1094

A VERIFICATION SUITE FOR C++ COMPILERS

Dissertation submitted to the Jawaharlal Nehru University
in partial fulfilment of the requirements
for the award of the Degree of
MASTER OF TECHNOLOGY

in
COMPUTER SCIENCE

by
K.RAMESH SHENOY

SCHOOL OF COMPUTER AND SYSTEMS SCIENCES
JAWAHARLAL NEHRU UNIVERSITY
NEW DELHI - 110067
INDIA
JANUARY 1994

CERTIFICATE

This is to certify that the Thesis entitled **"A VERIFICATION SUITE FOR C++ COMPILERS"** being submitted by me to J.N.U., in partial fulfillment of the requirements for the award of the degree of **M.Tech** in **Computer Science and Technology** is a record of original work done by me under the supervision of Dr. Pratul Dublish, School of Computer and System Sciences, during the Monsoon Semester 1993.

This work has not been submitted in part or full to any other university or institution for award of any degree.

[illegible]

K. RAMESH SHENOY

[illegible] 5-1-94

Prof. K.K. Bharadwaj
Dean, SC & SS
J.N.U.
New Delhi.

[illegible] 5/1/94

Supervisor

Dr. Pratul Dublish
Associate Professor
SC & SS, J.N.U.,
New Delhi.

ACKNOWLEDGEMENTS

I wish to acknowledge with a deep sense of gratitude the guidance and inspiration offered by my supervisor Dr. Pratul Dublish, Associate Professor, SC&SS, JNU. I am grateful for the precious time he spent with me discussing various topics involved with this project. It would have been impossible for me to come out successfully with out his suggestions and worthy criticisms during the course of this project.

I extend my sincere thanks to Prof. K.K. Bharadwaj, Dean, SC&SS, JNU for providing me the opportunity to undertake this dissertation. I would like to thank the authorities of SC&SS for providing me with the necessary facilities to complete my work.

My sincere thanks are due to many of my friends specially Mr. Sumitra Kumar Srivastava, who helped me and gave me their moral support during the project.

Finally I thank one and all who are directly or indirectly involved in this work.

[illegible]

K. Ramesh Shenoy

TO MY PARENTS

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION TO C++

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. It enables reasonably educated and experienced programmers write programs at a higher level of abstraction without loss of efficiency compared to C for applications that are demanding in time, space, inherent complexity and constraints from the execution environment [Str, 1993]. C++ was developed from the C programming language. C was chosen as the base language for C++ because,

a. it is versatile, terse and relatively low-level;

b. it is adequate for most system programming tasks;

c. it is available on a wide variety of hardware platforms and operating systems;

d. it fits into the UNIX programming environment.

The difference between C and C++ is primarily in the degree of emphasis on types and structures. C is expressive and permissive. C++ is even more expressive in the sense that it allows the user to define his own types. This helps the programmer define appropriate types in the software to model the real world entities.

C++ made *object oriented programming* and *data abstraction* available to the community of software developers that until then had considered such techniques and the languages that supported them such as Smalltalk, CLU, Simula, Ada, object oriented Lisp dialects etc with disdain.

C++ is already widely available and is in wide use for real application and system development. C++ has not been standardized as yet. The proposal for ANSI standardization was written by Dimtry Lenkov [Len, 1989]. Dimtry's proposal made a strong case for a careful and detailed definition of the C++ language.

His main motivation for doing so was the increasing popularity of C++ among software developers and the availability of several independent but incompatible C++ compilers. He argued that an early standardization of C++ is in the interest of software community since it will prevent the proliferation of incompatible C++ dialects.

The *ANSI C++ committee* was formed in December 1989 and *ISO C++ committee* in June 1991 for standardization of C++. These two committees decided to hold joint meetings for standardization of C++. The C++ committee had a difficult charter :

a. The definition of the language must be precise and comprehensive.

b. C/C++ compatibility had to be addressed.

c. Extensions beyond current C++ practice had to be considered.

d. Libraries had to be considered.

The aim of the ANSI and ISO C++ committees was to publish the complete draft working paper *(DWP)* for public review by late 1993 and to publish the official standard about two years later. However, the standardization effort is considerably behind the original schedule and the official standard is now expected in 1997.

1.2 THE NEED FOR A C++ VERIFICATION SUITE

There is an urgent need for a verification suite for C++ compilers because the language is complex and is not yet standardized in all its dimensions. The language is complex because of the active interplay of features like class, inheritance, dynamic binding etc. Further the standardization committee is coming out with a new version of the DWP every six months. So the compilers will have to be updated from time to time. In this context there is a need to test the compilers to ensure that they conform to the latest changes.

An independent verification suite is far more preferable in contrast to one written by compiler developer because :

a. Testing is essentially a destructive process and it is hard to be destructive on something one has created. It is natural for everyone to believe that the program they have written works well. So it is not easy for a software developer to test his own software with a proper frame of mind for testing [Jal, 1991].

b. If a software is tested by people not involved with developing the same, then they may succeed in finding those errors which might have occurred due to the fact that the developers did not understand the specifications clearly [Jal, 1991].

1.3 AN OVERVIEW OF THE THESIS

This thesis consists of test programs based on the functional specifications of the C++ language in the draft working paper dated 1st June 1993. The test programs are written for the following three chapters of the DWP :

a. Chapter - 9 Classes

b. Chapter -10 Derived Classes

c. Chapter -11 Member access control

In this thesis, the focus is on these chapters because of the following reasons :

a. The concept of class, derived class, and member access control are fundamental features of C++.

b. Many changes have been introduced in these chapters in the DWP as compared to the Annotated Reference Manual [Ell, 1990].

Broadly there are two different approaches for testing a software : functional testing and structural testing [Jal, 1991]. In functional testing the software or module to be tested is treated as a black box, and the test cases are decided based on the specifications of the system or the module. The focus is here on testing the external behaviour of the system. In structural testing the test cases are decided based on the internal structure or logic of the module to be tested.

This verification (or test) suite consists of test programs for functional testing of C++ compilers. This is so because of the following factors :-

a. Our goal is to evaluate the performance of *any* C++ compiler with respect to the functional specifications of the C++ language in the DWP.

b. Structural testing requires access to the source code of a compiler. However, the source code of commercial C++ compilers is, in general, not available.

c. A test suite based on functional testing approach can be used to evaluate *any* C++ compiler. In contrast, a test suite based on structural testing will only be useful for the compiler on whose internal structure it is based.

We expect that our verification suite will be useful for :-

a. Developers of C++ compilers, since this suite can be used for carrying out functional testing.

b. Users of C++ compilers to test and judge the quality of various C++ compilers available.

c. For learners of C++, since this suite consists of about two hundred programs based on most of the language features related to the chapters on classes, derived classes, and member access control.

The rest of this thesis is organized as follows:-

Chapter 2 provides details about different categories of test programs and explains how the suite is organized. Chapter 3 gives an overview of classes and explains about data members, function members, POD struct / POD union, scope and name lookup rules, nested classes and local classes.

Chapter 4 gives an overview of derived classes and underlying concepts like multiple base classes, virtual base classes, ambiguities, virtual functions, and abstract classes.

Chapter 5 gives an overview of member access control and focuses on issues like access specifiers, access declarations, protected member access, and friends.

Chapter 6 contains results and conclusions. Appendix A contains a copy of chapters 9-11 of the DWP. Appendix B gives the details about the contents of the floppy attached to this thesis.

CHAPTER 2

ORGANIZATION OF THE VERIFICATION SUITE

2.1 INTRODUCTION

This verification suite consists of a collection of test programs. Each test program is based on a specific feature given in the relevant portion of the DWP. There are two categories of test programs :

a. Positive test programs, that is, which contain no compile time errors as per the DWP specifications. These are named as *p*.cpp*.

b. Negative test programs, that is, which are expected to give a compile time error or warning as per the DWP specifications. These are named as *n*.cpp*.

Each test program is based on some feature specified in a particular para, section, and chapter of DWP because :

a. If a single program is written for all the features in a para then the program will become large and complex. Hence it will be difficult to manually check whether the program is correct or not.

b. If the program detects an error, i.e., a deviation from the DWP specifications, in the compiler under test, then it will be easier to pin point the error and locate its cause.

A test program detects an error in the compiler under test by checking any of the following:-

a. Whether the compiler is able to detect the compile time error in it or not.

b. Whether the conditions implied by DWP specifications are in fact true or not in a program during run time.

c. Whether the values of variables implied by DWP specifications are in fact the actual values in a program during run time.

2.2 NAMING CONVENTION FOR TEST PROGRAMS

The scheme that has been adopted for naming most of the test programs is as follows. The first two numeric characters immediately after *n* or *p* represent the chapter number, i.e., they are 09 for chapter 9, 10 for chapter 10, and 11 for chapter 11 of the DWP. The next two numeric characters represent the section number and the next two characters represent the paragraph number of the feature in the DWP, on which the test program is based. The last character represents the number of test written for the same para number, i.e., like *a* is used for the first test, *b* for the second test, *c* for the third test and so on. For example, consider the following:

a. Test program name p090202a.cpp, implies this is the first positive test program based on the feature given in chapter 9, section 9.2, para 2 of the DWP.

b. Test program name n110303c.cpp, implies this is the third negative test program based on the feature given in chapter 11, section 11.3, para 3 of DWP.

However there is a little deviation to this scheme for few test programs because if the above scheme is followed the name of the file may contain more than eight alpha-numeric characters before the period. However, MS-DOS allows only eight characters in a file name before the period. So where ever possible instead of using two numeric characters each for the chapter number, the section number, and the para number only one numeric character is used. For example,

a. Test program name n921102b.cpp, implies this is the second test program based on the feature given in chapter number 9, section number 9.2.1, para number 1, sub para number 2 of the DWP.

b. Test program name p093101d.cpp, implies this is the fourth test program based on the feature given in chapter number 9, section number 9.3.1, para number 1, of the DWP.

2.3 POSITIVE TESTS

A positive test program contains no error as per the DWP specifications. However if the compiler reports errors in it on compilation, then it implies that the *test has failed* and there are *errors* in the *C++ compiler* under test. If a positive test compiles successfully it is executed. During its execution, a positive test checks the values of its internal variables to verify whether the run-time behaviour of this compiled test program is as per the DWP specification. In particular, one of the following may occur when a positive test is executed :

a. If there is no error in the compiler then it will give no output.

b. An error is detected in the compiler by the program, *say p*.cpp,* during runtime since the value of the relational expression, on source line L, is found to be false(true), instead of true(false). In such a case the following output is produced,

```
ERROR: LINE NO: L
TEST PGM  p*.cpp FAILED.
```

For example, consider the test program p090003.cpp given below.

```
1.    #include <iostream.h>
2.    /* PGM NAME  :P090003.CPP
3.       REFER TO  :SEC :9.0, PARA :3, PG :9-1
4.       FEATURE   :OBJECTS OF AN EMPTY CLASS HAVE A NON-
5.                  ZERO SIZE AND HAVE DISTINCT ADDRESSES.
```

```
6.      */
7.     void main()
8.     {
9.      int tf=0;      // Flag for test fail.
10.     class A { }; // Class with no members, Empty class.
11.     A e1,e2;       // e1, e2 are objects of empty class.
12.     A* p1=&e1;
13.     if(p1==&e2){ cout<<"LINE NO : "<<__LINE__; tf=1; }
14.     // Checks whether the objects e1, e2 have distinct
15.     // addresses.
16.     int i=sizeof(e1);
17.     if(i==0){ cout<<"LINE NO : "<<__LINE__; tf=1; }
18.      //Checks whether the empty class is non zero.
19.     if(tf)cout<<"\n TEST PGM P090003.CPP FAILED. \n";
20.    }
```

For testing this feature two objects e1, e2 of empty class A are defined on line 11. On line 13 in the *if statement*, it is checked whether the objects e1, e2 have distinct addresses. If the objects e1, e2 do not have distinct addresses, then the relational expression which is inside *if statement* on line 13 is true and hence the program will give the output as below:

```
ERROR : LINE NO: 13
TEST PGM P090003.CPP FAILED.
```

c. If an error is detected in the compiler by the program, *say p*.cpp,* during runtime since it is found that on line number *L* of the program the value of variable *X* is *A* instead of *E* as per DWP specification. Then it will give output in the format shown below :

ERROR: LINE NO: *L*   :VAR NAME: *X*

ACTUAL VALUE: *A*   EXPECTED VALUE: *E*

TEST PGM *P*.CPP* FAILED.

Consider the following example below,

```
1. #include<iostream.h>
2. /* PGM NAME :P090205.CPP
3.    REFER TO :SEC:9.2, PARA:5, PG:9.4
4.    FEATURE  :A CLASS C1 MAY CONTAIN A POINTER OR REFERENCE
                TO AN OBJECT OF CLASS C1.
5.  */
6. class A
7. {
8.  public:
9.  A()
10. :t2(*this)
11. { }
12. A* t1;
13. A& t2; //Reference to an object of type A.
14. int i;
15. };
16. void main()
17. {
18. A a1,a2;
19. a1.t1=&a2;
20. int tf=0;
21. a1.i=19;
22. if(a1.t1->i!=a2.i){ ...... }
23. if(a1.t2.i!=19) { cout<<"\nERROR: LINE NO: "<<__LINE__
                        <<" :VAR NAME: a1.t2.i"
                        <<"\n ACTUAL VALUE: "<<a1.t2.i;
                        <<" EXPECTED VALUE: 19";tf=1;
                  }
24. if(tf)cout<<"\n TEST PGM P090205.CPP FAILED \n\n" ;
25. }
```

The feature mentioned is being tested by assigning a value 19 to *a1.i* on line 21, this same variable is being accessed by the variable t2 as defined on line 13. On line 23 it is being checked whether the actual value of *a1.t2.i* is same as expected value which is *19*. If the actual value is, say 5, because of an error in compiler, then the program will give an output as shown below:

```
ERROR: LINE NO: 23    :VAR NAME: a1.t2.i

ACTUAL VALUE: 5         EXPECTED VALUE: 19

TEST PGM P090205.CPP  FAILED.
```

Thus when the test programs with name p*.cpp give output message in the format described in case b and c above, it implies that *test has failed* and there is an *error* in the *C++ compiler* under test.

2.4 NEGATIVE TESTS

A negative test program contains a compile time error as per the DWP specification. For example consider the following test program,

```
/* PGM NAME :N090203A.CPP
   REFER TO :SEC:9.2, PARA:3, PG:9.4
   FEATURE  :A MEMBER MAY NOT BE AUTO.
 */

class A
{
 public:
 auto int i;              //ERROR:Cannot be auto.
};
void main() { }
```

As per the feature a class member may not be *auto* but the data member *int i* is defined as auto in class A which is an error.

A negative test program on compilation should give either an error or warning message. However, it has not yet been resolved by the C++ standardization committee in which cases the compiler should give an error message and in which cases the compiler should give a warning message. So some compilers may give an error and some other, a warning, for the same negative test program submitted for compilation.

Further, the error or warning message produced may not be related to the actual error in the program. This problem is caused because the text of error or warning message has not been standardized. So it is upto the user of the compiler to interpret the error or warning message by having a look at the test program.

If a test program with name n*.cpp fail to give an error or a warning message then it implies that the *test has failed* and there is an *error in C++ compiler* under test.

2.5 RUNNING THE VERIFICATION SUITE

The verification suite can be used for testing all DOS based C++ compilers by using command *auco* <[input]> where [input] is the command line compilation command of the compiler under test.

By using the above auco batch command each **.cpp* file is compiled and the generated code, if any, is executed.

Further two files *out.k* and *rpt.s* are produced. In the file out.k all the *.cpp files compiled are listed in a sorted order based on chapter, section and para of feature on which they are based.

Similarly in the file rpt.s all the error messages, if any, generated by *.cpp files (submitted for compilation) and the output produced, if any, are stored in a sorted order based on chapter, section and para of the feature concerned.

The only input that is to be given for using test programs in order to test a compiler, using auco command, is to give the appropriate command line compilation command. For example, if one is testing the *Borland C++* compiler for which *bcc* is the command line compilation command, one has to give the following command against system prompt (c:\>, after copying all the files from sub-directory *krs* of the floppy attached to this thesis into the hard disk c:\)

```
c:\>auco bcc
```

After giving the above command, files out.k and rpt.s are created. Thus by looking at each *.cpp file in out.k and, their corresponding generated messages on compilation or output, if any, in file rpt.s one can check whether the *test has failed [refer sec:2.3-2.4]* or not.

Further if one wants to test the compiler using only test programs written for a chapter of DWP, it can be done by using command *auco9* for chapter 9, *auco10* for chapter 10 and *auco11* for chapter 11, instead of using command *auco* for all the above three chapters.

However for using the test programs to test a C++ compiler which is not DOS based, appropriate driver programs will have to be written.

CHAPTER 3

OVERVIEW OF CLASS

3.1 INTRODUCTION

The purpose of this chapter is to give a brief overview, with suitable examples, of features associated with C++ classes. This being done for the sake of completeness. For more details about the C++ programming language one can refer to books written by Bjarne Stroustrup [Str, 1991] and stanley B. Lippman [Lip 1991]. Further for more insight about the concepts associated with Object Oriented Programming one can refer to books by Grady Booch [Boo, 1991] and Khoshafian et al.[Kho, 1990].

The *C++ class mechanism* provides the programmer with a tool for creating new types that can be used as conveniently as the built-in types. A type is the concrete representation of an idea or concept. The reason for designing a new type is to provide a concrete and specific definition of a concept that has no direct and obvious counterpart among the built-in types. For example, one might provide a type customer in a program dealing with bank database, a type book in a program designed for library management, or a type train in a program developed for railway reservation system.

