cout << s2 << endl ;
55 }
56
57 /*-----
58 stack constructor
59 s1 looks like this: 003B74F0 003B7530 003B7570 003B75B0 003B75F0
60 stack Copy constructor
61 s2 looks like this: 003B74F0 003B7530 003B7570 003B75B0 003B75F0
62 s2 and s1 looks like:
63 003B74F0 003B7530 003B7570 003B75B0 003B75F0
64 003B74F0 003B7530 003B7570 003B75B0 003B75F0
65 top of s1 = 400
66 stack destructor
67 stack destructor
68 -----*/
69 static void test_int_by_address() {
70     stack<int*> s1 ;
71     for (int i = 0; i < 5; i++) {
72         int* t = new int(i*100) ;
73         s1.push(t) ;
74     }
75     cout << "s1 looks like this: " ;
76     cout << s1 << endl ;
77
78     stack<int*> s2(s1) ; //Now s2 and s1 will be holding same pointers
79     cout << "s2 looks like this: " ;
80     cout << s2 << endl ;
81     cout << "s2 and s1 looks like this: " << endl ;
82     cout << s2 << endl << s1 << endl ;
83     int*& y = s1.top();
84     cout << "top of s1 = " << *(y) << endl ;
85     for (int i = 0 ; i < 4; i++) {
86         int*& y = s1.top();
87         delete y ;
88         s1.pop() ;
89     }
90
91     /* s2 is holding some pointers which is removed by s1 */
92     /* DO NOT do d2 = s1 */
93
94     while(!(s1.isempty())) {
95         int*& y = s1.top();
96         delete y ;
97         s1.pop() ;
98     }

```

```
...oftware\course\code\template\templateteststack\stack_test.cpp
99     /* Why we should NOT do for s2 */
100 }
101
102 /*
103
104 */
105 static void test_Int_by_value() {
106     stack<Int> s1 ;
107     for (int i = 0; i < 5; i++) {
108         s1.push(i * 100) ;
109     }
110    cout << "s1 looks like this: " ;
111    cout << s1 << endl ;
112    stack<Int> s2(s1) ;
113    cout << "s2 looks like this: " ;
114    cout << s2 << endl ;
115    cout << "s2 and s1 looks like this: " << endl ;
116    cout << s2 << endl << s1 << endl ;
117    cout << "top of s1 = " << s1.top() << endl ;
118    for (int i = 0 ; i < 4; i++) {
119        s1.pop() ;
120    }
121    cout << "After poping 4 elements s1 looks like this: " << endl ;
122    cout << s1 << endl ;
123    cout << "s2 looks like this: " << endl ;
124    cout << s2 << endl ;
125    s2 = s1 ;
126    cout << "After s2 = s1, s2 looks like this: " << endl ;
127    cout << s2 << endl ;
128 }
129
130 /*
131
132 */
133 static void test_Int_by_address() {
134     stack<Int*> s1 ;
135     for (int i = 0; i < 5; i++) {
136         Int* t = new Int(i*100) ;
137         s1.push(t) ;
138     }
139     cout << "s1 looks like this: " ;
140     cout << s1 << endl ;
141
142     stack<Int*> s2(s1) ; //Now s2 and s1 will be holding same pointers
143     cout << "s2 looks like this: " ;
144     cout << s2 << endl ;
145     cout << "s2 and s1 looks like this: " << endl ;
146     cout << s2 << endl << s1 << endl ;
147     Int*& y = s1.top();
```

```

...oftware\course\code\template\templateteststack\stack_test.cpp
148 cout << "top of s1 = " << *(y) << endl ;
149 for (int i = 0 ; i < 4; i++) {
150     Int*& y = s1.top();
151     delete y ;
152     s1.pop() ;
153 }
154
155 /* s2 is holding some pointers which is removed by s1 */
156 /* DO NOT do d2 = s1 */
157
158 while(!(s1.isempty())) {
159     Int*& y = s1.top();
160     delete y ;
161     s1.pop() ;
162 }
163 /* Why we should NOT do for s2 */
164 }
165
166 -----
167 stack constructor
168 s1 looks like this: course 0 course 1 course 2 course 3 course 4
169 stack Copy constructor
170 s2 looks like this: course 0 course 1 course 2 course 3 course 4
171 s2 and s1 looks like this:
172 course 0 course 1 course 2 course 3 course 4
173 course 0 course 1 course 2 course 3 course 4
174 top of s1 = course 4
175 After poping 4 elements s1 looks like this:
176 course 0
177 s2 looks like this:
178 course 0 course 1 course 2 course 3 course 4
179 stack equal operator
180 After s2 = s1, s2 looks like this:
181 course 0
182 stack destructor
183 stack destructor
184 -----
185 static void test_string() {
186     stack<string> s1 ;
187     for (int i = 0; i < 5; i++) {
188         char a[2] ;
189         a[0] = i + '0' ;
190         a[1] = '\0' ;
191         string s = "course " ;
192         s1.push(s + a) ;
193     }
194     cout << "s1 looks like this: " ;
195     cout << s1 << endl ;
196     stack<string> s2(s1) ;

```

```
197 cout << "s2 looks like this: " ;
198 cout << s2 << endl ;
199 cout << "s2 and s1 looks like this: " << endl ;
200 cout << s2 << endl << s1 << endl ;
201 cout << "top of s1 = " << s1.top() << endl ;
202 for (int i = 0 ; i < 4; i++) {
203     s1.pop() ;
204 }
205 cout << "After popping 4 elements s1 looks like this: " << endl ;
206 cout << s1 << endl ;
207 cout << "s2 looks like this: " << endl ;
208 cout << s2 << endl ;
209 s2 = s1 ;
210 cout << "After s2 = s1, s2 looks like this: " << endl ;
211 cout << s2 << endl ;
212 }
213
214
215 /*-----*/
216
217 -----*/
218 int main() {
219     stack<int>::set_verbose(true) ; //Note this call
220     test_int_by_value() ;
221     test_int_by_address();
222     test_Int_by_value() ;
223     test_Int_by_address();
224     test_string();
225     return 0 ;
226 }
```

VASUDEVAMURTHY

Chapter 11

Exception handling

11.1 Introduction

11.2 Problem

What is the problem?

```

class data {
public:
    data(int x, int y): _x(x), _y(y) {
        cout << "In data constructor of " << _x << "," << _y << "\n" ;
    }
    ~data() {
        cout << "In data destructor of " << _x << "," << _y << "\n" ;
    }
    int _x;
    int _y;
};

```

```

static double divide(const data& d) {
    if (d._y == 0) {
        exit(0); //I cannot return valid double here */
    }
    return (double(d._x) / double(d._y));
}

```

```

static void test() {
    data d1(3,2);
    cout << d1._x << "/" << d1._y << "=" << divide(d1) << endl;
    data d2(3,0);
    cout << d2._x << "/" << d2._y << divide(d2) << endl;
    data d3(3,8);
    cout << d3._x << "/" << d3._y << divide(d3) << endl;
    /* d1 and d2 destructors are NOT called */
}

```

MEMORY LEAK

In data constructor of 3,2
 $3/2=1.5$
 In data constructor of 3,0

Figure 11.1: Cannot handle unexpected input

```
1  /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename: p1.cpp  
4 -----*/  
5  
6 #include <iostream>  
7 #include <iostream>  
8 #include <cassert>  
9 using namespace std;  
10  
11 /*-----  
12 mydata  
13 -----*/  
14 class mydata {  
15 public:  
16     mydata(int x, int y):_x(x),_y(y) { cout << "In mydata constructor of " << _x << "," << _y << "\n" ;}  
17     ~mydata() { cout << "In mydata destructor of " << _x << "," << _y << "\n" ;}  
18     int _x ;  
19     int _y ;  
20 };  
21  
22 /*-----  
23 divide  
24 -----*/  
25 static double divide(const mydata& d) {  
26     if (d._y == 0) {  
27         exit(0) ; //I cannot return valid double here */  
28     }  
29     return (double(d._x) / double(d._y)) ;  
30 }  
31  
32 /*-----  
33 In mydata constructor of 3,2  
34 3/2= 1.5  
35 In mydata constructor of 3,0  
36 -----*/  
37 static void test() {  
38     mydata d1(3,2) ;  
39     cout << d1._x << "/" << d1._y << "=" << divide(d1) << endl ;  
40     mydata d2(3,0) ;  
41     cout << d2._x << "/" << d2._y << divide(d2) << endl ;  
42     mydata d3(3,8) ;  
43     cout << d3._x << "/" << d3._y << divide(d3) << endl ;  
44     /* d1 and d2 destructors are NOT called */  
45 }  
46  
47 /*-----  
48 main
```

```
49 -----*/  
50 int main() {  
51     test();  
52     return 0 ;  
53 }
```

11.3 Try and catch

TRY and CATCH

```

class data {
public:
    data(int x, int y):_x(x),_y(y) {
        cout << "In data constructor of " << _x << "," << _y << "\n" ;
    }
    ~data() {
        cout << "In data destructor of " << _x << "," << _y << "\n" ;
    }
    int _x;
    int _y;
};

static double divide(const data& d) throw(const char*){
    if (d._y == 0) {
        throw "In divide: DIVISION BY ZERO" ;
        //I cannot return valid double here */
    }
    return (double(d._x) / double(d._y)) ;
}

static void test() {
    data d1(3,2) ;
    cout << d1._x << "/" << d1._y << "=" << divide(d1) << endl ;
    data d2(3,0) ;
    cout << d2._x << "/" << d2._y << divide(d2) << endl ;
    data d3(3,8) ;
    cout << d3._x << "/" << d3._y << divide(d3) << endl ;
    /* d1 and d2 destructors are called */
}

static void try_test() {
    try {
        test();
    }
    catch(const char* s) {
        cout << "I am in try_test: " << s << endl ;
    }
}

```

NO MEMORY LEAK

In data constructor of 3,2
3/2= 1.5

In data constructor of 3,0

In data destructor of 3,0

In data destructor of 3,2

I am in try_test: In divide: DIVI

SEPARATING TRY AND CATCH

Figure 11.2: No more memory leak

|Why C setjmp and longjmp cannot be used in C++

One of the important aspects of C++ exception handing is: Unwinding the stack.

It is certainly possible that just prior to an exception being thrown, one or more user-defined types have already been instantiated. Furthermore, let's assume that the constructor functions for these types have allocated private resources on the heap. So when the exception is thrown, what happens to these resources as the instances go out of scope?

The answer is that the stack is automatically unwound and the destructor function is guaranteed to be called, thereby negating the possibility of memory leakage.

In C there is no concept of automatic invoking of destructors. Also C library functions setjmp and longjmp has no idea of calling destructors. This makes setjmp and longjmp unsatisfactory in C++. The C techniques of signals and setjmp/longjmp do not call destructors, so objects aren't properly cleaned up. This makes it virtually impossible to effectively recover from an exceptional condition because you'll always leave objects behind that haven't been cleaned up and that can no longer be accessed

Can you have a catch block with no preceding try block?

No. Note that an exception handler, also known as a catch block, can appear only after a try block or another exception handler. There can be no intervening statements. The process of throwing and catching an exception entails searching the exception handlers after the try block for the first one that can be invoked.

Figure 11.3: Need for try and catch

```
c:\work\software\course\code\exception\p2.cpp-----1
1  /*
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename: p2.cpp
4 */
5
6 #include <iostream>
7 #include <iostream>
8 #include <cassert>
9 using namespace std;
10
11 /*
12 mydata
13 */
14 class mydata {
15 public:
16     mydata(int x, int y):_x(x),_y(y) { cout << "In mydata constructor of " << _x << "," << _y << "\n" ;}
17     ~mydata() { cout << "In mydata destructor of " << _x << "," << _y << "\n" ;}
18     int _x ;
19     int _y ;
20 };
21
22 /*
23 divide
24 */
25 static double divide(const mydata& d) throw(const char*){
26     if (d._y == 0) {
27         throw "In divide: DIVISION BY ZERO" ;
28         //I cannot return valid double here */
29     }
30     return (double(d._x) / double(d._y)) ;
31 }
32
33 /*
34 In mydata constructor of 3,2
35 3/2= 1.5
36 In mydata constructor of 3,0
37 In mydata destructor of 3,0
38 In mydata destructor of 3,2
39 I am in try_test: In divide: DIVISION BY ZERO
40 */
41 static void test() {
42     mydata d1(3,2) ;
43     cout << d1._x << "/" << d1._y << "=" << divide(d1) << endl ;
44     mydata d2(3,0) ;
45     cout << d2._x << "/" << d2._y << divide(d2) << endl ;
46     mydata d3(3,8) ;
47     cout << d3._x << "/" << d3._y << divide(d3) << endl ;
48 /* d1 and d2 destructors are called */
```

```
49 }
50
51 /*-----*
52 Separating try and catch
53 -----*/
54 static void try_test() {
55     try {
56         test();
57     }
58     catch(const char* s) {
59         cout << "I am in try_test: " << s << endl;
60     }
61 }
62
63 /*-----*
64 main
65 -----*/
66 int main() {
67     try_test();
68     return 0;
69 }
```

11.4 Stack example

