NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

A Graphic Representation Scheme and a Tool for Natural Language Documentation of C++ Programs

by

Ian J. Hamilton

An M.C.S. Thesis

submitted to the School of Graduate Studies and Research
in partial fulfilment of the requirements
for the Master of Computer Science degree*

University of Ottawa
Ottawa, Ontario
Canada

* The Master of Computer Science Program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute for Computer Science

ISBN 0-315-75014-6

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

ACKNOWLEDGEMENT

I am grateful to my supervisor, Dr. Tuncer Ören, for his guidance and advice in the preparation of this thesis.

I wish to express my gratitude to my wife, Linda, and my sons, Matthew and David, for their patience and understanding throughout my studies.

ABSTRACT

A graphic representation scheme to represent C++ programs is presented as a set of related templates. The templates represent the three types of objects in a C++ program and their inheritance properties. A survey of existing C++ tools is presented followed by a brief introduction to the DoC++ (Documentation for C++) tool. The concepts in the C++ language underlying the DoC++ tool are discussed. The two parts of DoC++ are explained in detail. The Organized Representation for C++ (ORC++) tool provides templates for overviews and detailed views of C++ programs. The Natural language for C++ (NlC++) tool supports ORC++ by providing natural language to complement the ORC++ documentation. The implementation of the DoC++ system is discussed. Conclusions and suggestions for further research are made. Examples of the DoC++ output are included.

TRADEMARK ACKNOWLEDGEMENT

C-DOC is the trademark of Software Blacksmiths, Inc.

C++/Softbench is the trademark of Hewlett-Packard, Inc.

C++/Views is the trademark of CNS, Inc.

GUI_MASTER is the trademark of Vleermuis Software Research.

MS-DOS is the trademark of Microsoft Corporation.

ObjectCraft is the trademark of ObjectCraft, Inc.

OOSD/C++ is the trademark of Interactive Development Environments, Inc.

Saber-C++ is the trademark of Saber Software, Inc.

Software through Pictures is the trademark of Interactive Development Environments, Inc.

Teamwork/OOD is the trademark of Cadre Technologies, Inc.

Turbo C++ is the trademark of Borland, Inc.

Windows is the trademark of Microsoft, Inc.

TABLE OF CONTENTS

ACKNOWLEDGEMENT i

ABSTRACT ii

TRADEMARK ACKNOWLEDGEMENT iii

TABLE OF CONTENTS iv

LIST OF FIGURES vii

Chapter 1 INTRODUCTION 1

1.1 Origins of DoC++ 1

1.2 Object-oriented Programming 2

1.3 Software Development and CASE 2

1.4 Reverse Engineering 4

Chapter 2 SOFTWARE ENGINEERING TOOLS FOR C++ 5

2.1 Survey of Available Tools for C++ 5

2.2 DoC++ 7

Chapter 3 ORGANIZATION OF C++ PROGRAMS 8

3.1 Object Type Organization 8

3.1.1 Class Object Type 9

3.1.2 Struct Object Type 12

3.1.3 Union Object Type 13

3.2 Inheritance Properties 13

3.2.1 Single Inheritance with Private Inheritance Attribute 17

3.2.2 Single Inheritance with Public Inheritance Attribute 17

3.2.3 Cascaded Inheritance 17

3.2.4 Multiple Inheritance 22

Chapter 4 THE ORC++ REPRESENTATION SCHEME AND TOOL 25
4.1 Overview 25
4.1.1 Global Overview 25
4.1.2 Inheritance Overview 25
4.2 Detailed View 27
4.2.1 Objects 27
4.2.1.1 Object Members 27
4.2.1.2 Object Method Definitions 32
4.2.2 Non-member Methods 32

Chapter 5 THE NLC++ NATURAL LANGUAGE DESCRIPTION TOOL 39
5.1 Table of Contents 39
5.2 Numbering of ORC++ Sections 39
5.3 Introductions to ORC++ Sections 41
5.4 NlC++ Annotation in ORC++ Overview 41

Chapter 6 IMPLEMENTATION OF DOC++ 43
6.1 Batch Mode 43
6.2 Interactive Mode 43
6.3 Algorithm and Data Structures 45
6.4 DoC++ String Processing and C++ Knowledge 48

Chapter 7 CONCLUSIONS & RECOMMENDATIONS FOR FURTHER RESEARCH . 53

Chapter 8 REFERENCES 55

APPENDICES 57

LIST OF FIGURES IN APPENDICES 57

Appendix A EXAMPLE WITH A CLASS OBJECT TYPE 58

Appendix B EXAMPLE WITH A STRUCT OBJECT TYPE 64

Appendix C EXAMPLE WITH SINGLE INHERITANCE 70

Appendix D EXAMPLE WITH MULTIPLE INHERITANCE 80

Appendix E EXAMPLE SHOWING NLC++ 89

Appendix F GLOSSARY OF TERMS 108

LIST OF FIGURES

Fig. 3.1 Declaration Part of Class or Struct 10
Fig. 3.2 Declaration Part of Union 14
Fig. 3.3 Single Inheritance with Private Inheritance Attribute 18
Fig. 3.4 Single Inheritance with Public Inheritance Attribute 19
Fig. 3.5 Cascaded Inheritance with Private Inheritance Attribute 20
Fig. 3.6 Cascaded Inheritance with Public Inheritance Attribute 21
Fig. 3.7 Cascaded Inheritance with Private and Public Inheritance Attributes 23
Fig. 3.8 Multiple Inheritance with Private and Public Inheritance Attributes 24

Fig. 4.1 The Global Overview Template 26
Fig. 4.2 Single Inheritance Overview 28
Fig. 4.3 Multiple Inheritance Overview 28
Fig. 4.4 Template for Members of a Class (or Struct) . . 30
Fig. 4.5 Template for Members of a Union 33
Fig. 4.6 Template for Method Definitions - Class or Struct 34
Fig. 4.7 Template for Method Definitions - Union 35
Fig. 4.8 Graphic Representations of Sequential and Selection Blocks in Method Definitions 36
Fig. 4.9 Graphic Representations of Sequential and Repetition Blocks in Method Definitions 37
Fig. 4.10 Nesting Different Structural Blocks in Method Definitions 38

Fig. 5.1 N1C++ Table of Contents 40

Fig. 6.1 DoC++ Command Line Help Screen 44
Fig. 6.2 Main Data Tree 47

Fig. A.1 Program Listing - Example 1 60
Fig. A.2 Overview - Example 1 61
Fig. A.3 Detailed View - Example 1 62

Fig. B.1 Program Listing - Example 2 65
Fig. B.2 Overview - Example 2 66
Fig. B.3 Detailed View - Example 2 68

Fig. C.1 Program Listing - Example 3 71
Fig. C.2 Overview - Example 3 73
Fig. C.3 Detailed View - Example 3 74

Fig. D.1 Program Listing - Example 4 81
Fig. D.2 Overview - Example 4 83
Fig. D.3 Detailed View - Example 4 84

Fig. E.1 Table of Contents - Example 4 90
Fig. E.2 Overview with NlC++ - Example 4 91
Fig. E.3 Detailed View with NlC++ - Example 4 95

Chapter 1

INTRODUCTION

This thesis discusses the nature of C++ programs with emphasis on the organization of objects and inheritance. The developed scheme and the tool, DoC++ (Documentation for C++), a software engineering tool that documents software written in the object-oriented language, C++, are presented. The DoC++ tool consists of two components: ORC++ (Organized Representation for C++) and N1C++ (Natural language for C++) which are explained in greater detail later.

The DoC++ software is at the pilot project stage and is intended for the documentation of small programs rather than large software systems. However, the concepts could be implemented for large scale systems with interactive browsing. At this stage of development the DoC++ tool may be more suitable for training purposes as it provides much needed detail on the nature of the objects and the inheritance relations in C++ programs.

In this chapter the origins of DoC++ are discussed followed by a discussion of software engineering in general including object-oriented programming, computer-aided software engineering (CASE) tools, and reverse engineering. This will lay the foundation for viewing DoC++ in the context of general developments in the field of software engineering.

1.1 Origins of DoC++

This work was inspired by earlier work done at the University of Ottawa by Ören [Ören84] to provide a graphical representation scheme for computer algorithms and computer programs written in various structured languages. Ören and his associates developed software that would take a program and produce box structures that clearly indicated the beginning and end of control structures, such as the Pascal "while" repetition structure [Ayt84]. It would be clear to a reader of the documentation what code was executed inside the "while" loop. Nesting of structures was shown using a box within a box technique. Boxes enclosed selection and repetition structures of the language in question. This approach was in support of the structured programming approach to software engineering. Languages that were documented in this manner included FORTRAN, Pascal and C. This approach could be described as an example of reverse engineering which is discussed in greater detail below.

DoC++ includes more information about a program under study as it became clear that there was a need to document C++ programs with an emphasis on the objects in C++ programs and their

inheritance and encapsulation properties. The DoC++ system, however, began with an extension of Ören's documentation technique to C++, with some refinement, documenting the repetition and selection structures within the C++ function definitions.

1.2 Object-oriented Programming

C++ supports the object-oriented paradigm for computer programming and has the following properties: data abstraction, data encapsulation, inheritance and polymorphism. Other object-oriented languages include Smalltalk, Object Pascal, ADA, Eiffel, Actor, Metaphor, Objective-C and Common Lisp Object System (CLOS) [Tho90] [Lea91]. The properties of object-oriented languages allow for the re-use of code. The encapsulation and data abstraction of the object implies the interface to an object is through messages that are sent to the object through a well-specified interface. The user of the code sees the object as a black box and does not need to know the details of the inner workings of the object or worry that use of the code will have disastrous side-effects [Ber88]. Polymorphism allows the objects to be used in a wide variety of situations since this property allows for a simpler interface with the object since the same method name can be used for a variety of message types that may be sent to the object. The inheritance property allows the user of the code to use the properties of an object and add new properties to make a new object. The combination of these properties provides the basis of "off the shelf" software objects that developers could integrate into their projects [Cox90].

The emphasis on the re-use of object-oriented language code is related to the need to reduce the costs of software development. Chikofsky [Chi90] suggests that C++ may be an excellent choice for program development since it also allows existing C code to be used. The similarity between C and C++ reduces the costs of retraining staff to work with an object-oriented language. As well, C++ compilers are efficient. Choosing C++ as a language may be criticized on the basis of the relative difficulty in learning the language.

1.3 Software Development and CASE

Object-oriented languages, such as C++, can not be evaluated properly without viewing them within the context of the complete cycle of software development and the CASE environments that can aid in this development.

CASE tools can be subdivided into upper CASE tools and lower CASE tools. Upper CASE tools consist of graphical workbenches producing some of the following: entity-relationship diagrams, data-flow diagrams, state transition diagrams, flow-charts, and pseudo-code specification. Lower CASE tools traditionally refer to fourth generation languages that take the results of the upper CASE tools and are essentially application generators. Upper and

lower CASE tools can be integrated. Nilsson [Nil90] mentions the terms Integrated Software Development Environment (ISDE), Integrated Project Support Environment (IPSE), and Software Factory to refer to this integration.

The first CASE tools were based on DeMarco's structured analysis and design concept [DeM78]. This consisted of data-flow diagrams, to which a top-down decomposition methodology is applied, ending up with a structure chart that identifies the functions that are needed. Upper CASE tools were developed that allowed the software developer to graphically decompose data-flow diagrams and end up with a graphical version of the structure chart. This approach has been expanded and many CASE tools will take upper CASE results and perform lower CASE functions producing applications using code generators.

With the advent of another approach to software analysis and design, object-oriented analysis and design, the use of CASE tools based on structured analysis and design may not be appropriate. Ward [Ward90] discusses modifications to present CASE tools to adapt them to objects. Ward states that the addition of entity-relationship diagrams, event-response modelling and state-transition diagrams to CASE tools make them suitable for work with object-oriented systems.

Coad and Yourdon [Coad90] present an approach to object-oriented analysis and design that uses a new set of diagrams in the analysis stage. The diagrams represent five steps or layers of analysis and proceed from the highest level of abstraction to the lowest. First the subject layer is used to break the problem into manageable portions that are described in terms of broad objects, such as a "user." The object layer expands each of the sections in the subject layer to arrive at objects that contain data and methods that are exclusive to the data. The structure layer identifies the organization of objects since objects can consist of sub-objects. This layer shows the inheritance relationships of objects. The attribute layer shows a greater level of detail on data in each object. The service layer deals with the messages sent from object to object. Coad and Yourdon make the point that their method addresses the inheritance and encapsulation aspects of object-oriented systems. This method of analysis appears top-down in approach but does place the emphasis on objects.

Booch [Boo91] has proposed another approach to object-oriented analysis and design in his object-oriented software development (OOSD). In this bottom-up methodology Booch uses the class template, the class diagram, the object diagram, the module diagram and the state transition diagram as the elements of his design system. The first element, the class template, is a detailed list of properties for the class including: the visibility, the cardinality, the place this class has in a class hierarchy, and the interface with details on private, protected and public sections. The class diagram shows relationships between classes in a graphical fashion. The object diagram is

used to define the messages passed between the objects in the system. The implementation of the system is handled in the module diagram; this diagram shows, for example, which objects will be compiled in one file in a C++ program. The Booch approach shows a close connection between the analysis and design stages and the implementation stage. This approach allows for incremental development and simplifies prototyping and the scaling up of a project.

1.4 Reverse Engineering

Within the domain of software engineering certain terminology has evolved that helps one to organize work in this field. Drawing from traditional engineering fields the term, reverse engineering, has become part of the jargon of software engineering. Reverse engineering in the software field involves taking a finished product, i.e. code, and producing documentation and/or design information. The use of the term, reverse engineering, leads one to consider the opposite term, forward engineering, which consists of the usual method of developing software moving successively between the stages of definition, requirements, analysis, design, implementation, validation, and maintenance. CASE design tools are forward engineering tools and there is a tendency not to re-use code since it is often automatically generated for each new application.

According to Chikofsky [Chi90] the products of reverse engineering are documentation and design recovery which aid the software engineer in achieving the following three goals. First there will be improved comprehension of code and design. This is accomplished by various means including improved formatting of code and the production of alternate views and varying levels of detail and abstraction for the system. Secondly, automatic redocumentation will result in great time savings and ensure that documentation is done. Thirdly, software will be re-used since proper documentation will promote re-use. Reverse engineering may be advantageous when used in an object-oriented analysis and design environment.

Chapter 2

SOFTWARE ENGINEERING TOOLS FOR C++

A survey of existing tools specifically for the object-oriented language, C++, will be presented. Following this survey is an introduction to the DoC++ tool.

2.1 Survey of Available Tools for C++

The most basic tool required for software development, a compiler, is readily available for the C++ language. Although C++ was developed in the Unix environment, much work has been done to produce equivalent compilers for MS-DOS. Changes in the nature of C++ complicated the development of these products. Zortech, Borland and other companies now produce MS-DOS C++ compilers that come with an impressive list of features, including an editor, project management features, an extensive C function library, and a library of ready-to-use classes.

To produce good quality software more may be needed than a compiler. A survey of products available commercially to aid software development using C++ are mentioned below. Tools designed for workstations are mentioned first, followed by tools for the MS-DOS environment. The workstation tools are more powerful and more expensive.

The first workstation product, C++/Softbench from Hewlett-Packard, is described in detail by Arnistead [Arn90]. This product has been designed to aid in the design and management of large projects written in C++. It consists of seven tools and class libraries. The seven integrated tools are the C++ Developer, Static Analyzer, Program Builder, Program Editor, Program Debugger, Development Manager, and Mail. The C++ Developer tool allows for graphical construction of classes including an inheritance hierarchy. Classes can be added or deleted and the inheritance hierarchy changed using a graphical browser. From the graphical representation, source code templates can be automatically generated. The C++ Developer also allows the user to view and edit the members of a class. C++/Softbench's Static Analyzer allows the user to see information on the program that is specific to C++ such as the names of all of the classes. C++/Softbench comes with some features that are common to most recent compilers. The Program Builder helps the user compile a program with many source files. There is the Program Editor that can be replaced in the system with another editor if the user chooses. The Development Manager allows version control and the integration of other tools with the C++/Softbench system. C++/Softbench can be described as a CASE tool.

Another set of workstation CASE tools aimed at C++ has been developed by Interactive Development Environments (IDE). One of IDE's products, Software through Pictures, is an analysis and design tool using a wide variety of methods including DeMarco's traditional structured analysis and design approach and Booch's earlier approach to object-oriented analysis [Oman90]. Another product from IDE, OOSD/C++, allows the program developer to start with a graphical design editor and through the use of a C++ re-use library at the design phase, end up with a design that is suitable for implementation in C++ [JOOP91a]. Plans are being made to integrate these IDE products with Saber-C++ produced by Saber Software [JOOP91b].

Still another workstation C++ CASE tool is Teamwork/OOD from Cadre Technologies. This tool consists of a graphical editor for the design stage and a C++ code generator, automating the production of code from the design [JOOP92].

The first MS-DOS product that will be included in this survey of C++ tools, GUI_MASTER from Vleermuis Software Research in the Netherlands, is a tool for automatically producing the graphical user interface for a C++ program being written for MS-DOS. GUI_MASTER is a forward engineering tool that writes a skeleton program complete with pop-up menus. The user of this system must write all the code other than the code for the user interface [Byte90a].

The second MS-DOS product, C-DOC from Software Blacksmiths, is a MS-DOS C/C++ program analyzer and documenter. C-DOC's features include a hierarchy tree for calling and called methods, a table of contents, a cross-reference of identifiers, reformatting of code, and path complexity diagrams for each method. This product fits into the reverse engineering category [Byte90b].

Another MS-DOS product, C++/Views from CNS, is a productivity tool for use in the Windows 3.0 environment. C++/Views has a C++ class library, a class browser and an interface generator. The browser allows the user to use templates for editing C++ classes [Wei91] [JOOP91d].

ObjectCraft is a MS-DOS productivity tool that allows developers to do visual programming. The tool will automatically generate C++ code [JOOP91c]. Improvements now allow the importing of C++ files, the printing of the visual programming diagrams and the writing of C++ methods within ObjectCraft [JOOP91e].

2.2 DoC++

There is more than one stage of development in the DoC++ computer-aided software engineering tool. The first component of DoC++ involves a tool that takes an existing C++ program and documents the program using various templates that are filled in. This tool is called ORC++ and stands for Organized Representation for the C++ language. For C++ programs being examined, ORC++ provides a good overview of the program, an outline of the nature of inheritance in the program, details of the declarations for each object type, and a detailed view of the code in a program using a highly organized scheme.

The second component of the DOC++ tool consists of the N1C++ natural language description tool. This augments the ORC++ description in two ways. DoC++, if used, adds introductory remarks before each of the ORC++ descriptions and provides further details on inheritance, encapsulation and accessibility of various object types in a C++ program.

The DoC++ system provides for two modes of operation: a batch mode and an interactive mode. In the batch mode a C++ program is given as a command line argument and the system analyzes the program and sends the DoC++ documentation to a file. The output can be modified using command line switches.

In interactive mode the user can choose options from menus. Programs may be chosen for analysis. The user may specify the type of information that is desired.