A program that provides types that closely match the concepts of the application is usually easier to understand

and easier to modify than a program that does not. A well-chosen set of user defined types makes a program more concise; it also enables the compiler to detect illegal uses of objects that otherwise would not be detected until the program is tested [Str, 1991].

The fundamental idea behind defining a new type is to separate the implementation details of the type from the various operations that can be carried out on it. Such a separation can be expressed by channelising the use of the data structure and internal housekeeping routines through a specific interface.

3.2 THE CLASS DEFINITION

A class is a user defined type. A class definition has two parts : the class head, composed of the keyword *class* followed by the class *tag name*, and the class body, enclosed by a pair of curly braces, which must be followed by either a semicolon or a declaration list. For example,

```
class  SAMP
{
 private :

  char dat1;

 public :

  void getin() { .... }

  void getout() { .... }
};
```

Here a class with name *SAMP* has been defined. An object of type SAMP is created using the declaration below.

```
SAMP s1;
```

The member specification in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions, nested types and members constants. The class SAMP defined earlier contains a data member dat1 and two function members getin() and getout().

A member of a class can be private, protected, or public [refer sec:5.1]. These keywords control the level of access to members of a class in a program.

A class can also be defined using the keyword *struct*. The only difference between the keywords *class* and *struct* is that in a class the members are private by default, while in a struct they are public by default.

For example the class SAMP defined earlier can also be defined as

```
class SAMP
{
  char dat1;
   public :
   void getin() { .... }
   void getout() { .... }
```

```
      };
```

or as

```
 struct SAMP
 {
  private :

   char dat1;

  public :

   void getin() { .... }

   void getout() { .... }
};
```

The above definitions of type SAMP are equivalent.

A *union* is a class declared with the class-key union, its members are public by default and it holds only one member at a time.

3.3 DATA MEMBERS

The declaration of class data members is done in the same way as the ordinary variable declarations with the exception that an explicit initializer is not allowed. There may be zero or more data members of any type in a class.

In the class SAMP defined in earlier example dat1 is the data member of type char. Similarly other data members of type int, float, double etc. can be declared.

A class object can be declared as a data member only if its class definition has been seen. In cases where a class definition has not been seen, a forward declaration of class can be supplied. A forward declaration permits pointers and references to objects of the class to be declared as data members.

3.4 POINTERS TO MEMBERS

Pointers to members are the variables which contain an offset to the member from the starting point of the address of the object of a given type. The value of a pointer to member does not reveal its machine address, unlike in the case of pointers to ordinary variables which contain the machine address of variables to which they point. The pointer to members can be defined as follows :

```
class A
{
 public:
  char ch;
  int f(char);
};
char A::*pm1=&A::ch;  //pm1 contains the offset of ch.
char A::*pm2(char)=&A::f; //pm2 contains offset of f.
```

Here pm1 is declared as a pointer to member of *A* of type char and pm2 as a pointer to member of *A* of type int (char). They can be used like this,

```
A a1;              //a1 is object of type class A.
```

```
a1.ch='a';          //assign 'a' to a  character member ch of
                    //object a1 using member access operator '.'.
a1.*pm1='a';        // is equivalent to a1.ch='a';
                    // '.*' operator binds pm1 to address of a1.
a1.f('a');          //call the  function member f of object a1
                    //with argument 'a' directly.

a1.*pm2('a');       // is equivalent to a1.f('a');
                    // '.*' operator binds pm2 to address of a1.
```

3.5 REFERENCES

A references type, sometimes referred to as an alias, serves as an alternative name for the object with which it has been initialized. A reference type has to be necessarily initialized at the time of declaration and cannot be changed to refer to another object once it is initialized unlike in case of the pointers. The reference variable rf1 can be declared and initialized as shown below.

```
A a1;               //a1 is object of type class A defined above.
A& rf1=a1;          //rf1 is a reference variable declared and
                    //initialized to object a1.
rf1.ch='a';         // is equivalent to a1.ch='a';
rf1.f('a');         // is equivalent to a1.f('a');
```

3.6 MEMBER FUNCTIONS

Member functions of a class are the set of operations that may be applied to the objects of that class. A function declared as a member is called a member function and can only be used by the objects of that class.

For example, an object s1 of class SAMP defined earlier, can use the function member *void getin()* with a return type void for storing a character in data member *dat1*. Similarly the function member *void getout()* can be used to access the same.

Thus the member functions are nothing but the set of predefined operations that can be carried out on the data members. It acts as a interface to manipulate the data members.

Members functions are distinguished from ordinary functions by following attributes :

a. Member functions are defined within the scope of their class; ordinary functions are defined at file scope. This means that they are not visible outside the scope of the class.

b. Member functions have full access privilege to both the public and private members of the class while, in general, ordinary functions have access only to the public members of the class.

c. The member functions of one class, in general, have no access privileges to the members of another class.

3.7 CONSTRUCTORS AND DESTRUCTORS

Constructor is a special member function that has the same name as its class. It is executed automatically whenever an object of its type is created. It is mainly used for initialization of objects when they are created. A constructor like any ordinary function takes arguments, however no return type can be specified for a constructor. A constructor can also be used to initialize the data members unlike other functions. Constructors cannot be inherited, unlike other member functions.

The constructors can be defined as follows :

```
struct A
{
 int i;

 int j;

 A(k,l):i(k),j(l) { }
          //constructor, it has same name as class.
 .......

 .......
};
```

Here, A::A(k,l) acts as constructor. Using the constructor data member *i* is initialized with *k* and *j* is assigned with *l*.

Destructor is a special member function that has same name as its class preceded by a tilde. Conceptually a destructor reverses the effect of constructor. A destructor is used for doing special operations just before the destruction of an object. It is invoked automatically for an object prior to its destruction. Destructors cannot be inherited. A destructor takes no arguments and no return type can be specified for it. It is also invoked implicitly to deallocate all the objects in the file scope before program terminates.

The destructor can be defined as follows :

```
struct A
{
  static int ct;
  A() { ct++; }
  ~A() { ct--; }
   .......
};
int A::ct=0; // Initialized to zero.

void main() { ..... }
```

In above example, variable *ct* keeps the count of number of active objects. It is incremented by one, through constructor when an object of type A is created and decremented by one prior to destruction of an object of type A.

3.8 POD STRUCTURE / POD UNION

The POD struct / POD union in C++ ensures compatibility with C-struct / C-union. Formally a POD struct / POD union is same as C-struct / C-union which contain no constructor or destructor, no private or protected members, no virtual functions [refer sec:4.5], no base classes [refer sec:4.1], no references, and contain no pointers to members [Plu, 1993]. Consider the following example,

```
union A
{
 struct B
 {
   int j ;

   char c;
 } b1;

 struct C
 {
  int j;

  double d;
 } c1;
};
```

here, B and C are POD-structs and A is a POD-union.

3.9 SCOPE RULES FOR CLASSES

" Looking up names in C++ programs is a problem because of the need to reconcile with conflicting desire :

a. C programmers are accustomed to use a name from an outer scope and then redefine it later in the same scope.

b. Two nearby uses of the same name without an intervening definition of that name should mean the same thing. Moreover, a member function body explicitly written inline should mean the same thing when written out of line.

c. Reordering the members of a class should not change the meaning of the class for sake of understanding class definition easily.

Unfortunately, it is hard to meet all three of these criteria at once. ...." [Koe, 1992]. So after lot of consideration the following rules have been framed:-

1. <u>THE CLASS SCOPE RULE</u> : The scope of a name declared in a class consists not only of the text following the names declarator, but also of all functions bodies, default parameters and constructor initializers in that class (including such things in nested classes). For example consider,

```
#include <iostream.h>

/* PGM NAME :P921101C.CPP
   REFER TO :SEC:9.2.1, PARA:1, PG:9-6
```

```
    FEATURE   :THE SCOPE OF A NAME DECLARED IN A CLASS CONSISTS
               NOT ONLY OF THE TEXT FOLLOWING THE NAMES
               DECLARATOR, BUT ALSO DEFAULT PARARMETERS
               IN THAT CLASS, INCLUDING SUCH THINGS IN NESTED
               CLASSES.
*/

  int tf=0;

  class A
  {
   public:

    void f(int=i, char=ch);   //Default parameters.
    class AN
    {
     public:
     void f1(int=i, char=ch); //Default parameters.
    }a5;
    static int i;
    static char ch;
   };

  int A::i=5;
  char A::ch='a';

  void A::f(int i1, char ch1)
  {
   if(i1!=5){ cout<<"\n ERROR: LINE NO : "<<__LINE__
              <<"         VAR NAME :i1 "
              <<"\n ACTUAL VALUE : "<<i1
              <<" EXPECTED VALUE: 5 \n";tf=1;
        }
   if(ch1!='a'){ cout<<"\n ERROR: LINE NO : "<<__LINE__
                 <<"         VAR NAME :ch1 "
                 <<"\n ACTUAL VALUE : '"<<ch1
                 <<"'        EXPECTED VALUE:'a' \n";tf=1;
           }
  }

  void A::AN::f1(int i2, char ch2)
  {
   if(i2!=5){ cout<<"\n ERROR: LINE NO : "<<__LINE__
              <<"      VAR NAME :i2 "
              <<"\n ACTUAL VALUE : "<<i2
              <<"      EXPECTED VALUE: 5 \n";tf=1;
        }
   if(ch2!='a'){ cout<<"\n ERROR: LINE NO : "<<__LINE__
                 <<"     VAR NAME :ch2 "
                 <<"\n ACTUAL VALUE : '"<< ch2
```

```
                <<"'   EXPECTED VALUE: 'a' \n";tf=1;
         }
}

void main()
{
 A a1;
 a1.f();
 a1.a5.f1();
 if(tf)cout<<"\n TEST PGM P921101C.CPP FAILED \n";
}
```

As per the feature mentioned, the scope of static members *ch* and *i* declared in a class includes default parameters in the class and nested classes. In the functions A::f and A::AN::f defined above it is being checked whether it is actually so. If it is so then this program gives no output when it is executed or else gives the appropriate output, depending on which of the above if loops, the boolean condition is true.

2. <u>THE RECONSIDERATION RULE</u> : A name N used in a class S must refer to the same declaration when re-evaluated in its context and in the completed scope of S. For example,

```
1:     typedef int T;
2:     struct A
3:     {
4:      struct B
5:      {
6:       T f() { T x=0; return x; }
7:      };
8:      typedef double T;
9:     };
```

The reconsideration rule makes this program illegal [Sak, 1992]. Because when first encountered, the T in the declaration of f() on line 6 (but not the T in the body of f()) refers to ::T on line 1. However, B is not complete until A is completed on line 9. When f() is evaluated in the completed scope of B, the T in f()'s declaration refers to A::T on line 8.

3. <u>THE REORDERING RULE</u> : If reordering member declarations in a class yields an alternate valid program under the above two rules the program's meaning is undefined. For example look at the following [Koe , 1992]

```
struct y
{
 void f(long(p));

 typedef char p;
};
```

Assuming p does not already name type, the first use of p in the example above is as the name of f()'s formal parameter. But if we interchange its members

```
struct y
{
 typedef char p;

 void f(long(p));
};
```

the use of p in the declaration y::f above now does refer to y::p. Thus the swapping of these declarations has quietly changed the meaning of the class. Thus the above program's meaning is *undefined*, as per the reordering rule. The term undefined is used instead of *error* because the C++ standardization committee could not figure out for certain as to whether it is possible to detect two different valid programs in a single program that is written. It is likely that if someone comes up with an efficient algorithm to detect the same then the rule may be reconsidered.

3.10 NESTED CLASSES

A nested class is a class defined within another class. The name of a nested class is local to its enclosing class, that is, it is hidden within the class in which it is declared. A class is declared as a nested class if its use is limited to class within which it is defined. It has an advantage of minimizing the number of names in the global scope. For example consider the following,

```
class A
{
  ......
  class B { ..... } b1;
  public:
   .....
   class C { ..... } c1;
};
```

Here class A contains object b1 of nested type class B and object c1 of nested type class C as its members.

3.11 LOCAL CLASSES

A class defined within a function definition is called a local class. The member function of a local class must be defined within the class definition itself, since C++ does not support function defined within the function. A local class cannot have static data members. Consider the example below,

```
int i;

void h()
{
 .......

 .......

 class A
 {
  char c;
  public:

  char g1() { return c; }

  int g2() { return ::i }

 }a1;

 .......

 .......
}
```

Here, a local class A is defined within function h() and an object a1 of type class A is created. The scope of a local class is limited to its enclosing function scope. A class is declared local when its use is limited to the function within which it is defined. It has an advantage of minimizing the number of names in the global scope.

CHAPTER 4

OVERVIEW OF DERIVED CLASSES

4.1 INTRODUCTION

A concept does not exist in isolation; it co-exists with related concepts and derives much of its power from relationships with related concepts. Since we use classes to represent concepts the issue become how to represent relationships between concepts. The notion of a derived class and its associated language mechanisms is provided to express hierarchial relationships, that is, to express commonalty between classes [Str, 1991].

Derived classes provide a simple, flexible and efficient mechanism for defining a class by adding facilities to an existing class without reprogramming or recompilation. Using derived classes, one can provide a common interface for several different classes so that objects of those classes can be manipulated identically by other parts of a program. Consider the following example, if one is designing a software for library management of a university then the entities members, students, employees, faculty members, non teaching staff, under graduates, post graduates etc., have certain attributes in common. Also the rules governing their membership are common to some extent, so based on these one can form the class hierarchy as shown below:

$$\begin{tikzcd}
 & & & & & & \text{MEMBER} & & & & \\
 & & & \text{STUD} \arrow[urrr] & & & & & & \text{EMP} \arrow[ulll] & \\
 & \text{UG} \arrow[urr] & & & & \text{PG} \arrow[ull] & & & \text{FM} \arrow[ur] & & \text{NTM} \arrow[ul] \\
\text{PUG} \arrow[ur] & & \text{FUG} \arrow[ul] & & \text{PPG} \arrow[ur] & & \text{FPG} \arrow[ul] & & & &
\end{tikzcd}$$

CLASS HIERARCHY FOR LIBRARY MANAGEMENT SOFTWARE

where class MEMBER represents all members of library, class STUD represents all students members, class EMP represents all members who are employees of university, class FM represents faculty members, class NTM represents non teaching staff, class UG represents under graduate students, class PG represents post graduate students, class PUG represents part time under graduate students, class FUG represents full time under graduate students, class PPG represents part time post graduate students and class FPG represents full time post graduate students.

Based on above class hierarchy one can define attributes of each class so as to fully exploit the degree of commonality among various entities involved.

Inheritance is the process of creating new classes called *derived classes*, from existing classes which are then called the *base classes* of the derived classes. The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own. The base class remains unchanged by the process. The base class can be declared as *private, public,* and *protected* [refer sec:5.3]. For example consider,

```
struct base
{
 int i;
};

struct derived : base
{
   int j;
};
```

here, an object of class derived will have a sub-object of class base, where as objects of class base will not have any such sub-objects and they remain absolutely unaffected.

A derived class and its base classes can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from". This DAG is often referred to as class lattice.

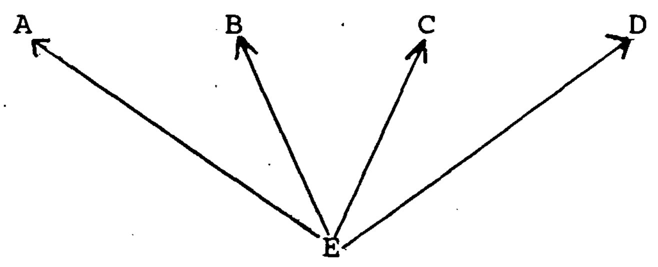
$$\begin{tikzcd}
\text{BASE} \\
\text{DERIVED} \arrow[u]
\end{tikzcd}$$

4.2 MULTIPLE BASE CLASSES

A class can be derived from multiple base classes. The use of more than one direct base class is called multiple inheritance. For example,

```
class A { .... };
class B { .... };
class C { .... };
class D { .... };
class E: public A, public B, public C, public D { .... };
```

here, an object of class E has sub-objects of class A, class B, class C and class D as shown below



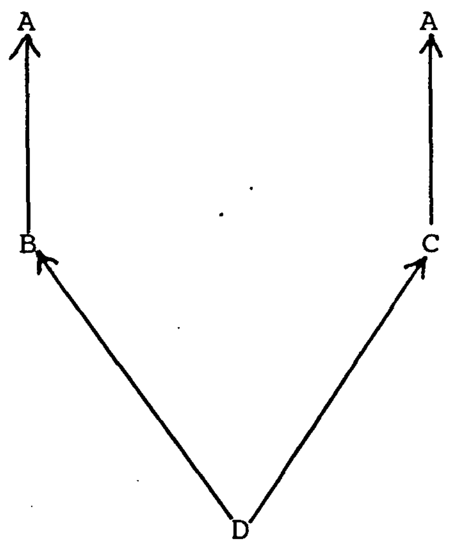
A class cannot be specified as a direct base class of a derived class more than once. For example,

```
class A { .... };
class B : public A, public A { .... };  //Not allowed.
```

However, a class can be an indirect base class more than once. For example,

```
class A { .... };
class B : public A  { .... };
class C : public A  { .... };
class D : public B, public C { .... };
```

Here, an object of class D has two sub-objects of class A as shown below



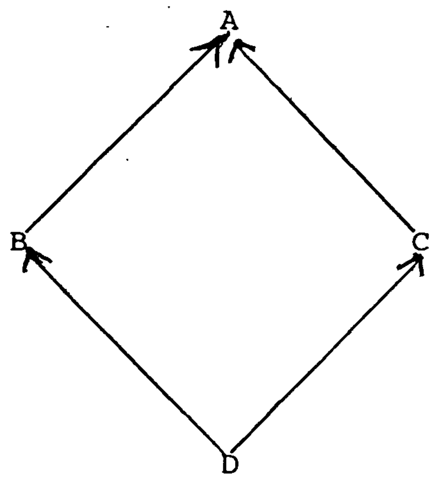
4.3 VIRTUAL BASE CLASS

In last example cited above an object of class D has two sub-objects of class A. Members of class A cannot be directly accessed from within class D because it will be ambiguous to do so, without specifying as to which of two sub-objects of class A one is referring to. This ambiguity can be eliminated by declaring base class A as virtual. Once a base class is specified as virtual then irrespective of number of places it is specified as virtual, all of them share a single sub-object of that virtual base class.