```
1  /*-----  
2  Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3  Filename: error.h  
4  -----*/  
5  #ifndef ERROR_H  
6  #define ERROR_H  
7  
8  #include <iostream>  
9  using namespace std;  
10  
11 #ifdef _MSC_VER  
12 #pragma warning(disable: 4996) /* Disable deprecation */  
13 #endif  
14  
15 /*-----  
16 class error  
17 -----*/  
18 namespace vasudevamurthy {  
19     class error{  
20     public:  
21         error(const char *name = NULL, const int num = -1): _name(NULL),_num(num) {  
22             if (name) {  
23                 cout << "In error constructor\n" ;  
24                 _name = new char[strlen(name) + 1] ;  
25                 strcpy(_name,name) ;  
26             }  
27         }  
28         virtual ~error() {  
29             cout << "In error destructor\n" ;  
30             delete [] _name ;  
31         }  
32  
33         error(const error& r) {  
34             cout << "In error copy constructor\n" ;  
35             _copy(r) ;  
36         }  
37  
38         error& operator=(const error& r) {  
39             cout << "In error Equal constructor\n" ;  
40             if (this != &r) {  
41                 delete [] _name ;  
42                 _copy(r) ;  
43             }  
44             return *this ;  
45         }  
46  
47         const char* name() const {return _name ;}  
48         int num() const {return _num; }  
49         virtual void print(ostream& o, char const *s) const{
```

```

c:\work\software\course\code\exception\istack\error.h
50     o << s << "Name = " << _name << " Index = " << _num << endl ;
51 }
52
53 private:
54     char *_name ;
55     int _num ;
56
57     void _copy(const error& r) {
58         int l = strlen(r._name + 1) ;
59         _name = new char[l] ;
60         strcpy(_name,r._name) ;
61         _num = r._num ;
62     }
63 };
64
65 /*-----
66 Underflow errors when popping an empty stack happens.
67 This problem is handled here
68 -----*/
69 class error_underflow: public error{
70 public:
71     error_underflow(const char *name = NULL, const int num = -1): error
72         (name,num) {
73         cout << "In error_underflow constructor\n" ;
74     }
75
76     virtual ~error_underflow() {
77         cout << "In error_underflow destructor\n" ;
78     }
79
80     error_underflow(const error_underflow& r):error(r) {
81         cout << "In error_underflow copy constructor\n" ;
82     }
83
84     error_underflow& operator=(const error_underflow& r) {
85         cout << "In error_underflow Equal constructor\n" ;
86         if (this != &r) {
87             error::operator=(r) ;
88         }
89         return *this ;
90     }
91
92     virtual void print(ostream& o, char const *s) const {
93         o << s ;
94         error::print(o,"UNDERFLOW Condition ") ;
95     };
96
97 */

```

```
98    Overflow errors when pushing elements to a stack that is already full.
99    This problem is handled here
100   -----
101 class error_overflow: public error{
102 public:
103     error_overflow(const char *name = NULL, const int num = -1): error
104         (name,num) {
105         cout << "In error_overflow constructor\n" ;
106     }
107     virtual ~error_overflow() {
108         cout << "In error_overflow destructor\n" ;
109     }
110     error_overflow(const error_overflow& r):error(r) {
111         cout << "In error_overflow copy constructor\n" ;
112     }
113
114     error_overflow& operator=(const error_overflow& r) {
115         cout << "In error_overflow Equal constructor\n" ;
116         if (this != &r) {
117             error::operator=(r) ;
118         }
119         return *this ;
120     }
121
122     virtual void print(ostream& o, char const *s) const {
123         o << s ;
124         error::print(o,"OVERFLOW Condition ") ;
125     }
126 };
127
128 /**
129 Out of memory heap space is handled here
130 -----
131 */
132 class error_out_of_memory: public error{
133 public:
134     error_out_of_memory(const char *name = NULL, const int num = -1): error
135         (name,num) {
136         cout << "In error_out_of_memory constructor\n" ;
137     }
138     virtual ~error_out_of_memory() {
139         cout << "In error_out_of_memory destructor\n" ;
140     }
141     error_out_of_memory(const error_out_of_memory& r):error(r) {
142         cout << "In error_out_of_memory copy constructor\n" ;
143     }
144 }
```

```
c:\work\software\course\code\exception\istack\error.h
145
146     error_out_of_memory& operator=(const error_out_of_memory& r) {
147         cout << "In error_out_of_memory Equal constructor\n" ;
148         if (this != &r) {
149             error::operator=(r) ;
150         }
151         return *this ;
152     }
153
154     virtual void print(ostream& o, char const *s) const {
155         o << s ;
156         error::print(o,"OUT_OF_MEMORY Condition ") ;
157     }
158 };
159 }
160
161 #endif
162
```

```
1  /*-----*
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3 Filename: istack.h
4 -----*/
5 #ifndef ISTACK_H
6 #define ISTACK_H
7
8 #include <iostream>
9 #include <cassert>
10 using namespace std;
11
12 /*-----*
13 class istack
14 -----*/
15 namespace vasudevamurthy {
16     class istack {
17     public:
18         enum {SIZE = 100} ;
19         istack(int n = SIZE) ;
20         ~istack() ;
21         istack(const istack& s) ;
22         istack& operator=(const istack& rhs) ;
23
24         int size() const {return _sp; }
25         int capacity() const {return _size ; }
26         bool isempty() const {return _sp ? false : true ; }
27         bool isfull() const {return (_sp < _size) ? false: true ;}
28         void push(const int& x) ;
29         int& top() const ;
30         void pop() ;
31         friend ostream& operator<<(ostream& o, const istack& s) ;
32         static void set_verbose(bool x) {_display = x ; }
33
34     private:
35         int* _array ;
36         int _size ;
37         int _sp ;
38         void _copy(const istack& s) ;
39         static bool _display ;
40
41     };
42 }
43
44#endif
45
46
```

```
1  /*
2  Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy
3  Filename: istack.cpp
4  */
5
6 #include "istack.h"
7 #include "error.h"
8 #include <iostream>
9 using namespace std;
10 using namespace vasudevamurthy ;
11
12 /*
13 static function declaration
14 */
15 bool istack::_display = true ;
16
17 /*
18 Constructor
19 */
20 istack::istack(int n):_size(n),_sp(0),_array(NULL) {
21     if (_display) {
22         cout << "istack constructor" << endl ;
23     }
24     _array = new int[n] ;
25     if (!_array) {
26         throw error_out_of_memory("istack",n) ;
27     }
28 }
29
30 /*
31 destructor
32 */
33 istack::~istack() {
34     if (_display) {
35         cout << "istack destructor" << endl ;
36     }
37     delete [] _array ;
38     _size = 0 ;
39     _sp = 0 ;
40 }
41
42 /*
43 _Copy
44 */
45 void istack::_copy(const istack& s){
46     _size = s._size ;
47     _sp = s._sp ;
48     _array = new int[_size] ;
49     for (int i = 0; i < _sp ; i++) {
```

```
c:\work\software\course\code\exception\istack\istack.cpp
50     _array[i] = s._array[i] ;
51 }
52 }
53
54 /*-----*
55 Copy Constructor
56 -----*/
57 istack::istack(const istack& s){
58     if (_display) {
59         cout << "istack Copy constructor" << endl ;
60     }
61     _copy(s) ;
62 }
63
64 /*-----*
65 equal Constructor
66 -----*/
67 istack& istack::operator=(const istack& s){
68     if (_display) {
69         cout << "istack equal operator" << endl ;
70     }
71     if (this != &s) {
72         delete [] _array ;
73         _copy(s) ;
74     }
75     return *this ;
76 }
77
78 /*-----*
79 push
80 -----*/
81 void istack::push(const int& x){
82     if (isfull()) {
83         throw error_overflow("istack",_sp) ;
84     }
85     _array[_sp++] = x ;
86 }
87
88 /*-----*
89 top
90 -----*/
91 int& istack::top() const{
92     if (isempty()) {
93         throw error_underflow("istack",_sp) ;
94     }
95     return _array[_sp-1] ;
96 }
97
98 /*-----*
```

```
c:\work\software\course\code\exception\istack\istack.cpp
99  pop
100  *-----*/
101 void istack::pop() {
102     if (isfull()) {
103         throw error_overflow("istack",_sp) ;
104     }
105     _sp-- ;
106 }
107
108 /*-----
109 Overloaded print operator
110 Because operator<< is global, we need to mention namespace
111 -----*/
112 namespace vasudevamurthy {
113     ostream& operator<<(ostream& o, const istack& s) {
114         for (int i = 0; i < s._sp; i++) {
115             cout << s._array[i] << " " ;
116         }
117         return o ;
118     }
119 }
120
```

```
1  /*-----  
2 Copyright (c) 2011 Author: Jagadeesh Vasudevamurthy  
3 Filename: istack.cpp istack_test.cpp  
4  
5 -----*/  
6 #include "istack.h"  
7 #include "error.h"  
8 #include <iostream>  
9 using namespace std;  
10 using namespace vasudevamurthy ;  
11  
12 /*-----  
13  
14 -----*/  
15 static void test() {  
16     istack s1(3) ;  
17     for (int i = 0; i < 5; i++) {  
18         s1.push(i * 100) ;  
19     }  
20     cout << "s1 looks like this: " ;  
21     cout << s1 << endl ;  
22     istack s2(s1) ;  
23     cout << "s2 looks like this: " ;  
24     cout << s2 << endl ;  
25     cout << "s2 and s1 looks like this: " << endl ;  
26     cout << s2 << endl << s1 << endl ;  
27     cout << "top of s1 = " << s1.top() << endl ;  
28     for (int i = 0 ; i < 4; i++) {  
29         s1.pop() ;  
30     }  
31     cout << "After poping 4 elements s1 looks like this: " << endl ;  
32     cout << s1 << endl ;  
33     cout << "s2 looks like this: " << endl ;  
34     cout << s2 << endl ;  
35     s2 = s1 ;  
36     cout << "After s2 = s1, s2 looks like this: " << endl ;  
37     cout << s2 << endl ;  
38 }  
39  
40 /*-----  
41 istack constructor  
42 In error constructor  
43 In error_overflow constructor  
44 istack destructor <----- delete [] _array ;  
45 test_exceptionOVERFLOW Condition Name = istack Index = 3  
46 In error_overflow destructor <----  
47 In error destructor <<--- delete [] _name of error  
48 -----*/  
49 static void test_exception() {
```

c:\work\software\course\code\exception\istack\istack_test.cpp

```
50   try {
51     test() ;
52   }
53   catch (const error_underflow& a) {
54     a.print(cout,"test_exception") ;
55   }
56   catch(const error_overflow& a) {
57     a.print(cout,"test_exception") ;
58   }
59   catch(const error_out_of_memory& a) {
60     a.print(cout,"test_exception") ;
61   }
62   catch(...) {
63     cout << "How it is possible test_exception\n" ;
64   }
65 }
66
67 /*-----*/
68
69 -----*/
70 int main() {
71   istack::set_verbose(true) ;
72   test_exception();
73   return 0 ;
74 }
```

Chapter 12

Standard Template Library

12.1 Introduction

12.2 util class

c:\work\software\course\objects\stl\complex\util.h

```
1  /*-----  
2 Copyright (c) 2014 Author: Jagadeesh Vasudevamurthy  
3 file: util.h  
4 -----*/  
5  
6 /*-----  
7 include this file for all the programs  
8 -----*/  
9 #ifndef UTIL_H  
10 #define UTIL_H  
11  
12 /*-----  
13 Basic include files  
14 -----*/  
15 #include <iostream>  
16 #include <fstream>  
17  
18 #include <iomanip>      // std::setprecision  
19 using namespace std;  
20  
21 #ifdef _WIN32  
22 #include <cassert>  
23 #include <ctime>  
24 //##include "vld.h" //Comment this line, if you have NOT installed Visual leak detector  
25 #else  
26 #include <assert.h>  
27 #include <time.h>  
28 #include <math.h> //required for log2  
29 #include <string.h> //for strlen,strcat and strcpy on linux  
30 #endif  
31  
32 //sprintf: This function or variable may be unsafe. Consider using sprintf_s instead.To disable deprecation,  
33 //use _CRT_SECURE_NO_WARNINGS  
34 //To overcome above warning  
35 #ifdef _MSC_VER  
36 #pragma warning(disable: 4996) /* Disable deprecation */  
37 #endif  
38  
39 /*-----  
40 class random number generator  
41 -----*/  
42 class Random {  
43 public:  
44     Random() { srand((unsigned)time(0)); }  
45     int get_random_number(int a = 0, int b = 10000) const {  
46         int upper_bound, lower_bound;  
47         if (a < b) { upper_bound = b - a; lower_bound = a; } else if (a >= b)  
48             { upper_bound = a - b; lower_bound = b; }  
49         return rand() % (upper_bound - lower_bound) + lower_bound;  
50     }  
51 };
```