The DoC++ tool is a reverse engineering method and aids the software engineer in achieving the three goals of reverse software engineering: program comprehension, automatic redocumentation, and reuse of code. Since the DoC++ tool automatically generates products that document C++ programs it is reasonable to describe it as a CASE tool but since DoC++ does not provide help in analyzing new projects or generate working systems it can not be seen as a complete CASE system. Due to its operation at the code level it could be seen as a lower CASE tool that may fit into the object-oriented software development (OOSD) scheme described by Booch [Boo91]. Booch's orientation in analysis and design is to stress the properties of classes and objects to be used in a system. DoC++ also stresses classes but from the reverse engineering viewpoint and hence may complement Booch's approach to software development.

Chapter 3

Organization of C++ Programs

C++ is C with some additions. To some extent this work clarifies the nature of C++ in general and the structure of C++ programs in particular. The ideas presented in this report and the related software may be useful in C++ language training as well as for documentation purposes.

C++ programs consist of some or all of the following items. At the start of the program there is an initial quantity of code that may contain "define" statements and "include" statements and some declarations. As well C++ programs contain functions which are also called methods using OOL terminology. The code for methods is differentiated from the declaration by calling it the definition of the method. Declarations may appear much as seen in C programs but in C++ there is a special type of declaration that is the focus of C++ programs and this is the object type declaration. The method definitions in C++ programs consist of both member methods, i.e. members of an object type, and non-member methods, i.e. not members of an object type.

3.1 Object Type Organization

An essential aspect of C++ is the nature of the different object components in the language.

In the C computer language, user-defined data types called structs may contain data of various types; in C++ this concept is expanded so that the user-defined data types also may contain methods that typically modify the stored data. This combination of data and methods declared in the same structure is the essence of C++ being an object oriented language.

There are three general object types: the class, the struct and the union. As in C's struct type, the three object types in C++ can be tailored to fit a specific data structure requirement. Consequently, there may be many object types in a program and for each of these object types there may be more than one variable (or instance) using this type. Unfortunately, the distinction between the object type and an instance of it is not always made. The term "object" sometimes is used to refer to both the general type and the specific instance. The same terminology problem exists with the terms, class and struct. Consequently, in this work the terms, "object type," "class type" and "struct type" are used where many authors would use "object," "class" and "struct".

3.1.1 Class Object Type

In C++ an object type may contain, in addition to data, functions (usually called methods in OOL terminology) that operate on the data in the object type. The data and methods that belong to an object type are called members of the object type. There are different categories of data and methods in an object type. Each category of data or methods may contain zero or more different data items or methods and each must be declared in the declaration of the object type. As well as declaring the methods within the type declaration it is also necessary to define the methods. The definition of the method can be done in the declaration but is usually done outside the declaration unless the code for the definition is quite short or the programmer wishes the code to be assembled inline rather than as a function call with its associated overhead.

The categories of data and methods in the class object type appear in Figure 3.1. Organizing the components of C++ is an essential element in this process of clarifying the nature of C++ programs.

To explain the organization of the elements in Figure 3.1 it is useful to focus first on the sub-elements that are common to each section. All of the sections except Friends contain data boxes. The data box is the sub-section symbolizing the declaration of data to be used, be it a simple standard data type such as an integer or a complex tree data type.

The data can be put in one of three sections: private, protected, or public. Some data can be put in one section, other data in another section. The choice of where to put the data depends on the C++ users decision on how much encapsulation is needed for the data in question.

Data put in the public section is not really encapsulated. One can access this data directly from a method outside the object type using the same syntax as seen in C, i.e. the instance of the object type followed by the dot operator followed in turn by the data identifier. If the instance of an object type that kept track of the number of people in a bank queue was called q1 and the public data in this object type was called number_of_people, then outside the object type one could write

```
q1.number_of_people++
```

to increment the counter.

Data put in the private section is encapsulated. It cannot be accessed as done with the public data. Outside methods may not refer to this data. Generally only member methods may access this data. The member methods that access the private data maybe in the private, protected, or public sections. These methods refer to the data without the name of the instance of the object type and the dot operator since the method refers to the data in the

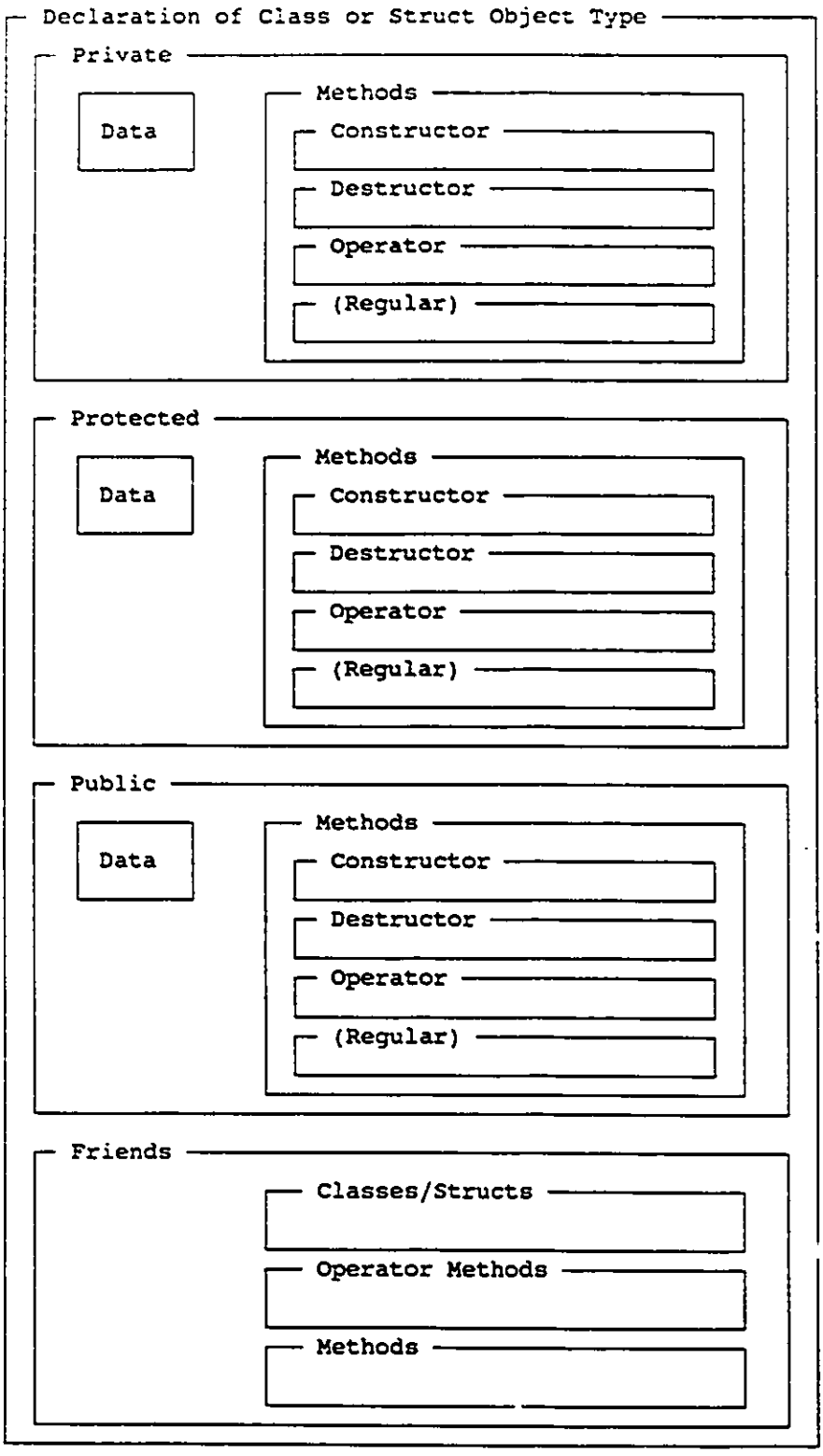


Figure 3.1 Declaration Part of Class or Struct

same instance of the object type that the method is in. Using the example given above with the bank queue, if the instance was once again q1, it would be illegal to write

```
q1.number_of_people++
```

outside the object type. Most likely number_of_people would be incremented by use of a method put in the public section of the object type. If this public method was called increment_the_queue then the syntax used outside the object type to increase the value in q1.number_of_people would be

```
q1.increment_the_queue().
```

Within the method increment_the_queue one would find the following code:

```
number_of_people++
```

The data number_of_people is encapsulated or hidden. It is not easy to inadvertently change the value of this data. An additional feature of this method of using C++ is data abstraction. It is only necessary for the outside world to know that the queue is being incremented. The exact details how the queue is kept is not known outside the object type.

Data put in the protected section is encapsulated but not quite to the degree that private data is encapsulated. Protected data have all the access properties of private data with one extra degree of access. This involves inheritance and will be discussed in greater depth later in this work. Protected data may be accessed by all the member methods of object types that are derived from the object type containing the protected data.

Now that the data has been discussed it appropriate to discuss the methods. Note that the private, protected, and public sections all contain the same four types of methods. The rules of access that apply to the data also apply to the methods. We normally hear the term, data hiding, that is used in conjunction with encapsulation. In C++ method hiding is also possible. The methods in the public section may be accessed directly from outside the object type. As with the public data this involves the use of the name of the instance of the object type followed by the dot operator followed by the name of the method. Methods in the protected and private sections may only be called by methods that can access them. This includes the public methods, friend methods and possibly methods in derived object types.

The first type of method listed in the private, protected and public sections of Figure 3.1 is the constructor method. Constructor methods are called automatically whenever an instance of an object type is declared. It is the code that typically allocates memory and initializes data.

The second type of method listed in the private, protected

and public sections of Figure 3.1 is the destructor method. Destructor methods are called automatically whenever the scope of an instance of an object type ends. This type of method is used typically to deallocate memory.

The third type of method listed in the private, protected and public sections of Figure 3.1 is the operator method. Operator methods are used to define the use of operators such as the + operator. Instead of using awkward syntax such as

```
c = add(a,b)
```

where a and b may be user-defined data types such as vectors or strings, C++ allows the programmer to write

```
c = a + b
```

The variable c will contain the vector sum of a and b or the concatenation of the two strings. However the operator has to be defined as a method. One can think of the operator method as being equivalent to the method operator+(a,b).

The last type of method listed in the private, protected and public sections of Figure 3.1 is the regular method. The term regular is in brackets in Figure 3.1 since it is a term created in this work to describe the methods that do not fit into the previous three types of methods mentioned above. These resemble the methods seen in C programs.

An exception to this encapsulation is the use of friends. Friends are a special type of member and may consist of individual methods or an entire class object type or struct object type in which there may be several methods. All of the methods that are declared to be friends may access the private data of a class object type even though the friend methods are outside the class object type.

In Figure 3.1 there are three types of friends. The last two types, operator methods and regular methods, are individual methods that have access to the private and protected members (data and methods) of an object type. These methods may be members of some other object type or may be methods that do not belong to any object type. The first type of friend, the class or struct refers to another object type. All of the methods in this other object type have access to the private and protected members (data and methods) in the object type in question. The use of friends overrides the encapsulation built into C++ and must be used with caution. In this work the emphasis in on encapsulation and not the methods used to avoid encapsulation.

3.1.2 Struct Object Type

Originally the class object type was an extension of the original C++ struct data type. With present versions of C++ the class and struct object types are the same except for some default

factors that are discussed in section 3.2. Hence Figure 3.1 shows the categories of data and methods for the struct object type. The original version of the struct did not have private and protected sections.

3.1.3 Union Object Type

Just as C has the union user-defined type, C++ also has this type but, in addition to holding data, unions also have methods and the organization is shown in Figure 3.2. Note that unions have only a public section.

3.2 Inheritance Properties

Inheritance in C++ programs is implemented with the object types, the class and the struct. These two object types may inherited data and methods from other class or struct types. Several levels of inheritance are possible and an inheritance hierarchy arises. In an inheritance hierarchy class types and struct types may appear together. An object type that inherits data and/or methods is called a derived object type; an object type from which data and/or methods are inherited is called a base object type. An object type can be derived from one object type and at the same time be the base object type for another derived object type.

An inheritance hierarchy can be expressed by means of an indented chart as follows:

```
class one
     struct two:one
          struct three:two
```

This indicates that class object type one is the parent of struct object type two which in turn is the parent of struct object type three. Instances of these types follow the same inheritance rules in the sense that an instance of object type three has properties described in object types one and two. Instances do not inherit from other instances.

Both the class object type and the struct object type may contain members divided into sections showing their level of encapsulation as seen in Figure 3.1.

In the class object type, until the keywords, public: or protected: appear in the declarations, the data and methods declared are considered to be private. Private is the default for class object types. In the struct object type, until the keywords, private: or protected: appear in the declarations, the data and methods declared are considered to be public. Public is the default for struct object types.

The connection between inheritance and the private, protected

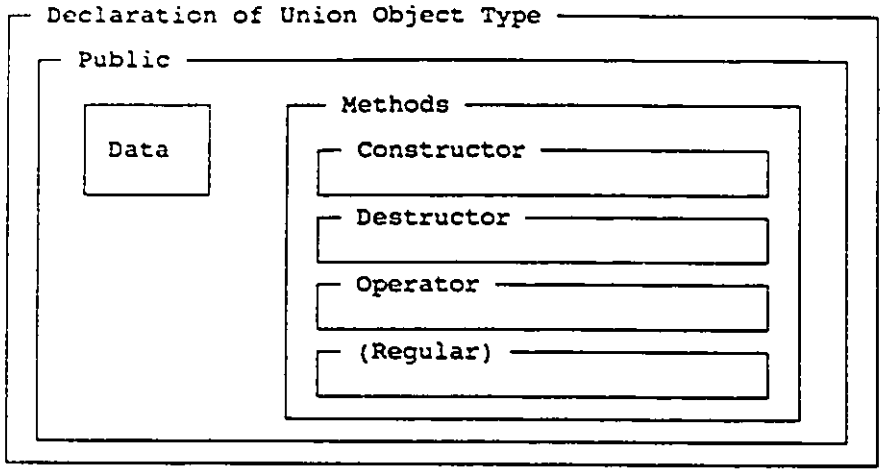


Figure 3.2 Declaration Part of Union

and public sections of the object types involved needs to be made. As data and/or methods are inherited their level of encapsulation may change. For example, data that is in the public section of a base class type may become part of the private section of the derived object type. Similarly data in the protected section of a base class type may become part of the private section of the derived object type. Data in the private section of a base class type becomes part of the derived object type but it can not be accessed directly using methods in the derived object type. This creates a new organizational sub-section, called the Buried Section in this thesis. Buried data can be accessed by using the public methods of the parent class which can access its private data which becomes the buried data of the derived class. The same situation exists for the four types of methods: (Regular) Methods, Operator Methods, Constructor Methods, and Destructor Methods. Each method may or may not be called depending on where it is in the buried-private-protected-public hierarchy of accessibility.

In the declaration of either a class or a struct type the following syntax is encountered:

```
class class_name : inheritance_attribute base_object_name

    or

struct struct_name : inheritance_attribute base_object_name
```

The inheritance attribute can be either public or private. The lack of an inheritance attribute results in defaults being applied. In the case of a struct type being the base object type the default is public. In the case of a class type being the base object type the default is private.

```
class beta : gamma
```

Example 3.1

In Example 3.1 beta is a class that inherits from a base object type with the name, gamma. If gamma is a struct type then the inheritance attribute is public. If gamma is a class type then the inheritance attribute is private.

```
class beta : private gamma
```

Example 3.2

In Example 3.2 beta is a class that inherits from a base object type with the name, gamma. If gamma is a class then the use of private is redundant since private is the default inheritance attribute. However if gamma is a struct then the

inheritance attribute is changed from the default which is public to private.

```
struct beta : public gamma
```

Example 3.3

In Example 3.3 beta is a struct that inherits from a base object type with the name gamma. If gamma is a struct then the use of public is redundant since public is the default inheritance attribute. However if gamma is a class then the inheritance attribute is changed from the default which is private to public.

From the above discussion of inheritance it is clear that an object type has the possibility of having, in addition to Private, Protected, and Public data and methods, data and methods that could be described as Buried (or Inaccessible), Inherited Private, Inherited Protected, and Inherited Public. The following list arranges the various sections in a class type or struct type in which inheritance is involved according to the degree of encapsulation that they provide to their members:

Buried (or Inaccessible) Members
Private Members & Inherited Private Members
Protected Members & Inherited Protected Members
Public Members & Inherited Public Members

The preceding concepts are the basis of the treatment of C++ inheritance in the ORC++ system. Examples in Appendices C, D and E of this thesis will clarify the approach. The nature of inheritance in C++ can be a source of confusion. It takes careful study of a program to determine what data an object inherits. An automated system will eliminate some of the careful searching required and this will be shown in the appendices of this thesis.

In the following four sections the term "single inheritance" refers to inheritance involving one base object type and one derived object type, either of which may be a class object type or a struct object type. This is analogous to a child inheriting from the child's parent (singular emphasized). The term "cascading inheritance" refers to single inheritance involving more than two generations. As well as inheriting data and methods directly from the previous generation, data and methods from the previous generation's previous generation are inherited indirectly. This is analogous to a child inheriting from the child's grandparent indirectly through the child's parent. Finally, the term "multiple inheritance" involves a derived object type inheriting from more than one base object type directly. This type of inheritance is analogous to a child inheriting from both parents, although in C++ the number of parents is unlimited.

3.2.1 Single Inheritance with Private Inheritance Attribute

In the inheritance declaration, the base object type being a class or the keyword "private" preceding the name of the base object type indicates that the inheritance attribute is "private." The resultant changes in the level of encapsulation between generations with this inheritance attribute are shown in the Figure 3.3. Arrows in this figure indicate that the private data and methods of the base object type become buried in the derived object type. Protected data and methods from the base become private in the derived. Public data and methods of the base also become private in the derived. The base object type's data and methods become more encapsulated as they are inherited.

3.2.2 Single Inheritance with Public Inheritance Attribute

In the inheritance declaration, the base object type being a struct or the keyword "public" preceding the name of the base object type indicates that the inheritance attribute is "public." The resultant changes in the level of encapsulation between generations with this inheritance attribute are shown in Figure 3.4. The double arrows in this figure indicate that the private data and methods of the base object type become buried in the derived object type just as was seen with the private inheritance attribute. This information is encapsulated. The arrows indicate that the protected data and methods from the base remain protected in the derived object type. The public data and methods of the base remain public in the derived object type. The level of encapsulation remains unchanged as the protected and public data and methods are inherited.

3.2.3 Cascaded Inheritance

Figure 3.5 shows the changes in level of encapsulation with three generations in which the inheritance attribute is private. The transitions between the first generation and the third generation is an extension of Figure 3.3. Under the second generation we now have a private section that contains both data and methods that are declared in the second generation as well as inherited data and methods.

Figure 3.6 shows the same type of cascaded inheritance over three generations. In this case the inheritance attribute between both the first and second generations and the second and third generations is public.

Figure 3.7 is a combination of the two previous figures. The single lines with arrows indicate inheritance with a private inheritance attribute; the double lines with arrows indicate inheritance with a public inheritance attribute. The attribute between the first generation and the second may be either private or public. Between the second and third generation the same may be said. All combinations of paths are considered; the inheritance attribute may change in subsequent generations.