A base class is specified as virtual by modifying its declaration with the keyword *virtual*. For example,

```
class A { .... };
class B :virtual public A { .... };
class C :virtual public A { .... };
class D : public B, public C { .... };
```

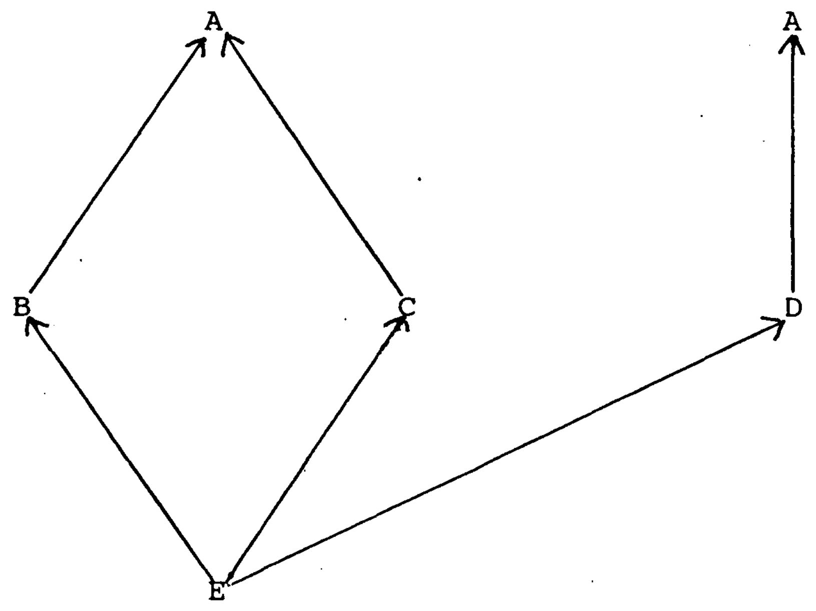
Here, an object of class D has one sub-objects of class A, as shown below



A class may have both virtual and non virtual base classes of a given type. Consider the example below,

```
class A { .... };
class B :virtual public A { .... };
class C :virtual public A { .... };
class D :public A { .... };
class E : public B, public C, public D { .... };
```

Here, an object of class E has two sub-objects of class A; class E's A and the virtual A shared by class B and class C, as shown below



4.4 AMBIGUITIES

Access to base class members from a derived class is ambiguous if the expression used refers to more than one enumerator, function, object or a type. Consider the following example,

```
struct A
{
 int i;

 int j;

 enum { E1,E2,E3 };

 char h1();

 int h2();
};

struct B
{
 int i;

 int j();

 enum { E2,E3,E1 };

 int h1();

 void h2();
};

class C : public A, public B {  };
```

then in the above example access to any of base class members through the object or pointer to derived class will be ambiguous since they are defined in both the classes.

Ambiguities can be resolved by qualifying a name with it class name. Like for example,

```
C c1;        // c1 object of class C is defined then use
```

```
c1.A::i or c1.B::i            // instead of c1.i;
c1.A::h1() or c1.B::h1()      // instead of c1.h1();
c1.A::E1 or c1.B::E1          // instead of c1.E1;
```

4.5 VIRTUAL FUNCTIONS

Virtual functions define type dependent operations within an inheritance hierarchy. Using virtual functions one can hide the implementation details of an inheritance hierarchy from the programs that make use of it.

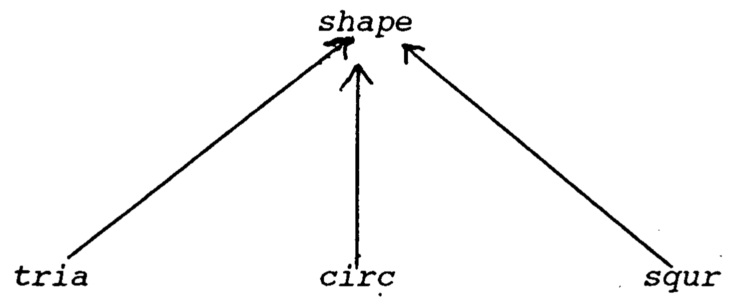
A virtual function is a special member function invoked through a public base class reference or pointer, it is bound dynamically at run time. The instance invoked is determined by the class type of the actual object addressed by the pointer or reference. Resolution of a virtual function is transparent to the user. A class that declares or inherits a virtual function is called a polymorphic class. Consider the following example,

```
class shape
{
 public:
 virtual void draw() { };
 .....
 .....
};
class tria : public shape
```

```
{
 public:
  void draw() { .... }              // draws a triangle.

   ......

   ......
};

class circ : public shape
{
 public:

  void draw() { .... }              // draws a circle.

   ......

   ......

};

class squr : public shape
{
 public:

  void draw() { .... }              // draws a square.

   ......

   ......
};

void main()
{
 tria t1;

 circ c1;

 squr s1;

 shape *ps;


 ps=&t1;

 ps->draw();   //will draw a triangle.


 ps=&c1;
```

```
  ps->draw();   //will draw a circle.
  ps=&s1;
  ps->draw();   //will draw a square.
}
```

Here, in the above example class hierarchy is as shown below



$$\begin{tikzcd}
 & shape & \\
tria \arrow[ur] & circ \arrow[u] & squr \arrow[ul]
\end{tikzcd}$$

The same function name *draw()* is used for first drawing triangle, then a circle and finally a square. The instance of *draw()* invoked when *ps->draw()* is used, is determined by the class type of the actual object addressed by the pointer *ps*.

4.6 ABSTRACT CLASSES

In the above example *tria::draw()* was used for drawing *triangle*, *circ::draw()* was used for drawing *circle* and *squr::draw()* was used for drawing a *square*, so in fact *shape::draw()* was used as an interface for which derived classes *tria*, *circ* and *squr* provided variety of implementations. In this situation it is better to define class *shape* as an abstract class rather than as a ordinary class for clarity.

The abstract class mechanism supports the notion of a general concept, such as a shape, of which only more concrete variants, such as circle, square etc., can actually be used. An abstract is a class that can be used only as a base class of some other class. No objects of an abstract class may be created except as a sub-objects of a class derived from it. A class is abstract if it has atleast one *pure virtual function* which may be inherited. A virtual function is specified pure by using a *pure-specifier* in the function declaration in the class declaration. The class Shape, in the example in earlier section, can be suitably redefined as an abstract class as follows :

```
class shape
{
 public:
  virtual void draw()=0;        //pure virtual function
  .....
  .....
};
```

In the declaration virtual void draw()=*0*, the equal sign has nothing to do with assignment; the value *0* is not assigned to anything. The =*0* syntax is simply to tell the compiler that a function will be *pure* .

An abstract class can neither be used as an function return type nor as an parameter type. For example if class A is an abstract class then,

```
A a1;     //ERROR: Object of abstract class cannot be created.
A mf1(); //ERROR: class A cannot be used as return type.
void mf2(A); //ERROR: class A cannot be used as an parameter
             //type.
```

CHAPTER 5

MEMBER ACCESS CONTROL

5.1 INTRODUCTION

The access control to members of a class is one of the important features of C++. Through the access control mechanism one can define different ways the data members of the class can be manipulated. This ensures a degree of modularity in any program that is written, which in turn makes the debugging of program easier. Further it provides the desired level of protection against accidental use of members of a class.

A member of a class can be *private*, *protected* or *public* :

A)Private:- If it is private, its name can be used only by member functions and friends of the class in which it is declared.

B)Protected:- If it is protected, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived form this class.

C)Public: If it is public, its name can be used by any function.

This reflects the view that there are three types of functions accessing a class:-

1. Functions implementing the class (its friends and members).

2. Functions implementing a derived class (the derived classes friends and members).

3. Other functions.

Members of a class declared with the key word *class* are *private* by default. Members of a class declared with the keywords *struct* or *union* are *public* by default.

5.2 ACCESS SPECIFIERS

Member declarations can be labelled by an access specifier. An access-specifier specifies the level of access for the members following it either till another access-specifier is encountered or until the end of the class. For example :

```
class A
{
 int i;      //A::i is private by default: 'class' used
 public:
  int j;     //A::j is public.
  char c;    //A::c is public.
};
```

In a class declaration access specifiers can be used any number of times and in any order. For example :

```
struct A
{
 int i;     //A::i is public by default : 'struct' used
 protected:
  char c;       //A::c is protected.
 private:
  float f;      //A::f is private.
 public :
  double d;     //A::d is public.
};
```

5.3 ACCESS SPECIFIER FOR BASE CLASSES

A base class can be declared *private*, *protected* or *public*. If a class is declared to be a base class for another class using

1) The *private* access specifier, then the protected and public members of the base class are accessible as private members of the derived class. The private members of base class are inaccessible to derived class. For example,

```
class B { .... };    // Base class.
struct D1 : private B { .... };  // Derived class.
```

2) The *protected* access specifiers, then the protected and public members of the base class are accessible as protected members of the derived class. The private members of base

class are inaccessible to derived class. For example,

```
class D2 : protected B { .... };
```

3) The *public* assess specifier, then the protected members of the base class are accessible as protected members of the derived class and the public members of the base class are accessible as public members of the derived class and the private members of the base class remain inaccessible to the derived class. For example,

```
class D5 : private B { .... };
```

A derived class can access the private members of its base classes only if it is a friend [refer sec:5.5] of its base classes. For example,

```
class D;        //Forward declaration.
class B
{
 double d;
 friend class D;
 ......
};
struct D : B
{
  void fd()
  {
   d=8.194; //  d is accessible since it is friend of B
}      };        }      };
```

here, in the function D::fd the private member *d* of B is assigned a value.

When no access specifier is used for a base class, private is assumed when the derived class is declared class and public is assumed when the derived class is declared class. For example,

```
class B { .... };
struct D1 : B { .... };  // B is public by default.
class D2 : B  { .... };  // B is private by default.
```

5.4 ACCESS DECLARATIONS

The protected or public members of a private or protected base class can be used at same level in derived class by making use of access declaration.

The access declaration of protected member of a private or protected base class must be given in the protected part of derived class. The access declaration of public member of a private or protected base class must be given in the public part of derived class. For example

```
class B
{
  protected :
    int i;
  public :
    char c;
};
```

```
class D1 : private B
{
 public :

  A::c;   //Restore access.

 protected:

  A::i;   //Restore access.
};
```

An access declaration cannot be used to enable access to a member that is inaccessible in the base class, nor can it be used to restrict access to a member that is accessible in the base class. For example,

```
class A
{
 public:

  int i;

 private:

  char c;
};

class D1 : private A
{
  public :

   A::c;  //Error: Cannot make c a public member of D1

  protected:

   A::i;  //Error: Cannot restrict access of i.
};
```

It is not possible to make a base class member accessible in a derived class using access declaration, if it already defines it. For example

```
class B
{
 public:

  char g();
};

class D1 : private B
{
 public:

  void g(int);

  B::g;  //Error: Two declarations of g;
};
```

5.5 FRIENDS

Friends are needed to enable unrestricted access to members of a class. A function declared as a friend is permitted to access even private and protected members of a class. It is not in the scope of class. It is called with a member access operator only if it is a member of another class. Friend declarations are not affected by access specifiers. For example,

```
class A
{
 friend void f(A);

 private:

  int i;

 protected:

  char c;

  public:

  void mf(B);
  A() { i=64; c='b'; }
 };

void f(A a1)
{
 if(a1.i!=64) { ....... }

 if(a1.c!='b') { ....... }
}

struct B
{
 friend void A::mf(B); //mf(B) is member of A.

 B() { j=37; ch='a'; }

 private:

  char ch;

 protected:

  int j;
};

 void A::mf(B b1)
 {
  if(b1.j!=37) { ...... }

  if(b1.ch!='a') { ..... }
 }

 void main()
 {
   A a;

   B b;

   f(a);

   a.mf(b);
 }
```

In the above example function void f(A) is allowed to access private and protected parts of class A since it is friend of class A and A::mf(B) is allowed to access the private and protected parts of class B since it is friend of class B.

Friendship is neither inherited nor transitive. For example,

```
class A1
{
 friend class A2;

 int j;
};

class A2
{
 friend class A3;
};

class A3
{
 void mf1(A1 a1)
 {
   a1.j=10;   //Error: A3 is not a friend of A1 despite

              //being a friend of A2.
 }
};

class D : public A2
{
 void mf2(A1 a)
 {
   a.j=19;    //Error: D is not a friend of A1 despite

              //being derived from a friend.
 }
};
```

5.6 PROTECTED MEMBER ACCESS

A friend or a member function of a derived class can access a protected non-static member of one of its base classes only through a pointer to, reference to, or object of the derived class itself. If this restriction is not there than it will be possible to access the base class part of an unrelated class as if it were its own, without the use of an explicit cast. However, this problem does not arise with the protected static member of a base class. Because in case of static members a single copy of it is shared by all the objects of its class and derived classes. Consider the following example.

```
class A
{
 protected:

 static int i;

 int j;
};

int A::i=0;

class B : public A  {   };

class C : public A
{
 public :

 friend void ff();

 void mf(B* , C*);
};

void ff()
{
 B b1;

 b1.i=10;     //Allowed since i is static.

 b1.j=5;      //Error: Because only objects of C can

              //access protected non static member of A.

 C c1;
 c1.j=1;     //Object of class C.
};

void C::mf(B *pb1, C *pc1)
{
 pb1->j=5;  //Error: Because only a pointer to C can

            //access protected non static member of A.

 pc1->j=1;  //pc1 is a pointer to class C.
};
```

CHAPTER 6

RESULTS AND CONCLUSIONS

6.1 RESULTS :

This verification suite consists of test programs based on most of the features mentioned in chapter 9, chapter 10, and chapter 11 of DWP. Test programs for few features could not be written because of following :

a. There is no point in testing a feature which is defined as undefined. Like in section 9.2.1.

b. There is no point in testing a feature which is defined as implementation dependent. Like in section 9.2 and section 9.6 of DWP.

This verification suite was used for functional testing of the Borland C++ version 3.1 compiler on a PC-AT. The following positive test programs have detected errors in it.

a. p921101a.cpp
b. p921101c.cpp
c. p921101d.cpp
e. p090701d.cpp
f. p100203a.cpp
g. p100203b.cpp
h. p100203c.cpp
i. p100203d.cpp
g. p100203e.cpp

The following is the list of negative test programs that have detected errors in Borland C++ version 3.1 compiler.

a. n921102a.cpp

b. n921102b.cpp

c. n921102e.cpp

d. n921102f.cpp

e. n110303a.cpp

f. n110303b.cpp

g. n110405.cpp

The following is the section wise analysis of the above test programs :

6.2 FAILURES RELATED TO NAME LOOKUP

```
 #include<iostream.h>

/* PGM NAME :P921101A.CPP
   REFER TO :SEC:9.2.1, PARA:1, PG:9-6
   FEATURE  :THE SCOPE OF A NAME DECLARED IN A CLASS CONSISTS
             NOT ONLY OF THE TEXT FOLLOWING THE NAMES
             DECLARATOR, BUT ALSO  OF ALL FUNCTION BODIES
             IN THAT CLASS, INCLUDING SUCH THINGS IN NESTED
             CLASSES.
*/

class A
{
  public:

   char g1() { char c=ch; return c; }
   int  g2() { int  j=i; return j; }

   class AN
   {
    public:
     char p1() { char c1=ch; return c1; }
     int  p2() { int i1=i; return i1; }
   } a5;

    static char ch;
    static int i;
};

char A::ch='a';
int A::i=10;


void main()
{
 int tf=0;          //Flag for test fail.
 A t1;

 if(t1.g1()!='a'){ cout<<"\n ERROR: LINE NO : "<<__LINE__
                   <<"        VAR NAME :t1.g1() "
                   <<"\n ACTUAL VALUE :  '"<< t1.g1()
                   <<"'       EXPECTED VALUE: 'a'\n";tf=1;
              }
 if(t1.g2()!=10){ cout<<"\n ERROR: LINE NO : "<<__LINE__
                  <<"        VAR NAME :t1.g2() "
                  <<"\n ACTUAL VALUE :  "<<t1.g2()
                  <<"        EXPECTED VALUE: 10\n";tf=1;
             }

 if(t1.a5.p1()!='a'){ cout<<"\n ERROR: LINE NO : "<<__LINE__
                      <<"    VAR NAME :t1.a5.p1() "
                      <<"\n ACTUAL VALUE :  '"<<t1.a5.p1()
                      <<"'    EXPECTED VALUE: 'a'\n";tf=1;
                  }
 if(t1.a5.p2()!=10){ cout<<"\n ERROR: LINE NO : "<<__LINE__
                     <<"     VAR NAME :t1.a5.p2() "
                     <<"\n ACTUAL VALUE :  "<<t1.a5.p2()
                     <<"     EXPECTED VALUE: 10\n";tf=1;
                }

 if(tf)cout<<"\n TEST PGM P921101A FAILED \n";
}
```