```
c:\work\software\course\objects\stl\complex\util.h
    { upper_bound = a - b; lower_bound = b; }
48
49     return(lower_bound + rand() % upper_bound);
50 }
51 /* no body can copy random or equal random */
52 Random(const Random& x) = delete;
53 Random& operator=(const Random& x) = delete;
54 private:
55
56 };
57
58 /*-----
59 All external here
60 -----*/
61 extern int Strcmp(const char* s1, const char* s2);
62 extern void Strcpy(char* s1, const char* s2);
63 extern void print_integer(const int& x);
64 extern void print_integer(const int*& x);
65 extern void print_integer(int& x);
66 extern void print_integer(int*& x);
67 extern int intAscending_order(const int& c1, const int& c2);
68 extern int intAscending_order(int* const& c1, int* const& c2);
69 extern int intDescending_order(const int& c1, const int& c2);
70 extern int intDescending_order(int* const& c1, int* const& c2);
71 extern void delete_int(int*& c);
72 extern void delete_charstar(char*& c);
73 extern int charcompare(const char& c1, const char& c2);
74 extern void print_char(char& c);
75 extern void print_string(char*& c);
76 extern void free_string(char*& c);
77 extern int stringDescending_order(char* const& c1, char* const& c2);
78 extern int stringAscending_order(char* const& c1, char* const& c2);
79
80
81 #endif
82
83 //EOF
84
85
```

```
1  /*
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3  Filename: util.cpp
4  */
5 #include "util.h"
6
7 /*
8 strcmp
9 */
10 int Strcmp(const char* s1, const char* s2) {
11     for (; *s1 == *s2; ++s1, ++s2)
12         if (*s1 == 0) {
13             return 0;
14         }
15     return *(unsigned char *)s1 < *(unsigned char *)s2 ? -1 : 1;
16 }
17
18 /*
19 strcpy
20 */
21 void Strcpy(char* s1, const char* s2) {
22     int i = 0;
23     while (1) {
24         s1[i] = s2[i];
25         if (s2[i] == '\0') {
26             return;
27         }
28         i++;
29     }
30 }
31
32 /*
33 Print an integer
34 */
35 void print_integer(const int& x) {
36     cout << x << " ";
37 }
38
39 /*
40 Print an integer
41 */
42 void print_integer(const int*& x) {
43     cout << *x << " ";
44 }
45
46 /*
47 Print an integer
48 */
49 void print_integer(int& x) {
```

```
50     cout << x << " ";
51 }
52
53 /*-----
54 Print an integer
55 -----*/
56 void print_integer(int*& x) {
57     cout << *x << " ";
58 }
59
60 /*-----
61 7 9 1
62 9 7 -1
63 -----*/
64 int intAscendingOrder(const int& c1, const int& c2) {
65     if (c1 == c2) {
66         return 0;
67     }
68     if (c1 < c2) {
69         return 1;
70     }
71     return -1;
72 }
73
74 /*-----
75 pointer
76 -----*/
77 int intAscendingOrder(int* const& c1, int* const& c2) {
78     return intAscendingOrder(*c1, *c2);
79 }
80
81 /*-----
82 7 9 -1
83 9 7 1
84 -----*/
85 int intDescendingOrder(const int& c1, const int& c2) {
86     int x = intAscendingOrder(c1, c2);
87     return -x;
88 }
89
90 /*-----
91 pointer
92 -----*/
93 int intDescendingOrder(int* const& c1, int* const& c2) {
94     return intDescendingOrder(*c1, *c2);
95 }
96
97 /*-----
98 Delete a int object
```

```
C:\work\software\course\objects\stl\complex\util.cpp
99  -----
100 void delete_int(int*& c) {
101     delete(c);
102 }
103
104 /**
105 Delete a char * object
106 */
107 void delete_charstar(char*& c) {
108     delete[] c;
109 }
110
111 /**
112 char compare
113 */
114 int charcompare(const char& c1, const char& c2) {
115     if (c1 == c2) {
116         return 0;
117     }
118     if (c1 > c2) {
119         return 1;
120     }
121     return -1;
122 }
123
124 /**
125 char print
126 */
127 void print_char(char& c) {
128     cout << c << " ";
129 }
130
131 /**
132 string print
133 */
134 void print_string(char*& c) {
135     cout << c << " ";
136 }
137
138 /**
139 Delete c which is allocated by new []
140 */
141 void free_string(char*& c) {
142     delete[] c;
143 }
144
145 /**
146 henry  zoo  is in descending order: -1
147 tom    idiot is in decending order:   1
```

```
148 -----*/
149 int string_descending_order(char* const& c1, char* const& c2) {
150     int x = strcmp(c1, c2);
151     return x;
152 }
153
154 /*-----
155 henry  zoo  is in ascending order: -1   1
156 tom    idiot is in ascending order:  1  -1
157 -----*/
158 int stringAscending_order(char* const& c1, char* const& c2) {
159     int x = string_descending_order(c1, c2);
160     return -x;
161 }
162
163
```

12.3 complex class

```

1  /*-----
2 Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3 Filename: complex.h
4 -----*/
5 #ifndef complex_h
6 #define complex_h
7
8 #include "util.h"
9
10 class complex {
11 public:
12     complex(int x = 0, int y = 0);
13     ~complex();
14     complex(const complex& b);
15     complex& operator=(const complex& rhs);
16     friend bool operator==(const complex& c1, const complex& c2);
17     friend bool operator!=(const complex& c1, const complex& c2);
18     friend bool operator<(const complex& s1, const complex& s2);
19     friend ostream& operator<<(ostream& o, const complex& c);
20     friend ostream& operator<<(ostream& o, complex* const& c);
21     void setxy(int x, int y) { _x = x; _y = y; _release(); _buildstring(); }
22     void getxy(int& x, int &y) const { x = _x; y = _y; }
23     bool display() const { return _display; }
24     static void set_display(bool x) {
25         _display = x;
26     }
27
28 private:
29     int _x;
30     int _y;
31     char* _string;
32     static bool _display; /* ONLY ONCE for all object */
33     void _alloc(int l) {
34         _string = new char[l];
35     }
36     void _release() {
37         delete[] _string;
38     }
39     void _copy(const complex& from);
40     void _buildstring();
41 };
42
43 /*-----
44 extern functions
45 -----*/
46 extern void print_complex(const complex& c);
47 extern void print_complex(const complex*& c);
48 extern void print_complex(complex& c);
49 extern void print_complex(complex*& c);

```

25

-4

→ 25-i4\0

Data structure

```
c:\work\software\course\objects\stl\complex\complex.h
50 extern int complex_larger_compare( const complex& c1, const complex& c2);
51 extern int complex_smaller_compare( const complex& c1, const complex& c2);
52 extern int complex_larger_compare( complex* const& c1, complex* const& c2);
53 extern int complex_smaller_compare( complex* const& c1, complex* const& c2);
54 extern void delete_complex(complex*& c);
55 extern int complexAscending_order( const complex& c1, const complex& c2);
56 extern int complexAscending_order( complex* const& c1, complex* const& c2);
57 extern int complexDescending_order( const complex& c1, const complex& c2);
58 extern int complexDescending_order( complex* const& c1, complex* const& c2);
59
60 #endif
61
62
```

```
1  /*-----  
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3  Filename: complex.cpp  
4  compile: g++ complex.cpp  
5  Memory leaked: 0 bytes  
6  -----*/  
7  #include "complex.h"  
8  
9  /*-----  
10 static definition - only once at the start  
11 Change to false, if you don't need verbose  
12 -----*/  
13 bool complex::_display = true;  
14  
15  
16 /*-----  
17 Constructor  
18 -----*/  
19 complex::complex(int x, int y) :_x(x), _y(y), _string(nullptr) {  
20     if (display()) {  
21         cout << "in complex constructor:" << endl;  
22     }  
23     _buildstring();  
24 }  
25  
26 /*-----  
27 Destructor  
28 -----*/  
29 complex::~complex() {  
30     if (display()) {  
31         cout << "In complex Destructor " << _string << endl;  
32     }  
33     _release();  
34 }  
35  
36 /*-----  
37 Helper: copy function  
38 -----*/  
39 void complex::_copy(const complex& from) {  
40     _x = from._x;  
41     _y = from._y;  
42     int l = strlen(from._string) + 1;  
43     _alloc(l);  
44     strcpy(_string, from._string);  
45 }  
46  
47 /*-----  
48 Copy constructor  
49 -----*/
```

```

C:\work\software\course\objects\stl\complex\complex.cpp
50 complex::complex(const complex& b) {
51     if (display()) {
52         cout << "In complex copy Constructor " << b._string << endl;
53     }
54     _copy(b);
55 }
56
57 /*-----*
58 equal operator
59 -----*/
60 complex& complex::operator=(const complex& rhs) {
61     if (rhs.display()) {
62         cout << "In complex equal operator " << rhs._string << endl;
63     }
64     if (this != &rhs) {
65         _release();
66         _copy(rhs);
67     }
68     return *this;
69 }
70
71 /*-----*
72 Equal Equal operator
73 -----*/
74 bool operator==(const complex& c1, const complex& c2) {
75     if (c1.display()) {
76         cout << "in complex == operator \n";
77     }
78     return ((c1._x == c2._x) && (c1._y == c2._y)) ? true : false;
79 }
80
81 /*-----*
82 NOT Equal operator
83 -----*/
84 bool operator!=(const complex& c1, const complex& c2) {
85     if (c1.display()) {
86         cout << "in complex != operator \n";
87     }
88     return (!(c1 == c2));
89 }
90
91 /*-----*
92 < operator is used by set, map and hashmap
93 -----*/
94 bool operator<(const complex& s1, const complex& s2) {
95     int x1, y1;
96     s1.getxy(x1, y1);
97     int x2, y2;
98     s2.getxy(x2, y2);

```

```
99     return (x1 < x2) ? true : false;
100 }
101
102 /*-----
103 Print
104 -----*/
105 ostream& operator<<(ostream& o, const complex& c) {
106     o << c._string;
107     return o;
108 }
109
110 /*-----
111 Print
112 -----*/
113 ostream& operator<<(ostream& o, complex* const& c) {
114     return (o << *c);
115 }
116
117 /*-----
118 build a string from x and y
119 2, 3 == 2+i3
120 2,-200 == 2-i200 ;
121 -2,4 == -2+i4;
122 -2,-4 == -2-i4;
123
124 strlen() returns the length of a string, excluding the null
125 char *j = "jag" ;
126 strlen = 3; j[0] = j j[1] = a j[2] = g ; j[3] = '\0' ;
127 */
128 void complex::_buildstring() {
129     char sx[20], sy[20];
130     sprintf(sx, "%d", _x);
131     char s[20];
132     s[0] = '\0';
133     strcat(s, sx);
134     int m = _y;
135     if (_y < 0) {
136         m = -(_y);
137         strcat(s, "-i");
138     } else {
139         strcat(s, "+i");
140     }
141     sprintf(sy, "%d", m);
142     strcat(s, sy);
143     int l = strlen(s) + 1;
144     _alloc(l);
145     strcpy(_string, s);
146 }
```

We can print from
complex*

C:\work\software\course\objects\stl\complex\complex.cpp

```

148  /*
149   print a complex number
150  -----
151 void print_complex(const complex& c) {
152     cout << c << " ";
153 }
154
155  /*
156 print a complex number
157 -----
158 void print_complex(const complex*& c) {
159     cout << *c << " ";
160 }
161
162  /*
163 print a complex number
164 -----
165 void print_complex(complex& c) {
166     cout << c << " ";
167 }
168
169  /*
170 print a complex number
171 -----
172 void print_complex(complex*& c) {
173     cout << *c << " ";
174 }
175
176  /*
177 c1 > c2
178 -----
179 int complex_larger_compare(const complex& c1, const complex& c2) {
180     int r1, i1, r2, i2;
181     c1.getxy(r1, i1);
182     c2.getxy(r2, i2);
183
184     if ((r1 == r2) && (i1 == i2)) {
185         return 0;
186     }
187     if ((r1 > r2) && (i1 > i2)) {
188         return 1;
189     }
190     return -1;
191 }
192
193  /*
194 c1 < c2
195 -----
196 int complex_smaller_compare(const complex& c1, const complex& c2) {

```

Users can register these functions

```
C:\work\software\course\objects\stl\complex\complex.cpp
197     int x = complex_larger_compare(c1, c2);
198     return -x;
199 }
200
201 /*-----
202 c1 > c2
203 complex* const& c1
204 Constant pointer, non constant data
205 -----*/
206 int complex_larger_compare(complex* const& c1, complex* const& c2) {
207     return (complex_larger_compare(*c1, *c2));
208 }
209
210 /*-----
211 c1 < c2
212 complex* const& c1
213 Constant pointer, non constant data
214 -----*/
215 int complex_smaller_compare(complex* const& c1, complex* const& c2) {
216     return (complex_smaller_compare(*c1, *c2));
217 }
218
219 /*-----
220 Delete a complex object
221 -----*/
222 void delete_complex(complex*& c) {
223     delete(c);
224 }
225
226 /*-----
227 -----*/
228 /*-----
229 int complexAscending_order(const complex& c1, const complex& c2) {
230     int r1, i1, r2, i2;
231     c1.getxy(r1, i1);
232     c2.getxy(r2, i2);
233     int x1 = r1 + i1;
234     int x2 = r2 + i2;
235     if (x1 == x2) {
236         return 0;
237     }
238     if (x1 < x2) {
239         return 1;
240     }
241     return -1;
242 }
243
244 /*-----*/
```