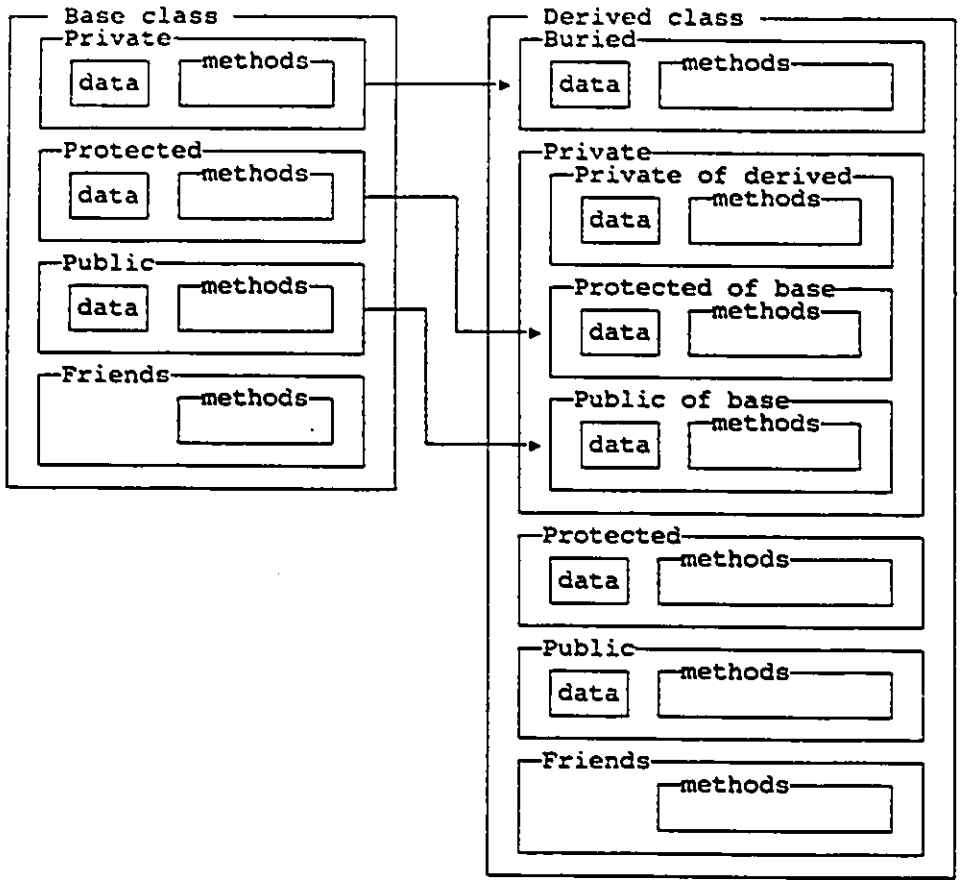


Figure 3.3 Single Inheritance with Private Inheritance Attribute

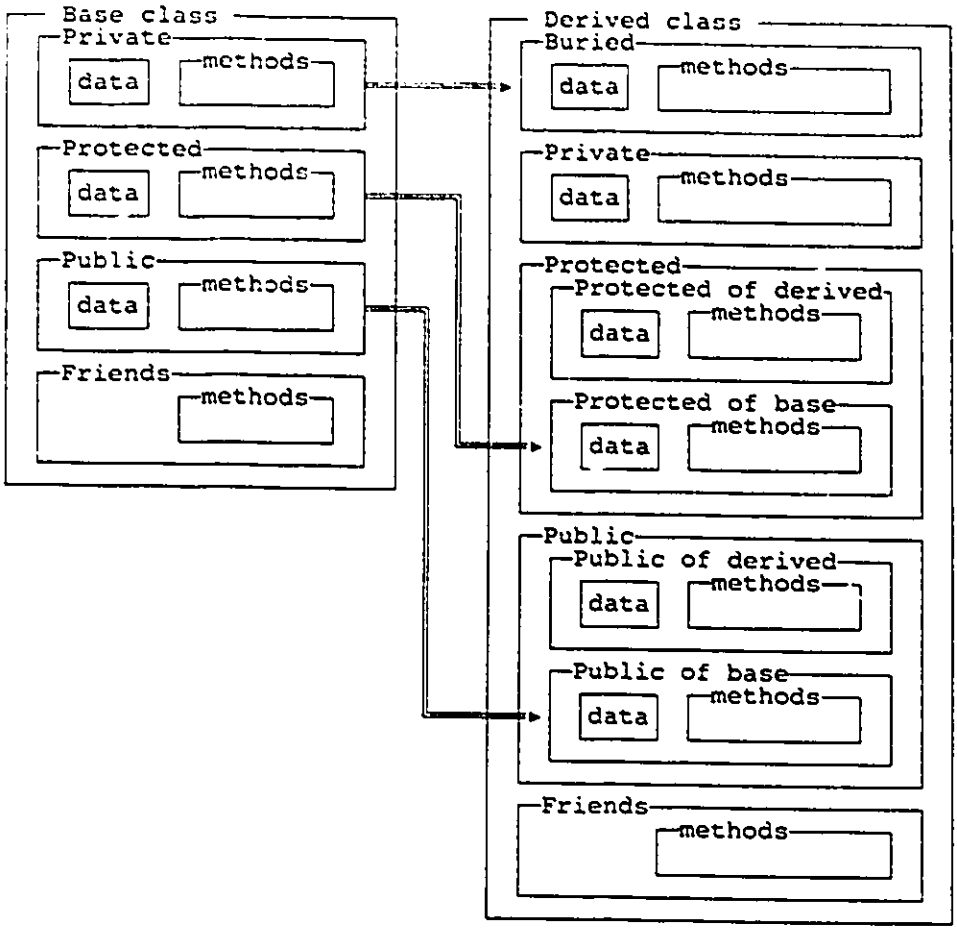


Figure 3.4 Single Inheritance with Public Inheritance Attribute

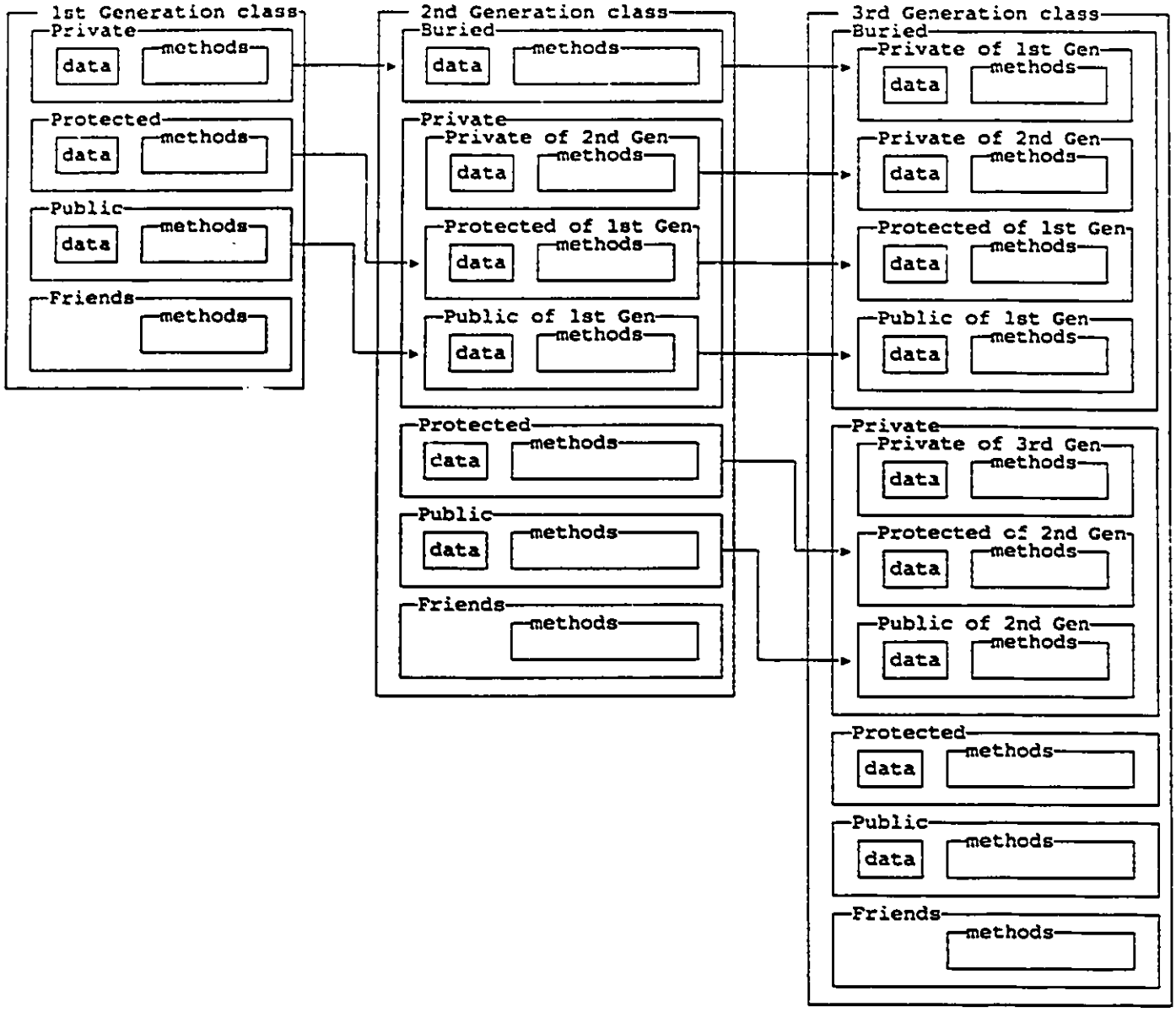


Figure 3.5 Cascaded Inheritance with Private Inheritance Attribute

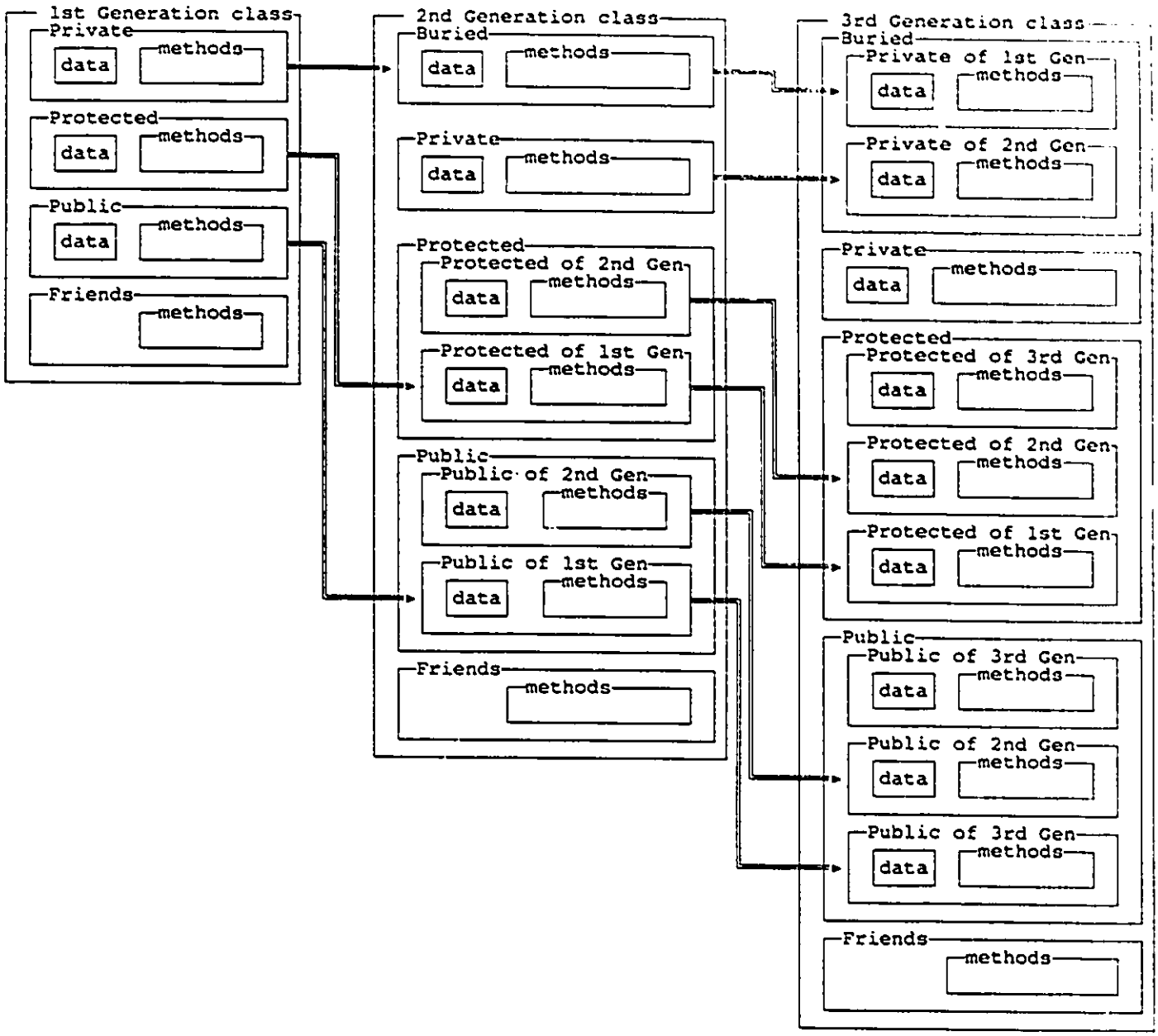


Figure 3.6 Cascaded Inheritance with Public Inheritance Attribute

As the data and methods from both the first and second generations are inherited indirectly or directly by the third generation we have increasingly complex combinations of possible inherited data and methods. The private section of the third generation may contain its own declared private members, protected members and public members from the second generation, and protected members and public members from the first generation. This assumes that the first and second generations have protected and public members. If they do not have any of these members there is nothing to inherit. Figure 3.7 has been made as general as possible. Also the types of data and methods in the private section of the third generation is dependent on the inheritance attributes between the first, second and third generations. Having protected members from the first generation in the third generations private section is possible only if the inheritance attribute between the first and second generation is public and the inheritance attribute between the second and third generation is private. The Friend Section has been deleted in Figure 3.7 because of space considerations.

3.2.4 Multiple Inheritance

Figure 3.8 shows multiple inheritance involving two base object types, base1 and base2, and one derived object type, derived. As with earlier figures, single lines with arrows indicate a private inheritance attribute, and double lines with arrows indicate a public inheritance attribute. The two base object types are placed on either side of the derived object type. As with cascaded inheritance the derived object type has buried, private, protected, and public sections that contain data and members from various sources.

The inheritance attribute between base1 and derived may be either private or public. Similarly the inheritance attribute between base2 and derived may be either private or public. It is possible for the public members of base1 to become private members in derived and for the public members of base2 to become public members of derived.

Inheritance can involve combinations of cascaded and multiple inheritance. This can lead to quite complicated inheritance paths that defy the graphical representation technique used in the figures in this section. However there is a need to know what is inherited and at what level it is encapsulated. This is another justification for the development of the DoC++ tool.