As per the feature mentioned, the scope of static members *ch* and *i* declared in a class includes all function bodies in the class and nested classes. As such the above program contains no errors. However, the Borland C++ compiler gives errors as shown below :

```
                **** ****  REPORT  OF  TEST  **** ****

Borland C++   Version 3.1 Copyright (c) 1992 Borland
International
p921101a.cpp:
Error p921101a.cpp 21: Undefined symbol 'ch' in function
A::AN::p1()
Error p921101a.cpp 22: Undefined symbol 'i' in function
A::AN::p2()
*** 2 errors in Compile ***

        Available memory 1618007

                        **** ****  -X-  **** ****
```

```
#include <iostream.h>

/* PGM NAME :P921101C.CPP
   REFER TO :SEC:9.2.1, PARA:1, PG:9-6
   FEATURE  :THE SCOPE OF A NAME DECLARED IN A CLASS CONSISTS
             NOT ONLY OF THE TEXT FOLLOWING THE NAMES
             DECLARATOR, BUT ALSO DEFAULT PARAMETERS
             IN THAT CLASS, INCLUDING SUCH THINGS IN NESTED
             CLASSES.
*/

int tf=0;

class A
{
 public:

  void f(int=i, char=ch);   //Default parameters.
  class AN
```

```
   {
    public:
    void f1(int=i, char=ch); //Default parameters.
   }a5;
   static int i;
   static char ch;
  };

int A::i=5;
char A::ch='a';

void A::f(int i1, char ch1)
{
 if(i1!=5){ cout<<"\n ERROR: LINE NO : "<<__LINE__
            <<"        VAR NAME :i1 "
            <<"\n ACTUAL VALUE : "<<i1
            <<" EXPECTED VALUE: 5 \n";tf=1;
      }
 if(ch1!='a'){ cout<<"\n ERROR: LINE NO : "<<__LINE__
               <<"        VAR NAME :ch1 "
               <<"\n ACTUAL VALUE : '"<<ch1
               <<"'       EXPECTED VALUE:'a' \n";tf=1;
         }
}

void A::AN::f1(int i2, char ch2)
{
 if(i2!=5){ cout<<"\n ERROR: LINE NO : "<<__LINE__
            <<"     VAR NAME :i2 "
            <<"\n ACTUAL VALUE : "<<i2
            <<"     EXPECTED VALUE: 5 \n";tf=1;
      }
 if(ch2!='a'){ cout<<"\n ERROR: LINE NO : "<<__LINE__
               <<"     VAR NAME :ch2 "
               <<"\n ACTUAL VALUE : '"<< ch2
               <<"'    EXPECTED VALUE: 'a' \n";tf=1;
         }
}

void main()
{
 A a1;
 a1.f();
 a1.a5.f1();
 if(tf)cout<<"\n TEST PGM P921101C.CPP FAILED \n";
}
```

As per the feature mentioned, the scope of static members *ch* and *i* declared in a class includes all default parameters in the class and nested classes. As such the above program contains no errors. However, the Borland C++ compiler gives errors as shown below :

```
                **** ****  REPORT  OF  TEST  **** ****

Borland  C++   Version  3.1  Copyright  (c)  1992  Borland
International
p921101c.cpp:
Error p921101c.cpp 19: Undefined symbol 'i'
Error p921101c.cpp 19: Undefined symbol 'ch'
*** 2 errors in Compile ***

        Available memory 1619040
                      **** ****  -X-  **** ****
```

```
#include <iostream.h>

/* PGM NAME :P921101D.CPP
   REFER TO :SEC:9.2.1, PARA:1, PG:9-6
   FEATURE  :THE SCOPE OF A NAME DECLARED IN A CLASS CONSISTS
             NOT ONLY OF THE TEXT FOLLOWING THE NAMES
             DECLARATOR, BUT ALSO CONSTRUCTOR INITIALIZERS IN
             THAT CLASS, INCLUDING  SUCH THINGS IN NESTED
             CLASSES.
 */

class A
{
  public:
   char ch1;
   int i1;
   A():i1(i),ch1(ch){ }
   class AN
```

```
    {
     public:
     char ch2;
     int i2;
     AN():i2(i),ch2(ch){ }
    } a5;

  static char ch;
  static int i;
};

char A::ch='a';
int A::i=10;


void main()
{
 int tf=0;          //Flag for test fail.
 A t1;

 if(t1.ch1!='a'){ cout<<"\n ERROR: LINE NO : "<<__LINE__
              <<"      VAR NAME :t1.ch1 "
              <<"\n ACTUAL VALUE : '"<<t1.ch1
              <<"'      EXPECTED VALUE:'a' \n";tf=1;
         }
 if(t1.i1!=10){ cout<<"\n ERROR: LINE NO : "<<__LINE__
           <<"      VAR NAME :t1.i1 "
           <<"\n ACTUAL VALUE : "<<t1.i1
           <<" EXPECTED VALUE: 10 \n";tf=1;
      }

 if(t1.a5.ch2!='a'){ cout<<"\n ERROR: LINE NO : "<<__LINE__
              <<"      VAR NAME :t1.a5.ch2 "
              <<"\n ACTUAL VALUE : '"<<t1.a5.ch2
              <<"'      EXPECTED VALUE:'a' \n";tf=1;
         }
 if(t1.a5.i2!=10){ cout<<"\n ERROR: LINE NO : "<<__LINE__
           <<"      VAR NAME :t1.a5.i2 "
           <<"\n ACTUAL VALUE : "<<t1.a5.i2
           <<" EXPECTED VALUE: 10 \n";tf=1;
      }

 if(tf)cout<<"\n TEST PGM P921101D.CPP FAILED \n";
}
```

As per the feature mentioned, the scope of static members *ch* and *i* declared in a class includes constructor initializers in that class and nested classes. As such the above program contains no errors. However, the Borland C++ compiler gives errors as shown below :

```
                **** ****  REPORT  OF  TEST  **** ****


Borland C++   Version 3.1 Copyright (c) 1992 Borland
International
p921101d.cpp:
Error p921101d.cpp 21: Undefined symbol 'i' in function
A::AN::AN()
Error p921101d.cpp 21: Undefined symbol 'ch' in function
A::AN::AN()
*** 2 errors in Compile ***

        Available memory 1620328
                      **** ****  -X-  **** ****
```

```
#include<iostream.h>

/* PGM NAME :N921102A.CPP
   REFER TO :SEC:9.2.1, PARA:2, PG:9-6
   FEATURE  :A NAME N USED IN A CLASS S MUST REFER TO THE
             SAME DECLARATION WHEN RE-EVALUATED IN ITS
             CONTEXT AND IN THE COMPLETED SCOPE OF S.
 */

enum { i=1 };
class A
{
 char v[i];      //ERROR: 'i' REFERS TO ::i
                //BUT WHEN RE-EVALUATED IS A::i
enum { i=2 };
};

void main() { }
```

The above program is illegal as per the feature mentioned above, because when first encountered, the *i* in the declaration of *v* on line 4 refers to global *i* on line 1. However, when it is re-evaluated in the completed scope of class *A* it refers to declaration on line 5. Thus it is illegal as per feature mentioned above. However, the Borland C++ compiler does not give either an error or a warning as shown below :

```
              **** ****  REPORT  OF  TEST  **** ****

Borland C++   Version 3.1 Copyright (c) 1992 Borland
International
n921102a.cpp:
Turbo Link   Version 5.1 Copyright (c) 1992 Borland
International

    Available memory 1623600

                      **** ****  -X-  **** ****
```

```
#include<iostream.h>

/* PGM NAME :N921102B.CPP
   REFER TO :SEC:9.2.1, PARA:2, PG:9-6
   FEATURE  :A NAME N USED IN A CLASS S MUST REFER TO THE
              SAME DECLARATION WHEN RE-EVALUATED IN ITS
              CONTEXT AND IN THE COMPLETED SCOPE OF S.
 */

1: typedef char *T;

2: class A
```

```
3: {
4:  T a;            //ERROR: 'T' REFERS TO ::T
                    //BUT WHEN RE-EVALUATED IS A::T
5:  typedef long T;
6:  T b;
7: };
   void main() { }
```

The above program is illegal as per the feature mentioned above, because when first encountered, the *T* in the declaration of *a* on line 4 refers to global *T* on line 1. However, when it is re-evaluated in the completed scope of class *A* it refers to declaration on line 5. Thus it is illegal as per feature mentioned above. However, the Borland C++ compiler does not give either an error or a warning as shown below :

```
                **** ****  REPORT  OF  TEST  **** ****


Borland C++   Version  3.1  Copyright  (c)  1992  Borland
International
n921102b.cpp:
Turbo Link   Version  5.1  Copyright  (c)  1992  Borland
International

    Available memory 1623600

                        **** ****  -X-  **** ****
```

```
/* PGM NAME :N921102E.CPP
   REFER TO :SEC:9.2.1, PARA:2, PG:9-6
   FEATURE  :A NAME N USED IN A CLASS S MUST REFER TO THE
             SAME DECLARATION WHEN RE-EVALUATED IN ITS
```

```
                CONTEXT AND IN THE COMPLETED SCOPE OF S.
 */

1: typedef int **I;
2: struct A
3: {
4:  struct B
5:  {
6:   I f() { I i=0; return i; }
7:  };
8:  typedef float I;
9: };
   void main() { }
```

The above program is illegal as per the feature mentioned above, because when first encountered, the *I* in the declaration of *f* on line 6 (but not the I in the body of f) refers to global *I* on line 1. However, B is not complete until A is completed on line 9. when f is re-evaluated in the completed scope of *B*, the I in f's declaration refers to A::I declaration on line 8. Thus it is illegal as per feature mentioned above. However, the Borland C++ compiler does not give either an error or a warning as shown below :

```
                **** ****  REPORT  OF  TEST  **** ****

Borland C++   Version 3.1 Copyright (c) 1992 Borland
International
n921102e.cpp:
Turbo Link   Version 5.1 Copyright (c) 1992 Borland
International

    Available memory 1694364
```

**** **** -X- **** ****

```
/* PGM NAME :N921102F.CPP
   REFER TO :SEC:9.2.1, PARA:2, PG:9-6
   FEATURE  :A NAME N USED IN A CLASS S MUST REFER TO THE
              SAME  DECLARATION WHEN RE-EVALUATED IN ITS
              CONTEXT AND IN THE COMPLETED SCOPE OF S.
 */

1: char **T;
2: struct A
3: {
4:  char s[ sizeof(T) ];
5:  int T;
6: };
   void main() { }
```

The above program is illegal as per the feature mentioned above because when first encountered, the *T* in the declaration of *s* on line 4 refers to global *T* on line 1. However, when it is re-evaluated in the completed scope of *A* it refers to declaration on line 5. Thus it is illegal as per feature mentioned above. However, the Borland C++ compiler does not give either an error or a warning as shown below :

**** **** REPORT OF TEST **** ****

```
Borland C++   Version 3.1 Copyright (c) 1992 Borland
International
n921102f.cpp:
Turbo Link   Version 5.1 Copyright (c) 1992 Borland
```

```
International

    Available memory 1695468
```

**** **** -X- **** ****

6.3 FAILURES RELATED TO NESTED CLASSES

```
#include<iostream.h>

/* PGM NAME :P090701D.CPP
   REFER TO :SEC:9.7, PARA:1, PG:9-12.
   FEATURE :A NESTED CLASS MAY BE DECLARED IN A CLASS AND
             LATER DEFINED IN THE SAME OR AN ENCLOSING
             SCOPE.
 */

    class A
    {
     public:
      class B;       //FORWARD DECLARATION OF NESTED CLASS.
15:   class C;
      class B        //DEFINITION OF NESTED CLASS.
      {
       public:
        B(){ i=100; }
        int i;
        int f() { return i; }
      };
     };
 25: class A::C       //DEFINITION OF NESTED CLASS.
     {
      public:
       C() { h='a'; }
       char h;
       char g() { return h; }
     };

    void main()
```

```
     {
      int tf=0;      //Flag for test fail.

      A::B b;
      if(b.f()!=100){ cout<<"\n LINE NO: "<<__LINE__; tf=1; }

      A::C c;
40:   if(c.g()!='a'){ cout<<"\n LINE NO: "<<__LINE__; tf=1; }

      if(tf)cout<<"\n TEST FAILED. \n";
     }
```

In the program listed above, the nested class C is declared on line 15 and later defined on line 25. On line 40 the function member of object *c* of class C is being used. As per the feature mentioned their are no errors in the above program. However, the Borland C++ compiler gives errors as shown below :

```
                    **** ****  REPORT  OF  TEST  **** ****

Borland C++   Version 3.1 Copyright (c) 1992 Borland
International
p090701d.cpp:
Error p090701d.cpp 25: Multiple declaration for 'A::C'
Error p090701d.cpp 40: 'g' is not a member of 'C' in function
main()
*** 2 errors in Compile ***

    Available memory 1620772

                         **** ****  -X-  **** ****
```

6.4 FAILURES RELATED TO VIRTUAL FUNCTIONS

```
/* PGM NAME :P100203A.CPP
   REFER TO :SEC:10.2, PARA:3, PG:10-6.
   FEATURE  :IT IS A DIAGNOSABLE ERROR FOR THE RETURN TYPE OF
             AN OVERRIDING FUNCTION TO DIFFER FROM THE RETURN
             TYPE OF THE OVERRIDDEN FUNCTION UNLESS THE
             RETURN TYPE OF THE OVERRIDDEN FUNCTION IS
             POINTER OR REFERENCE TO (POSSIBLY CV-QUALIFIED)
             A CLASS A, AND THE RETURN TYPE OF THE OVERRIDING
             FUNCTION IS POINTER OR REFERENCE (RESPECTIVELY)
             TO CLASS B SUCH THAT A IS AN UNAMBIGUOUS DIRECT
             OR INDIRECT BASE CLASS OF B, ACCESSIBLE IN THE
             CLASS OF THE OVERRIDING FUNCTION, AND THE CV-
             QUALIFICATION IN THE RETURN TYPE OF THE
             OVERRIDING FUNCTION IS LESS THAN OR EQUAL TO THE
             CV-QUALIFICATION IN THE RETURN TYPE OF THE
             OVERRIDDEN FUNCTION. IN THAT CASE WHEN THE
             OVERRIDING FUNCTION IS CALLED AS THE FINAL
             OVERRIDER OF THE  OVERRIDDEN FUNCTION, ITS
             RESULT IS CONVERTED TO THE TYPE RETURNED BY THE
             (STATICALLY CHOSEN) OVERRIDDEN FUNCTION.


*/

class A { };

struct B : private A      //Line no: 28
{
 B() { i=55; }
 int i;
 friend class D;
};

struct C
{
 virtual A* vf() { A a1; return(&a1); }
};

 class D : public C
 {
  public:
   B* vf() { B b1; return(&b1); }
          //Legal:A which is a direct base class of B is
          //accessible in class D.
 };

 void g()
```

```
{
 D d;
 B* bp=d.vf();
 if(bp->i!=55){ cout<<"\n ERROR: LINE NO : "<<__LINE__
                    <<"     VAR NAME : bp->i"
                    <<"\n ACTUAL VALUE :  "<<bp->i
                    <<"     EXPECTED VALUE: 55 \n"
                    <<"\n TEST PGM P100203A.CPP FAILED. \n";
               }

}
void main()
{
 g();
}
```

In the above test program D::vf() does not conflict with the base C::vf() because the return type of overridden function C::vf() is pointer to class A and return type of overriding function D::vf() is pointer to class B, where A is direct base class of B, and is accessible in the class D because class D is declared as friend of class B on line 28. Further the CV-qualification of the return type of overriding function is same as that of overridden function. Thus as per the feature mentioned above their is no error in the program. However, the Borland C++ gives errors as shown below:

```
**** ****  REPORT  OF  TEST  **** ****


Borland C++   Version 3.1   Copyright  (c)  1992  Borland
International
p100203a.cpp:  Error  p100203a.cpp  43:   Virtual   function
'D::vf()' conflicts  with base class 'C'
```

```
*** 1 errors in Compile ***

   Available memory 1619468
                    **** ****  -X-  **** ****
```

Similar errors were reported by Borland C++ compiler in test programs p100203b.cpp, p100203c.cpp, p100203d.cpp, and p100203e.cpp which are all based on above feature. In all these programs only the CV-qualification of the return type pointers of overriding and overridden functions are different compared to above listed program. However, there are no errors in them as per feature mentioned above.