```
C:\work\software\course\objects\stl\complex\complex.cpp
246  -----
247  int complexAscendingOrder( complex* const& c1, complex* const& c2) {
248      return (complexAscendingOrder(*c1, *c2));
249  }
250
251  /*
252  -----
253  */
254  int complexDescendingOrder( const complex& c1, const complex& c2) {
255      int x = complexAscendingOrder(c1, c2);
256      return -x;
257  }
258
259  /*
260  -----
261  */
262  int complexDescendingOrder( complex* const& c1, complex* const& c2) {
263      return (complexDescendingOrder(*c1, *c2));
264  }
265
266
267 //EOF
```

12.4 vector class

Constructors		
<code>vector<T> a</code>	Default constructor	$O(1)$
<code>vector<T> a(size)</code>	Default constructor with explicit size	$O(\text{size})$
<code>vector<T> a(size,T)</code>	Default constructor with explicit size and initial value	$O(\text{size})$
Size		
<code>a.size()</code>	Number of elements currently held in vector a	$O(1)$
<code>a.capacity()</code>	Maximum elements vector a can hold	$O(1)$
<code>a.clear()</code>	All elements are removed size = 0 and capacity = unchanged	$O(n)$
Element access		
<code>a[i]</code> Returns object by alias		
<code>a[i]</code>	Access i th element of array a i must be less than size of a boundary checking is not done	$O(1)$
<code>a.at[i]</code>	Access i th element of array a Boundary check is done before access. Throws exception if $(i < 0) \text{ } (i \geq \text{size})$	$O(1)$
<code>a.push_back(i)</code>	Add element to the end of the array a.	$O(1)$ OR $O(n)$
<code>a.pop_back()</code>	Remove element from the end of the array a.	$O(1)$

Figure 12.1: `vector` class

12.4. VECTOR CLASS

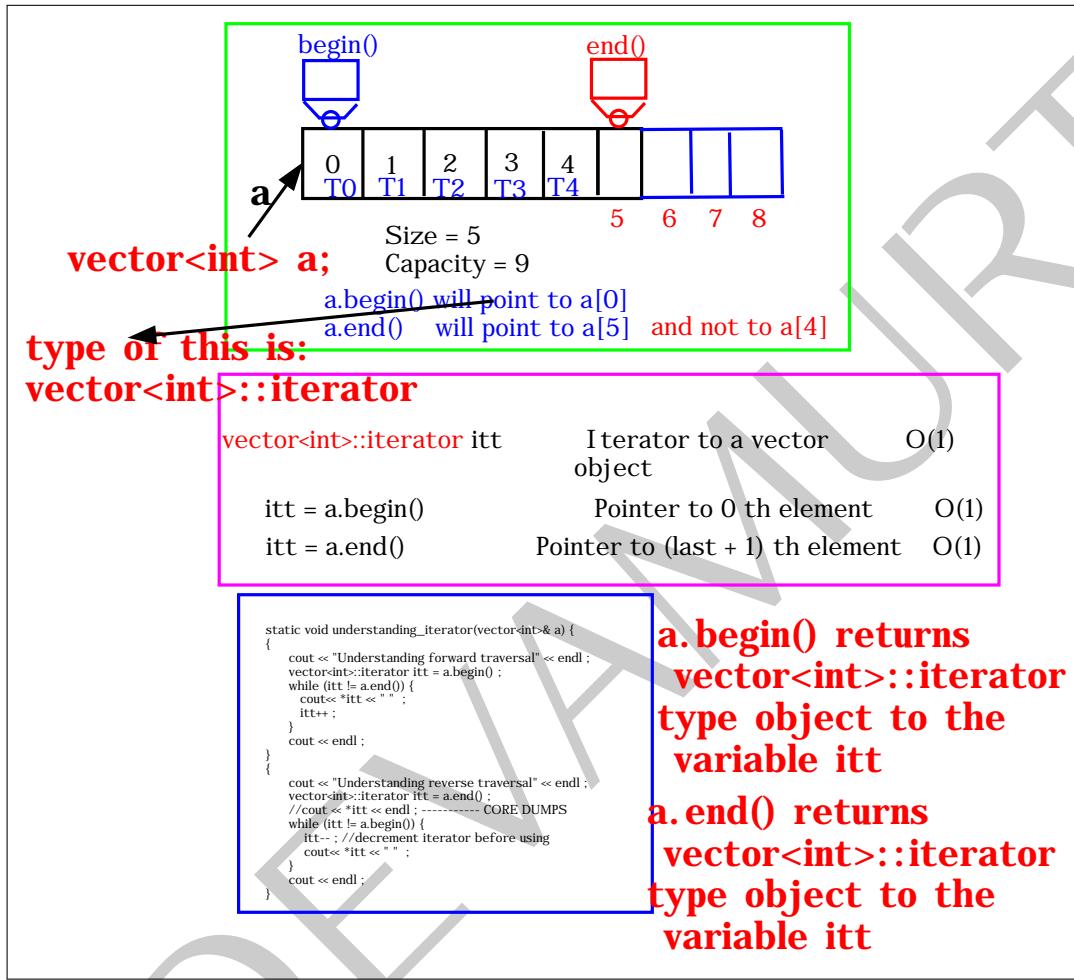


Figure 12.2: Concept of iterators

```

1  /*
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3  Filename: svector.cpp
4  compile: g++ svector.cpp ../complex/complex.cpp ../complex/util.cpp
5  Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
6  */
7  #include <vector>
8  #include <stdexcept> //Without this catch will NOT work on Linux
9  #include "../complex/complex.h"
10
11 /*
12 Multiply integer by 10
13 */
14 static void multiply_by_10(int& x) {
15     x = x * 10;
16 }
17
18 /*
19 apply a function pf on elements of vector a
20 */
21 template <typename T>
22 static void apply(const char* s, vector<T>& a, void(*pf)(T& x)) {
23     cout << s << endl;
24     cout << "-----" << endl;
25     auto itt = a.begin();
26     while (itt != a.end()) {
27         T& p = *itt;
28         pf(p);
29         itt++;
30     }
31     cout << endl;
32     /* apply using range */
33     for (auto& p : a) {
34         pf(p);
35     }
36 }
37
38 /*
39 a[0] ..... a[9]
40
41 begin() will point to a[0]
42 end() will NOT POINT to a[9], but to one element past a[9]
43 That means real end is: end()-1 ;
44 */
45 static void understanding_iterator(vector<int>& a) {
46 {
47     cout << "Understanding forward traversal" << endl;
48     vector<int>::iterator itt = a.begin();
49     while (itt != a.end()) {

```

pf is a function ptr.

The function:

output: void

input: An object T by reference

example:

void multiply_by_10(complex& x){
}

543

```

C:\work\software\course\objects\stl\vector\svector.cpp
50     cout << *itt << " ";
51     itt++;
52 }
53 cout << endl;
54 }
55 {
56     cout << "Understanding forward traversal using auto" << endl;
57     auto itt = a.begin();
58     while (itt != a.end()) {
59         cout << *itt << " ";
60         itt++;
61     }
62     cout << endl;
63 }
64 {
65     cout << "Understanding forward traversal using range" << endl;
66     for (const auto& i : a) {
67         cout << i << " ";
68     }
69     cout << endl;
70 }
71 {
72 {
73     cout << "Understanding reverse traversal" << endl;
74     vector<int>::iterator itt = a.end();
75     //cout << *itt << endl ; ----- CORE DUMPS
76     while (itt != a.begin()) {
77         itt--; //decrement iterator before using
78         cout << *itt << " ";
79     }
80     cout << endl;
81 }
82 {
83     cout << "Understanding reverse traversal using auto" << endl;
84     auto itt = a.end();
85     //cout << *itt << endl ; ----- CORE DUMPS
86     while (itt != a.begin()) {
87         itt--; //decrement iterator before using
88         cout << *itt << " ";
89     }
90     cout << endl;
91 }
92 cout << "-----" << endl;
93 }
94
95 /*-----
96 Understanding access using
97 1. a[i]
98 2. a.at(i) -- Same as a[i], but does out of bound checks

```

```
99 and throws exception that you can catch
100
101 -----*/
102 template <typename T>
103 static void understanding_access(const vector<T>& a, int i) {
104     //int p = a[i] ; //core dumps
105     try {
106         int p = a.at(i); //Throws exception. out of bound checks
107
108     } catch (std::out_of_range) {
109         cout << " accessing a[" << i << "] but size of array is " << a.size() << endl;
110     }
111     cout << "-----" << endl;
112 }
113
114 /*-----
115 print1
116 -----*/
117 template <typename T>
118 static void print1(const char* s, const vector<T>& a) {
119     cout << s << endl;
120     cout << "-----" << endl;
121     cout << "size = " << a.size() << " ";
122     cout << "capacity = " << a.capacity() << " ";
123     cout << endl;
124     for (int i = 0; i < int(a.size()); i++) {
125         cout << "a[" << i << "] = " << a[i] << " ";
126     }
127     cout << endl;
128     cout << "-----" << endl;
129 }
130
131 /*-----
132 basic
133 -----*/
134 static void basic() {
135     vector<int> a;
136     print1("begin with", a);
137     for (int i = 0; i < 5; i++) {
138         a.push_back(i);
139     }
140     print1("After Inserting 5 elements", a);
141     understanding_iterator(a);
142
143     understanding_access(a, 6);
144
145     a.pop_back();
146     a.pop_back();
```

```

C:\work\software\course\objects\stl\vector\svector.cpp
147     print1("After deleting last 2 elements", a);
148
149     apply("multiply by 10", a, multiply_by_10);
150     cout << endl;
151     apply("print using iterator", a, print_integer);
152     cout << endl;
153
154     a.clear();
155     print1("Using clear", a);
156 }
157
158 /*-----
159 questions
160 -----*/
161 static void questions() {
162     vector<int> a(10, -1);
163     print1("Is it -1", a);
164     cout << "I am reading without assigning a[5] = " << a[5] << endl;
165
166     complex c(99, -420);
167     vector<complex> b(6, c);
168     print1("Is it 99,-420", b);
169     cout << "I am reading without assigning b[5] = " << b[5] << endl;
170
171 //cout << b[7] << endl ; -- Crashes because you need to push_back before U ↵
172     access
173 }
174
175 /*-----
176 problem
177
178 -----*/
179 static void problem() {
180     vector<complex> vc;
181     print1("begin with ", vc);
182     vc.push_back(0);
183     vc.push_back(1);
184     vc.push_back(2);
185
186     print1("After Inserting 0, 1 and 2 ", vc);
187
188     complex& c1 = vc[1];
189     cout << " c1 = " << c1 << endl;
190
191     for (int i = 3; i < 10; ++i) {
192         vc.push_back(i);
193     }
194     print1("After Inserting 3 to 10 ", vc);

```

MORAL: DO NOT STORE ALIAS TO THE CONTAINER OBJECT

Dynamic array

```
195
196 //WHAT HAPPENS HERE ??????????????
197 if (0) {
198     cout << " c1 = " << c1 << endl;
199 }
200 }
201
202 /*-----*
203 array of integer pointers
204 -----*/
205 static void test_vector_of_ptr_integers(int n) {
206     vector<int*> a;
207     for (int i = 0; i < n; i++) {
208         a.push_back(new(int)(i * 10));
209     }
210     vector<int*> b(a);
211     apply("Contents of vector b", b, print_integer);
212     bool equal = true;
213     for (int i = 0; i < n; i++) {
214         if (*(a[i]) != *(b[i])) {
215             equal = false;
216             break;
217         }
218     }
219     equal ? cout << " array a == b\n " : cout << " array a != b\n ";
220 //YOU CANNOT DO THIS with vector
221 //cout << "We have not inserted 25th element. Let us see what we get " <<
222 //    endl ;
223     for (int i = 0; i < n; i++) {
224         /* we allocated contents in a. So we delete it */
225         delete(a[i]);
226     }
227     /* Why you should NOT do this */
228     /*
229     for (int i = 0; i < n; i++) {
230         delete(b[i]) ;
231     }
232     */
233 }
234
235 /*-----*
236 array of user defined type
237 -----*/
238 static void test_vector_of_udt() {
239     const int N = 5;
240     vector<complex> a(N); N is just starting size. You can
241     for (int i = 0; i < N; i++) add more than N. Dynamic array
242         a[i].setxy(i, -i);
```

```

243     }
244     vector<complex> b(a);
245     apply("Contents of vector b", b, print_complex);
246     bool equal = true;
247     for (int i = 0; i < N; i++) {
248         if (a[i] != b[i]) {
249             equal = false;
250             break;
251         }
252     }
253     equal ? cout << " array a == b\n" : cout << " array a != b\n";
254     for (int i = 0; i < 3; i++) {
255         complex c((i + 100), -(i + 100));
256         a.push_back(c);
257     }
258     apply("Contents of vector a", a, print_complex);
259     b = a;
260 }
261
262 /*-----*
263 array of user defined pointer type
264 -----*/
265 static void test_vector_of_ptr_udt(int n) {
266     vector<complex*> a;
267     for (int i = 0; i < n; i++) {
268         a.push_back(new(complex)(i, -i));
269     }
270     vector<complex*> b(a);
271     apply("Contents of vector b", b, print_complex);
272     bool equal = true;
273     for (int i = 0; i < n; i++) {
274         if (*(a[i]) != *(b[i])) {
275             equal = false;
276             break;
277         }
278     }
279     equal ? cout << " array a == b\n" : cout << " array a != b\n";
280     for (int i = 0; i < n; i++) {
281         /* we allocated contents in a. So we delete it */
282         delete(a[i]);
283     }
284     /* Why you should NOT do this */
285     /*
286     for (int i = 0; i < n; i++) {
287         delete(b[i]);
288     }
289     */
290 }
291

```