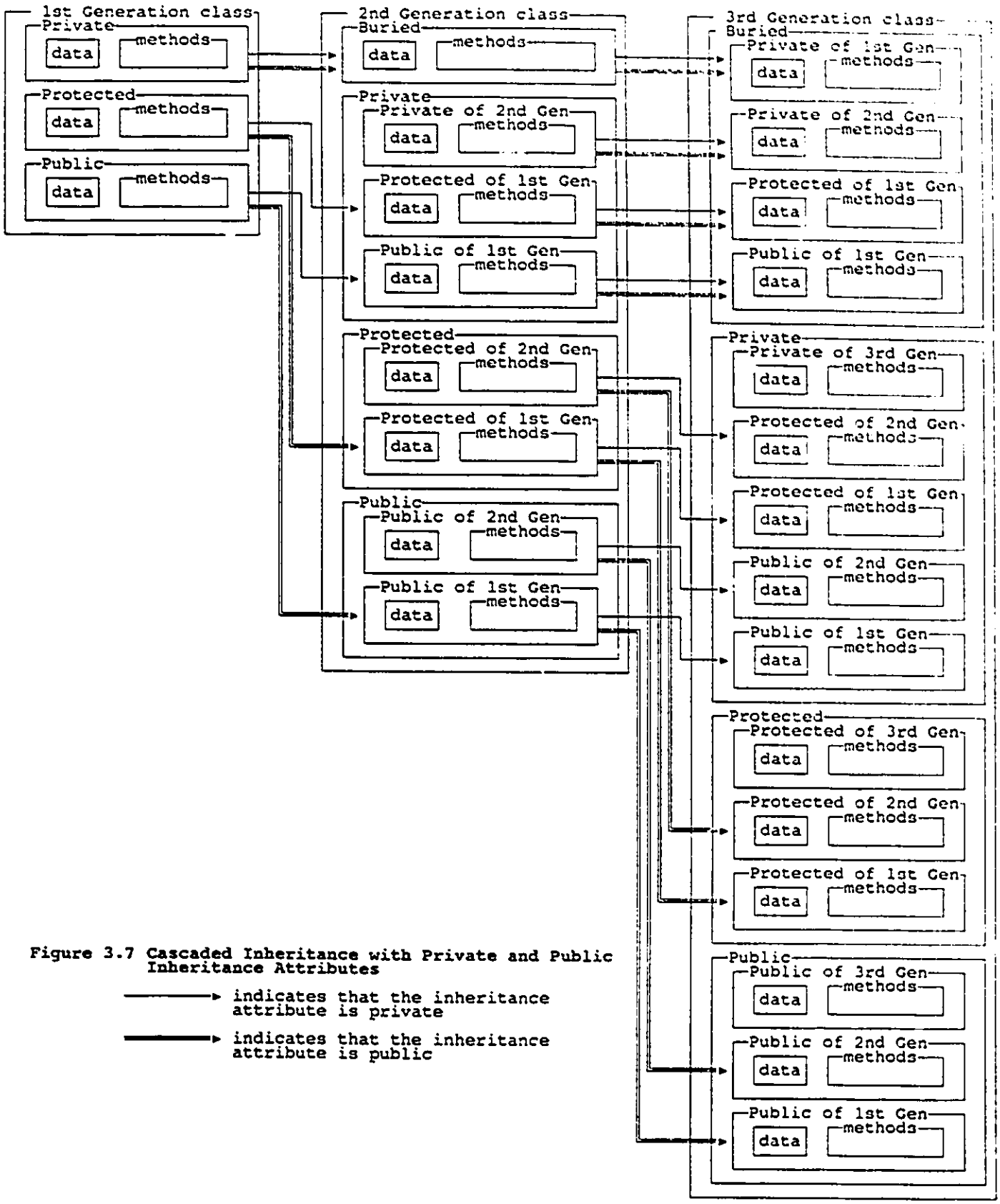


Figure 3.7 Cascaded Inheritance with Private and Public Inheritance Attributes

——→ indicates that the inheritance attribute is private

——→ indicates that the inheritance attribute is public

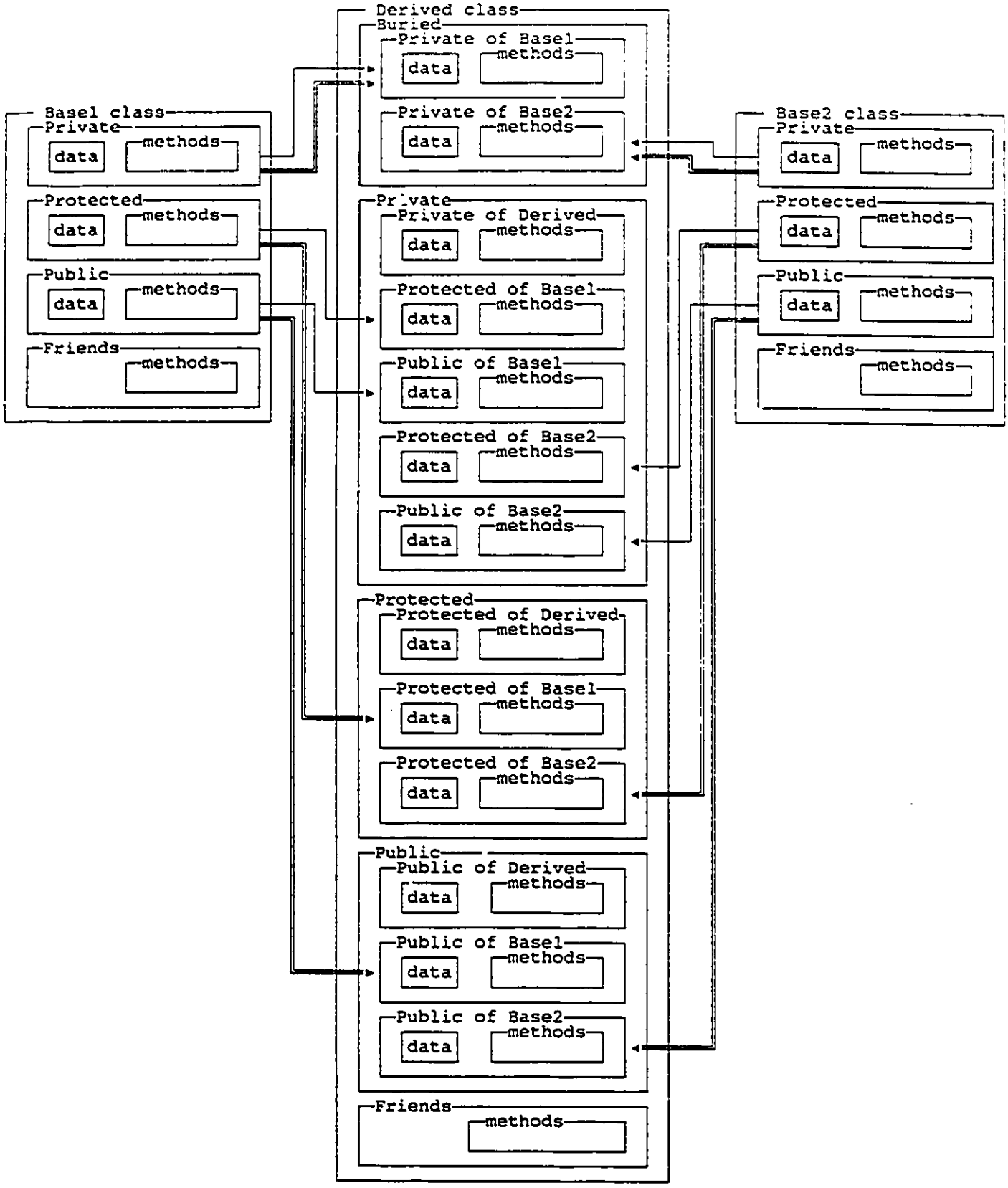


Figure 3.8 Multiple Inheritance with Private and Public Inheritance Attributes

——► indicates that the inheritance attribute is private
═══► indicates that the inheritance attribute is public

Chapter 4

THE ORC++ REPRESENTATION SCHEME AND TOOL

The ORC++ representation scheme and tool is subdivided into two components. One is a set of overviews; one is a detailed view of the object types in the program and the non-member methods. All these views are presented in a manner consistent with the previously discussed C++ program organization scheme.

4.1 Overview

The overview consists of two parts:

1. a global overview, and
2. an inheritance overview.

4.1.1 Global Overview

The most general component of the overview is the Global Overview. The intent of this overview is to give the broadest view of the program, showing the initial macro definitions, file inclusions and declarations and then the names and types of all the object types in the program, followed by the names of all non-member methods.

The five possible subsections of the Global Overview are shown in Figure 4.1. The object types are organized in a particular order: class object types first, followed by struct object types and union object types. Note that the template expands and contracts for the particular C++ program being analyzed. If there are no components in a certain category the box for this category will not appear. If a certain category has more than one element then the box expands and all the elements are shown, one per line.

4.1.2 Inheritance Overview

The Inheritance Overview diagram gives an overall view of the nature of inheritance among object types used in the program. It is important to see at a glance the parent(s) and grandparent(s) of an object.

Global Overview of program:	<program name>

		Number	Names
Initial	Includes		
	Defines		
	Declarations		
Classes			
Structs			
Unions			
Non-member methods			

Figure 4.1 The Global Overview template

Two skeletal diagrams for the Inheritance Overview are shown in Figures 4.2 and 4.3. Figure 4.2 shows single inheritance and indicates that the "derived object type" is a part of the larger "base object type" since the "derived object type" is the label for a box that is inside the box labelled "base object type." Only one generation of inheritance is involved.

Figure 4.3 shows multiple inheritance as well as multi-generation inheritance. The "object type 4" inherits from both "object type 2" and "object type 5." This is multiple inheritance and is shown graphically since the box with the label "object type 4" is contained within the two other boxes.

Multi-generation inheritance (also called cascaded inheritance) is demonstrated since "object type 4" inherits indirectly from "object type 1" by inheriting directly from "object type 2."

In the examples in the appendix the issues of inheritance beyond one generation and multiple inheritance will be addressed in greater detail. The Inheritance Overview provides automatic documentation of these aspects of inheritance and provides help in an area that requires careful analysis.

4.2 Detailed View

The Detailed View allows the user to see two types of detailed information. This view shows information on the objects and on the non-member methods used in the program. The Detailed View for objects is explained in section 4.3.1. The Detailed View for non-member methods is covered in section 4.3.2.

4.2.1 Objects

This part of the Detailed View provides information on the object types used in the program. There are two components to each object: the members in the object and the definitions for the methods in the object. Each of these are documented in the Detailed View and are explained in sections 4.3.1.1 and 4.3.1.2 respectively.

4.2.1.1 Object Members

Information on the members that make up an object is provided in an organized fashion following the philosophy of the DOC++ approach. Two types of members are of concern: the members that are declared to be part of the object type and the members that are inherited from other object types.

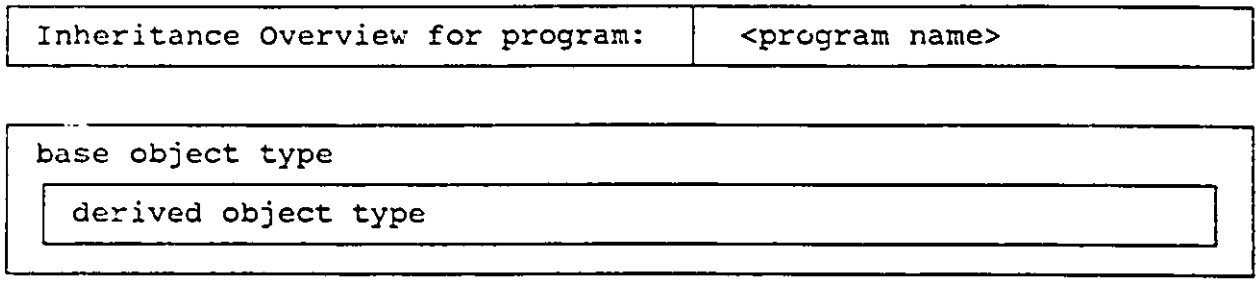


Figure 4.2 Single Inheritance Overview

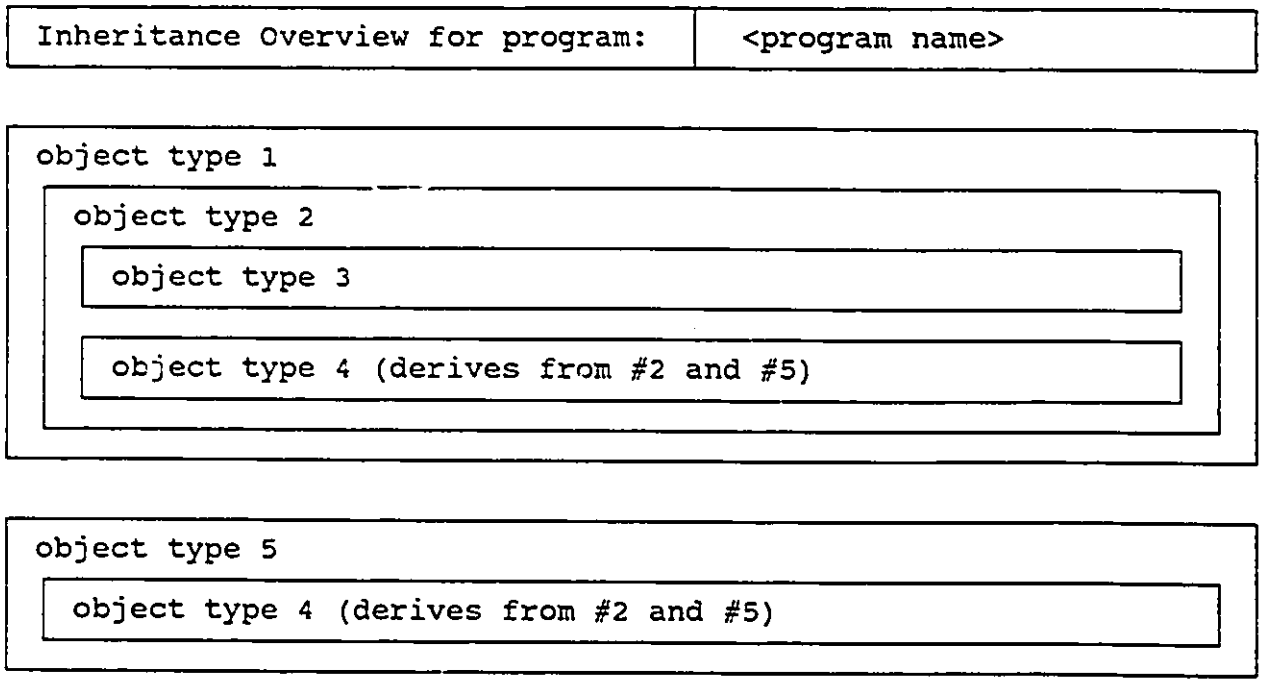


Figure 4.3 Multiple Inheritance Overview

The declared members may be in the following categories:

Private Members
Protected Members
Public Members
Friends

Class object types and struct object types may inherit members from other object types and consequently may indirectly have additional non-declared members belonging to the following member categories:

Buried Members
Inherited Private Members
Inherited Protected Members
Inherited Public Members

Union object types are not allowed to inherit and may only have members in the public category.

In both of these lists of member categories, the order of presentation moves from most encapsulated to least encapsulated. Within all of the categories except "Friends" the members may be further divided into data and the four types of methods: constructor, destructor, operator and regular.

The combination of the declared member categories and the inherited member categories results in this list of all possible member categories:

Buried Members
Private Members
Inherited Private Members
Protected Members
Inherited Protected Members
Public Members
Inherited Public Members
Friends

As with the other object overviews, the order of presentation is from most encapsulated to least encapsulated. In cases where two groups have the same level of encapsulation, as with private members and inherited private members, the declared members are presented first.

Figure 4.4 is the template for object members in the Detailed View and shows all the object member categories mentioned above. This overview contains both declared and inherited members. This allows the user to see all members of a class or struct object type, including the inherited members which may not be evident in the object type's declaration.

Members of CLASS or STRUCT	<class name> or <struct name>
of the Program:	<program name>

Inherited Buried	Data		
	Methods	Constructor	
		Destructor	
		Operator	
		(Regular)	
Private	Data		
	Methods	Constructor	
		Destructor	
		Operator	
		(Regular)	
Inherited Private	Data		
	Methods	Constructor	
		Destructor	
		Operator	
		(Regular)	
Protected	Data		
	Methods	Constructor	
		Destructor	
		Operator	
		(Regular)	

Figure 4.4 Template for Members of a Class (or Struct)

Inherited Protected	Data		
	Methods	Constructor	
		Destructor	
		Operator	
		(Regular)	
Public	Data		
	Methods	Constructor	
		Destructor	
		Operator	
		(Regular)	
Inherited Public	Data		
	Methods	Constructor	
		Destructor	
		Operator	
		(Regular)	
Friends	Classes/Structs		
	Methods	Operator	
		(Regular)	

Note: The template is followed by either one of the following notes:

"All method definitions are implemented." or
"The method definitions preceded by a "-" are not implemented."

Figure 4.4 (cont'd) Template for Members of a Class (or Struct)

Some methods that are declared to be part of an object type may not be implemented, i.e. there is no method definition. If this is the case the method will be proceeded by a "-" in this template and at the end of the template, a message will appear to indicate that the declaration contains some methods that are not implemented. If all the methods are implemented a message to that effect appears beneath the template.

Figure 4.4 is the template for both class and struct object types; both may inherit members from other object types and the template has space for inherited members. The organization for the union type is shown is Figure 4.5. Union types may have only public members and may not take part in inheritance.

4.2.1.2 Object Method Definitions

The Detailed View shows, in addition to the members for each object, the actual code for all methods declared in an object type. The method definitions are presented in an organized representation that clearly classifies the method types and demarcates the control structures used in the methods.

The template for the types of methods found in class or struct object types is shown in Figure 4.6. The template for union object types is in Figure 4.7. The method definitions for all methods belonging to the same type, such as constructor methods, are grouped together.

Figures 4.8 & 4.9 show the control structure organization used inside the method definitions. Each structure has an obvious beginning and an obvious end. The code controlled by this structure is inside a box and it is clear that this code is included in the structure. Included are the repetition structures: the do while loop, the while loop, and the for loop; and the selection structures: the switch, the simple if and the compound if else. This representation scheme allows for nesting of these structures as well. Figure 4.10 illustrates nesting of C++ structures.

4.2.2 Non-member Method Definitions

As well as the Detailed View of objects, DoC++ supplies an organized representation scheme for the non-member methods, methods that are not part of any object. These methods are shown using the same graphical methods used with the method definitions for the objects.

Members of UNION	<union name>
of the Program:	<program name>

Public	Data		
	Methods	Constructor	
		Destructor	
		Operator	
		(Regular)	

Note: The template is followed by either one of the following notes:

"All method definitions are implemented." or
"The method definitions preceded by a "-" are not implemented."