6.5 FAILURES RELATED TO ACCESS CONTROL

```
/* PGM NAME   :N110303A.CPP
   REFER TO   :SEC:11.3, PARA:3, PG:11-4.
   FEATURE    :AN ACCESS DECLARATION MAY NOT BE USED TO
               RESTRICT ACCESS TO A MEMBER THAT IS ACCESSIBLE
               IN THE BASE CLASS, NOR MAY IT BE USED TO ENABLE
               ACCESS TO A MEMBER THAT IS NOT ACCESSIBLE IN
               THE BASE CLASS.
 */

class A
{
 public:
  float f;
};

class B : private A
{

};

class C : private B
{
 public:
  A::f;          //ERROR: Attempt to grant access.
};
void main() { }
```

In the above program attempt is being made to grant access to A::f in C, using access declaration, even though it is not accessible in B which is the immediate base class of C. As per feature mentioned above it is illegal to do so. However, the Borland C++ compiler does not give either an error or a warning as shown below :

**** **** REPORT OF TEST **** ****

```
Borland C++   Version 3.1 Copyright (c) 1992 Borland
International
n110303a.cpp:
Turbo Link   Version 5.1 Copyright (c) 1992 Borland
International

    Available memory 1698484
```

**** **** -X- **** ****

```
/* PGM NAME   :N110303B.CPP
   REFER TO   :SEC:11.3, PARA:3, PG:11-4.
   FEATURE    :AN ACCESS DECLARATION MAY NOT BE USED TO
              RESTRICT ACCESS TO A MEMBER THAT IS ACCESSIBLE
              IN THE BASE CLASS, NOR MAY IT BE USED TO ENABLE
              ACCESS TO A MEMBER THAT IS NOT ACCESSIBLE IN
              THE BASE CLASS.
```

```
 */

class A
{
 protected:
  float f;
};
class B : private A
{

};

class C : private B
{
 protected:
  A::f;          //ERROR: Attempt to grant access.
};

void main() { }
```

In the above program attempt is being made to grant access to A::f in C, using access declaration, even though it is not accessible in B which is the immediate base class of C. As per feature mentioned above it is illegal to do so. However, the Borland C++ compiler does not give either an error or a warning as shown below :

```
                **** ****  REPORT  OF  TEST  **** ****

Borland C++   Version 3.1 Copyright (c) 1992 Borland
International
n110303b.cpp:
Turbo Link   Version 5.1 Copyright (c) 1992 Borland
International

    Available memory 1698468


                        **** ****  -X-  **** ****
```

6.6 FAILURES RELATED TO FRIENDS

```
#include <iostream.h>

/* PGM NAME :N110405.CPP
   REFER TO :SEC:11.4, PARA:5, PG:11-6.
   FEATURE :A GLOBAL FRIEND FUNCTION MAY BE DEFINED IN A
             CLASS DEFINITION OTHER THAN A LOCAL CLASS
             DEFINITION.
 */

void g()
{
 class A
 {
  public:
   int i;
   A() { i=1; }
   friend int f(A a1) { return a1.i; } //ERROR:
 };
 A a2;
 if(f(a2)!=1){ cout<<"\n ERROR: LINE NO: "<<__LINE__
                   <<"\n TEST FAILED. \n"; }

void main()
{
 g();
}
```

In the program listed above a friend function *f* is defined in a local class. But as per the feature mentioned above it is illegal to do so. However, the Borland C++ compiler does not give either an error or a warning as shown below :

```
              **** ****  REPORT  OF  TEST  **** ****

Borland C++  Version 3.1 Copyright (c) 1992 Borland
International
n110405.cpp:
```

```
Turbo Link  Version 5.1 Copyright (c) 1992 Borland
International

    Available memory 1619272
```

**** **** -X- **** ****

6.7 CONCLUSION:

Thus to sum up the following defects were observed in Borland C++ version 3.1 compiler.

a. The name and scope rules have not been implemented by it.

b. It does not allow a nested class to be defined outside the class in which it has been declared.

c. A problem involving return type of virtual functions have been detected.

d. A problem regarding granting of access using access declaration has been detected.

e. It allows global friend functions to be defined in a local class definition.

The verification suite was able to detect so few errors in the Borland C++ compiler version 3.1 because it is an industrial strength product which has already been extensively tested. However, we expect a higher number of errors for compilers under development.

This verification suite consists of test programs for only a part of DWP. This work can be carried over further and test programs can be written based on other chapters, e.g., templates, exception handling etc.

APPPENDIX A

RELEVANT PORTION OF DWP

9 Classes

1 A *class* is a user-defined type. A class definition specifies the representation of objects of the class and the set of operations that can be applied to such objects. This chapter presents the syntax and semantics for simple classes.
2 The definition of both `static` and non-`static` members is discussed, and the scope rules involving classes and functions – including local and nested classes containing member functions – are described. The mechanisms for controlling the layout of class objects, for conforming to externally imposed formats, and for maintaining compatibility with C layouts (`structs`, `unions` and bit-fields) are presented.
3 Derived classes (that is, inheritance), access control, and special member functions are discussed in the next three chapters.

9 Classes

1 A class is a type. Its name becomes a *class-name* (§9.1), that is, a reserved word within its scope.

class-name:
    identifier
    template-class-id

Class-specifiers and *elaborated-type-specifiers* (§7.1.6) are used to make *class-names*. An object of a class consists of a (possibly empty) sequence of members.

class-specifier:
    class-head { *member-specification*$_{opt}$ }

class-head:
    class-key identifier$_{opt}$ *base-clause*$_{opt}$
    class-key nested-class-specifier base-clause$_{opt}$

class-key:
```
class
struct
union
```

2 The name of a class can be used as a *class-name* even within the *member-specification* of the class specifier itself. A *class-specifier* is commonly referred to as a class definition. A class is considered defined when its *class-specifier* has been seen even though its member functions are in general not yet defined.

3 Objects of an empty class have a nonzero size.

4 Class objects may be assigned, passed as arguments to functions, and returned by functions (except objects of classes for which copying has been restricted; see §12.8). Other plausible operators, such as equality comparison, can be defined by the user; see §13.4.

5 A *structure* is a class declared with the *class-key* `struct`; its members and base classes (§10) are public by default (§11). A *union* is a class declared with the *class-key* `union`; its members are public by default and it holds only one member at a time (§9.5).

9.1 Class Names

1 A class definition introduces a new type. For example,

```
struct X { int a; };
struct Y { int a; };
X a1;
Y a2;
int a3;
```

declares three variables of three different types. This implies that

```
a1 = a2;        // error: Y assigned to X
a1 = a3;        // error: int assigned to X
```

are type mismatches, and that

```
int f(X);
int f(Y);
```

declare an overloaded (§13) function `f()` and not simply a single function `f()` twice. For the same reason,

```
struct S { int a; };
struct S { int a; };  // error, double definition
```

is an error because it defines `S` twice.

2 A class definition introduces the class name into the scope where it is defined and hides any class, object, function, or other declaration of that name in an enclosing scope (§3.2). If a class name is declared in a scope where an object, function, or enumerator of the same name is also declared the class can be referred to only using an *elaborated-type-specifier* (§7.1.6). For example,

```
struct stat {
    // ...
};

stat gstat;               // use plain 'stat' to
                          // define variable

int stat(struct stat*); // redefine 'stat' as function

void f()
{
    struct stat* ps;      // 'struct' prefix needed
                          // to name struct stat
    // ...
    stat(ps);             // call stat()
    // ...
}
```

An *elaborated-type-specifier* with a *class-key* used without declaring an object or function introduces a class name exactly like a class definition but without defining a class. For example,

```
struct s { int a; };

void g()
{
    struct s;  // hide global struct 's'
    s* p;      // refer to local struct 's'
    struct s { char* p; };  // declare local struct 's'
}
```

Such declarations allow definition of classes that refer to each other. For example,

```
class vector;

class matrix {
    // ...
    friend vector operator*(matrix&, vector&);
};

class vector {
    // ...
    friend vector operator*(matrix&, vector&);
};
```

Declaration of `friends` is described in §11.4, operator functions in §13.4.

3 An *elaborated-type-specifier* (§7.1.6) can also be used in the declarations of objects and functions. It differs from a class declaration in that if a class of the elaborated name is in scope the elaborated name will refer to it. For example,

```
struct s { int a; };

void g(int s)
{
    struct s* p = new struct s;    // global 's'
    p->a = s;                      // local 's'
}
```

4 A name declaration takes effect immediately after the *identifier* is seen. For example,

```
class A * A;
```

first specifies `A` to be the name of a class and then redefines it as the name of a pointer to an object of that class. This means that the elaborated form `class A` must be used to refer to the class. Such artistry with names can be confusing and is best avoided.

5 A *typedef-name* (§7.1.3) that names a class is a *class-name*; see also §7.1.3.

9.2 Class Members

member-specification:
  member-declaration member-specification$_{opt}$
  access-specifier : *member-specification$_{opt}$*

member-declaration:
  decl-specifier-seq$_{opt}$ member-declarator-list$_{opt}$;
  function-definition ;$_{opt}$
  qualified-id ;

member-declarator-list:
  member-declarator
  member-declarator-list , *member-declarator*

member-declarator:
  declarator pure-specifier$_{opt}$
  identifier$_{opt}$: *constant-expression*

pure-specifier:
  = 0

1 The *member-specification* in a class definition declares the full set of members of the class; no member can be added elsewhere. Members of a class are data members, member functions (§9.3), nested types, and member constants. Data members and member functions are static or nonstatic; see §9.4. Nested types are classes (§9.1, §9.7) and enumerations (§7.2) defined in the class, and arbitrary types declared as members by use of a typedef declaration (§7.1.3). The enumerators of an enumeration (§7.2) defined in the class are member constants of the class. Except when used to declare friends (§11.4) or to adjust the access to a member of a base class (§11.3), *member-declarations* declare members of the class, and each such *member-declaration* must declare at least one member name of the class. A member may not be declared twice in the *member-specification*, except that a nested class may be declared and then later defined.

2 Note that a single name can denote several function members provided their types are sufficiently different (§13). Note that a *member-declarator* cannot contain an *initializer* (§8.4). A member can be initialized using a constructor; see §12.1.

3 A member may not be `auto`, `extern`, or `register`.

4 The *decl-specifier-seq* can be omitted in function declarations only. The *member-declarator-list* can be omitted only after a *class-specifier*, an *enum-specifier*, or a *decl-specifier-seq* of the form `friend` *elaborated-type-specifier*. A *pure-specifier* may be used only in the declaration of a virtual function (§10.2).

5 Non-`static` (§9.4) members that are class objects must be objects of previously declared classes. In particular, a class `cl` may not contain an object of class `cl`, but it may contain a pointer or reference to an object of class `cl`. When an array is used as the type of a nonstatic member all dimensions must be specified.

6 A simple example of a class definition is

```
struct tnode {
    char tword[20];
    int count;
    tnode *left;
    tnode *right;
};
```

which contains an array of twenty characters, an integer, and two pointers to similar structures. Once this definition has been given, the declaration

```
tnode s, *sp;
```

declares `s` to be a `tnode` and `sp` to be a pointer to a `tnode`. With these declarations, `sp->count` refers to the `count` member of the structure to which `sp` points; `s.left` refers to the `left` subtree pointer of the structure `s`; and `s.right->tword[0]` refers to the initial character of the `tword` member of the `right` subtree of `s`.

7 Nonstatic data members of a class declared without an intervening *access-specifier* are allocated so that later members have higher addresses within a class object. The order of allocation of nonstatic data members separated by an *access-specifier* is implementation dependent (§11.1). Implementation alignment requirements may cause two adjacent members not to be allocated immediately after each other; so may requirements for space for managing virtual functions (§10.2) and virtual base classes (§10.1); see also §5.4.

8 If two types `T1` and `T2` are the same type, then `T1` and `T2` are *layout-compatible* types.

9 Two POD-struct (§8.4.1) types are layout-compatible if they have the same number of members, and corresponding members (in order) have layout-compatible types.

10 Two POD-union (§8.4.1) types are layout-compatible if they have the same number of members, and corresponding members (in any order) have layout-compatible types.

Shouldn't this be the same *set* of types?

11 Two enumeration types are layout-compatible if they have the same sets of enumerator values.

Shouldn't this be the same *underlying type*?

12 If a POD-union contains several POD-structs that share a common initial sequence, and if the POD-union object currently contains one of these POD-structs, it is permitted to inspect the common initial part of any of them. Two POD-structs share a common initial sequence if corresponding members have layout-compatible types (and, for bit-fields, the same widths) for a sequence of one or more initial members.

13 A pointer to a POD-struct object, suitably converted, points to its initial member (or if that member is a bit-field, then to the unit in which it resides) and vice versa. There may therefore be unnamed padding within a POD-struct object, but not at its beginning, as necessary to achieve appropriate alignment.

14 The range of nonnegative values of a signed integral type is a subrange of the corresponding unsigned integral type, and the representation of the same value in each type is the same.

15 Even if the implementation defines two or more basic types to have the same representation, they are nevertheless different types.

16 The representations of integral types shall define values by use of a pure binary numeration system.

Does this mean two's complement? Is there a definition of "pure binary numeration system?"

17 The qualified or unqualified versions of a type are distinct types that have the same representation and alignment requirements.

18 A qualified or unqualified `void*` shall have the same representation and alignment requirements as a qualified or unqualified `char*`.

19 Similarly, pointers to qualified or unqualified versions of layout-compatible types shall have the same representation and alignment requirements.

20 If the program attempts to access the stored value of an object other than through an lvalue of one of the following types:

- the declared type of the object,
- a qualified version of the declared type of the object,
- a type that is the signed or unsigned type corresponding to the declared type of the object,
- a type that is the signed or unsigned type corresponding to a qualified version of the declared type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type.[17]

the result is undefined.

21 A function member (§9.3) with the same name as its class is a constructor (§12.1). A static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class.

[17] The intent of this list is to specify those circumstances in which an object may or may not be aliased.

9.2.1 Scope Rules for Classes

1 The following rules describe the scope of names declared in classes.

1. The scope of a name declared in a class consists not only of the text following the name's declarator, but also of all function bodies, default parameters, and constructor initializers in that class (including such things in nested classes).

2. A name N used in a class S must refer to the same declaration when re-evaluated in its context and in the completed scope of S.

3. If reordering member declarations in a class yields an alternate valid program under (1) and (2), the program's meaning is undefined.

4. A declaration in a nested declarative region hides a declaration whose declarative region contains the nested declarative region.

5. A declaration within a member function hides a declaration whose scope extends to or past the end of the member function's class.

6. The scope of a declaration that extends to or past the end of a class definition also extends to the regions defined by its member definitions, even if defined lexically outside the class (this includes both function member bodies and static data member initializations).

2 For example:

```
typedef int  c;
enum { i = 1 };

class X {
    char  v[i];  // error: 'i' refers to ::i
                 // but when reevaluated is X::i
    int  f() { return sizeof(c); }  // okay: X::c
    char  c;
    enum { i = 2 };
};

typedef char*  T;
struct Y {
    T  a;     // error: 'T' refers to ::T
              // but when reevaluated is Y::T
    typedef long  T;
    T  b;
};

struct Z {
    int  f(const R);  // error: 'R' is parameter name
                      // but swapping the two declarations
                      // changes it to a type
    typedef int  R;
};
```

9.3 Member Functions

1 A function declared as a member (without the `friend` specifier; §11.4) is called a member function, and is called for an object using the class member syntax (§5.2.4). For example,

```
struct tnode {
    char tword[20];
    int count;
    tnode *left;
    tnode *right;
    void set(char*, tnode* l, tnode* r);
};
```

Here set is a member function and can be called like this:

```
void f(tnode n1, tnode n2)
{
    n1.set("abc",&n2,0);
    n2.set("def",0,0);
}
```

2 The definition of a member function is considered to be within the scope of its class. This means that (provided it is nonstatic §9.4) it can use names of members of its class directly. Such names then refer to the members of the object for which the function was called.

3 A static local variable in a member function always refers to the same object. A static member function can use only the names of static members, enumerators, and nested types directly. If the definition of a member function is lexically outside the class definition, the member function name must be qualified by the class name using the :: operator. For example,

```
void tnode::set(char* w, tnode* l, tnode* r)
{
    count = strlen(w+1);
    if (sizeof(tword)<=count)
        error("tnode string too long");
    strcpy(tword,w);
    left = l;
    right = r;
}
```

The notation tnode::set specifies that the function set is a member of and in the scope of class tnode. The member names tword, count, left, and right refer to members of the object for which the function was called. Thus, in the call n1.set("abc",&n2,0), tword refers to n1.tword, and in the call n2.set("def",0,0) it refers to n2.tword. The functions strlen, error, and strcpy must be declared elsewhere.

4 Members may be defined (§3.1) outside their class definition if they have already been declared but not defined in the class definition; they may not be redeclared. See also §3.3. Function members may be mentioned in friend declarations after their class has been defined. Each member function that is called must have exactly one definition in a program. |

5 The effect of calling a nonstatic member function (§9.4) of a class X for something that is not an object of class X is undefined.

9.3.1 The this Pointer

1 In a nonstatic (§9.3) member function, the keyword this is a non-lvalue expression whose value is the address of the object for which the function is called. The type of this in a member function of a class X is X* unless the member function is declared const or volatile; in those cases, the type of this is const X* or volatile X*, respectively. A function declared const and volatile has a this with the type const volatile X*. See also §19.3.3. For example,

```
struct s {
    int a;
    int f() const;
    int g() { return a++; }
    int h() const { return a++; } // error
};

int s::f() const { return a; }
```

The `a++` in the body of `s::h` is an error because it tries to modify (a part of) the object for which `s::h()` is called. This is not allowed in a `const` member function where `this` is a pointer to `const`, that is, `*this` is a `const`.

2 A `const` member function (that is, a member function declared with the `const` qualifier) may be called for `const` and non-`const` objects, whereas a non-`const` member function may be called only for a non-`const` object. For example,

```
void k(s& x, const s& y)
{
    x.f();
    x.g();
    y.f();
    y.g();       // error
}
```

The call `y.g()` is an error because `y` is `const` and `s::g()` is a non-`const` member function that could (and does) modify the object for which it was called.

3 Similarly, only `volatile` member functions (that is, a member function declared with the `volatile` specifier) may be invoked for `volatile` objects. A member function can be both `const` and `volatile`.

4 Constructors (§12.1) and destructors (§12.4) may be invoked for a `const` or `volatile` object. Constructors (§12.1) and destructors (§12.4) cannot be declared `const` or `volatile`.

9.3.2 Inline Member Functions

1 A member function may be defined (§8.3) in the class definition, in which case it is `inline` (§7.1.2). Defining a function within a class definition is equivalent to declaring it `inline` and defining it immediately after the class definition; this rewriting is considered to be done after preprocessing but before syntax analysis and type checking of the function definition. Thus

```
int b;
struct x {
    char* f() { return b; }
    char* b;
};
```

is equivalent to

```
int b;
struct x {
    char* f();
    char* b;
};

inline char* x::f() { return b; } // moved
```

Thus the `b` used in `x::f()` is `X::b` and not the global `b`. See also _class.local.type_. |

2 Member functions can be defined even in local or nested class definitions where this rewriting would be syntactically incorrect. See §9.8 for a discussion of local classes and §9.7 for a discussion of nested classes.