```
292  /*-----  
293  main  
294  -----*/  
295  int main() {  
296      basic();  
297      questions();  
298      problem();  
299      test_vector_of_ptr_integers(5);  
300      test_vector_of_udt();  
301      test_vector_of_ptr_udt(5);  
302      return 1;  
303  }  
304  
305
```

12.5 string class

```
1  /*-----  
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3  Filename: string.cpp  
4  compile: g++ sting.cpp  
5  -----*/  
6  #include <iostream>  
7  #include <string>  
8  #include <vector>  
9  
10 using namespace std;  
11 #ifdef _MSC_VER  
12 #pragma warning(disable: 4996) /* Disable deprecation */  
13 #endif  
14  
15 /*-----  
16 print1  
17 -----*/  
18 static void print1(const char* title, const string& s) {  
19     cout << title ;  
20     cout << s << endl ;  
21     cout << "size = " << s.size() << endl ;  
22 }  
23  
24 /*-----  
25 basic tests  
26 s4 is an empty string  
27 s = : abcd  
28 size = 4  
29 s[3] = d  
30 s1 = : abcd  
31 size = 4  
32 s1 == s  
33 s != abc  
34 s = : abcdbabu  
35 size = 8  
36 s = : abcdbabuz  
37 size = 9  
38 s = : abcdbabuzT  
39 size = 10  
40 s = : abcdbabuzT78  
41 size = 12  
42 s after clear:  
43 size = 0  
44 -----*/  
45 static void test_basic(){  
46     string s4 ;  
47     /* How to test the string is empty */  
48     if (s4.empty()) {  
49         cout << "s4 is an empty string" << endl ;
```

```

50    }
51    string s = "abcd" ;
52    print1("s = : ",s) ;
53    cout << "s[3] = " << s[3] << endl ;
54    string s1(s) ;
55    print1("s1 = : ",s1) ;
56    if (s1 == s) {
57        cout << "s1 == s" << endl ;
58    }
59    if (s != "abc") {
60        cout << "s != abc" << endl ;
61    }
62    s = s + "babu" ;
63    print1("s = : ",s) ;
64    s = s + 'z' ;
65    print1("s = : ",s) ;
66    s.append("T") ;
67    print1("s = : ",s) ;
68    /* How to append an integer to string */
69    int x = 78 ;
70    char temp[10] ;
71    sprintf(temp,"%d",x) ;
72    s = s + temp ;
73    print1("s = : ",s) ;
74    /* How to clear a string */
75    s.clear() ;
76    print1("s after clear: ",s) ;
77 }
78
79 /*-----
80 testing cin
81
82 Enter your name: jag murthy phd
83 s after getline >> s: jag murthy phd
84 size = 14
85 Enter your Country: san jose ca
86 s after getline >> s: san jose ca
87 size = 11
88 -----*/
89 static void test_cin() {
90     string s ;
91     cout << "Enter your name: " ;
92     getline(cin,s) ;
93     print1("s after getline >> s: ",s);
94     cout << "Enter your Country: " ;
95     getline(cin,s) ;
96     print1("s after getline >> s: ",s);
97 }
98

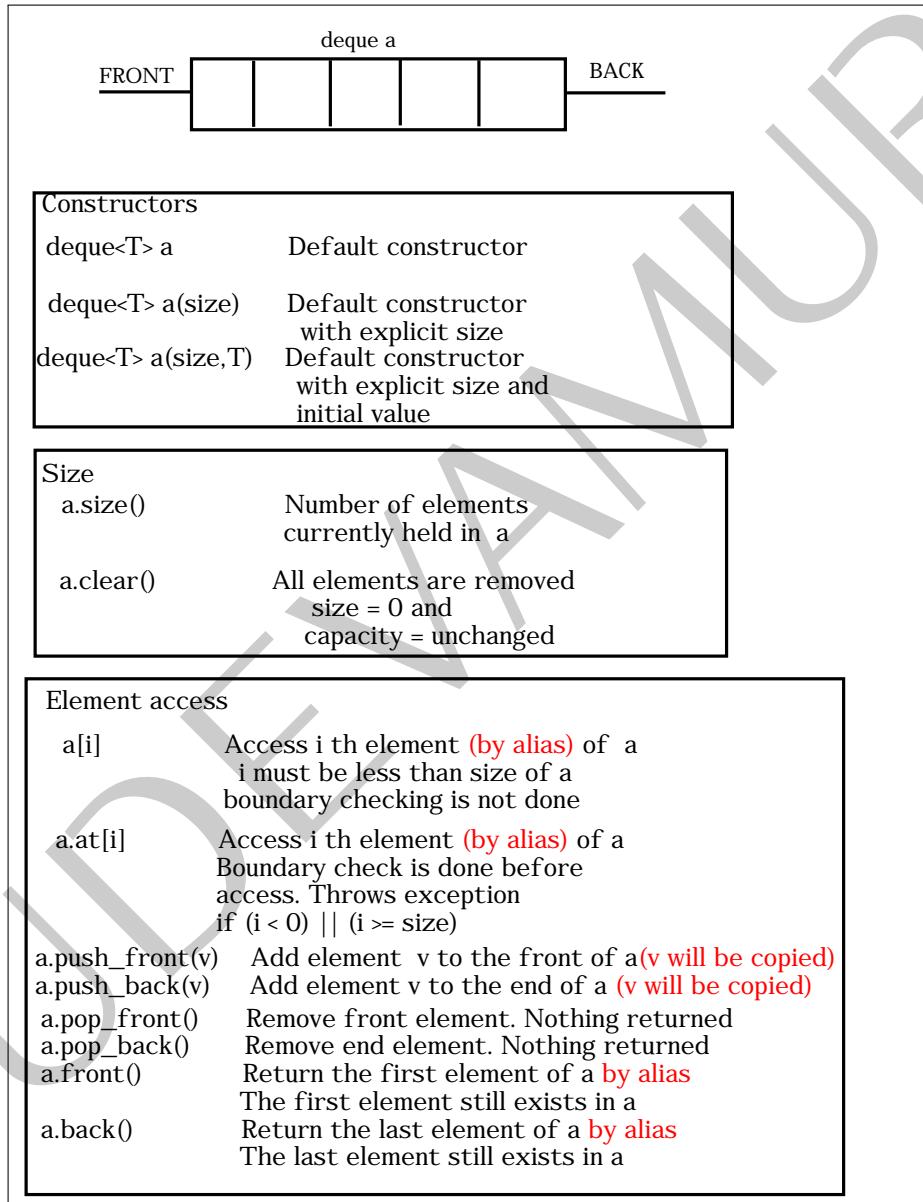
```

```
99  /*-----
100 testing iterators
101
102 a = : A quick brown fix jumps over a lazy dog
103 size = 39
104 Understanding forward traversal
105 A q u i c k   b r o w n   f i x   j u m p s   o v e r   a   l a z y   d o g
106 Understanding reverse traversal
107 g o d   y z a l   a   r e v o   s p m u j   x i f   n w o r b   k c i u q   A
108 -----*/
109 static void test_iterators(){
110     string a("A quick brown fix jumps over a lazy dog") ;
111     print1("a = : ",a) ;
112     {
113         cout << "Understanding forward traversal" << endl ;
114         string::iterator itt = a.begin() ;
115         while (itt != a.end()) {
116             cout<< *itt << " " ;
117             itt++ ;
118         }
119         cout << endl ;
120     }
121     {
122         cout << "Understanding forward traversal using auto" << endl ;
123         auto itt = a.begin() ;
124         while (itt != a.end()) {
125             cout<< *itt << " " ;
126             itt++ ;
127         }
128         cout << endl ;
129     }
130     {
131         cout << "Understanding forward traversal using range" << endl ;
132         for(const char& c:a) {
133             cout << c << " " ;
134         }
135         cout << endl ;
136     }
137     {
138         cout << "Understanding reverse traversal" << endl ;
139         string::iterator itt = a.end() ;
140         //cout << *itt << endl ; ----- CORE DUMPS
141         while (itt != a.begin()) {
142             itt-- ; //decrement iterator before using
143             cout<< *itt << " " ;
144         }
145         cout << endl ;
146     }
147 }
```

```
C:\work\software\course\objects\stl\string\string.cpp
148     cout << "Understanding reverse traversal using auto" << endl ;
149     auto itt = a.end() ;
150     //cout << *itt << endl ; ----- CORE DUMPS
151     while (itt != a.begin()) {
152         itt-- ; //decrement iterator before using
153         cout<< *itt << " " ;
154     }
155     cout << endl ;
156 }
157 }
158
159 /*-----
160 testing find
161
162 str: Hello, can you find UCSC ext? at Santa clara UCSC
163 size = 49
164 First occurrence of UCSC was found at: 20
165 -----*/
166 static void test_find() {
167     string str("Hello, can you find UCSC ext? at Santa clara UCSC");
168     string::size_type position = str.find("UCSC");
169     print1("str: ",str) ;
170     cout << "First occurrence of UCSC was found at: " << position << endl;
171 }
172
173 /*-----
174 "A quick brown fix jumps over a lazy dog"
175 token[0] = A
176 token[1] = quick
177 token[2] = brown
178 token[3] = fix
179 token[4] = jumps
180 token[5] = over
181 token[6] = a
182 token[7] = lazy
183 token[8] = dog
184 -----*/
185 static int tokenize(const string& str, vector<string>& tokens, const string& delimiters = " ") {
186     // Skip delimiters at beginning.
187     string::size_type lastPos = str.find_first_not_of(delimiters, 0);
188     // Find first "non-delimiter".
189     string::size_type pos      = str.find_first_of(delimiters, lastPos);
190
191     while (string::npos != pos || string::npos != lastPos) {
192         // Found a token, add it to the vector.
193         tokens.push_back(str.substr(lastPos, pos - lastPos));
194         // Skip delimiters. Note the "not_of"
195         lastPos = str.find_first_not_of(delimiters, pos);
```

```
C:\work\software\course\objects\stl\string\string.cpp
196     // Find next "non-delimiter"
197     pos = str.find_first_of(delimiters, lastPos);
198 }
199 return tokens.size() ;
200 }
201
202 /*-----
203 "UCSC extension|UCSC|California|United States of America|North America|World", ↵
204   |
205 token[0] = UCSC extension
206 token[1] = UCSC
207 token[2] = California
208 token[3] = United States of America
209 token[4] = North America
210 token[5] = World
211 -----*/
212 static void test_tokenizer(const string& str, const string& delimiters = " ") {
213     vector<string> tokens;
214     int size = tokenize(str, tokens, delimiters);
215     auto itt = tokens.begin();
216     int i = 0;
217     while (itt != tokens.end()) {
218         string& s = *itt;
219         cout << "token[" << i++ << "] = " << s << endl;
220         itt++;
221     }
222 }
223
224
225 /*-----
226 main
227 -----*/
228 int main() {
229     test_basic();
230     test_cin();
231     test_iterators();
232     test_find();
233     test_tokenizer("A quick brown fix jumps over a lazy dog");
234     test_tokenizer("UCSC extension|UCSC|California|United States of America|North ↵
235       America|World", "|");
236     return EXIT_SUCCESS;
237 }
```

12.6 deque class

Figure 12.3: **deque** class

12.6. DEQUE CLASS

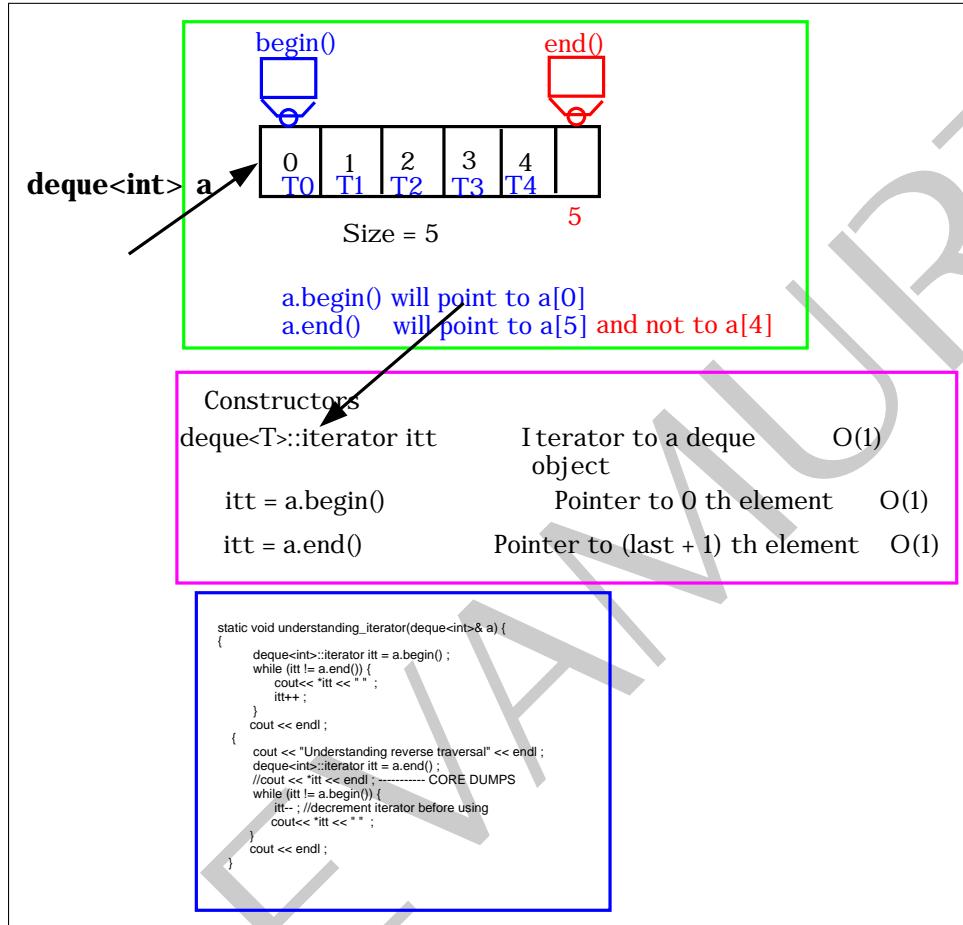


Figure 12.4: `deque` class