Figure 4.5 Template for Members of a Union

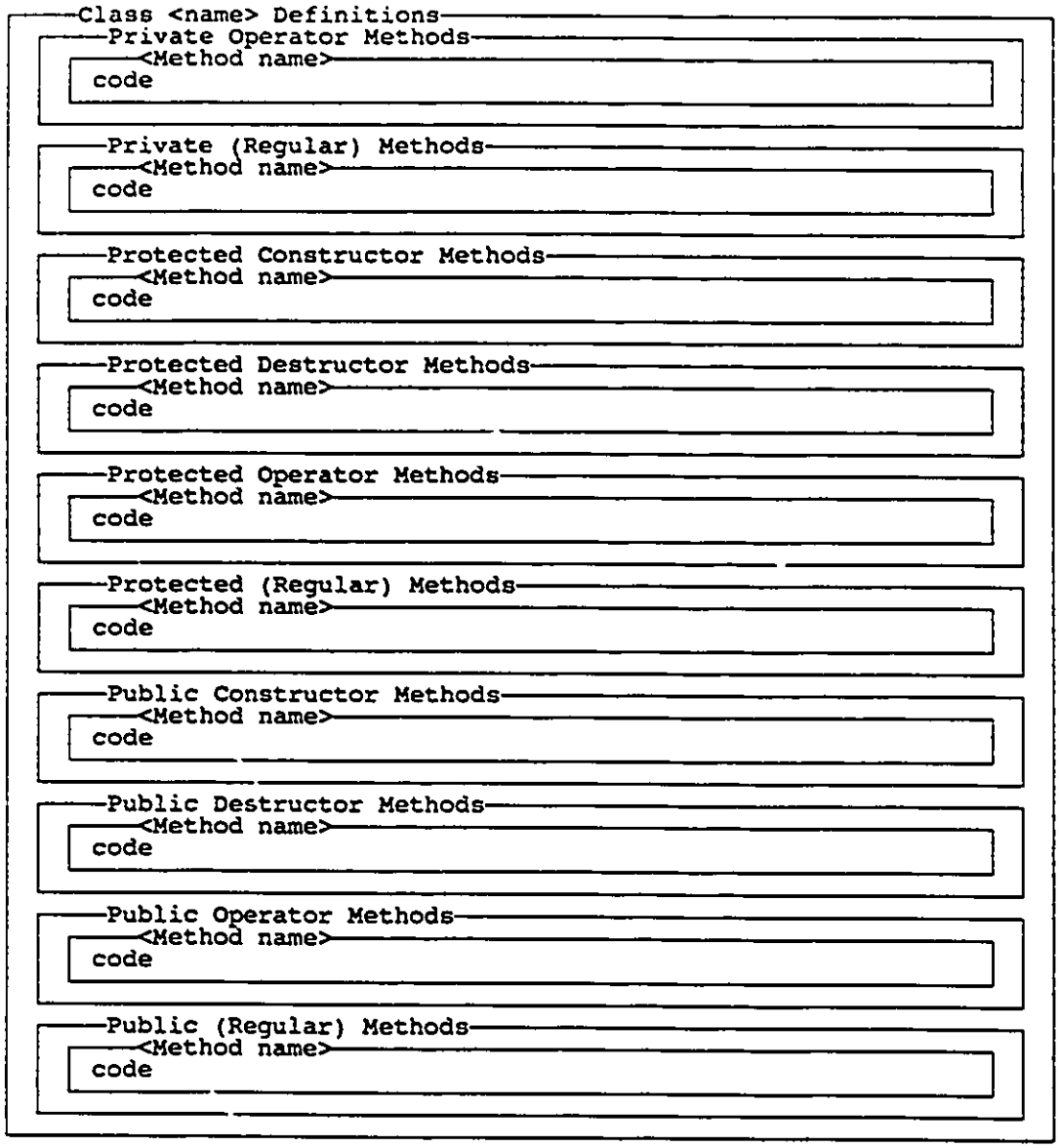


Figure 4.6 Template for Method Defininitons - Class or Struct

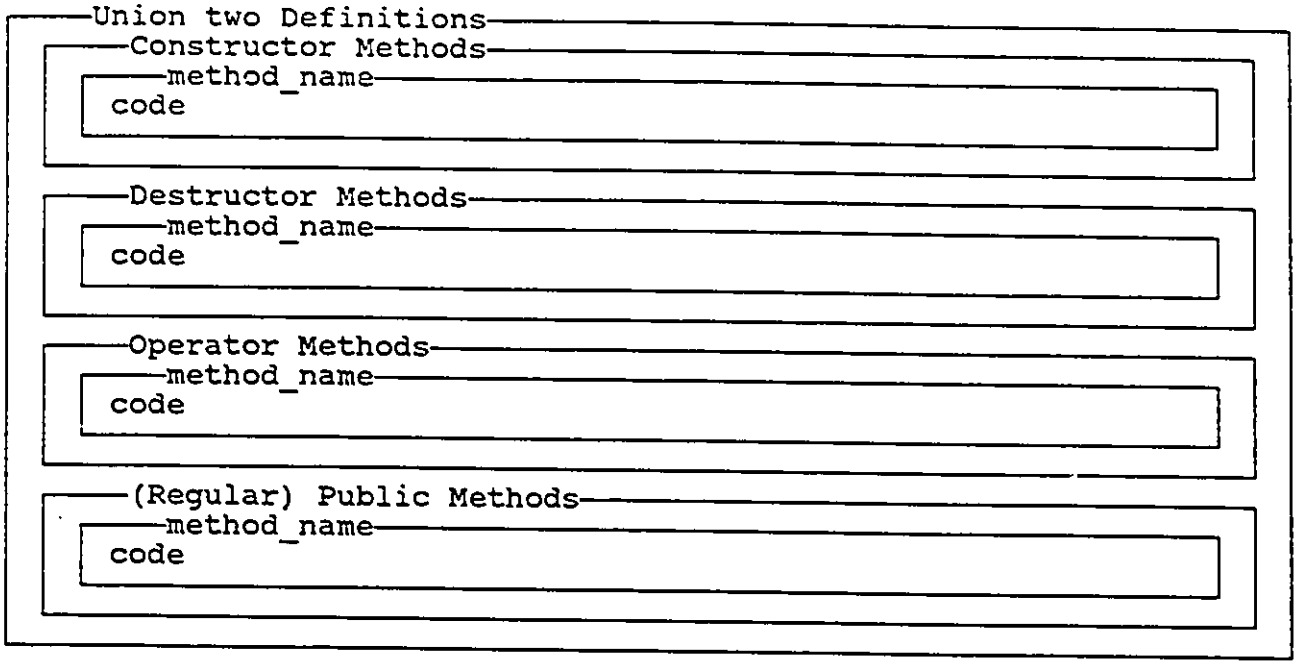


Figure 4.7 Template for Method Definitions - Union

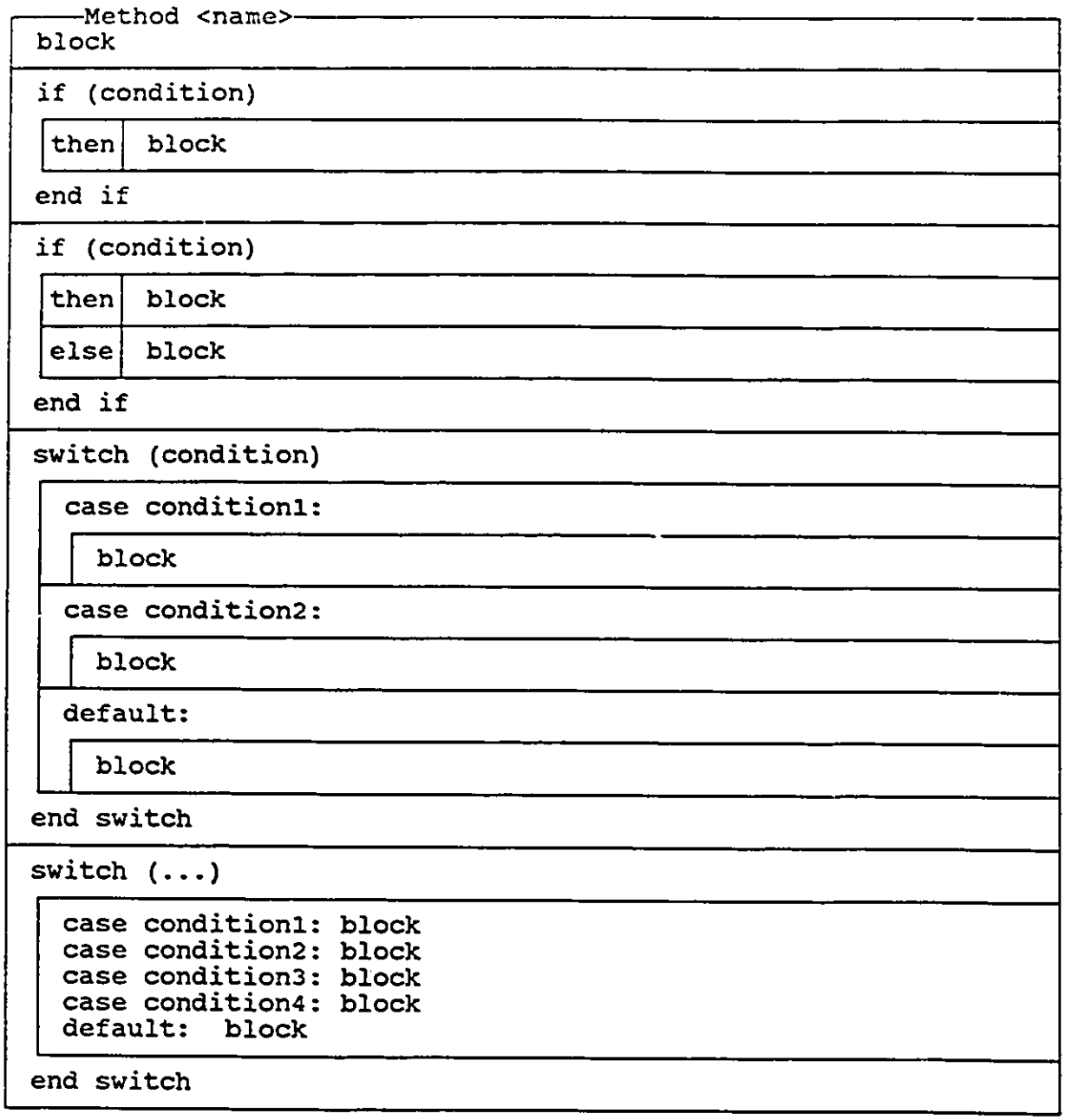


Figure 4.8 Graphic Representation of Sequential and Selection Blocks in Method Definitions

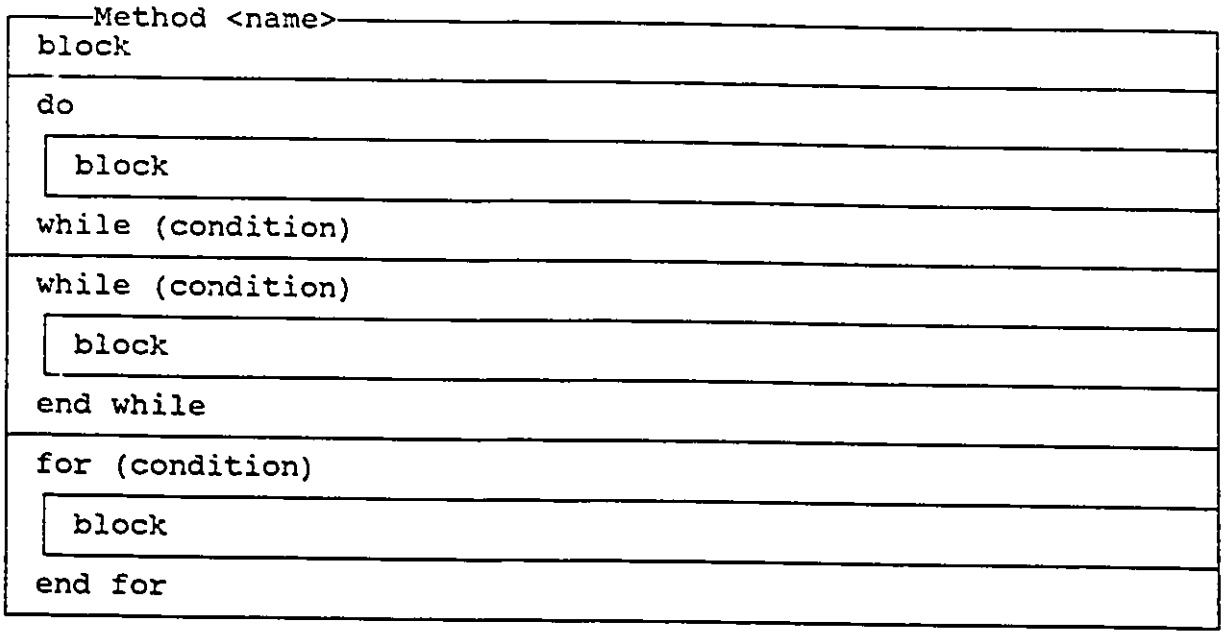


Figure 4.9 Graphic Representation of Sequential and Repetition Blocks in Method Definitions

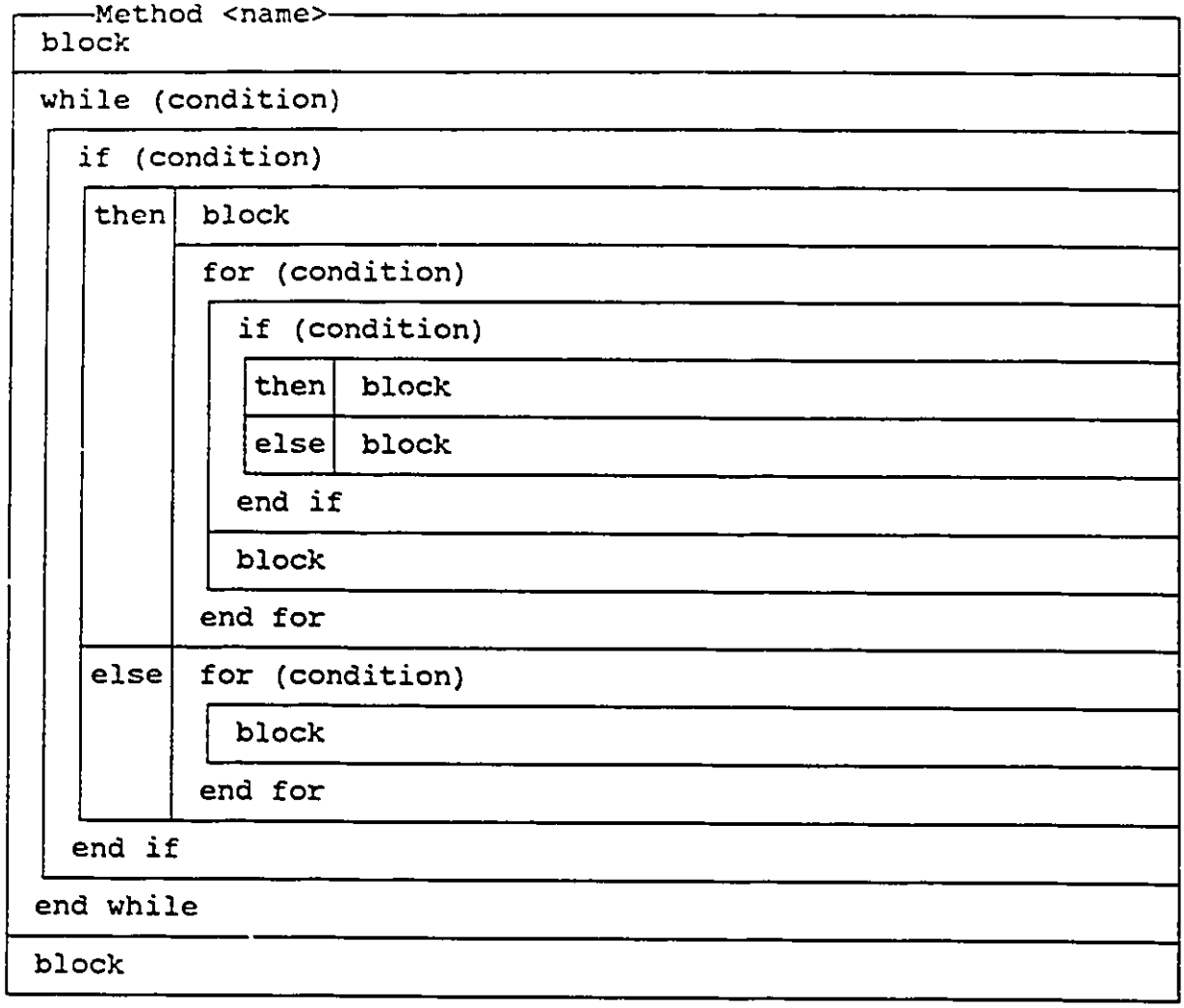


Figure 4.10 Nesting Different Structural Blocks in Method Definitions

Chapter 5

THE N1C++ NATURAL LANGUAGE DESCRIPTION TOOL

A second, optional, component of the DoC++ tool, N1C++, is used to augment the ORC++ description of a C++ program.

The N1C++ tool is composed of four options designed to enhance the box structures used by ORC++. A table of contents with numbered sections and subsections may be produced. The user may choose to use the titles and numbering from the table of contents as labels with the ORC++ output. Introductory comments before each ORC++ section corresponding to the titles and numbering in the table of contents may be chosen to clarify the section coming next.

In addition, N1C++ provides annotation in natural language form in three places in the ORC++ Overview. First, there is a natural language description to follow the Global Overview. Secondly, a N1C++ description of the inheritance in the program follows the Inheritance Overview. Thirdly, the nature of the encapsulation and inheritance for each object type appears in natural language form after the Object Overview for each object type. Users may suppress the N1C++ descriptions.

An example with the N1C++ options appears in Appendix E.

5.1 Table of Contents

If the user chooses, N1C++ will produce a table of contents for the ORC++ output. This involves the use of titles and numbering of sections and sub-sections. If all possible aspects of a C++ program are present then the table of contents would appear as shown Figure 5.1.

The absence of one component in the table results in renumbering.

5.2 Numbering of ORC++ Sections

The user may choose to have the titles and numbering scheme used in the table of contents appear before the appropriate sections of the ORC++ description. The user may choose to have these titles and numbering even if the table of contents is not used.

1.0 Overview

    1.1 Global Overview

    1.2 Inheritance Overview

2.0 Detailed View

    2.1 Objects

        2.11 Classes

            2.111 first class object type name

              ...

        2.12 Structs

            2.121 first struct object type name

              ...

        2.13 Unions

            2.131 first union object type name

              ...

    2.2 Non-member Methods

Figure 5.1 NlC++ Table of Contents

For example, before the Detailed View's Non-member Methods the system will produce the title "2.2 Non-member Methods" before the ORC++ description of the Non-member Methods. It assumed that "2.2" is the correct number for the section. If there are no objects the section number would be "2.1."

5.3 Introductions to ORC++ Sections

After the title and section number and before the ORC++ component, it is possible to produce a general description of the section. This introduction to the section will aid in the documentation process and in aiding the education process with respect to C++ and the DoC++ system.

Introductions are available for the Overview, Global Overview, Inheritance Overview, Detailed View, Detailed View for Objects, Detailed View for Classes, Detailed View for Structs, Detailed View for Unions, and Detailed View for Non-member Methods.

5.4 NlC++ Annotation in ORC++ Overview

Annotation is provided in three areas of the ORC++ Overview: the Global Overview, the Inheritance Overview and for each object in the Detailed View. The purpose of these annotations is to supplement the ORC++ overviews and the annotations follow the appropriate overview. It is expected that the user will read the overview and the annotation together.

The Global Overview annotation aids in the understanding of the program by stating what program components are not present in the program under study. The Global Overview itself has been constructed to show only what is present in the program. The annotation may tell the user, for example, that there are no class object types in the program or how many of each object type are in the program, drawing attention to these important details.

The Inheritance Overview annotation explains the Inheritance Overview's box diagram. For each object type the annotation states the objects above and below the object type in the inheritance hierarchy showing both the names of objects for which the object in question is a base object type and the names of objects that the object in question inherits from. Inheritance may involve many levels. As well as explaining the box diagram, this annotation provides a list of objects related to a given object.

The Detailed View annotation is created for each object type in the program. The annotation is organized in the same way as the Detailed View Members. Data and methods are discussed in sections, such as the Public Section. For each section the accessibility of the data and methods in that section is discussed.

There are two aspects of accessibility that need to be addressed. The first concern is how the data and methods can be accessed. Some data and methods, such as public members, can be accessed from outside the object using the object name followed by the dot operator and then the data or method name. Other data and methods, such as private members, may not be accessed from outside the object, at least not directly. Data and methods may also be accessed from methods that are within the object itself. For example, private data may be accessed by methods in other sections such as the public section, the private section itself, and other sections depending on the members in the object type and inheritance relationships.

The second aspect of accessibility to be considered is what other data and methods the methods in the section in question can and cannot access themselves. For example, the methods in the public section can access various data and methods in other sections of the same object type, such as the private section data and methods and the inherited private section data and methods. Some sections such as the buried section can not be directly accessed. These sections are accessed by going through a method that does have access to the buried section.

Chapter 6

IMPLEMENTATION OF DOC++

The DoC++ tool is currently implemented in the MS-DOS environment. The code for the system is written in C. Porter [Por89] was the main resource for developing the graphical interface and the tree data structures. DoC++ is efficient; the four C++ example programs given in the appendix are processed and produce complete DoC++ output in 6 sec., 12 sec., 20 sec., 20 sec. respectively on an IBM PC XT.

The two modes, batch and interactive, will be discussed followed by a general description of the algorithm and related data structures and some of the knowledge used in the code to implement the tool.

6.1 Batch Mode

In batch mode the user invokes the DoC++ program using the command <docpp> followed by a parameter list which includes the name of input and output files plus options. If no parameters follow the <docpp> command then the program starts the interactive mode. The command line help screen illustrates the options that are possible in batch mode. This help screen is shown in Figure 6.1.

As well as specifying the files to be used the user may choose to see only part of the DoC++ documentation. For example it is possible to see the Inheritance Overview for the program, stack.cpp by giving the batch command, <docpp i=stack.cpp ov:i>.

Combinations of parameters are also allowed. For example, the command, <docpp i=stack.cpp dv:cm opt:a> will show the members and not the method definitions for only all class object types. Each of these class object types will be followed by the annotation.

By default the program will send the output to a file called OUT.OCP and both the complete Overview and the complete Detailed View will be included but none of the N1C++ options. The use of only the <opt:> option results in the overview and detailed view with all of the N1C++ options.

6.2 Interactive Mode

The interactive mode contains all the functionality seen in the batch mode. The use of pop-up menus and the ability to browse through the output facilitates the use of DoC++. The menus

```
docpp   - Interactive Version
docpp ? - Batch Version HELP
docpp  i=inputfile  |  o=outputfile  |
     ov:g|i  |  dv:c|s|u|m|d|n  |  opt:t|i|n|a

ov:    - Complete Overview
ov:g     - Global Overview
ov:i     - Inheritance Overview

dv:    - Complete Detailed View
dv:c     - include Classes
dv:s     - include Structs
dv:u     - include Unions
dv:m     - Members Only
dv:d     - Method Definitions Only
dv:n     - include Non-member Methods

opt:   - All of the following NlC++ options
opt:t    - Table of Contents
opt:i    - Section Introductions
opt:n    - Section Numbering
opt:a    - Annotation
```

Figure 6.1 DoC++ Command Line Help Screen

reflect the general approach used in this report. On the main menu the choices are File, Overview, Detailed View, N1C++ and Quit.

In the File submenu the user may type in input and output file names. As well the user may choose an input file name from a directory and change directories using pop-up menus. Toggles exist in the File submenu to control output to the screen or to the file or both.

The Overview submenu allows the user to choose either the Global Overview or the Inheritance Overview or both. This submenu will produce output on the screen that can be browsed. This allows the user to take a similar view of two programs and make comparisons.