9.4 Static Members

1 A data or function member of a class may be declared `static` in the class definition. There is only one copy of a static data member, shared by all objects of the class and any derived classes in a program. A static member is not part of objects of a class. Static members of a global class have external linkage (§3.3). The declaration of a static data member in its class definition is *not* a definition and may be of an incomplete type. A definition is required elsewhere; see also §19.3.

2 A static member function does not have a `this` pointer so it can access nonstatic members of its class only by using `.` or `->`. A static member function cannot be `virtual`. There cannot be a static and a nonstatic member function with the same name and the same parameter types.

3 Static members of a local class (§9.8) have no linkage and cannot be defined outside the class definition. It follows that a local class cannot have static data members.

4 A static member mem of class `cl` can be referred to as `cl::mem` (§5.1), that is, independently of any object. It can also be referred to using the `.` and `->` member access operators (§5.2.4). When a static member is accessed through a member access operator, the expression on the left side of the `.` or `->` is not evaluated. The static member mem exists even if no objects of class `cl` have been created. For example, in the following, `run_chain`, `idle`, and so on exist even if no `process` objects have been created:

```
class process {
    static int no_of_processes;
    static process* run_chain;
    static process* running;
    static process* idle;
    // ...
public:
    // ...
    int state();
    static void reschedule();
    // ...
};
```

and `reschedule` can be used without reference to a `process` object, as follows:

```
void f()
{
    process::reschedule();
}
```

5 Static members of a global class are initialized exactly like global objects and only in file scope. For example,

```
void process::reschedule() { /* ... */ };
int process::no_of_processes = 1;
process* process::running = get_main();
process* process::run_chain = process::running;
```

Static members obey the usual class member access rules (§11) except that they can be initialized (in file scope). The initializer of a static member of a class has the same access rights as a member function, as in `process::run_chain` above.

6 The type of a static member does not involve its class name; thus the type of `process :: no_of_processes` is `int` and the type of `&process :: reschedule` is `void(*)()`.

9.5 Unions

1 A union may be thought of as a class whose member objects all begin at offset zero and whose size is sufficient to contain any of its member objects. At most one of the member objects can be stored in a union at any time. A union may have member functions (including constructors and destructors), but not virtual (§10.2) functions. A union may not have base classes. A union may not be used as a base class. An object of a class with a constructor or a destructor or a user-defined assignment operator (§13.4.3) cannot be a member of a union. A union can have no `static` data members.

Shouldn't we prohibit references in unions?

2 A union of the form

```
union { member-specification } ;
```

is called an anonymous union; it defines an unnamed object (and not a type). The names of the members of an anonymous union must be distinct from other names in the scope in which the union is declared; they are used directly in that scope without the usual member access syntax (§5.2.4). For example,

```
void f()
{
    union { int a; char* p; };
    a = 1;
    // ...
    p = "Jennifer";
    // ...
}
```

Here a and p are used like ordinary (nonmember) variables, but since they are union members they have the same address.

3 A global anonymous union must be declared `static`. An anonymous union may not have `private` or `protected` members (§11). An anonymous union may not have function members.

4 A union for which objects or pointers are declared is not an anonymous union. For example,

```
union { int aa; char* p; } obj, *ptr = &obj;
aa = 1;        // error
ptr->aa = 1;   // ok
```

The assignment to plain aa is ill formed since the member name is not associated with any particular object.

5 Initialization of unions that do not have constructors is described in §8.4.1.

9.6 Bit-Fields

1 A *member-declarator* of the form

identifier$_{opt}$: *constant-expression*

specifies a bit-field; its length is set off from the bit-field name by a colon. Allocation of bit-fields within a class object is implementation dependent. Fields are packed into some addressable allocation unit. Fields straddle allocation units on some machines and not on others. Alignment of bit-fields is implementation dependent. Fields are assigned right-to-left on some machines, left-to-right on others.

2 An unnamed bit-field is useful for padding to conform to externally-imposed layouts. Unnamed fields are not members and cannot be initialized. As a special case, an unnamed bit-field with a width of zero specifies alignment of the next bit-field at an allocation unit boundary.

3 A bit-field may not be a static member. A bit-field must have integral or enumeration type (§3.6.1). It is implementation dependent whether a plain (neither explicitly signed nor unsigned) `int` field is signed or unsigned. The address-of operator `&` may not be applied to a bit-field, so there are no pointers to bit-fields. Nor are there references to bit-fields.

9.7 Nested Class Declarations

1 A class may be defined within another class. A class defined within another is called a *nested* class. The name of a nested class is local to its enclosing class. The nested class is in the scope of its enclosing class. Except by using explicit pointers, references, and object names, declarations in a nested class can use only type names, static members, and enumerators from the enclosing class.

```
int x;
int y;

class enclose {
public:
    int x;
    static int s;

    class inner {

        void f(int i)
        {
            x = i;   // error: assign to enclose::x
            s = i;   // ok: assign to enclose::s
            ::x = i; // ok: assign to global x
            y = i;      // ok: assign to global y
        }

        void g(enclose* p, int i)
        {
            p->x = i;   // ok: assign to enclose::x
        }

    };
};

inner* p = 0;   // error 'inner' not in scope
```

Member functions of a nested class have no special access to members of an enclosing class; they obey the usual access rules (§11). Member functions of an enclosing class have no special access to members of a nested class; they obey the usual access rules. For example,

```
class E {
    int x;

    class I {
        int y;
        void f(E* p, int i)
        {
            p->x = i;   // error: E::x is private
        }
    };

    int g(I* p)
    {
        return p->y;    // error: I::y is private
    }
};
```

Member functions and static data members of a nested class can be defined in the global scope. For example,

```
class enclose {
    class inner {
        static int x;
        void f(int i);
    };
};
```

```
        typedef enclose::inner ei;
        int ei::x = 1;

        void enclose::inner::f(int i) { /* ... */ }
```

A nested class may be declared in a class and later defined in the same or an enclosing scope. For example:

```
        class E {
            class I1;        // forward declaration of nested class
            class I2;
            class I1 {};    // definition of nested class
        };
        class E::I2 {};     // definition of nested class
```

Like a member function, a friend function defined within a class is in the lexical scope of that class; it obeys the same rules for name binding as the member functions (described above and in §10.4) and like them has no special access rights to members of an enclosing class or local variables of an enclosing function (§11).

9.8 Local Class Declarations

1 A class can be defined within a function definition; such a class is called a *local* class. The name of a local class is local to its enclosing scope. The local class is in the scope of the enclosing scope. Declarations in a local class can use only type names, static variables, `extern` variables and functions, and enumerators from the enclosing scope. For example,

```
        int x;
        void f()
        {
            static int s ;
            int x;
            extern int g();

            struct local {
                int g() { return x; }     // error: 'x' is auto
                int h() { return s; }     // ok
                int k() { return ::x; }   // ok
                int l() { return g(); }   // ok
            };
            // ...
        }

        local* p = 0;   // error: 'local' not in scope
```

2 An enclosing function has no special access to members of the local class; it obeys the usual access rules (§11). Member functions of a local class must be defined within their class definition. A local class may not have static data members.

9.9 Nested Type Names

1 Type names obey exactly the same scope rules as other names. In particular, type names defined within a class definition cannot be used outside their class without qualification. For example,

```
class X {
public:
    typedef int I;
    class Y { /* ... */ };
    I a;
};

I b;      // error
Y c;      // error
X::Y d;  // ok
X::I e;  // ok
```

10

Derived Classes

1 This chapter explains *inheritance*. A class can be *derived* from one or more other classes, which are then called *base* classes of the derived class. The derived class inherits the properties of its base classes, including its data members and member functions. In addition, the derived class can override *virtual* functions of its bases and declare additional data members, functions, and so on. Access to class members is checked for ambiguity.

2 Sharing among the (base) classes that make up a class can be expressed using *virtual base classes*. Classes can be declared *abstract* to ensure that they are used only as base classes.

3 The final section of this chapter (§10.4) is a summary of the C++ scope rules.

10 Derived Classes

1 A list of base classes may be specified in a class declaration using the notation:

base-clause:
    : *base-specifier-list*

base-specifier-list:
    base-specifier
    base-specifier-list , *base-specifier*

base-specifier:
    qualified-class-specifier
    `virtual` *access-specifier*$_{opt}$ *qualified-class-specifier*
    access-specifier `virtual`$_{opt}$ *qualified-class-specifier*

access-specifier:
    `private`
    `protected`
    `public`

The *class-name* in a *base-specifier* must denote a previously declared class (§9), which is called a *direct base class* for the class being declared. A class `B` is a base class of a class `D` if it is a direct base class of `D` or a direct base class of one of `D`'s base classes. A class is an *indirect* base class of another if it is a base class but not a direct base class. A class is said to be (directly or indirectly) *derived* from its (direct or indirect) base classes. For the meaning of *access-specifier* see §11. Unless redefined in the derived class, members of a base class can be referred to as if they were members of the derived class. The base class members are said to be *inherited* by the derived class. The scope resolution operator `::` (§5.1) may be used to refer to a base member explicitly. This allows access to a name that has been redefined in the derived class. A derived class can itself serve as a base class subject to access control; see §11.2. A pointer to a derived class may be implicitly converted to a pointer to an accessible unambiguous base class (§4.6). A reference to a derived class may be implicitly converted to a reference to an accessible unambiguous base class (§4.7).

2 For example,

```
class Base {
public:
    int a, b, c;
};

class Derived : public Base {
public:
    int b;
};

class Derived2 : public Derived {
public:
    int c;
};
```

3 Here, an object of class `Derived2` will have a sub-object of class `Derived` which in turn will have a sub-object of class `Base`. A derived class and its base classes can be represented by a directed acyclic graph (DAG) where an arrow means "directly derived from." A DAG of classes is often referred to as a "class lattice." For example,

$$\begin{tikzcd}
\texttt{Base} \\
\texttt{Derived} \arrow[u] \\
\texttt{Derived2} \arrow[u]
\end{tikzcd}$$

Note that the arrows need not have a physical representation in memory and the order in which the sub-objects appear in memory is unspecified.

4 Name lookup proceeds from the original class (the named class in the case of a *qualified-id*) along the edges of the lattice until the name is found. If a name is found in more than one class in the lattice, the access is ambiguous (see §10.1.1) unless one occurrence of the name hides[18] all the others. A name `B::f` *hides* a name `A::f` if its class `B` has `A` as a base and the instance of `B` containing `B::f` has the instance of `A` containing `A::f` as a sub-object. The second part of this definition is trivially satisfied when multiple inheritance is not used. For example,

```
void f()
{
    Derived2 x;
    x.a = 1;              // Base::a
    x.b = 2;              // Derived::b
    x.c = 3;              // Derived2::c
    x.Base::b = 4;        // Base::b
    x.Derived::c = 5;     // Base::c
    Base* bp = &x;        // standard conversion:
                          // Derived2* to Base*
}
```

assigns to the five members of `x` and makes `bp` point to `x`.

5 Note that in the *class-name* `::` *id-expression* notation, *id-expression* need not be a member of *class-name*; the notation simply specifies a class in which to start looking for *id-expression*.

6 Initialization of objects representing base classes can be specified in constructors; see §12.6.2.

[18] This criterion is called "dominance" in the ARM.

10.1 Multiple Base Classes

1 A class may be derived from any number of base classes. For example,

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class D : public A, public B, public C { /* ... */ };
```

The use of more than one direct base class is often called multiple inheritance.

2 The order of derivation is not significant except possibly for default initialization by constructor (§12.1), for cleanup (§12.4), and for storage layout (§5.4, §9.2, §11.1).

3 A class may not be specified as a direct base class of a derived class more than once but it may be an indirect base class more than once.

```
class B { /* ... */ };
class D : public B, public B { /* ... */ };  // illegal

class L { /* ... */ };
class A : public L { /* ... */ };
class B : public L { /* ... */ };
class C : public A, public B { /* ... */ };   // legal
```

Here, an object of class C will have two sub-objects of class L as shown below.

$$\begin{tikzcd}
L & & L \\
A \arrow[u] & & B \arrow[u] \\
& C \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

4 The keyword virtual may be added to a base class specifier. A single sub-object of the virtual base class is shared by every base class that specified the base class to be virtual. For example,

```
class V { /* ... */ };
class A : virtual public V { /* ... */ };
class B : virtual public V { /* ... */ };
class C : public A, public B { /* ... */ };
```

Here class C has only one sub-object of class V, as shown below.

$$\begin{tikzcd}
& V & \\
A \arrow[ur] & & B \arrow[ul] \\
& C \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

5 A class may have both virtual and nonvirtual base classes of a given type.

```
class B { /* ... */ };
class X : virtual public B { /* ... */ };
class Y : virtual public B { /* ... */ };
class Z : public B { /* ... */ };
class AA : public X, public Y, public Z { /* ... */ };
```

Here class AA has two sub-objects of class B: Z's B and the virtual B shared by X and Y, as shown below.

$$\begin{tikzcd}
& B & & B \\
X \arrow[ur] & & Y \arrow[ul] & Z \arrow[u] \\
& AA \arrow[ul] \arrow[ur] \arrow[urr] & &
\end{tikzcd}$$

10.1.1 Ambiguities

1 Access to base class members must be unambiguous. Access to a base class member is ambiguous if the *id-expression* or *qualified-id* used does not refer to a unique function, object, type, or enumerator. The check for ambiguity takes place before access control (§11). For example,

```
class A {
public:
    int a;
    int (*b)();
    int f();
    int f(int);
    int g();
};

class B {
    int a;
    int b();
public:
    int f();
    int g;
    int h();
    int h(int);
};

class C : public A, public B {};

void g(C* pc)
{
    pc->a = 1;  // error: ambiguous: A::a or B::a
    pc->b();    // error: ambiguous: A::b or B::b
    pc->f();    // error: ambiguous: A::f or B::f
    pc->f(1);   // error: ambiguous: A::f or B::f
    pc->g();    // error: ambiguous: A::g or B::g
    pc->g = 1;  // error: ambiguous: A::g or B::g
    pc->h();    // ok
    pc->h(1);   // ok
}
```

If the name of an overloaded function is unambiguously found overloading resolution also takes place before access control. Ambiguities can be resolved by qualifying a name with its class name. For example,

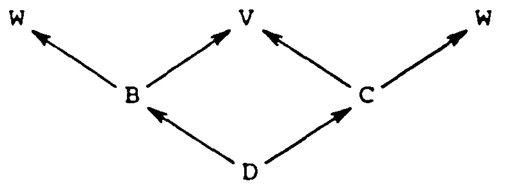
```
class A {
public:
    int f();
};

class B {
public:
    int f();
};

class C : public A, public B {
    int f() { return A::f() + B::f(); }
};
```

A single function, object, type, or enumerator may be reached through more than one path through the directed acyclic graph of base classes. This is not an ambiguity. For example,

```
class V { public: int v; };
class A {
public:
    int a;
    static int   s;
    enum { e };
};
class B : public A, public virtual V {};
class C : public A, public virtual V {};

class D : public B, public C { };

void f(D* pd)
{
    pd->v++;          // ok: only one 'v' (virtual)
    pd->s++;          // ok: only one 's' (static)
    int i = pd->e;    // ok: only one 'e' (enumerator)
    pd->a++;          // error, ambiguous: two 'a's in 'D'
}
```

When virtual base classes are used, a hidden function, object, or enumerator may be reached along a path through the inheritance DAG that does not pass through the hiding function, object, or enumerator. This is not an ambiguity. The identical use with nonvirtual base classes is an ambiguity; in that case there is no unique instance of the name that hides all the others. For example,

```
class V { public: int f();  int x; };
class W { public: int g();  int y; };
class B : public virtual V, public W
{
public:
    int f();  int x;
    int g();  int y;
};
class C : public virtual V, public W { };

class D : public B, public C { void g(); };
```



The names defined in V and the left hand instance of W are hidden by those in B, but the names defined in the right hand instance of W are not hidden at all.

```
void D::g()
{
    x++;          // ok: B::x hides V::x
    f();          // ok: B::f() hides V::f()
    y++;          // error: B::y and C's W::y
    g();          // error: B::g() and C's W::g()
}
```

An explicit or implicit conversion from a pointer or reference to a derived class to a pointer or reference to one of its base classes must unambiguously refer to a unique object representing the base class. For example,

```
class V { };
class A { };
class B : public A, public virtual V { };
class C : public A, public virtual V { };
class D : public B, public C { };

void g()
{
    D d;
    B* pb = &d;
    A* pa = &d;  // error, ambiguous: C's A or B's A ?
    V* pv = &d;  // fine: only one V sub-object
}
```

10.2 Virtual Functions

1 Virtual functions support dynamic binding and object-oriented programming. A class that declares or inherits a virtual function is called a *polymorphic class.*

2 If a virtual member function vf is declared in a class Base and in a class Derived, derived directly or indirectly from Base, a member function vf with the same name and same parameter list as Base::vf is declared, then Derived::vf is also virtual (whether or not it is so declared) and it *overrides*[19] Base::vf. For convenience we say that any virtual function overrides itself. Then in any well-formed class, for each virtual function declared in that class or any of its direct or indirect base classes there is a unique *final overrider* that overrides that function and every other overrider of that function.

3 It is a diagnosable error for the return type of an overriding function to differ from the return type of the overridden function unless the return type of the overridden function is pointer or reference (possibly cv-qualified) to a class B, and the return type of the overriding function is pointer or reference (respectively) to a class D such that B is an unambiguous direct or indirect base class of D, accessible in the class of the overriding function, and the cv-qualification in the return type of the overriding functin is less than or equal to the cv-qualification in the return type of the overridden function. In that case when the overriding function is called as the final overrider of the overridden function, its result is converted to the type returned by the (statically chosen) overridden function. See §5.2.2. For example,

```
class B {};
class D : private B { friend class Derived; };
struct Base {
    virtual void vf1();
    virtual void vf2();
    virtual void vf3();
    virtual B*   vf4();
    void f();
};

struct No_good : public Base {
    D*  vf4();          // error: B (base class of D) inaccessible
};
```

[19] A function with the same name but a different parameter list (see §13) as a virtual function is not necessarily virtual and does not override. The use of the virtual specifier in the declaration of an overriding function is legal but redundant (has empty semantics). Access control (§11) is not considered in determining overriding.