```
1  /*
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3  Filename: sdeque.cpp
4  compile: g++ ../complex/util.cpp ../complex/complex.cpp sdeque.cpp
5  Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
6  */
7 #include <iostream>
8 #include <deque>
9 #include <stdexcept> //Without this catch will NOT work on Linux
10
11
12 #ifdef _WIN32
13 #include "..\complex\complex.h"
14 #else
15 #include "../complex/complex.h"
16 #endif
17
18 /*
19 Multiply integer by 10
20 */
21 static void multiply_by_10(int& x) {
22     x = x * 10;
23 }
24
25 /*
26 print1
27 */
28 template <typename T>
29 static void print1(const char* s, const deque<T>& a) {
30     cout << s << endl;
31     cout << "-----" << endl;
32     cout << "size = " << a.size() << " ";
33     cout << endl;
34     for (int i = 0; i < int(a.size()); i++) {
35         cout << "a[" << i << "] = " << a[i] << " ";
36     }
37     cout << endl;
38     cout << "-----" << endl;
39 }
40
41 /*
42 apply a function pf on elements of deque a
43 */
44 template <typename T>
45 static void apply(const char* s, deque<T>& a, void(*pf)(T& x)) {
46     cout << s << endl;
47     cout << "-----" << endl;
48     for (auto& p : a) {
49         pf(p);
```

C:\work\software\course\objects\stl\deque\sdeque.cpp

```

50     }
51     cout << endl;
52 }
53
54 /*-----*
55 a[0] ..... a[9]
56
57 begin() will point to a[0]
58 end() will NOT POINT to a[9], but to one element past a[9]
59 That means real end is: end()-1 ;
60 -----*/
61 static void understanding_iterator(deque<int>& a) {
62 {
63     cout << "Understanding forward traversal" << endl;
64     deque<int>::iterator itt = a.begin();
65     while (itt != a.end()) {
66         cout << *itt << " ";
67         itt++;
68     }
69     cout << endl;
70 }
71 {
72     cout << "Understanding forward traversal using auto" << endl;
73     auto itt = a.begin();
74     while (itt != a.end()) {
75         cout << *itt << " ";
76         itt++;
77     }
78     cout << endl;
79 }
80 {
81     cout << "Understanding forward traversal using range" << endl;
82     for (const auto& i : a) {
83         cout << i << " ";
84     }
85     cout << endl;
86 }
87 {
88     cout << "Understanding reverse traversal" << endl;
89     deque<int>::iterator itt = a.end();
90     //cout << *itt << endl ; ----- CORE DUMPS
91     while (itt != a.begin()) {
92         itt--; //decrement iterator before using
93         cout << *itt << " ";
94     }
95     cout << endl;
96 }
97 {
98     cout << "Understanding reverse traversal using auto" << endl;

```

```
99     auto itt = a.end();
100    //cout << *itt << endl ; ----- CORE DUMPS
101    while (itt != a.begin()) {
102        itt--; //decrement iterator before using
103        cout << *itt << " ";
104    }
105    cout << endl;
106 }
107 cout << "-----" << endl;
108 }

109 /*-----
110 Understanding access using
111 1. a[i]
112 2. a.at(i) -- Same as a[i], but does out of bound checks
113 and throws exception that you can catch
114
115 -----
116 static void understanding_at_access(const deque<int>& a, int i) {
117     //int p = a[i] ; //core dumps
118     try {
119         int p = a.at(i); //Throws exception. out of bound checks
120     } catch (std::out_of_range) {
121         cout << " accessing a[" << i << "] but size of array is " << a.size() << endl;
122     }
123     cout << "-----" << endl;
124 }
125
126 */
127 /*-----
128 a.front() -- Returns first element of the deque
129 a.back() -- Returns last element of the deque
130 a.push_front(v) -- v is inserted as the first element of the deque
131 a.push_back(v) -- v is inserted as the back element of the deque
132 a.pop_front() - First element of the deque is removed. Nothing is returned
133 a.pop_back() - Last element of the deque is removed. Nothing is returned
134
135 -----
136 static void understanding_access(deque<int>& a) {
137     print1("begin with", a);
138     for (int i = 0; i < 5; i++) {
139         a.push_back(i);
140     }
141     print1("After Inserting 5 elements from the back", a);
142     for (int i = 0; i < 5; i++) {
143         a.push_front(i * 10 + 1);
144     }
145     print1("After Inserting 5 elements from the front", a);
146 }
```

C:\work\software\course\objects\stl\deque\sdeque.cpp

```

147     int& y = a.front();
148     cout << "The front of the deque has " << y << endl;
149     int& z = a.back();
150     cout << "The back of the deque has " << z << endl;
151
152     print1("After front and back operation", a);
153
154 //y = a.pop_front() ; WRONG. pop cannot return ;
155 a.pop_front();
156 a.pop_front();
157 print1("After Removing two elements from the front", a);
158 a.pop_back();
159 a.pop_back();
160 print1("After Removing two elements from the back", a);
161
162     int x = a.size() - 1;
163     y = a[x];
164     cout << "Random access of a[" << x << "] = " << y << endl;
165     understanding_at_access(a, a.size());
166 }
167
168 /*-----
169 basic
170 -----*/
171 static void basic() {
172     deque<int> a;
173     print1("begin with", a);
174     understanding_access(a);
175     understanding_iterator(a);
176
177     apply("multiply by 10", a, multiply_by_10);
178     cout << endl;
179     apply("print using iterator", a, print_integer);
180     cout << endl;
181
182     a.clear();
183     print1("Using clear", a);
184 }
185
186 /*-----
187 array of integer pointers
188 -----*/
189 static void test_deque_of_ptr_integers() {
190     deque<int*> a;
191     print1("begin with", a);
192     for (int i = 0; i < 5; i++) {
193         a.push_back(new int(i));
194     }
195     apply("After Inserting 5 elements from the back", a, print_integer);

```

```
196  for (int i = 0; i < 5; i++) {
197      a.push_front(new int(i * 10 + 1));
198  }
199  apply("After Inserting 5 elements from the front", a, print_integer);
200
201  int*& y = a.front();
202  cout << "The front of the deque has " << *y << endl;
203  int*& ba = a.back();
204  cout << "The back of the deque has " << *ba << endl;
205
206 //y = a.pop_front() ; WRONG. pop cannot return ;
207 delete(a.front());
208 a.pop_front();
209
210 delete(a.front());
211 a.pop_front();
212
213 apply("After Removing two elements from the front", a, print_integer);
214
215 delete(a.back());
216 a.pop_back();
217
218 delete(a.back());
219 a.pop_back();
220 apply("After Removing two elements from the back", a, print_integer);
221
222 deque<int*> b(a);
223 apply("Contents of deque b:", b, print_integer);
224 for (int i = 0; i < int(a.size()); i++) {
225     /* we allocated contents in a. So we delete it */
226     delete(a[i]);
227 }
228 /* Why you should NOT do this */
229 /*
230 for (int i = 0; i < n; i++) {
231 delete(b[i]) ;
232 }
233 */
234 }
235
236 /**
237 array of user defined type
238 */
239 static void test_deque_of_udt() {
240     deque<complex> a;
241     print1("begin with", a);
242     for (int i = 0; i < 5; i++) {
243         a.push_back(complex(i, -i));
244     }
```

C:\work\software\course\objects\stl\deque\sdeque.cpp

```

245  print1("After Inserting 5 elements from the back", a);
246  for (int i = 0; i < 5; i++) {
247      a.push_front(complex((i * 10 + 1), -(i * 10 + 1)));
248  }
249  print1("After Inserting 5 elements from the front", a);
250
251  complex& y = a.front();
252  cout << "The front of the deque has " << y << endl;
253  complex& z = a.back();
254  cout << "The back of the deque has " << z << endl;
255  {
256      /* see what happens if U get complex by value */
257      complex vf = a.front();
258  }
259
260 //y = a.pop_front() ; WRONG. pop cannot return ;
261 a.pop_front();
262 a.pop_front();
263 print1("After Removing two elements from the front", a);
264 a.pop_back();
265 a.pop_back();
266 print1("After Removing two elements from the back", a);
267
268 int x = a.size() - 1;
269 complex &zz = a[x]; //Copy constructor called here
270 cout << "Random access of a[" << x << "] = " << zz << endl;
271
272 deque<complex> b(a);
273 //apply("Contents of deque b",b,print_complex) ;
274 bool equal = true;
275 for (int i = 0; i < int(a.size()); i++) {
276     if (a[i] != b[i]) {
277         equal = false;
278         break;
279     }
280 }
281 equal ? cout << " array a == b\n " : cout << " array a != b\n ";
282 }
283
284 /*
285 array of user defined pointer type
286 */
287 static void test_deque_of_ptr_udt() {
288     deque<complex*> a;
289     print1("begin with", a);
290     for (int i = 0; i < 5; i++) {
291         a.push_back(new complex(i, -i));
292     }
293     apply("After Inserting 5 elements from the back", a, print_complex);

```

```
294     for (int i = 0; i < 5; i++) {
295         a.push_front(new complex(i * 10 + 1, i * 10 + 1));
296     }
297     apply("After Inserting 5 elements from the front", a, print_complex);
298
299     complex*& af = a.front();
300     cout << "The front of the deque has " << *af << endl;
301     complex*& ab = a.back();
302     cout << "The back of the deque has " << *ab << endl;
303
304 //y = a.pop_front() ; WRONG. pop cannot return ;
305 delete(a.front());
306 a.pop_front();
307
308 delete(a.front());
309 a.pop_front();
310
311 apply("After Removing two elements from the front", a, print_complex);
312
313 delete(a.back());
314 a.pop_back();
315
316 delete(a.back());
317 a.pop_back();
318
319 apply("After Removing two elements from the back", a, print_complex);
320
321 deque<complex*> b(a);
322 apply("Contents of deque b:", b, print_complex);
323 for (int i = 0; i < int(a.size()); i++) {
324     /* we allocated contents in a. So we delete it */
325     delete(a[i]);
326 }
327 /* Why you should NOT do this */
328 /*
329 for (int i = 0; i < n; i++) {
330 delete(b[i]) ;
331 }
332 */
333 }
334
335 /**
336 main
337 -----
338 int main() {
339     basic();
340     test_deque_of_ptr_integers();
341     test_deque_of_udt();
342     test_deque_of_ptr_udt();
```

```
343     return 1;  
344 }  
345 //EOF  
346  
347  
348
```

12.6.1 stack class

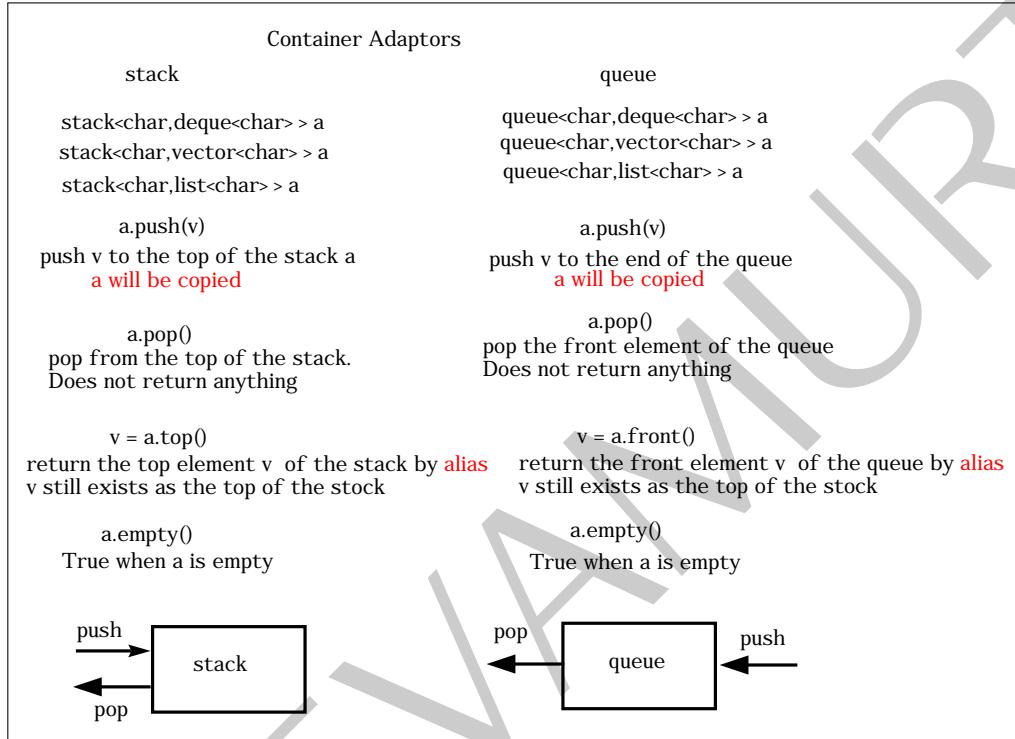


Figure 12.5: stack and queue class