The Detailed View submenu controls the choice of a complete detailed view or part of it, namely seeing only all the class object types, all the struct object types, all the union object types or the non-member methods. For the three object types the user may choose to see just the members or just the method definitions or both. In addition the Detailed View submenu allows the user to choose individual class object types, struct object types or union object types from pop-up menus.

The N1C++ submenu allows the user to do the following: produce a table of contents for the complete DoC++ documentation; toggle on or off the section numbering following the titles and numbering used in the table of contents; toggle on or off the introductions to each section; toggle on or off the annotations that may accompany the Overview and Detailed View. As well the user may choose a complete DoC++ documentation which will include the table of contents followed both the Overview and Detailed View with the three N1C++ toggles on.

6.3 Algorithm and Data Structures

The system reads a C++ program that is to be analyzed from disk and stores it in memory. This is done to speed up processing. Then the program is analyzed. Keywords are found and subsequent steps take the original program and divide it up, storing the various components of the program in a tree data structure. The data structure fits the organization of programs presented earlier in this work.

The tree consists of primary nodes for declarations, class object types, struct object types, union object types and non-member methods. Each of these primary nodes has various branches depending on the node in question. The class object type node allows for creation of a branch when the first class object type is found. Subsequent class object types are further branches. For each class or struct object type there is a node for each of the subsections of a class or struct object type, i.e. private, protected, public and friend sections. The private, protected

and public sections have further nodes for data, constructor methods, destructor methods, operator methods and regular methods. At these nodes the names of the data or methods are branches. Branching off from the names of the method is the definition of the method.

A partial representation of the tree is shown Figure 6.2. Items that have the same level of indentation are connected by means of left branch pointers. Further indentation results in a right branch pointer to the indented item. The organization of the tree reflects the organized representation scheme used in the DoC++ output. The branches at the "destructor_head," "operator_head" and "regular_head" are the same as the branch below the "constructor_head." For each section a branch equivalent to the branch below "private_section" is created but not shown in Figure 6.2 due to space considerations. For each new class object type the tree creates a branch similar to the one shown for "class_1" in Figure 6.2.

Placing the C++ program on this tree data structure facilitates the analysis of the program without wasting memory. Branches of the tree are created only if needed. The formatted output such as overviews or the box structures used to delineate structures such as while loops are not stored in memory. They are generated when requested from the raw information on the tree.

A second tree data structure is used to keep track of inheritance. The names of the objects involved in inheritance and information such as the inheritance attribute are stored at the nodes on this tree. Each node has three pointers: a left pointer, a right pointer and an up pointer. The left and right pointers allow movement through the tree and the objects are put in the tree according to their level of inheritance. A typical inheritance tree may be represented by the following indented chart which may be analyzed.

```
object_1
  object_2
  object_3
    object_4
object_5
  object_6
```

The left pointer of object_1 points to object_5; the right pointer of object_1 points to object_2; the left pointer of object_2 points to object_3; the right pointer of object_3 points to object_4; the right pointer of object_5 points to object_6. Object_1 is the parent of object_2 and object_3. Object_3, in turn, is the parent of object_4. Object_5 is not related to object_1 but is the parent of object_6. By recursively moving through the tree, moving from right pointer to left pointer one encounters all the nodes in the order, object_1 .. object_2 .. object_3 .. object_4 .. object_5 .. object_6. In many cases in the program it is necessary to check every node of the inheritance tree and recursion allows this to be done easily.

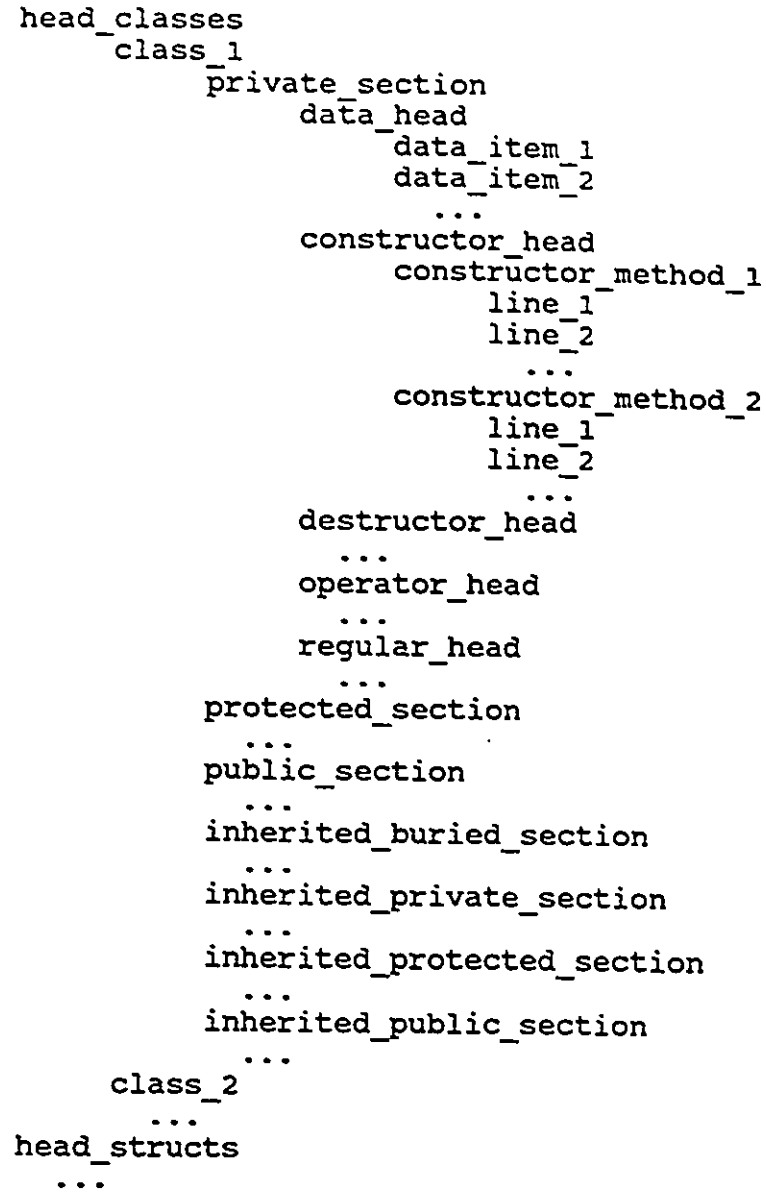


Figure 6.2 Main Data Tree

The need for the up pointer arises since object_1 is the parent of both object_2 and object_3 but object_1 points directly only at object_2 using its right pointer. Consequently, the up pointer is used to point to the parent from each derived object type. The up pointers for both object_2 and object_3 point to object_1; the up pointer for object_4 points to object_3; the up pointer for object_6 points to object_5. This data structure allows the program to generate the Inheritance Overview diagram and the inheritance annotation. To find the derived object types for one node we simple move recursively through the branch that is to the right of it. Moving to the right of object_1 we see a branch containing object_2, object_3 and object_4. These three object types are derived from object_1. Conversely, using the up pointers the program is able to produce a list of object types from which a particular object inherits. For example, following up pointers from object_4 we see object_3 and then object_1. Object_4 is indeed a descendant of object_3 and object_1.

Output to the screen is put into a doubly linked list. This allows the user to scroll up and down through the output.

6.4 DoC++ String Processing and C++ Knowledge

Knowledge about C++ is coded into the DoC++ tool. Much of this knowledge is demonstrated in the string processing of the program being analyzed by the DoC++ tool.

Classification of the elements of this simple C++ program by DoC++ is outlined below in Example 6.1.

```
#include <stdio.h>
class one
{
private:
     int x;
     char y;
public:
     void show_me(void);
};
void one :: show_me (void)
{
     if  ((x == 65)
       || (y == 'A'))
     {
          puts("The condition is met.");
     }
}
```

Example 6.1

DoC++ first reads the file and places the C++ code in memory. Some initial preparsing is done. Leading blanks on all lines are eliminated to make further string searching easier. In addition, the "if" statement and its conditions are put on one line rather

than being split between two lines. DoC++ has routines to allow long lines to spill over to more than one line in the output but needs to have the initial code all on one line.

After the preparsing DoC++ reads through the file looking for strings to trigger actions. The first string recognized in this C++ program is "#include ." The first line is put on the main data structure under the node for file inclusions.

DoC++ continues reading the program and recognizes the string "class." Immediately, a new branch is added to the main data tree to handle the data that is to follow. The string following "class" is processed to look for a class name and inheritance. Inheritance is indicated by the occurence of "::" on this line. In this case there is no inheritance.

Now the information inside the class is processed. DoC++ is looking for the string "};" to end the class. DoC++ encounters the string "private:" instead. This indicates that whatever follows is a private member of the class.

DoC++ looks at the next line and determines that it is not a method declaration since it does not have an argument list. Hence, "int x;" is a data declaration. This information is added to the main data structure under the node for private data for the class, one. The next line is also added to this node.

Then DoC++ encounters the keyword "public:" which instructs DoC++ to add the following line to the public section of the data structure. The line, "void show_me(void)," is analyzed and DoC++ determines that this is a method since it has arguments. DoC++ tries to see if this method is a special type, i.e. a constructor, a destructor, or an operator method, but finds out that it is none of these so it is left with the conclusion that show_me is a public regular method. Then the end of the class is triggered by the "};" string.

After the class declaration has been processed, DoC++ looks at the next line and sees a method definition. The string "::" indicates to DoC++ that this is a method definition that has been previously declared as part of an object type. The C++ code does not indicate what type of object is concerned but DoC++ will parse the name of the object by looking for all the text immediately to the left of the "::" string. The name of the method is determined by DoC++ by taking all the text after the "::" string. Incidently the spaces to the right and left of the "::" string are eliminated. After the name of the object type and the method name are known DoC++ searches the main data structure for a match. When the match is found the subsequent code is added to the data structure. DoC++ keeps track of the number of left and right braces to know when the method is completed. Contents of the methods are simply added to the data structure at this stage. Parsing of keywords, such as "if," occur when output is requested.

For class and struct object types, members (either data or

methods) may be generally classified as Private, Protected, Public or Friend. Each of these categories may have several types of members and there may be more than one member of one type. The keywords, private:, protected: and public: indicate the members that follow (except for members of category Friend) belong to that category. To change the category, a new keyword is inserted before the list of members belonging to the new category. DoC++ searches for the three keyword strings mentioned and adds the components found after these keywords to the correct section of the main data tree.

It is possible to eliminate the keywords and allow default settings to be used. For class object types, the default is private; for struct object types the default is public.

```
struct two
{
     int x;
     char y;
     void show_me(void);
};
```

Example 6.2

Example 6.2 illustrates how all three members can be in the public category since this is the default for struct object types.

```
class three
{
     int x;
     char y;
public:
     void show_me(void);
};
```

Example 6.3

Example 6.3 illustrates the situation where the keyword has been left out at the top of the object type declaration. All the members are private by default until the keyword, public:, is encountered. The member after this keyword is categorized as public.

Friends do not follow this syntax. Each friend member must have the keyword friend immediately preceding the name of each friend member. The friends may be declared in the private, protected or public section of the declaration. This does not make them part of these categories. They are in a category of their own.

In Example 6.4 the method, show_it, is a friend. It is not categorized as private or public.

```
class four
{
private:
     int x;
     friend void show_it(void);
public:
     void show_me(void);
};
```

Example 6.4

The next area of concern is the sub-categorization of the members in each of the categories mentioned above and what string information the DoC++ program uses to do this further categorization. All three categories, private, protected, and public may consist of data and methods (also called functions). The methods may be further divided into Constructor methods, Destructor methods, Operator methods, and Regular methods. DoC++ uses the fact that a constructor method has the same name as the object type to classify constructor method members.

```
class five
{
     int x;
public:
     five(int,int);
};
```

Example 6.5

Example 6.5 shows that there is a method, five, with the same name as the class object type. Hence it is a public constructor method. This particular method has two parameters, both of type integer. Other constructor methods with a different set of parameters is allowed. This is what makes C++ polymorphic.

Destructor methods are indicated by the use of a method name identical to the object type name except for a preceeding tilde (~) symbol. In the above case the destructor would be declared as ~five(void). Note that constructors and destructors do not return values as do other types of methods, for example, int show_me(int) which is a method that has an integer argument and returns an integer. Since constructors and destructors do not return anything, they do not have a type to be returned in front of the method name. DoC++ uses the appearance of the tilde followed immediately by the object type name to determine that a destructor has been encountered.

Operator methods are determined by the text string, "operator" in the method declaration.

In Example 6.6 the text "operator" indicates that the plus symbol is being defined as an operator. DoC++ simply looks for this string and uses it to class this operator method as a public operator method.

```
class six
{
     int x;
public:
     int operator+(int);
};
```

Example 6.6

Methods that do not belong to the above three categories are categorized as Regular methods. This term is not part of the C++ vocabulary but is used in the DoC++ project to classify members that do not belong to the other three recognized categories.

The last sub-category is data. Whatever is not a method is data. DoC++ looks for the methods first. If one of the types of methods is not found then the member is classified as data.

The above string processing and subsequent attachment of C++ program code to the main data structure is a key element of the DoC++ tool.

Chapter 7

CONCLUSIONS & RECOMMENDATIONS FOR FURTHER RESEARCH

This report has shown a graphic scheme and a reverse engineering tool for organizing and documenting C++ programs. The DoC++ scheme is an attempt to improve understanding of encapsulation and inheritance in existing C++ programs. The current implementation of DoC++ may prove beneficial in C++ language training. Later implementations with additional functionalities may support the analysis of large C++ programs. The two parts of the system, ORC++ and N1C++, are complementary.

The DoC++ pilot project has achieved its objectives as it produces C++ software documentation as planned. Given a C++ program, DoC++ makes it easier to see the components of the program and understand the inheritance and encapsulation properties of the program. The examples in the appendices demonstrate that it is much easier to analyze a C++ program with the DoC++ tool. The complexity of understanding a C++ program is demonstrated in the program listing in Example 4 in Appendix D. The inheritance and encapsulation relationships are complicated even in this relatively short C++ program. This explains why the DoC++ tool is needed.

This work has extended the efforts to produce automatic documentation software to the C++ language. As well as formatting the code of C++ methods using a box structures, the tool produces alternate views of C++ programs including overviews of a program in general and inheritance and encapsulation in particular.

The DoC++ tool may be viewed as a reverse engineering CASE tool supporting object-oriented software development. Reverse engineering is attained through the extraction of design information such as the number of object types, the inheritance and encapsulation properties of the object types and the formatting of the code. Other aspects of the design can not be generated through this method and should be produced by other means. An advantage of an automatic documentation method such as DoC++ is the ability to produce new documentation automatically after a programming change has been made. This fits in with Booch's view of incremental software development.

Re-use of software modules is an aim of DoC++. It should be used within the context of an overall object-oriented software development method such as that proposed by Booch [Boo91].

Enhancements to DoC++ could include the production of the N1C++ portion in natural languages other than English. As well the view of the code in the method definitions could be presented

at a higher level with only the box structures without detailed code showing.

The objective of the project was to create a working version of the program and scaling up the project was not part of the original plan. More significant changes would be needed to take this pilot project to a viable commercial product. The scaled-up implementation could be done in a workstation environment, such as the Sun, with the use of appropriate development tools. The present implementation was done on a PC without the use of software development tools other than a C compiler and a limited library of functions. A more powerful implementation would not be limited by the machine's memory as is the case with the present implementation. Changes would have to be made so that the improved implementation would handle larger programs with many modules stored in separate files. There would be a need for greater use of disk storage to avoid running out of memory. Also modifications would be needed to produce documentation output that would be appropriate for larger programs. For example, it may be useful to know in which module an object is physically located.

DoC++ is a static analyzer. Future work could involve augmenting this static analysis with dynamic analysis. An analysis of message passing between objects may be a useful component of such a system.

Plans for future work involve taking the ORC++ graphic representation scheme and using it in the top-down design of C++ programs. Ören states that two steps are planned [Ören91]. The first is called ADC++ (Algorithm Design for C++) and is a tool for designing and maintaining systems and algorithms compatible with the C++ language. The second step is APC++ (Algorithm-directed Programming in C++). This will be a syntax-directed programming tool that will take an algorithm created with ADC++ and help the programmer produce a syntactically correct C++ program.

Chapter 8

REFERENCES

[Arn90] Arnistead, M. and Burnham, J. "HP C++/Softbench: a development environment for C++." Journal of Object-oriented Programming (JOOP), Vol.3, No.4, November/December, 1990, pp.42-60.

[Ayt84] Aytac, K. and Ören, T.I. (1984). "ORPAS - Organized Representation of Pascal Programs." Technical Report TR-83.01, Computer Science Department, University of Ottawa, Canada.

[Ber88] Berry, J. (1988). The Waite Group's C++ Programming. Indianapolis,IA: Howard W. Sams & Company.

[Boo91] Booch, G. (1991). Object Oriented Design: with Applications. New York: Benjamin-Cummings.

[Byte91a] Advertisement. BYTE, August, 1991, p.117.

[Byte91b] Advertisement. BYTE, August, 1991, p.229.

[Coad90] Coad, P. and Yourdon, E. (1989). Object-Oriented Analysis. Englewood Cliffs, NJ: Prentice-Hall.

[Chi90] Chikofsky, E. and Cross, J. "Reverse Engineering and Design Recovery: A Taxonomy." IEEE Software, January, 1990, pp.13-17.

[Cox90] Cox, B. "There is a Silver Bullet." BYTE, October, 1990, pp.209-218.

[DeM78] DeMarco, T. (1978). Structured Analysis and System Specification. Englewood Cliffs, N.J.: Prentice-Hall.

[JOOP91a] Product News. JOOP, Vol.3, No.6, February, 1991, p.65.

[JOOP91b] Product News. JOOP, Vol.4, No.2, May, 1991, p.73.

[JOOP91c] Product News. JOOP, Vol.4, No.2, May, 1991, p.74.

[JOOP91d] Product News. JOOP, Vol.4, No.3, June, 1991, p.78.

[JOOP91e] Product News. JOOP, Vol.4, No.7, Nov./Dec., 1991, p.68.

[JOOP92] Product News. JOOP, Vol.4, No.8, Jan., 1992, p.66.

[Lea91] Leavens, G. "Introduction to the Literature on Object-Oriented Design, Programming and Languages." OOPS Messenger, Vol.2, No.4, October, 1991.

[Nil90] Nilsson, E. (1990). "CASE Tools and Software Factories." Lecture Notes in Computer Science. Goos, G. and Hartmanis, J. (ed.) Proceedings of the Second Nordic Conference CAiSE '90, Stockholm, Sweden. Berlin: Springer-Verlag. pp.42-60.

[Oman90] Oman, P. "CASE Analysis and Design Tools." IEEE Software, May, 1990, pp. 37-41.

[Ören84] Ören, T.I. (1984). "Graphical Representation of Pseudocodes and Computer Programs: A Unifying Technique and a Family of Documentation Programs." Proceedings of the Educational Computing Conference of IEEE Computer Society, D.C. Rine (ed.). San Jose, CA, Oct. 18-20, 1983. New York: IEEE Computer Society. pp. 81-89.

[Ören91] Ören, T.I., Hamilton, I.J. and Ören, Y.C. (1991). "ORC++: A Graphic Scheme and a Redocumentation Tool for C++." Submitted.

[Por89] Porter, K. (1989). Stretching Turbo C. New York: Brady Books.