```
struct Derived : public Base {
    void vf1();         // virtual and overrides Base::vf1()
    void vf2(int);      // not virtual, hides Base::vf2()
    char vf3();         // error: invalid difference in return type only
    D*  vf4();          // okay: returns pointer to derived class
    void f();
};

void g()
{
    Derived d;
    Base* bp = &d;      // standard conversion:
                        // Derived* to Base*
    bp->vf1();          // calls Derived::vf1()
    bp->vf2();          // calls Base::vf2()
    bp->f();            // calls Base::f() (not virtual)
    B*  p = bp->vf4();  // calls Derived::pf() and converts the
                        //  result to B*
    Derived*  dp = &d;
    D*  q = dp->vf4();  // calls Derived::pf() and does not
                        //  convert the result to B*
    dp->vf2();          // ill formed: argument mismatch
}
```

4 That is, the interpretation of the call of a virtual function depends on the type of the object for which it is called (the dynamic type), whereas the interpretation of a call of a nonvirtual member function depends only on the type of the pointer or reference denoting that object (the static type). See §5.2.2.

5 The `virtual` specifier implies membership, so a virtual function cannot be a global (nonmember) (§7.1.2) function. Nor can a virtual function be a static member, since a virtual function call relies on a specific object for determining which function to invoke. A virtual function can be declared a `friend` in another class. A virtual function declared in a class must be defined or declared pure (§10.3) in that class.

6 Following are some examples of virtual functions used with multiple base classes:

```
struct A {
    virtual void f();
};

struct B1 : A {   // note non-virtual derivation
    void f();
};

struct B2 : A {
    void f();
};

struct D : B1, B2 {  // D has two separate A sub-objects
};

void foo()
{
    D   d;
    // A*  ap = &d; // would be ill formed: ambiguous
    B1*  b1p = &d;
    A*   ap = b1p;
    ap->f();  // calls D::B1::f
    dp->f();  // ill formed: ambiguous
}
```

In class `D` above there are two occurrences of class `A` and hence two occurrences of the virtual member function `A::f`. The final overrider of `B1::A::f` is `B1::f` and the final overrider of `B2::A::f` is `B2::f`.

7 The following example shows a function that does not have a unique final overrider:

```
struct A {
    virtual void f();
};

struct VB1 : virtual A {    // note virtual derivation
    void f();
};

struct VB2 : virtual A {
    void f();
};

struct Error : VB1, VB2 {  // ill-formed
};

struct Okay : VB1, VB2 {
    void f();
};
```

Both VB1::f and VB2::f override A::f but there is no overrider of both of them in class Error. This error requires a diagnosis. Class Okay is well formed, however, because Okay::f is a final overrider.

8 The following example uses the well-formed classes from above.

```
struct VB1a : virtual A {  // does not declare f
};

struct Da : VB1a, VB2 {
};

void foe()
{
    VB1a*  vb1ap = new Da;
    vb1ap->f();  // calls VB2:f
}
```

9 Explicit qualification with the scope operator (§5.1) suppresses the virtual call mechanism. For example,

```
class B { public: virtual void f(); };
class D : public B { public: void f(); };

void D::f() { /* ... */ B::f(); }
```

Here, the function call in D::f really does call B::f and not D::f.

10.3 Abstract Classes

1 The abstract class mechanism supports the notion of a general concept, such as a shape, of which only more concrete variants, such as circle and square, can actually be used. An abstract class can also be used to define an interface for which derived classes provide a variety of implementations.

2 An *abstract class* is a class that can be used only as a base class of some other class; no objects of an abstract class may be created except as sub-objects of a class derived from it. A class is abstract if it has at least one *pure virtual function* (which may be inherited: see below). A virtual function is specified *pure* by using a *pure-specifier* (§9.2) in the function declaration in the class declaration. A pure virtual function need be defined only if explicitly called with the *qualified-id* syntax (§5.1). For example,

```
class point { /* ... */ };
class shape {           // abstract class
    point center;
    // ...
public:
    point where() { return center; }
    void move(point p) { center=p; draw(); }
    virtual void rotate(int) = 0;  // pure virtual
    virtual void draw() = 0;       // pure virtual
    // ...
};
```

An abstract class may not be used as an parameter type, as a function return type, or as the type of an explicit conversion. Pointers and references to an abstract class may be declared. For example, |

```
shape x;            // error: object of abstract class
shape* p;           // ok
shape f();          // error
void g(shape);      // error
shape& h(shape&);   // ok
```

3 Pure virtual functions are inherited as pure virtual functions. For example,

```
class ab_circle : public shape {
    int radius;
public:
    void rotate(int) {}
    // ab_circle::draw() is a pure virtual
};
```

Since shape::draw() is a pure virtual function ab_circle::draw() is a pure virtual by default. The alternative declaration,

```
class circle : public shape {
    int radius;
public:
    void rotate(int) {}
    void draw(); // must be defined somewhere
};
```

would make class circle nonabstract and a definition of circle::draw() must be provided.

4 An abstract class may be derived from a class that is not abstract, and a pure virtual function may override a virtual function which is not pure.

5 Member functions can be called from a constructor of an abstract class; the effect of calling a pure virtual function directly or indirectly for the object being created from such a constructor is undefined.

10.4 Summary of Scope Rules

1 The scope rules for C++ programs can now be summarized. These rules apply uniformly for all names (including *typedef-names* (§7.1.3) and *class-names* (§9.1)) wherever the grammar allows such names in the context discussed by a particular rule. This section discusses lexical scope only; see §3.3 for an explanation of linkage issues. The notion of point of declaration is discussed in (§3.2).

2 Any use of a name must be unambiguous (up to overloading) in its scope (§10.1.1). Only if the name is found to be unambiguous in its scope are access rules considered (§11). Only if no access control errors are found is the type of the object, function, or enumerator named considered.

3 A name used outside any function and class or prefixed by the unary scope operator :: (and *not* qualified by the binary :: operator or the -> or . operators) must be the name of a global object, function, or enumerator.

4 A name specified after X::, after obj., where obj is an X or a reference to X, or after ptr->, where ptr is a pointer to X must be the name of a member of class X or be a member of a base class of X. In addition, ptr in ptr-> may be an object of a class Y that has operator->() declared so

`ptr->operator->()` eventually resolves to a pointer to `X` (§13.4.6).

5 A name that is not qualified in any of the ways described above and that is used in a function that is not a class member must be declared before its use in the block in which it occurs or in an enclosing block or globally. The declaration of a local name hides previous declarations of the same name in enclosing blocks and at file scope. In particular, no overloading occurs of names in different scopes (§13.4).

6 A name that is not qualified in any of the ways described above and that is used in a function that is a nonstatic member of class `X` must be declared in the block in which it occurs or in an enclosing block, be a member of class `X` or a base class of class `X`, or be a global name. The declaration of a local name hides declarations of the same name in enclosing blocks, members of the function's class, and global names. The declaration of a member name hides declarations of the same name in base classes and global names.

7 A name that is not qualified in one of the ways described above and is used in a static member function of a class `X` must be declared in the block in which it occurs, in an enclosing block, be a static member of class `X`, or a base class of class `X`, or be a global name.

8 A function parameter name in a function definition (§8.3) is in the scope of the outermost block of the function (in particular, it is a local name). A function parameter name in a function declaration (§8.2.5) that is not a function definition is in a local scope that disappears immediately after the function declaration. A default parameter is in the scope determined by the point of declaration (§3.2) of its parameter, but may not access local variables or nonstatic class members; it is evaluated at each point of call (§8.2.6).

9 A *ctor-initializer* (§12.6.2) is evaluated in the scope of the outermost block of the constructor it is specified for. In particular, it can refer to the constructor's parameter names.

11

Member Access Control

1 This chapter explains mechanisms for control of access to class members. Access control is based on the use of the keywords public, private, and protected to control access to individual members of a class and on the use of private, protected, and public specifiers to control access to base class members in a derived class object. The friend mechanism provides a way of granting individual functions and classes access to members of a class.

2 Access control applies uniformly to function members, data members, member constants, and nested types.

11 Member Access Control

1 A member of a class can be

private; that is, its name can be used only by member functions and friends of the class in which it is declared.

protected; that is, its name can be used only by member functions and friends of the class in which it is declared and by member functions and friends of classes derived from this class (see §11.5).

public; that is, its name can be used by any function.

2 Members of a class declared with the keyword class are private by default. Members of a class declared with the keywords struct or union are public by default. For example,

```
class X {
    int a;  // X::a is private by default
};

struct S {
    int a;  // S::a is public by default
};
```

11.1 Access Specifiers

1 Member declarations may be labeled by an *access-specifier* (§10):

access-specifier : *member-specification*$_{opt}$

An *access-specifier* specifies the access rules for members following it until the end of the class or until another *access-specifier* is encountered. For example,

```
class X {
    int a;  // X::a is private by default: 'class' used
public:
    int b;  // X::b is public
    int c;  // X::c is public
};
```

Any number of access specifiers is allowed and no particular order is required. For example,

```
struct S {
    int a;  // S::a is public by default: 'struct' used
protected:
    int b;  // S::b is protected
private:
    int c;  // S::c is private
public:
    int d;  // S::d is public
};
```

2 The order of allocation of data members with separate *access-specifier* labels is implementation dependent (§9.2).

11.2 Access Specifiers for Base Classes

1 If a class is declared to be a base class (§10) for another class using the `public` access specifier, the `public` members of the base class are accessible as `public` members of the derived class and `protected` members of the base class are accessible as `protected` members of the derived class (but see §13.1). If a class is declared to be a base class for another class using the `protected` access specifier, the `public` and `protected` members of the base class are accessible as `protected` members of the derived class. If a class is declared to be a base class for another class using the `private` access specifier, the `public` and `protected` members of the base class are accessible as `private` members of the derived class. Private members of a base class remain inaccessible even to derived classes unless `friend` declarations within the base class declaration are used to grant access explicitly.

2 In the absence of an *access-specifier* for a base class, `public` is assumed when the derived class is declared `struct` and `private` is assumed when the class is declared `class`. For example,

```
class B { /* ... */ };
class D1 : private B { /* ... */ };
class D2 : public B { /* ... */ };
class D3 : B { /* ... */ };     // 'B' private by default
struct D4 : public B { /* ... */ };
struct D5 : private B { /* ... */ };
struct D6 : B { /* ... */ };    // 'B' public by default
class D7 : protected B { /* ... */ };
struct D8 : protected B { /* ... */ };
```

Here `B` is a public base of `D2`, `D4`, and `D6`, a private base of `D1`, `D3`, and `D5`, and a protected base of `D7` and `D8`.

3 Because of the rules on pointer conversion (§4.6), a static member of a private base class may be inaccessible as an inherited name, but accessible directly. For example,

```
class B {
public:
        int mi;             // nonstatic member
        static int si;      // static member
};
class D : private B {
};
class DD : public D {
        void f();
};

void DD::f() {
        mi = 3;            // error: mi is private in D
        si = 3;            // error: si is private in D
        B  b;
        b.mi = 3;          // okay (b.mi is different from this->mi)
        b.si = 3;          // okay (b.si is the same as this->si)
        B::si = 3;         // okay
        B* bp1 = this; // error: B is a private base class
        B* bp2 = (B*)this;  // okay with cast
        bp2->mi = 3;    // okay and bp2->mi is the same as this->mi
}
```

4 Members and friends of a class X can implicitly convert an X* to a pointer to a private or protected immediate base class of X.

11.3 Access Declarations

1 The access of public or protected member of a private or protected base class can be restored to the same level in the derived class by mentioning its *qualified-id* in the public (for public members of the base class) or protected (for protected members of the base class) part of a derived class declaration. Such mention is called an *access declaration*.

2 For example,

```
class A {
public:
    int z;
    int z1;
};

class B : public A {
    int a;
public:
    int b, c;
    int bf();
protected:
    int x;
    int y;
};
```

```
class D : private B {
    int d;
public:
    B::c;  // adjust access to 'B::c'
    B::z;  // adjust access to 'A::z'
    A::z1; // adjust access to 'A::z1'
    int e;
    int df();
protected:
    B::x;  // adjust access to 'B::x'
    int g;
};

class X : public D {
    int xf();
};

int ef(D&);
int ff(X&);
```

The external function ef can use only the names c, z, z1, e, and df. Being a member of D, the function df can use the names b, c, z, z1, bf, x, y, d, e, df, and g, but not a. Being a member of B, the function bf can use the members a, b, c, z, z1, bf, x, and y. The function xf can use the public and protected names from D, that is, c, z, z1, e, and df (public), and x, and g (protected). Thus the external function ff has access only to c, z, z1, e, and df. If D were a protected or private base class of X, xf would have the same privileges as before, but ff would have no access at all.

3 An access declaration may not be used to restrict access to a member that is accessible in the base class, nor may it be used to enable access to a member that is not accessible in the base class. For example,

```
class A {
public:
    int z;
};

class B : private A {
public:
    int a;
    int x;
private:
    int b;
protected:
    int c;
};

class D : private B {
public:
    B::a;  // make 'a' a public member of D
    B::b;  // error: attempt to grant access
           // can't make 'b' a public member of D
    A::z;  // error: attempt to grant access
protected:
    B::c;  // make 'c' a protected member of D
    B::x;  // error: attempt to reduce access
           // can't make 'x' a protected member of D
};

class E : protected B {
public:
    B::a;  // make 'a' a public member of E
};
```

The names c and x are protected members of E by virtue of its protected derivation from B. An access declaration for the name of an overloaded function adjusts the access to all functions of that name in the base class. For example,

```
class X {
public:
    f();
    f(int);
};

class Y : private X {
public:
    X::f;  // makes X::f() and X::f(int) public in Y
};
```

4 The access to a base class member cannot be adjusted in a derived class that also defines a member of that name. For example,

```
class X {
public:
    void f();
};

class Y : private X {
public:
    void f(int);
    X::f;  // error: two declarations of f
};
```

11.4 Friends

1 A friend of a class is a function that is not a member of the class but is permitted to use the private and protected member names from the class. The name of a friend is not in the scope of the class, and the friend is not called with the member access operators (§5.2.4) unless it is a member of another class. The following example illustrates the differences between members and friends:

```
class X {
    int a;
    friend void friend_set(X*, int);
public:
    void member_set(int);
};

void friend_set(X* p, int i) { p->a = i; }
void X::member_set(int i) { a = i; }

void f()
{
    X obj;
    friend_set(&obj,10);
    obj.member_set(10);
}
```

2 When a `friend` declaration refers to an overloaded name or operator, only the function specified by the parameter types becomes a friend. A member function of a class X can be a friend of a class Y. For example,

```
class Y {
    friend char* X::foo(int);
    // ...
};
```

All the functions of a class X can be made friends of a class Y by a single declaration using an *elaborated-*

type-specifier[20] (§9.1):

```
        class Y {
            friend class X;
            // ...
        };
```

Declaring a class to be a friend also implies that private and protected names from the class granting friendship can be used in the class receiving it. For example,

```
        class X {
            enum { a=100 };
            friend class Y;
        };

        class Y {
            int v[X::a];  // ok, Y is a friend of X
        };

        class Z {
            int v[X::a];  // error: X::a is private
        };
```

3 If a class or function mentioned as a friend has not been declared, its name is entered in the smallest non-class scope that encloses the friend declaration.

4 A function first declared in a friend declaration is equivalent to an `extern` declaration (§3.3, §7.1.1).

5 A global (but not a member) `friend` function may be defined in a class definition other than a local class definition (§9.8). The function is then `inline` and the rewriting rule specified for member functions (§9.3.2) is applied. A `friend` function defined in a class is in the (lexical) scope of the class in which it is defined. A friend function defined outside the class is not.

6 Friend declarations are not affected by *access-specifiers* (§9.2).

7 Friendship is neither inherited nor transitive. For example,

```
        class A {
            friend class B;
            int a;
        };

        class B {
            friend class C;
        };

        class C  {
            void f(A* p)
            {
                p->a++;  // error: C is not a friend of A
                         // despite being a friend of a friend
            }
        };

        class D : public B  {
            void f(A* p)
            {
                p->a++;  // error: D is not a friend of A
                         // despite being derived from a friend
            }
        };
```

[20] Note that the *class-key* of the *elaborated-type-specifier* is required.

11.5 Protected Member Access

1 A friend or a member function of a derived class can access a protected static member of a base class. A friend or a member function of a derived class can access a protected nonstatic member of one of its base classes only through a pointer to, reference to, or object of the derived class itself (or any class derived from that class). When a protected member of a base class appears in a *qualified-id* in a friend or a member function of a derived class the *nested-class-specifier* must name the derived class. For example,

```
class B {
protected:
    int i;
};

class D1 : public B {
};

class D2 : public B {
    friend void fr(B*,D1*,D2*);
    void mem(B*,D1*);
};

void fr(B* pb, D1* p1, D2* p2)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    p2->i = 3;  // ok (access through a D2)
    int B::*  pmi_B = &B::i;    // illegal
    int D2::* pmi_D2 = &D2::i;  // ok
}

void D2::mem(B* pb, D1* p1)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    i = 3;      // ok (access through 'this')
}

void g(B* pb, D1* p1, D2* p2)
{
    pb->i = 1;  // illegal
    p1->i = 2;  // illegal
    p2->i = 3;  // illegal
}
```

11.6 Access to Virtual Functions

1 The access rules (§11) for a virtual function are determined by its declaration and are not affected by the rules for a function that later overrides it. For example,

```
class B {
public:
    virtual f();
};

class D : public B {
private:
    f();
};
```

```
void f()
{
    D d;
    B* pb = &d;
    D* pd = &d;

    pb->f();  // ok: B::f() is public,
              // D::f() is invoked
    pd->f();  // error: D::f() is private
}
```

Access is checked at the call point using the type of the expression used to denote the object for which the member function is called (B* in the example above). The access of the member function in the class in which it was defined (D in the example above) is in general not known.

11.7 Multiple Access

1 If a name can be reached by several paths through a multiple inheritance graph, the access is that of the path that gives most access. For example,

```
class W { public: void f(); };
class A : private virtual W { };
class B : public virtual W { };
class C : public A, public B {
    void f() { W::f(); }  // ok
};
```

Since W::f() is available to C::f() along the public path through B, access is allowed.

APPPENDIX B

CONTENS OF FLOPPY

The floppy attached to this thesis consists of the following files as shown in next page. It contains a total of 219 *.cpp files, sr*.kr files and tn*.bat files each. For each *.cpp file there is an associated sr*.kr file and tn*.bat file.

These are all used during batch processing. All these files are in sub-directory named *KRS*. For using this verification suite all the files must loaded in toto in a separate sub-directory in the hard disk for simplicity.

Then this suite can be used for testing all DOS based C++ compilers by using command *auco* <[input]> where [input] is the command-line-compilation-command of the compiler under test for more detail refer to section 2.5 of this thesis.

However for using the test programs to test a C++ compiler which is not $\overset{\text{DoS}}{\wedge}$based, appropriate driver programs will have to be written after copying only all the *.cpp files.