```
1  /*-----  
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3  Filename: sstack.cpp  
4  compile: g++ ../complex/util.cpp ../complex/complex.cpp sstack.cpp  
5  Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6  -----*/  
7  
8 #include <iostream>  
9 #include <stack>  
10 #include <deque>  
11 #include <vector>  
12 #include <list>  
13  
14 #ifdef _WIN32  
15 #include "..\complex\complex.h"  
16 #else  
17 #include "../complex/complex.h"  
18 #endif  
19  
20 /*-----  
21 basic  
22  
23 Note that the stack is a container adaptor that  
24 can be built from  
25 1. deque  
26 2. vector  
27 3. list  
28 ex: stack<char,vector<char> > a  
29 -----*/  
30 static void test_stack_of_char() {  
31     //Use deque as a container  
32     stack<char, deque<char>> a;  
33  
34     //Use vector as a container.  
35     //stack<char,vector<char> > a ;  
36  
37     //Use list as a container.  
38     //stack<char,list<char> > a ;  
39  
40     cout << "The following elements are added to the stack\n";  
41     for (int i = 0; i < 5; i++) {  
42         char ch = char(i + 'a');  
43         cout << ch << " ";  
44         a.push(ch);  
45     }  
46     cout << endl;  
47     cout << "Number of element in the stack = " << a.size() << endl;  
48     cout << "Take out the elements from the stack\n";  
49     while (a.empty() == false) {
```

```
50     cout << a.top() << " ";
51     a.pop();
52 }
53 cout << endl;
54 }
55
56 /*-----
57 stack of complex objects
58 -----*/
59 static void test_stack_of_udt() {
60     //Use deque as a container.
61     stack<complex, deque<complex> > a;
62
63     cout << "The following elements are added to the stack\n";
64     for (int i = 0; i < 5; i++) {
65         complex ch = complex(i, -i);
66         cout << ch << " ";
67         a.push(ch);
68     }
69     cout << endl;
70     cout << "Number of element in the stack = " << a.size() << endl;
71     cout << "Take out the elements from the stack\n";
72     while (a.empty() == false) {
73         complex& c = a.top();
74         cout << c << " ";
75         a.pop();
76     }
77     cout << endl;
78 }
79
80 /*-----
81 stack of pointer to complex objects
82 -----*/
83 static void test_stack_of_pointer_to_udt() {
84     stack<complex*, deque<complex*> > a;
85
86     cout << "The following elements are added to the stack\n";
87     for (int i = 0; i < 5; i++) {
88         complex* ch = new complex(i, -i);
89         cout << *ch << " ";
90         cout << endl;
91         a.push(ch);
92     }
93     cout << endl;
94     cout << "Number of element in the stack = " << a.size() << endl;
95     cout << "Take out the elements from the stack\n";
96     while (a.empty() == false) {
97         complex*& u = a.top();
98         cout << u << " ";
```

C:\work\software\course\objects\stl\deque\sstack.cpp

```
99     delete(u);
100    a.pop();
101 }
102 cout << endl;
103 }
104
105 /*-----
106 main
107 -----*/
108 int main() {
109     test_stack_of_char();
110     test_stack_of_udt();
111     test_stack_of_pointer_to_udt();
112     return 1;
113 }
114
```

12.6.2 queue class

```
1  /*-----  
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3  Filename: squeue.cpp  
4  compile: g++ ..\complex\util.cpp ..\complex\complex.cpp squeue.cpp  
5  Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6  -----*/  
7  
8 #include <iostream>  
9 #include <queue>  
10 #include <deque>  
11 #include <vector>  
12 #include <list>  
13  
14 #ifdef _WIN32  
15 #include "..\complex\complex.h"  
16 #else  
17 #include "../complex/complex.h"  
18 #endif  
19  
20 /*-----  
21 basic  
22  
23 Note that the queue is a container adaptor that  
24 can be built from  
25 1. deque  
26 2. vector  
27 3. list  
28 ex: queue<char,vector<char> > a  
29  
30 -----*/  
31 static void test_queue_of_char() {  
32     //Use deque as a container.  
33     //queue<char,deque<char> > a ;  
34  
35     //Use vector as a container.  
36     //queue<char,vector<char> > a ;  
37  
38     //Use list as a container.  
39     queue<char, deque<char> > a;  
40  
41  
42     cout << "The following elements are added to the queue\n" ;  
43     for (int i = 0; i < 5; i++) {  
44         char ch = char(i + 'a');  
45         cout << ch << " ";  
46         a.push(ch);  
47     }  
48     cout << endl;  
49     cout << "Number of element in the queue = " << a.size() << endl;
```

```
50    cout << "Take out the elements from the queue\n";
51    while (a.empty() == false) {
52        cout << a.front() << " ";
53        a.pop();
54    }
55    cout << endl;
56 }
57
58 /*-----
59 queue of complex objects
60 -----*/
61 static void test_queue_of_udt() {
62     //Use deque as a container.
63     queue<complex, deque<complex> > a;
64
65     cout << "The following elements are added to the queue\n";
66     for (int i = 0; i < 5; i++) {
67         complex ch = complex(i, -i);
68         cout << ch << " ";
69         a.push(ch);
70     }
71     cout << endl;
72     cout << "Number of element in the queue = " << a.size() << endl;
73     cout << "Take out the elements from the queue\n";
74     while (a.empty() == false) {
75         complex& c = a.front();
76         cout << c << " ";
77         a.pop();
78     }
79     cout << endl;
80 }
81
82 /*-----
83 queue of pointer to complex objects
84 -----*/
85 static void test_queue_of_pointer_to_udt() {
86     queue<complex*, deque<complex*> > a;
87
88     cout << "The following elements are added to the queue\n";
89     for (int i = 0; i < 5; i++) {
90         complex* ch = new complex(i, -i);
91         cout << *ch << " ";
92         cout << endl;
93         a.push(ch);
94     }
95     cout << endl;
96     cout << "Number of element in the queue = " << a.size() << endl;
97     cout << "Take out the elements from the queue\n";
98     while (a.empty() == false) {
```

C:\work\software\course\objects\stl\deque\squeue.cpp

```
99     complex*& c = a.front();
100    cout << c << " ";
101    delete(c);
102    a.pop(); //compiler fails if container is a vector. WHY ??
103 }
104 cout << endl;
105 }
106
107 /*-----*/
108 main
109 -----*/
110 int main() {
111    test_queue_of_char();
112    test_queue_of_udt();
113    test_queue_of_pointer_to_udt();
114    return 1;
115 }
116
```

12.7 unorderedmap or hash class

Unordered_map

Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.

Internally, the elements in the `unordered_map` are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average).

Figure 12.6: `unordered_map` class

```
1  /*-----  
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3  Filename: unorderedmap.h  
4  -----*/  
5  
6 #ifndef unorderedmap_h  
7 #define unorderedmap_h  
8  
9 /*  
10 unordered_map will be part of the new C++ standard.  
11 unordered_map is implemented using hashing.  
12 */  
13  
14 #ifdef _WIN32 //Will not work in windows7 if U write WIN32  
15 #include "..\complex\complex.h"  
16 #else  
17 #include "../complex/complex.h"  
18 #endif  
19  
20 #include <unordered_map>  
21 #include <string>  
22  
23 /*-----  
24 hash functions  
25 -----*/  
26 struct hash_functions {  
27     static int _hash_func(int const k) {  
28         int key = k;  
29         key = ~key + (key << 15); // key = (key << 15) - key - 1;  
30         key = key ^ (key >> 12);  
31         key = key + (key << 2);  
32         key = key ^ (key >> 4);  
33         key = key * 2057; // key = (key + (key << 3)) + (key << 11);  
34         key = key ^ (key >> 16);  
35         return key;  
36     }  
37  
38     static int _hash_func(const char* s) {  
39         unsigned h = 0;  
40         int n = strlen(s);  
41         for (int i = 0; i < n; i++) {  
42             unsigned r = s[i];  
43             h = 31 * h + r;  
44         }  
45         return h;  
46     }  
47  
48     static int _hash_func(const string& s) {  
49         return _hash_func(s.c_str());
```

```

50     }
51
52     static int _hash_func(const complex& s) {
53         int x1, y1;
54         s.getxy(x1, y1);
55         return _hash_func(x1 + y1);
56     }
57
58     static int _hash_func(const complex* ptr) {
59         const char* p = reinterpret_cast<const char*>(ptr);
60         int x = _hash_func(p);
61         return x;
62     }
63 };
64
65 /*-----
66 Generic hasher that can take any key T
67 The object must have == operator defined
68 -----*/
69 template <typename T>
70 struct value_based_generic_hasher {
71     //Hash function
72     //MUST HAVE operator() defined
73     size_t operator()(const T& key) const {
74         size_t hvalue = hash_functions::_hash_func(key);
75         return hvalue;
76     }
77 };
78
79 /*-----
80 T must pointer to an object.
81 It looks at the guts for hash function
82 -----*/
83 template <typename T>
84 struct content_based_generic_hasher {
85     //Hash function
86     //MUST HAVE operator() defined
87     size_t operator()(const T& key) const {
88         size_t hash_value = hash_functions::_hash_func(*(key));
89         return hash_value;
90     }
91 };
92
93 /*-----
94 T must be pointer to an object.
95 It looks at the guts for equal function
96 The object must have == operator defined
97 -----*/
98 template <typename T>

```

```
...ware\course\objects\stl\unorderedmap(hash)\unorderedmap.h
99 struct content_based_generic_equal {
100     //Hash function
101     //MUST HAVE operator() defined
102     bool operator()(const T& left, const T& right) const {
103         return (*left == *right);
104     }
105 };
106
107
108
109 #endif
110
```

```
1  /*-----  
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy  
3  Filename: unorderedmap.cpp  
4  compile: g++ unorderedmap.cpp ../complex/complex.cpp ../complex/util.cpp  
5  Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)  
6  -----*/  
7  #include "unorderedmap.h"  
8  
9  /*  
10 unordered_map will be part of the new C++ standard.  
11 unordered_map is implemented using hashing.  
12 */  
13  
14 /*-----  
15 test names  
16 -----*/  
17 static void test_names() {  
18     unordered_map<string, char> grades;  
19  
20     // Illustrating insertion to map  
21     grades["john"] = 'a';  
22     grades["josh"] = 'd';  
23  
24     //Illustrating Insertion to map  
25     grades.insert(pair<string, char>("robert", 'b'));  
26     pair<unordered_map<string, char>::iterator, bool> res = grades.insert  
        (pair<string, char>("adam", 'b'));  
27     if (res.second) {  
28         cout << "adam is inserted " << endl;  
29     }  
30     {  
31         auto res = grades.insert(pair<string, char>("adam1", 'b'));  
32         if (res.second) {  
33             cout << "adam1 is inserted " << endl;  
34         }  
35     }  
36  
37     // Illustrating map requires unique key  
38     pair<unordered_map<string, char>::iterator, bool> res1 = grades.insert  
        (pair<string, char>("adam", 'c'));  
39     if (res1.second) {  
40         cout << "adam is inserted " << endl;  
41     } else {  
42         cout << "adam is NOT inserted as it is already there " << endl;  
43     }  
44  
45     // Illustrating changing value  
46     // Now adam improves grade to a from b  
47     grades.erase("adam");
```

```

...re\course\objects\stl\unorderedmap(hash)\unorderedmap.cpp
48     auto res2 = grades.insert(pair<string, char>("adam", 'a'));
49     if (res2.second) {
50         cout << "adam is inserted " << endl;
51     } else {
52         cout << "adam is NOT inserted as it is already there " << endl;
53     }
54
55
56 //Illustrating map traversal
57 auto p = grades.begin();
58 while (p != grades.end()) {
59     cout << p->first << " " << p->second << endl;
60     p++;
61 }
62
63 //Illustrating Finding a key in hash
64 if (grades.find("tom") == grades.end()) {
65     cout << "tom is NOT in the hash" << endl;
66 } else {
67     cout << "tom is in the hash" << endl;
68 }
69 if (grades.find("adam") == grades.end()) {
70     cout << "adam is NOT in the hash" << endl;
71 } else {
72     cout << "adam is in the hash" << endl;
73 }
74
75 //Illustrating Finding a key in hash using []
76 cout << "Grade of josh is " << grades["josh"] << endl;
77 //Illustrating Finding a key in hash using find
78 auto itr1 = grades.find("josh");
79 if (itr1 != grades.end()) {
80     cout << "Grade of josh is " << itr1->second << endl;
81 }
82 itr1 = grades.find("jag");
83 if (itr1 != grades.end()) {
84     cout << "Grade of jag is " << itr1->second << endl;
85 } else {
86     cout << "Student jag is not there. Hence he has no grade\n";
87 }
88
89 //If the item is NOT there, the key is automatically inserted
90 //Value will zero for built-in type
91 //Value will be default constructor for UDT
92 cout << "Grade of jag is " << grades["jag"] << endl;
93
94 //Illustrating size of hash
95 cout << "Number of element is hash is: " << grades.size() << endl;
96 //Illustrating cleaing hash

```