[Tho90] Thomas, D. and Lalonde W. (1990). in Advances in Artificial Intelligence in Software Engineering. Ören, T.I. (ed.) Greenwich, CN: JAI Press Inc. pp.57-106.

[Ward90] Ward, P. (1990). "The Use of Current Generation CASE Tools in Object-oriented Analysis and Design." CASE on Trial. Spurr, K. and Layzell, P. (ed.) Chichester, England: John Wiley & Sons. pp.105-123.

[Wie88] Wiener, R. and Pinson, L. (1988). An Introduction to Object-Oriented Programming and C++. Reading, MA: Addison-Wesley.

[Wie91] Wiener, R. "C++/Views for Microsoft Windows." JOOP, Vol.4, No.2, May, 1991, pp.71-73.

APPENDICES

LIST OF FIGURES FOR APPENDICES

Fig. A.1 Program Listing - Example 1 60
Fig. A.2 Overview - Example 1 61
Fig. A.3 Detailed View - Example 1 62

Fig. B.1 Program Listing - Example 2 65
Fig. B.2 Overview - Example 2 66
Fig. B.3 Detailed View - Example 2 68

Fig. C.1 Program Listing - Example 3 71
Fig. C.2 Overview - Example 3 73
Fig. C.3 Detailed View - Example 3 74

Fig. D.1 Program Listing - Example 4 81
Fig. D.2 Overview - Example 4 83
Fig. D.3 Detailed View - Example 4 84

Fig. E.1 Table of Contents - Example 4 90
Fig. E.2 Overview with N1C++ - Example 4 91
Fig. E.3 Detailed View with N1C++ - Example 4 95

Appendix A

EXAMPLE WITH A CLASS OBJECT TYPE

The program shown in Figure A.1 is adapted from Wiener [Wie88] (pp. 71-72). This example illustrates the use of a C++ class object type, stack, to provide data encapsulation and data abstraction. The object type, stack, contains the data structure necessary to store whatever needs to be put on to the stack. In this particular example, the stack is implemented by means of an array of integers. This implementation is encapsulated and a user of this class need not know the details of the implementation. The stack object type allows access to the data structure only by means of calls to public methods, push and pop, and hence data abstraction is attained.

The Overview, shown in Figure A.2, is produced by the DoC++ program and consists of two "sub-overviews." Although the user can specify only one part of the Overview to be viewed, Figure A.2 consists of the Global Overview and the Inheritance Overview.

From the Global Overview it is clear that the program consists of, at the level of least detail, some file inclusions, one class object type, and two non-member methods.

There is no inheritance in this program and the Inheritance Overview simply indicates that there is no inheritance in this program.

Figure A.3 demonstrates the Detailed View for this program. This consists of the Members for the only object type, class stack, and the method definitions for all the methods in this object type as well as the method definitions for the two methods that are not members of any object type. We find that class stack has some Private Data as well as some Public Methods - one Constructor, one Destructor, and two Regular Methods. Also we are told that all the methods are implemented. A consistent order is used to describe the Members and the Method Definitions despite the order in which the original code is written. In the program listing (Figure A.1) the destructor method, ~stack, appears after the two Regular Methods. The logical connection between constructors and destructors supports the proximity of these two components of the object type in the Detailed View. The organization also groups similar items such as the definitions for the two non-member methods, reverse_name and main.

Within the Method Definitions for the object type, stack, the types of methods are in the order used with the Members. The code for each method appears in the definition and this code is presented using box structures to enhance the understandability of the code. In Figure A.3 the selection structure used in the method stack::push has a clearly shown condition and it is

obvious which code is executed if the condition is true since the end of the selection structure is demarcated with "end if" and the appropriate box structure.

```
#include <stdio.h>
#include <string.h>
class stack
{
    int *top;
    int *bottom;
  public:
    stack();
    void push(int);
    int pop();
    ~stack();
};
stack::stack()
{
  top = bottom = new int[100];
}
void stack::push(int c)
{
  if ((top - bottom) < 100)
    *top++ = c;
}
int stack::pop()
{
  if (--top >= bottom)
    return *top;
}
stack::~stack()
{
  delete bottom;
}
char* reverse_name(char *name)
{
  stack s;
  char *reverse;
  for (int i = 0; i < strlen(name); i++)
    s.push(name[i]);
  reverse = new char[ strlen(name) + 1];
  for (i = 0; i < strlen(name); i++)
    reverse[i] = s.pop();
  reverse[strlen(name)] = '\0';
  return reverse;
}
main()
{
  char your_name[20];
  char name_backwards[20];
  printf("\n Enter your name: ");
  scanf("%s",your_name);
  printf("\n Your name backwards is %s",
            reverse_name(your_name));
}
```

Figure A.1 Program Listing - Example 1

DoC++ 1991 Aug 24 Time 07:18

====== Overview of program: example1 ==========

Global Overview of program:	example1

		Number	Names
Initial	Includes	2	#include <stdio.h> #include <string.h>
Classes		1	stack
Non-member methods		2	char* reverse_name(char *name) main()

No Inheritance is declared in this program.

====== End of Overview of program: example1 ==========

Figure A.2 Overview - Example 1

====== Detailed View of program: example1 ==========

Members of CLASS	stack
of the Program:	example1

Private	Data		int *top int *bottom
Public	Methods	Constructor	stack()
		Destructor	~stack()
		(Regular)	void push(int) int pop()

All method definitions are implemented

Class stack Definitions

Public Constructor Methods

stack::stack()

```
top = bottom = new int[100];
```

Public Destructor Methods

stack::~stack()

```
delete bottom;
```

Public (Regular) Methods

void stack::push(int c)

```
if ((top - bottom) < 100)
    then | *top++ = c;
end if
```

int stack::pop()

```
if (--top >= bottom)
    then | return *top;
end if
```

Figure A.3 Detailed View - Example 1.

```
Non-Member Methods Definitions
  char* reverse_name(char *name)
stack s;
char *reverse;

for (int i = 0; i < strlen(name); i++)
    s.push(name[i]);
end for

reverse = new char[ strlen(name) + 1];

for (i = 0; i < strlen(name); i++)
    reverse[i] = s.pop();
end for

reverse[strlen(name)] = '\0';
return reverse;

  main()
char your_name[20];
char name_backwards[20];
printf("\n Enter your name: ");
scanf("%s",your_name);
printf("\n Your name backwards is %s", reverse_name(your_name))
    ;
```

====== End of Detailed View of program: example1 ====================

Figure A.3 (cont'd) Detailed View - Example 1

Appendix B

EXAMPLE WITH A STRUCT OBJECT TYPE

In figure B.1 we see a C++ program that demonstrates the use of a struct object type called data. This program looks much like a C program with a few additions. The description of the components of the program that follows indicates that there is much knowledge that is read into a C++ program that could be done automatically by the DoC++ system.

In the struct object type data there is the usual data structure consisting of a string, name and a pointer to another struct of the same type, next. As well the struct object type data contains three functions or methods that work with the data in the struct. In this example the struct type contains only public members which at one time struct object types were only capable of having.

One of these methods, data(char*), is called a constructor and is identified by having the same name as the struct data itself. The purpose of the constructor is to automatically do what is necessary when an instance of the struct comes into existence. In this case the string, inname, which is the argument to the function data(char* inname) is copied into the string, name, which is in the struct data itself. Also the pointer to the next node in the linked list is set pointing to NULL. The other methods are so-called "regular" methods, since they do not fit into the categories of constructor, destructor or operator methods.

Note that the struct is generally used as an area to declare the data and methods to be used in the struct. The methods declared in this area have also to be defined. Short method definitions may be included in the declaration area but it is more usual and methodical to provide separate definitions for each method. In this program there are three such method definitions as well as the main() method definition.

Much effort is expended in even a simple program such as this to make connections between the declarations of methods and their definitions. Even the identification of various method types and data may not be obvious.

Figure B.2 shows the Overview for the program in Figure B.1. The first part of this overview, the Global Overview indicates what is generally in the program, i.e. it contains some preliminary file inclusions, one struct called data, and one non-member method called main(). There is no inheritance in this program as indicated.

```
#include <stdio.h>
#include <string.h>
struct data
{
  char name[80];
  data* next;
  data(char*);
  void data_show(void);
  void add_to_list(char*);
};
data::data(char* inname)
{
   strcpy(name, inname);
   next = NULL;
}
void data::data_show(void)
{
  data* current;
  current = this;
  while (current->next != NULL)
  {
    puts(current->name);
    current = current->next;
  }
  puts(current->name);
}
void data::add_to_list(char* inname)
{
  data* current;
  current = this;
  while (current->next != NULL)
  {
    current = current->next;
  }
  current->next = new data(inname);
  current->next->next = NULL;
}
main()
{
  data obj("Alpha");
  puts(obj.name);
  data obj2("Beta");
  puts(obj2.name);
  obj.data_show();
  obj2.data_show();
  obj2.add_to_list("Gamma");
  obj2.add_to_list("Delta");
  obj2.add_to_list("Epsilon");
  obj2.data_show();
  strcpy(obj2.name,"Alpha");
  obj2.data_show();
}
```

Figure B.1 Program Listing - Example 2

DoC++ 1991 Aug 24 Time 07:23

====== Overview of program: example2 ==========

Global Overview of program:	example2

		Number	Names
Initial	Includes	2	#include <stdio.h> #include <string.h>
Structs		1	data
Non-member methods		1	main()

No Inheritance is declared in this program.

====== End of Overview of program: example2 ==========

Figure B.2 Overview - Example 2

In addition to the Overview a more detailed representation scheme, the Detailed View is produced. It is intended to show the Members for each object as well as delimiting the control structures in Method Definitions. Figure B.3 shows the DoC++ Detailed View for the program in Figure B.1. Note that the methods data_show and add_to_list are both grouped under the subtitle, (Regular) Public Methods. This helps the reader to understand the nature of the program more quickly. Also the constructor method is identified with a label. The reader does not have to look for a method with the same name as the object. This can be confusing for program readers since the name of the object, data is used as a prefix for all the method definitions and in the pointer. It takes much effort to read a program. Why not let the computer aid in this endeavour? This methodology provides automatic documentation.

Note the use of block structure for the while loops in the definitions for the methods data_show and add_to_list. This clarifies which code is controlled by the while condition. In the program listing (Figure B.1) the indenting makes this clear but only because the program was properly indented.

====== Detailed View of program: example2 ==========

Members of STRUCT	data
of the Program:	example2

Public	Data		char name[80] data* next
	Methods	Constructor	data(char*)
		(Regular)	void data_show(void) void add_to_list(char*)

All method definitions are implemented

Struct data Definitions

Public Constructor Methods

data::data(char* inname)

```
strcpy(name, inname);
next = NULL;
```

Public (Regular) Methods

void data::data_show(void)

```
data* current;
current = this;
while (current->next != NULL)
    puts(current->name);
    current = current->next;
end while
puts(current->name);
```

void data::add_to_list(char* inname)

```
data* current;
current = this;
while (current->next != NULL)
    current = current->next;
end while
current->next = new data(inname);
current->next->next = NULL;
```

Figure B.3 Detailed View - Example 2

```
-----Non-Member Methods Definitions-----------------------------------
   -----main()-----------------------------------------------------
   data obj("Alpha");
   puts(obj.name);
   data obj2("Beta");
   puts(obj2.name);
   obj.data_show();
   obj2.data_show();
   obj2.add_to_list("Gamma");
   obj2.add_to_list("Delta");
   obj2.add_to_list("Epsilon");
   obj2.data_show();
   strcpy(obj2.name,"Alpha");
   obj2.data_show();
```

====== End of Detailed View of program: example2 ==========

Figure B.3 (cont'd) Detailed View - Example 2

Appendix C

EXAMPLE WITH SINGLE INHERITANCE

The program shown in Figure C.1 demonstrates inheritance in C++. Figure C.2 consists of the Global Overview, which provides a quick overall view of the program's components, and the Inheritance Overview.

The Global Overview indicates that there is one class object type and there are three struct object types.

The Inheritance Overview (indicates that there are three "generations" involved in the inheritance. It is clear from the boxes in the Inheritance Overview that struct object types, data3 and data4 inherit from class object type, data2 which in turn inherits from struct object type data. The diagram shows that both data3 and data4 inherit indirectly from the struct object type, data as well.

It is important in this example to understand the types of members involved in the four object types and their interrelationships. The Detailed View shown in Figure C.3 provides us with the required information.

The diagram for Members for each object type shows that derived object types contain more data than what first meets the eye. More importantly the classification or ease-of-access must be known. This follows the rules stated in section 3.2 of this work. We see class object type, data2 really has inherited public data and methods that are inherited from the struct object type, data, in addition to the data and methods declared in data2 itself.

An analysis of the Members for the struct object type, data3 indicates that it has inherited buried, inherited private, and public members. The inherited members come from class object type, data2 and from struct object type, data.

As data or methods are inherited they may change their classification going from being public, i.e. easily accessible, at one extreme to buried, i.e. not directly accessible at the other extreme.

In the Members for object type data3, we see that data3 picked up some Inherited Buried Data from data2. The member in question is data2::char extra[8], a structure allowing the storage of an 80-character string. This was Private Data in class object type, data2. It becomes more encapsulated in struct object type, data3. In addition we see that data3 picked up some

```
#include <stdio.h>
#include <string.h>
struct data
{
  char name[80];
  data(char*);
  void data_show(void);
};
class data2:data
{
  char extra[80];
public:
  data2(char*,char*);
  void data_show2(void);
};
struct data3:data2
{
  char more[80];
  data3(char*,char*,char*);
  void data_show3(void);
};
struct data4:public data2
{
  char more2[80];
  data4(char*,char*,char*);
  void data_show4(void);
};
data::data(char* inname)
{
   strcpy(name, inname);
}
data2::data2(char* nam,char* nam2):(nam)
{
  strcpy(extra, nam2);
}
data3::data3(char* nam,char* nam2,
             char* nam3):(nam,nam2)
{
  strcpy(more, nam3);
}
```

Figure C.1 Program Listing - Example 3

```
data4::data4(char* nam,char* nam2,
             char* nam4):(nam,nam2)
{
  strcpy(more2, nam4);
}
void data::data_show(void)
{
  puts(name);
}
void data2::data_show2(void)
{
  data_show(); puts(extra);
}
void data3::data_show3(void)
{
  data_show2(); puts(more);
  puts(name);
}
void data4::data_show4(void)
{
  data_show2(); puts(more2);
  puts(name);
}
main()
{
  data obj("Alpha");
  data2 obj2("Beta","Gamma");
  data3 obj3("Kappa","Lambda","Mu");
  data4 obj4("1","2","3");
  obj.data_show();
  obj2.data_show2();
  obj3.data_show3();
  obj4.data_show4();
  puts(obj4.name);
}
```

Figure C.1 (cont'd) Program Listing - Example 3

====== Overview of program: example3 ==========

Global Overview of program:	example3

		Number	Names
Initial	Includes	2	#include <stdio.h> #include <string.h>
Classes		1	data2
Structs		3	data data3 data4
Non-member methods		1	main()

Inheritance Overview of program:	example3

struct data

class data2:data

struct data3:data2

struct data4:public data2

====== End of Overview of program: example3 ==========

Figure C.2 Overview - Example 3

====== Detailed View of program: example3 ==========

Members of CLASS	data2
of the Program:	example3

Private	Data		char extra[80]
Public	Methods	Constructor	data2(char*,char*)
		(Regular)	void data_show2(void)
Inherited Public	Data		from struct data::char name[80]
	Methods	Constructor	struct data::data(char*)
		(Regular)	struct data::void data_ show(void)

All method definitions are implemented

```
Class data2 Definitions
  Public Constructor Methods
    data2::data2(char* nam,char* nam2):(nam)
    strcpy(extra, nam2);
  Public (Regular) Methods
    void data2::data_show2(void)
    data_show();
    puts(extra);
```

Figure C.3 Detailed View - Example 3

Members of STRUCT	data
of the Program:	example3

Public	Data		char name[80]
	Methods	Constructor	data(char*)
		(Regular)	void data_show(void)

All method definitions are implemented

```
Struct data Definitions
  Public Constructor Methods
    data::data(char* inname)
      strcpy(name, inname);
  Public (Regular) Methods
    void data::data_show(void)
      puts(name);
```

Figure C.3 (cont'd) Detailed View - Example 3

Members of STRUCT	data3
of the Program:	example3

Inherited Buried	Data		from class data2::char extra[80]
Inherited Private	Data		from struct data::char name[80]
	Methods	Constructor	class data2::data2(char*,char*) struct data::data(char*)
		(Regular)	class data2::void data_ show2(void) struct data::void data_ show(void)
Public	Data		char more[80]
	Methods	Constructor	data3(char*,char*,char*)
		(Regular)	void data_show3(void)

All method definitions are implemented

```
Struct data3 Definitions
  Public Constructor Methods
    data3::data3(char* nam,char* nam2,char* nam3):(nam,nam2)
    strcpy(more, nam3);

  Public (Regular) Methods
    void data3::data_show3(void)
    data_show2();
    puts(more);
    puts(name);
```

Figure C.3 (cont'd) Detailed View - Example 3

Members of STRUCT	data4
of the Program:	example3

Inherited Buried	Data		from class data2::char extra[80]
Public	Data		char more2[80]
	Methods	Constructor	data4(char*,char*,char*)
		(Regular)	void data_show4(void)
Inherited Public	Data		from struct data::char name[80]
	Methods	Constructor	class data2::data2(char*,char*) struct data::data(char*)
		(Regular)	class data2::void data_ show2(void) struct data::void data_ show(void)

All method definitions are implemented

```
Struct data4 Definitions
    Public Constructor Methods
        data4::data4(char* nam,char* nam2,char* nam4):(nam,nam2)
        strcpy(more2, nam4);

    Public (Regular) Methods
        void data4::data_show4(void)
        data_show2();
        puts(more2);
        puts(name);
```

Figure C.3 (cont'd) Detailed View - Example 3

```
Non-Member Methods Definitions
    main()
    data obj("Alpha");
    data2 obj2("Beta","Gamma");
    data3 obj3("Kappa","Lambda","Mu");
    data4 obj4("1","2","3");
    obj.data_show();
    obj2.data_show2();
    obj3.data_show3();
    obj4.data_show4();
    puts(obj4.name);
```

====== End of Detailed View of program: example3 ======================

Figure C.3 (cont'd) Detailed View - Example 3

Inherited Private Data and Methods from both struct object type, data and class object type, data2. The members inherited from data2 were Public in data2 but become Inherited Private in data3. The members inherited indirectly by data3 from struct object type, data are also Inherited Private because they were in the Public section of a struct object type that was inherited by a class object type, data2 which classified them as Inherited Public in data2. When this in turn was inherited by data3, since it was being inherited from a class object type, the level of encapsulation increased and it was placed in the Inherited Private section of data3. This once again follows the rules for inheritance stated in section 3.2. The automatic processing of these complex rules is helpful to users who may be unsure of the inheritance relation within a program.

A look at the Members for the struct object type, data4 indicates that it inherits somewhat differently. The use of the keyword, "public" in the declaration for this object type, i.e. "struct data4::public data2" causes the public members inherited from class object type, data2 and struct object type, data to be placed in the Inherited Public section of data4.

Figure C.3, also contains the Method Definitions for each object type as well as the method definition for the non-member method, main.