```
 Volume in drive A has no label
 Directory of A:\KRS

[.]             [..]            ARUN.BAT        ARUN1           ARUN3
ARUN4           ARUNK           AUCO.BAT        AUCO10.BAT      AUCO11.BAT
AUCO9.BAT       CLEAR.BAT       N090001A.CPP    N090001B.CPP    N090101A.CPP
N090101B.CPP    N090101C.CPP    N090102A.CPP    N090102B.CPP    N090103A.CPP
N090103B.CPP    N090201B.CPP    N090201C.CPP    N090201D.CPP    N090201E.CPP
N090201F.CPP    N090203A.CPP    N090203B.CPP    N090203C.CPP    N090205A.CPP
N090221A.CPP    N090221B.CPP    N090221C.CPP    N090221D.CPP    N090301.CPP
N090402A.CPP    N090402B.CPP    N090402C.CPP    N090403.CPP     N090501A.CPP
N090501B.CPP    N090501C.CPP    N090501E.CPP    N090502.CPP     N090503A.CPP
N090503B.CPP    N090503C.CPP    N090503D.CPP    N090504A.CPP    N090504B.CPP
N090701A.CPP    N090701B.CPP    N090701C.CPP    N090701D.CPP    N090802A.CPP
N090802B.CPP    N090901A.CPP    N090901B.CPP    N093101A.CPP    N093102.CPP
N093103A.CPP    N093202A.CPP    N093202B.CPP    N100103.CPP     N100104A.CPP
N100104B.CPP    N100111A.CPP    N100111B.CPP    N100111C.CPP    N100111D.CPP
N100111E.CPP    N100111F.CPP    N100111G.CPP    N100111H.CPP    N100111I.CPP
N100111J.CPP    N100202B.CPP    N100203A.CPP    N100203B.CPP    N100205A.CPP
N100302A.CPP    N100302B.CPP    N100302C.CPP    N100303.CPP     N100304A.CPP
N100304B.CPP    N110001A.CPP    N110001B.CPP    N110001C.CPP    N110001D.CPP
N110002.CPP     N110002B.CPP    N110101A.CPP    N110101B.CPP    N110101C.CPP
N110101D.CPP    N110201A.CPP    N110201B.CPP    N110201C.CPP    N110201D.CPP
N110201E.CPP    N110202A.CPP    N110203.CPP     N110301A.CPP    N110301B.CPP
N110303A.CPP    N110303B.CPP    N110303C.CPP    N110303D.CPP    N110304A.CPP
N110402A.CPP    N110402B.CPP    N110402C.CPP    N110405.CPP     N110407A.CPP
N110407B.CPP    N110501A.CPP    N110501B.CPP    N110501C.CPP    N110501D.CPP
N110501E.CPP    N110501F.CPP    N110501G.CPP    N110501H.CPP    N110601.CPP
N921102A.CPP    N921102B.CPP    N921102C.CPP    N921102D.CPP    N921102E.CPP
N921102F.CPP    P090002.CPP     P090003.CPP     P090004A.CPP    P090005A.CPP
P090005C.CPP    P090102A.CPP    P090102B.CPP    P090103A.CPP    P090103B.CPP
P090201A.CPP    P090202A.CPP    P090205.CPP     P090207A.CPP    P090209A.CPP
P090210A.CPP    P090212A.CPP    P090213A.CPP    P090301.CPP     P090302.CPP
P090303.CPP     P090401.CPP     P090402.CPP     P090404.CPP     P090405.CPP
P090501A.CPP    P090501B.CPP    P090502.CPP     P090503.CPP     P090504.CPP
P090701A.CPP    P090701B.CPP    P090701C.CPP    P090701D.CPP    P090801.CP
P090801A.CPP    P090801B.CPP    P090801C.CPP    P090801D.CPP    P090801E.CPP
P090802.CPP     P090901A.CPP    P093101A.CPP    P093102.CPP     P093103A.CPP
P093103B.CPP    P093201.CPP     P100101.CPP     P100103.CPP     P100104.CPP
P100105.CPP     P100111A.CPP    P100111B.CPP    P100111C.CPP    P100111F.CPP
P100202A.CPP    P100202B.CPP    P100202C.CPP    P100203A.CPP    P100203B.CPP
P100203C.CPP    P100203D.CPP    P100203E.CPP    P100204.CPP     P100209.CPP
P100302A.CPP    P100302B.CPP    P100302C.CPP    P100302D.CPP    P100304A.CPP
P110001A.CPP    P110001B.CPP    P110001C.CPP    P110001D.CPP    P110001E.CPP
P110002A.CPP    P110002B.CPP    P110101A.CPP    P110101B.CPP    P110201A.CPP
P110201B.CPP    P110201C.CPP    P110201D.CPP    P110202A.CPP    P110203.CPP
P110204A.CPP    P110301A.CPP    P110301B.CPP    P110303A.CPP    P110401A.CPP
P110402A.CPP    P110402B.CPP    P110406A.CPP    P110501A.CPP    P110501B.CPP
P110601.CPP     P110701.CPP     P921101A.CPP    P921101C.CPP    P921101D.CPP
P921104.CPP     P921105.CPP     SR00010.KR      SR00020.KR      SR00030.KR
```

SR00040.KR	SR00050.KR	SR00060.KR	SR00080.KR	SR00090.KR
SR00100.KR	SR00110.KR	SR00120.KR	SR00130.KR	SR00140.KR
SR00150.KR	SR00160.KR	SR00170.KR	SR00180.KR	SR00190.KR
SR00200.KR	SR00210.KR	SR00220.KR	SR00230.KR	SR00240.KR
SR00260.KR	SR00270.KR	SR00280.KR	SR00290.KR	SR00300.KR
SR00310.KR	SR00320.KR	SR00330.KR	SR00340.KR	SR00345.KR
SR00360.KR	SR00363.KR	SR00400.KR	SR00410.KR	SR00420.KR
SR00421.KR	SR00430.KR	SR00432.KR	SR00433.KR	SR00440.KR
SR00450.KR	SR00460.KR	SR00461.KR	SR00462.KR	SR00463.KR
SR00470.KR	SR00480.KR	SR00490.KR	SR00493.KR	SR00496.KR
SR00499.KR	SR00506.KR	SR00509.KR	SR00512.KR	SR00515.KR
SR00518.KR	SR00521.KR	SR00524.KR	SR00527.KR	SR00528.KR
SR00529.KR	SR00530.KR	SR00540.KR	SR00550.KR	SR00560.KR
SR00570.KR	SR00580.KR	SR00590.KR	SR00600.KR	SR00610.KR
SR00620.KR	SR00630.KR	SR00640.KR	SR00650.KR	SR00660.KR
SR00670.KR	SR00680.KR	SR00690.KR	SR00700.KR	SR00710.KR
SR00720.KR	SR00730.KR	SR00740.KR	SR00741.KR	SR00750.KR
SR00760.KR	SR00770.KR	SR00780.KR	SR00781.KR	SR00790.KR
SR00800.KR	SR00810.KR	SR00820.KR	SR00830.KR	SR00840.KR
SR00850.KR	SR00860.KR	SR00870.KR	SR00880.KR	SR00890.KR
SR00900.KR	SR00910.KR	SR00911.KR	SR00920.KR	SR01010.KR
SR01020.KR	SR01030.KR	SR01040.KR	SR01043.KR	SR01050.KR
SR01060.KR	SR01065.KR	SR01070.KR	SR01075.KR	SR01080.KR
SR01090.KR	SR01100.KR	SR01110.KR	SR01115.KR	SR01120.KR
SR01130.KR	SR01140.KR	SR01150.KR	SR01160.KR	SR01190.KR
SR01210.KR	SR01220.KR	SR01230.KR	SR01240.KR	SR01250.KR
SR01260.KR	SR01270.KR	SR01280.KR	SR01281.KR	SR01282.KR
SR01283.KR	SR01290.KR	SR01300.KR	SR01310.KR	SR01320.KR
SR01330.KR	SR01340.KR	SR01350.KR	SR01360.KR	SR01370.KR
SR01380.KR	SR01390.KR	SR01400.KR	SR01405.KR	SR01410.KR
SR01420.KR	SR01430.KR	SR01440.KR	SR01450.KR	SR01460.KR
SR01470.KR	SR01480.KR	SR01490.KR	SR01500.KR	SR01510.KR
SR01515.KR	SR01520.KR	SR01530.KR	SR01540.KR	SR01550.KR
SR01555.KR	SR01560.KR	SR01570.KR	SR01580.KR	SR01590.KR
SR01600.KR	SR01610.KR	SR01620.KR	SR01630.KR	SR01640.KR
SR01650.KR	SR01660.KR	SR01670.KR	SR01680.KR	SR01690.KR
SR01700.KR	SR01710.KR	SR01730.KR	SR01760.KR	SR01770.KR
SR01780.KR	SR01790.KR	SR01795.KR	SR01800.KR	SR01810.KR
SR01820.KR	SR01830.KR	SR01840.KR	SR01850.KR	SR01860.KR
SR01870.KR	SR01880.KR	SR01890.KR	SR01900.KR	SR01910.KR
SR01920.KR	SR01950.KR	SR01960.KR	SR01970.KR	SR01980.KR
SR01990.KR	SR02000.KR	SR02001.KR	SR02002.KR	SR02003.KR
SR02004.KR	SR02005.KR	SR02006.KR	SR02010.KR	SR02020.KR
SR02030.KR	TN00010.BAT	TN00020.BAT	TN00030.BAT	TN00040.BAT
TN00050.BAT	TN00060.BAT	TN00080.BAT	TN00090.BAT	TN00100.BAT
TN00110.BAT	TN00120.BAT	TN00130.BAT	TN00140.BAT	TN00150.BAT
TN00160.BAT	TN00170.BAT	TN00180.BAT	TN00190.BAT	TN00200.BAT
TN00210.BAT	TN00220.BAT	TN00230.BAT	TN00240.BAT	TN00250.BAT
TN00260.BAT	TN00270.BAT	TN00280.BAT	TN00290.BAT	TN00300.BAT
TN00310.BAT	TN00320.BAT	TN00330.BAT	TN00335.BAT	TN00350.BAT

TH-5160

```
TN00353.BAT     TN00390.BAT     TN00400.BAT     TN00410.BAT     TN00411.BAT
TN00420.BAT     TN00422.BAT     TN00423.BAT     TN00430.BAT     TN00440.BAT
TN00450.BAT     TN00451.BAT     TN00452.BAT     TN00453.BAT     TN00460.BAT
TN00470.BAT     TN00480.BAT     TN00490.BAT     TN00500.BAT     TN00510.BAT
TN00520.BAT     TN00530.BAT     TN00540.BAT     TN00550.BAT     TN00560.BAT
TN00570.BAT     TN00580.BAT     TN00590.BAT     TN00591.BAT     TN00592.BAT
TN00600.BAT     TN00610.BAT     TN00620.BAT     TN00630.BAT     TN00640.BAT
TN00650.BAT     TN00660.BAT     TN00670.BAT     TN00680.BAT     TN00690.BAT
TN00700.BAT     TN00710.BAT     TN00720.BAT     TN00730.BAT     TN00740.BAT
TN00750.BAT     TN00760.BAT     TN00770.BAT     TN00780.BAT     TN00790.BAT
TN00800.BAT     TN00810.BAT     TN00820.BAT     TN00830.BAT     TN00840.BAT
TN00850.BAT     TN00860.BAT     TN00870.BAT     TN00880.BAT     TN00890.BAT
TN00900.BAT     TN00910.BAT     TN00920.BAT     TN00930.BAT     TN00940.BAT
TN00950.BAT     TN00960.BAT     TN00970.BAT     TN00980.BAT     TN00990.BAT
TN01000.BAT     TN01010.BAT     TN01020.BAT     TN11030.BAT     TN11040.BAT
TN11050.BAT     TN11060.BAT     TN11070.BAT     TN11073.BAT     TN11080.BAT
TN11085.BAT     TN11090.BAT     TN11095.BAT     TN11100.BAT     TN11110.BAT
TN11120.BAT     TN11130.BAT     TN11140.BAT     TN11150.BAT     TN11160.BAT
TN11185.BAT     TN11190.BAT     TN11200.BAT     TN11210.BAT     TN11215.BAT
TN11220.BAT     TN11240.BAT     TN11250.BAT     TN11260.BAT     TN11270.BAT
TN11280.BAT     TN11281.BAT     TN11282.BAT     TN11283.BAT     TN11290.BAT
TN11300.BAT     TN11310.BAT     TN11320.BAT     TN11330.BAT     TN11340.BAT
TN11350.BAT     TN11360.BAT     TN11370.BAT     TN11380.BAT     TN11390.BAT
TN11400.BAT     TN11410.BAT     TN11415.BAT     TN11420.BAT     TN21430.BAT
TN21440.BAT     TN21450.BAT     TN21460.BAT     TN21470.BAT     TN21480.BAT
TN21490.BAT     TN21500.BAT     TN21510.BAT     TN21520.BAT     TN21525.BAT
TN21530.BAT     TN21540.BAT     TN21550.BAT     TN21560.BAT     TN21565.BAT
TN21570.BAT     TN21580.BAT     TN21590.BAT     TN21600.BAT     TN21610.BAT
TN21620.BAT     TN21630.BAT     TN21640.BAT     TN21650.BAT     TN21660.BAT
TN21670.BAT     TN21680.BAT     TN21690.BAT     TN21700.BAT     TN21710.BAT
TN21720.BAT     TN21740.BAT     TN21770.BAT     TN21780.BAT     TN21790.BAT
TN21800.BAT     TN21805.BAT     TN21810.BAT     TN21820.BAT     TN21823.BAT
TN21826.BAT     TN21830.BAT     TN21840.BAT     TN21850.BAT     TN21860.BAT
TN21870.BAT     TN21880.BAT     TN21890.BAT     TN21900.BAT     TN21910.BAT
TN21940.BAT     TN21950.BAT     TN21960.BAT     TN21970.BAT     TN21980.BAT
TN21990.BAT     TN21991.BAT     TN21992.BAT     TN21993.BAT     TN21994.BAT
TN21995.BAT     TN21996.BAT     TN22000.BAT     TN22010.BAT     TN22020.BAT
        670 file(s)     214433 bytes
                        364544 bytes free
```

BIBLIOGRAPHY

[Boo, 1991] Grady Booch: Object Oriented Design with applications. Benjamin/ Cummings 1991. ISBN 0-8053-0091-0.

[Ell, 1990] Marget A. Ellis and Bjarne Stroustrup : The Annotated C++ reference manual. Addison-Wesley. Reading, Massachusetts. 1990. ISBN 0-2010-51459-1.

[Jal, 1991] Pankaj Jalote : An integrated approach to Software Engineering. Narosa publishing house. 1991. ISBN 81-85198-63-2.

[Kho, 1990] Setrag Khoshafian and Razmik Abnous : Object Orientation: Concepts, Languages, databases, user interfaces. John Wiley 1990. ISBN 0-471-51802-6.

[Koe, 1992] Andrew koeing : What's in a name?. Journal of Object oriented programing. September 1992.

[Len, 1989] Dimtry Lenkov : C++ standardization proposal. ANSI X3J11, Document No 016, 1989.

[Lip, 1991] Stanley B.Lippman : C++ primer. Addison Wesley 1991. ISBN 0-201-54848-8.

[Plu, 1993] Thomas Plum: Paper on Layout Rules. ISO WG21, Document No N0228, 1993.

[Sak, 1992] Dan saks : Scope and Name lookup rules. C++ Report. July-August 1992.

[Str, 1991] Bjarne Stroustrup :The C++ programming language. Addison Wesely 1991. ISBN 0-201-53992-6.

[Str, 1993] Bjarne Stroustrup: A History of C++ : 1979-1991. ISO WG21, Document No N0248, 1993.