```
97     grades.clear();
98     cout << "Number of element is hash is after clear : " << grades.size() <<
99         endl;
100    }
101 /**
102 test_names_using_your_hash_function
103 */
104 static void test_names_using_your_hash_function() {
105     unordered_map<string, char, value_based_generic_hasher<const string> >
106         grades;
107     // Illustrating insertion to map
108     grades["john"] = 'a';
109     grades["josh"] = 'd';
110
111     //Illustrating Insertion to map
112     grades.insert(pair<string, char>("robert", 'b'));
113     auto res = grades.insert(pair<string, char>("adam", 'b'));
114     if (res.second) {
115         cout << "adam is inserted " << endl;
116     }
117
118     // Illustrating map requires unique key
119     auto res1 = grades.insert(pair<string, char>("adam", 'c'));
120     if (res1.second) {
121         cout << "adam is inserted " << endl;
122     } else {
123         cout << "adam is NOT inserted as it is already there " << endl;
124     }
125
126     // Illustrating changing value
127     // Now adam improves grade to a from b
128     grades.erase("adam");
129     auto res2 = grades.insert(pair<string, char>("adam", 'a'));
130     if (res2.second) {
131         cout << "adam is inserted " << endl;
132     } else {
133         cout << "adam is NOT inserted as it is already there " << endl;
134     }
135
136     //Illustrating map traversal
137     auto p = grades.begin();
138     while (p != grades.end()) {
139         cout << p->first << " " << p->second << endl;
140         p++;
141     }
142
143     //Illustrating Finding a key in hash
```

```

...re\course\objects\stl\unorderedmap(hash)\unorderedmap.cpp
144     if (grades.find("tom") == grades.end()) {
145         cout << "tom is NOT in the hash" << endl;
146     } else {
147         cout << "tom is in the hash" << endl;
148     }
149     if (grades.find("adam") == grades.end()) {
150         cout << "adam is NOT in the hash" << endl;
151     } else {
152         cout << "adam is in the hash" << endl;
153     }
154
155 //Illustrating Finding a key in hash using []
156 cout << "Grade of josh is " << grades["josh"] << endl;
157 //Illustrating Finding a key in hash using find
158 auto itr1 = grades.find("josh");
159 if (itr1 != grades.end()) {
160     cout << "Grade of josh is " << itr1->second << endl;
161 }
162 itr1 = grades.find("jag");
163 if (itr1 != grades.end()) {
164     cout << "Grade of jag is " << itr1->second << endl;
165 } else {
166     cout << "Student jag is not there. Hence he has no grade\n";
167 }
168
169 //If the item is NOT there, the key is automatically inserted
170 //Value will zero for built-in type
171 //Value will be default constructor for UDT
172 cout << "Grade of jag is " << grades["jag"] << endl;
173
174 //Illustrating size of hash
175 cout << "Number of element is hash is: " << grades.size() << endl;
176
177 //Illustrating cleaing hash
178 grades.clear();
179 cout << "Number of element is hash is after clear : " << grades.size() << endl;
180 }
181
182 /*-----
183 test udt
184 -----*/
185 static void test_udt() {
186     unordered_map<complex, char, value_based_generic_hasher<const complex> >
187         myhash;
188
189     // Illustrating insertion to map
190     for (int i = 0; i < 26; i++) {
191         complex c(i, -i);

```

```
191     myhash[c] = i + 'a';
192 }
193
194 //Illustrating map traversal
195 auto p = myhash.begin();
196 while (p != myhash.end()) {
197     cout << p->first << " " << p->second << endl;
198     p++;
199 }
200
201 //Illustrating find
202 {
203     {
204         complex c(24, -24);
205         auto p = myhash.find(c);
206         if (p == myhash.end()) {
207             cout << "The object " << c << " is not in the hash\n";
208         } else {
209             cout << "The object " << c << " exists. " << "The value associated with ?>
210                 this object is = " << p->second << endl;
211         }
212     }
213     {
214         complex c(214, -24);
215         auto p = myhash.find(c);
216         if (p == myhash.end()) {
217             cout << "The object " << c << " is not in the hash\n";
218         } else {
219             cout << "The object " << c << " exists. " << "The value associated with ?>
220                 this object is = " << p->second << endl;
221         }
222     }
223
224
225 /*-----*
226 test pointer udt
227 -----*/
228 static void test_pointer_to_udt() {
229     unordered_map<complex*, char, value_based_generic_hasher<const complex*> >    ?
230         myhash;
231     complex* c24 = NULL;
232     complex* c214 = new complex(241, -24);
233
234 // Illustrating insertion to map
235 for (int i = 0; i < 26; i++) {
236     complex* c = new complex(i, -i);
237     myhash[c] = i + 'a';
```

```

...re\course\objects\stl\unorderedmap(hash)\unorderedmap.cpp
237     if (i == 24) {
238         c24 = c;
239     }
240 }
241
242 //Illustrating map traversal
243 auto p = myhash.begin();
244 while (p != myhash.end()) {
245     cout << p->first << " " << p->second << endl;
246     p++;
247 }
248
249 //Illustrating find
250 {
251     complex* c = c24;
252     {
253         auto p = myhash.find(c);
254         if (p == myhash.end()) {
255             cout << "The object " << c << " is not in the hash\n";
256         } else {
257             cout << "The object " << c << " exists. " << "The value associated with this object is = " << p->second << endl;
258         }
259     }
260     {
261         complex* c = c214;
262         auto p = myhash.find(c);
263         if (p == myhash.end()) {
264             cout << "The object " << c << " is not in the hash\n";
265         } else {
266             cout << "The object " << c << " exists. " << "The value associated with this object is = " << p->second << endl;
267         }
268     }
269 }
270
271 //Free all the memory
272 {
273     auto p = myhash.begin();
274     while (p != myhash.end()) {
275         delete (p->first);
276         p++;
277     }
278 }
279 delete c214;
280 //You should not delete c24. Why ?
281 }
282
283 */

```

```
284 test pointer udt
285 -----
286 static void test_pointer_to_udt_find_through_object() {
287     unordered_map<complex*, char, content_based_generic_hasher<const complex*>, content_based_generic_equal<const complex*> > myhash;
288
289     // Illustrating insertion to map
290     for (int i = 0; i < 26; i++) {
291         complex* c = new complex(i, -i);
292         myhash[c] = i + 'a';
293     }
294
295     //Illustrating map traversal
296     auto p = myhash.begin();
297     while (p != myhash.end()) {
298         cout << p->first << " " << p->second << endl;
299         p++;
300     }
301
302     //illustering find
303     {
304         complex c(24, -24);
305         {
306             auto p = myhash.find(&c);
307             if (p == myhash.end()) {
308                 cout << "The object " << c << " is not in the hash\n";
309             } else {
310                 cout << "The object " << c << " exists. " << "The value associated with this object is = " << p->second << endl;
311             }
312         }
313         {
314             complex c(214, -24);
315             auto p = myhash.find(&c);
316             if (p == myhash.end()) {
317                 cout << "The object " << c << " is not in the hash\n";
318             } else {
319                 cout << "The object " << c << " exists. " << "The value associated with this object is = " << p->second << endl;
320             }
321         }
322     }
323
324     //free all the memory
325     {
326         auto p = myhash.begin();
327         while (p != myhash.end()) {
328             delete (p->first);
329             p++;
330         }
331     }
332 }
```

```
...re\course\objects\stl\unorderedmap(hash)\unorderedmap.cpp
330     }
331 }
332
333 }
334
335 /*-----*
336 main
337 -----*/
338 int main() {
339     test_names();
340     test_names_using_your_hash_function();
341     test_udt();
342     test_pointer_to_udt();
343     test_pointer_to_udt_find_through_object();
344     return 1;
345 }
346
```

12.8 set class

```

1  /*
2  Copyright (c) 2018 Author: Jagadeesh Vasudevamurthy
3  Filename: set.cpp
4  compile: g++ set.cpp
5  Memory leaked: 0 bytes (0%); potentially leaked: 0 bytes (0%)
6  */

```

```

7
8 #include <iostream>
9 using namespace std;
10 #include <set>
11 #ifdef _WIN32
12 //#include "vld.h"
13 #endif
14
15 /*-----*/
16 An Obj
17 -----
```

```

18 class Obj {
19 public:
20     //Constructor
21     Obj(const char *s) {
22         cout << "In constructor " << s << endl;
23         int l = strlen(s);
24         _size = l;
25         _dc = new int[l];
26         for (int i = 0; i < l; ++i) {
27             _dc[i] = s[i] - '0';
28         }
29     }
30     //Destructor
31     ~Obj() {
32         cout << "Object deleted is " << *this;
33         cout << endl;
34         _delete();
35     }
36
37     void _delete() {
38         delete[] _dc;
39         _dc = 0;
40     }
41
42     void _copy(const Obj& from) {
43         _size = from._size;
44         _dc = new int[_size];
45         cout << "IN COPY CONSTRUCTOR = " << from;
46         for (int i = 0; i < _size; ++i) {
47             _dc[i] = from._dc[i];
48         }
49         cout << endl;
```

Sets are containers that store unique elements following a specific order.

In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

Internally, the elements in a set are always sorted following a specific strict weak ordering criterion indicated by its internal comparison object (of type Compare).

Set containers are generally slower than unordered_set containers to access individual elements by their key, but they allow the direct iteration on subsets based on their order.

Sets are typically implemented as binary search trees.

```

50    }
51
52    //Copy constructor
53    Obj(const Obj& rhs) {
54        _copy(rhs);
55    }
56
57    //Equal operator.
58    Obj& operator=(const Obj& rhs) {
59        cout << "IN EQUAL OPERATOR\n";
60        if (this != &rhs) {
61            _delete();
62            _copy(rhs);
63        }
64        return *this;
65    }
66
67    //print
68    friend ostream& operator<<(ostream& o, const Obj& c) {
69        for (int i = 0; i < c._size; ++i) {
70            cout << c._dc[i];
71        }
72        return o;
73    }
74
75
76    //less than operator
77    friend bool operator<(const Obj& c1, const Obj& c2) {
78        cout << "IN MY < operator " << c1 << " and " << c2 << endl;
79        if (c1._size < c2._size) {
80            return true;
81        }
82        if (c1._size > c2._size) {
83            return false;
84        }
85        //At this point equal size
86        for (int i = 0; i < c1._size; ++i) {
87            if (c1._dc[i] < c2._dc[i]) {
88                return true;
89            }
90            if (c1._dc[i] > c2._dc[i]) {
91                return false;
92            }
93        }
94        //At this point, exactly same
95        return false;
96        //Remember this rule: ALWAYS HAVE COMPARISON FUNCTION RETURN FALSE FOR
97        //EQUAL VALUE
98    }

```

You need to implement < which is used in BST

Set uses only < to see whether two objects are equal
 $(c1 == c2) \Rightarrow (\neg(c1 < c2) \wedge \neg(c2 < c1))$

C:\work\software\course\objects\stl\set\set.cpp

```

98
99  //== operator
100 friend bool operator==(const Obj& c1, const Obj& c2) {
101     cout << "IN MY == operator " << c1 << " and " << c2 << endl;
102     return (!(c1 < c2) && !(c2 < c1)));
103 }
104
105 //!= operator
106 friend bool operator!=(const Obj& c1, const Obj& c2) {
107     cout << "IN MY != operator " << c1 << " and " << c2 << endl;
108     return !(c1 == c2));
109 }
110
111 private:
112     int _size;
113     int* _dc;
114 };
115
116 /*-----
117 find
118 -----*/
119 static bool find(const set<Obj>& s, const Obj& o) {
120     cout << "In find " << o << endl;
121     set<Obj>::iterator itt = s.find(o);
122     return (itt != s.end()) ? true : false;
123 }
124
125 /*-----
126 test
127 -----*/
128 void test() {
129     set<Obj> s;
130     Obj o1("1245");
131     Obj o2("1345");
132     if (o1 == o2) {
133         cout << "(o1 == o2)" << endl;
134     } else {
135         cout << "(o1 != o2)" << endl;
136     }
137     if (o1 != o2) {
138         cout << "(o1 != o2)" << endl;
139     } else {
140         cout << "(o1 == o2)" << endl;
141     }
142     //From this point, put a break point in == and != operator
143     //Set never call == operator or != operator even if you written one.
144     //It calls only < operator
145     s.insert(o1);
146     s.insert(o2);

```

C:\work\software\course\objects\stl\set\set.cpp

```
147
148     Obj f("12451");
149     if (find(s, f)) {
150         cout << "Already there\n";
151     } else {
152         cout << "NOT there\n";
153     }
154
155     Obj f1("1245");
156     if (find(s, f1)) {
157         cout << "Already there\n";
158     } else {
159         cout << "NOT there\n";
160     }
161 }
162
163 /*-----
164 main
165 -----*/
166 int main() {
167     test();
168     return 1;
169 }
170 //EOF
171
172
```