Appendix D

EXAMPLE WITH MULTIPLE INHERITANCE

Current versions of C++ allow for multiple inheritance. An object type may inherit from more than one base. This is demonstrated in the following program listing shown in Figure D.1. The DoC++ tool offers various views of this program. Figure D.2 provides us with the Overview. The Global Overview gives us the names of the object types. The Inheritance Overview shows the inheritance hierarchy involved. Note that the struct type, derived2 inherits from both class type, derived1 and struct type, base2. A more complete inheritance description is given in the Members section of the Detailed View shown in Figure D.3. This indicates in detail how various components of base object types are inherited by derived object types.

In the program listing the following information is given about the struct, derived2.

```
struct derived2:private derived1,base2
{
  char name[80];
  derived2(void);
  void data_show(void);
};
```

As will be shown below, there is much more to be known about derived2 and its relationship with the other object types in the program.

It is clear from the Inheritance Overview that derived2 inherits from derived1 and from base2. This is multiple inheritance. In addition, derived2 inherits from base1 through derived1. This is a case of multi-generation or cascaded inheritance and is not to be confused with multiple inheritance. Note that derived2 appears inside base1's box and inside base2's box. This is how DoC++ indicates that multiple inheritance is involved.

The Members section of the Detailed View shows how the inherited members of each object type are classified or encapsulated. The Detailed View for the struct object type, derived2 mentioned above shows twelve members rather than the three that are declared. The other nine members are inherited. This representation of the derived2 indicates inheritance from derived1, base1, and base2.

```
#include <stdio.h>
#include <string.h>
struct base1
{
  char name[80];
  base1(void);
  void data_show(void);
};
struct base2
{
  char name[80];
  base2(void);
  void data_show(void);
};
class derived1:base1
{
  char name[80];
public:
  derived1(void);
  void data_show(void);
};
struct derived2:private derived1,base2
{
  char name[80];
  derived2(void);
  void data_show(void);
};
struct derived3:public derived1
{
  char name[80];
  derived3(void);
  void data_show(void);
};
base1::base1(void)
{
  strcpy(name, "base1");
}
base2::base2(void)
{
  strcpy(name, "base2");
}
derived1::derived1(void)
{
  strcpy(name, "derived1");
}
derived2::derived2(void)
{
  strcpy(name, "derived2");
}
```

Figure D.1 Program Listing - Example 4

```
derived3::derived3(void)
{
  strcpy(name, "derived3");
}
void base1::data_show(void)
{
  puts(name);
}
void base2::data_show(void)
{
  puts(name);
}
void derived1::data_show(void)
{
  base1::data_show();
  puts(name);
}
void derived2::data_show(void)
{
  derived1::data_show();
  base2::data_show();
  puts(name);
}
void derived3::data_show(void)
{
  derived1::data_show();
  puts(name);
}
main()
{
  base1 obj1;
  base2 obj2;
  derived1 obj3;
  derived2 obj4;
  derived3 obj5;
  obj1.data_show();
  puts("----");
  obj2.data_show();
  puts("----");
  obj3.data_show();
  puts("----");
  obj4.data_show();
  puts("----");
  obj5.data_show();
}
```

Figure D.1 (cont'd) Program Listing - Example 4

====== Overview of program: example4 ==========

Global Overview of program:	example4

		Number	Names
Initial	Includes	2	#include <stdio.h> #include <string.h>
Classes		1	derived1
Structs		4	base1 base2 derived2 derived3
Non-member methods		1	main()

Inheritance Overview of program:	example4

struct base1

class derived1:base1

struct derived2:private derived1,base2

struct derived3:public derived1

struct base2

struct derived2:private derived1,base2

====== End of Overview of program: example4 ==========

Figure D.2 Overview - Example 4

====== Detailed View of program: example4 ==========

Members of CLASS	derived1
of the Program:	example4

Private	Data		char name[80]
Public	Methods	Constructor	derived1(void)
		(Regular)	void data_show(void)
Inherited Public	Data		from struct base1::char name[80]
	Methods	Constructor	struct base1::base1(void)
		(Regular)	struct base1::void data_ show(void)

All method definitions are implemented

```
Class derived1 Definitions
  Public Constructor Methods
    derived1::derived1(void)
      strcpy(name, "derived1");
  Public (Regular) Methods
    void derived1::data_show(void)
      base1::data_show();
      puts(name);
```

Figure D.3 Detailed View - Example 4

Members of STRUCT	base1
of the Program:	example4

Public	Data		char name[80]
	Methods	Constructor	base1(void)
		(Regular)	void data_show(void)

All method definitions are implemented

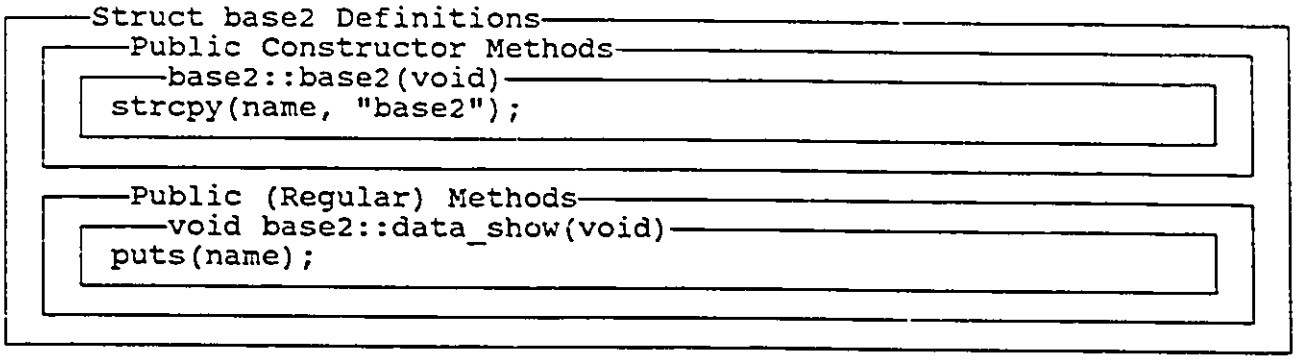
```
Struct base1 Definitions
    Public Constructor Methods
        base1::base1(void)
        strcpy(name, "base1");
    Public (Regular) Methods
        void base1::data_show(void)
        puts(name);
```

Members of STRUCT	base2
of the Program:	example4

Public	Data		char name[80]
	Methods	Constructor	base2(void)
		(Regular)	void data_show(void)

All method definitions are implemented

Figure D.3 (cont'd) Detailed View - Example 4

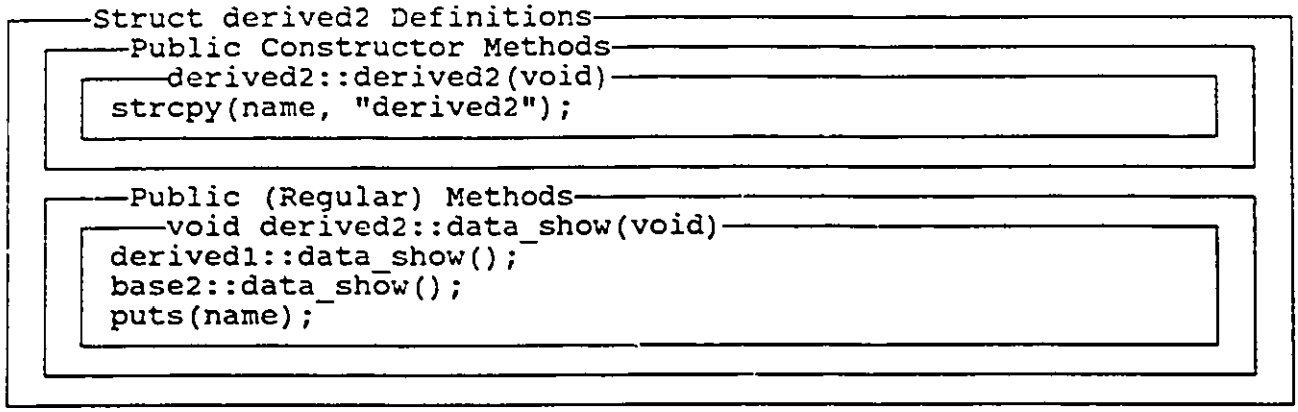


Members of STRUCT	derived2
of the Program:	example4

Inherited Buried	Data		from class derived1::char name[80]
Inherited Private	Data		from struct base1::char name[80]
	Methods	Constructor	class derived1::derived1(void) struct base1::base1(void)
		(Regular)	class derived1::void data_ show(void) struct base1::void data_ show(void)
Public	Data		char name[80]
	Methods	Constructor	derived2(void)
		(Regular)	void data_show(void)
Inherited Public	Data		from struct base2::char name[80]
	Methods	Constructor	struct base2::base2(void)
		(Regular)	struct base2::void data_ show(void)

All method definitions are implemented

Figure D.3 Detailed View - Example 4



Members of STRUCT	derived3
of the Program:	example4

Inherited Buried	Data		from class derived1::char name[80]
Public	Data		char name[80]
	Methods	Constructor	derived3(void)
		(Regular)	void data_show(void)
Inherited Public	Data		from struct base1::char name[80]
	Methods	Constructor	class derived1::derived1(void) struct base1::base1(void)
		(Regular)	class derived1::void data_ show(void) struct base1::void data_ show(void)

All method definitions are implemented

Figure D.3 (cont'd) Detailed View - Example 4

```
-----Struct derived3 Definitions-----
  -----Public Constructor Methods-----
    -----derived3::derived3(void)-----
     strcpy(name, "derived3");

  -----Public (Regular) Methods-----
    -----void derived3::data_show(void)-----
     derived1::data_show();
     puts(name);
```

```
-----Non-Member Methods Definitions-----
  -----main()-----
   base1 obj1;
   base2 obj2;
   derived1 obj3;
   derived2 obj4;
   derived3 obj5;
   obj1.data_show();
   puts("----");
   obj2.data_show();
   puts("----");
   obj3.data_show();
   puts("----");
   obj4.data_show();
   puts("----");
   obj5.data_show();
```

```
====== End of Detailed View of program: example4 ====================
```

Figure D.3 (cont'd) Detailed View - Example 4

Appendix E

EXAMPLE SHOWING N1C++

The N1C++ features may be demonstrated by seeing the output for the example seen in Appendix D.

In Figure E.1 the Table of Contents for the program is given. Subsequent diagrams that use the N1C++ numbering option use the same titles and numbering used in Figure E.1.

Figure E.2 is the Overview for the program from example 4 and it shows three N1C++ features together. These are the section numbering, the section introductions, and the annotation. The annotations seen here, for the Global Overview and for the Inheritance Overview, are two of the three types of annotation available in the DoC++ system.

The Global Overview annotation tells the user that there are no unions in the program.

The Inheritance Overview annotation helps the user read through the box diagram by the Inheritance Overview used to inheritance relationships in the program. The annotated text assumes the user reads the diagram from top to bottom. For each of the object types the annotation states the objects above and below the object type in the inheritance hierarchy. For example, the annotation gives the user the following information about derived1:

```
The class derived1 is a base object type for:
  struct derived2
  struct derived3
The class derived1 is a derived object type that inherits from:
  struct base1
```

It is clear from this annotation that derived1 has object types above and below it in the inheritance hierarchy. Although the Inheritance Overview diagram shows this as well, some users may be more comfortable with the annotation.

The Inheritance Overview annotation for derived2 shows that derived2 is not a base object but is involved in multiple inheritance as the following segment from Figure E.2 shows.

```
The struct derived2 is involved in multiple inheritance.
The struct derived2 is a derived object type that inherits from:
  class derived1
  struct base1
  and
  struct base2
```

Table of Contents for program:	example4

1.0 Overview

    1.1 Global Overview

    1.2 Inheritance Overview

2.0 Detailed View

    2.1 Objects

        2.1.1 Classes

            2.1.1.1 derived1

        2.1.2 Structs

            2.1.2.1 base1

            2.1.2.2 base2

            2.1.2.3 derived2

            2.1.2.4 derived3

    2.2 Non-member Methods

Figure E.1 Table of Contents - Example 4

1.0 Overview

Overview Section - Introduction

This section contains information on the general structure and contents of the program. The overview consists of two sub-sections: 1) the Global Overview showing the major components in a program; and 2) the Inheritance Overview.

1.1 Global Overview

- Global Overview - Introduction

This section is intended to provide the broadest view of the program, showing five possible components of a C++ program: initial code such as macro definitions, included files and global declarations; class object type names; struct object type names; union object type names; and non-member method names.

- The Global Overview is shown below in Figure 1.1.

Global Overview of program:	example4

		Number	Names
Initial	Includes	2	#include <stdio.h> #include <string.h>
Classes		1	derived1
Structs		4	base1 base2 derived2 derived3
Non-member methods		1	main()

Figure 1.1

- Global Overview - Annotation:

The Initial Section of the Global Overview contains no macro definitions or declarations that are not declared within an object type or a method. It does contain two file inclusions.

This program does not contain any union object types. There is one class object type and four struct object types in this program.

There is only one Non-member Method in this program.

Figure E.2 Overview with N1C++ - Example 4

1.2 Inheritance Overview

■ Inheritance Overview - Introduction

This section shows relationships, if any, among object types in the program "example4." Objects in the middle of the box diagram are inheriting from object types towards the outside.

In the annotation that follows the Inheritance Overview the inheritance properties of each object are discussed. An object may be a base object for other objects; an object may inherit from other objects.

■ The Inheritance Overview is shown below in Figure 1.2.

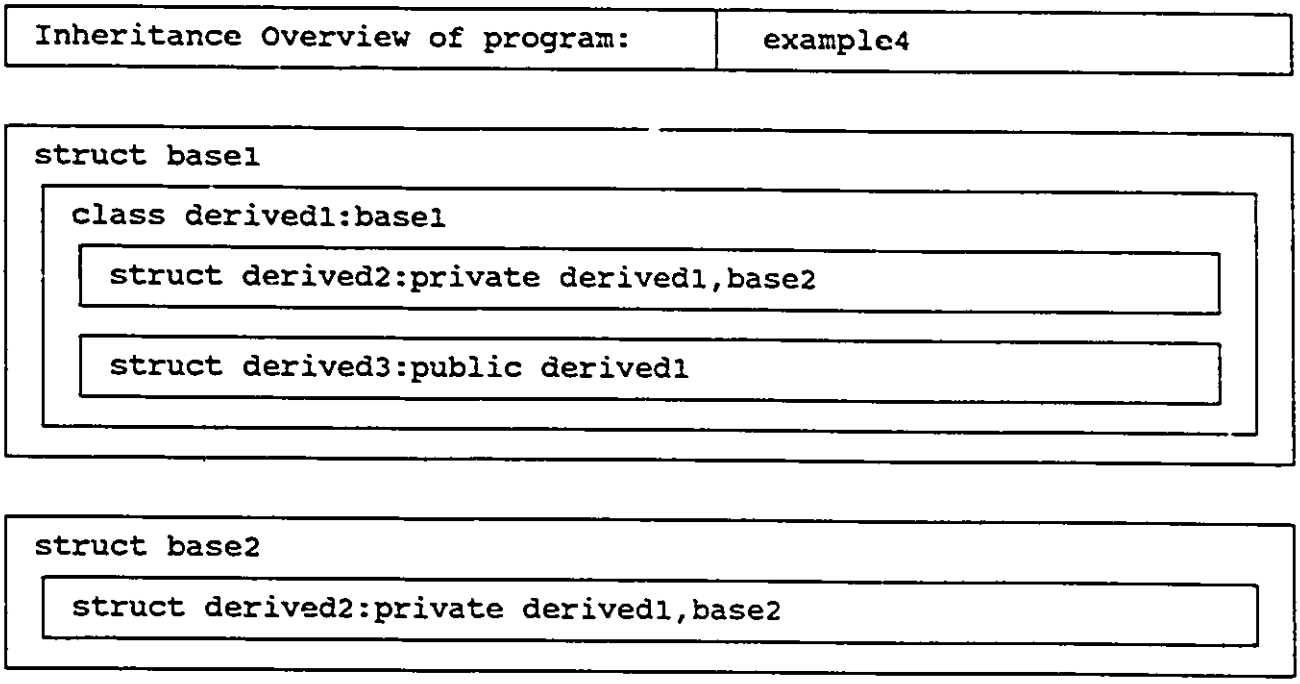


Figure 1.2

Figure E.2 (cont'd) Overview with NlC++ - Example 4

■ Inheritance Overview - Annotation

- struct base1
  The struct base1 is a base object type for:
    class derived1
    struct derived2
    struct derived3

- class derived1
  The class derived1 is a base object type for:
    struct derived2
    struct derived3
  The class derived1 is a derived object type that inherits from:
    struct base1

- struct derived2
  The struct derived2 is involved in multiple inheritance.
  The struct derived2 is a derived object type that inherits from:
    class derived1
    struct base1
    and
    struct base2

- struct derived3
  The struct derived3 is a derived object type that inherits from:
    class derived1
    struct base1

- struct base2
  The struct base2 is a base object type for:
    struct derived2

Figure E.2 (cont'd) Overview with NlC++ - Example 4

In addition DoC++ searches through the Inheritance Overview and finds all the object types that derived2 inherits from.

In Figure E.3 the last type of annotation is seen. This is the annotation of the members for each object type. A study of one such object type, derived2, demonstrates the knowledge of accessibility that the NlC++ gives the user. Below are the details for the Public Section of derived2.

- Public
There are methods and data in the Public Section. All are accessible from outside the object type.
The public data and methods can also be accessed by the following:
  the methods in the Public section
The public methods can access the following:
  the data and the methods in the Inherited Private section
  the data and the methods in the Inherited Public section
  the data and the methods in the Public section
The public methods cannot access the following:
  the data in the Buried section

The annotation shows how public data and methods can be accessed and what other data and methods the public methods can and cannot access.

This detailed annotation is available for all the sections of each object type.

2.0 Detailed View

Detailed View - Introduction

The Detailed View provides information on Objects (the Members and the Method Definitions) and on the Non-member Methods.

For each Object the documentation includes the Members (both declared and inherited) and the Method Definitions for each member method. In the method definitions selection and repetition structures are clearly indicated.

The Objects are classified into three types (classes, structs and unions).

2.1 Objects

Detailed View - Objects - Introduction

Three types of objects are possible in C++ programs: classes, structs and unions.

2.1.1 Classes

Detailed View - Classes - Introduction

This section documents both the Members and the Method Definitions for each class object type.

The user may choose to view only one of these for each class object type.

The Members are subdivided into general categories: private members, protected members, public members, friends, and four categories of inherited members. For all of these categories except friends, the members are categorized further into data, constructor methods, destructor methods, operator methods, and regular methods (i.e. not in one of the other three method categories).

The Method Definitions are categorized into the these general categories: private, protected and public. There are no Method Definitions for the friends and inherited members since they are defined elsewhere. The three general categories used are further subdivided into constructor methods, destructor methods, operator methods, and regular methods. Data is not defined.

In class object types data and methods are by default in the private category.

Figure E.3 Detailed View with N1C++ - Example 4

2.1.1.1 class derived1

- Members

The members of class derived1 are shown in Figure 2.1.1.1.A.

Members of CLASS	derived1
of the Program:	example4

Private	Data		char name[80]
Public	Methods	Constructor	derived1(void)
		(Regular)	void data_show(void)
Inherited Public	Data		from struct base1::char name[80]
	Methods	Constructor	struct base1::base1(void)
		(Regular)	struct base1::void data_ show(void)

All method definitions are implemented

Figure 2.1.1.1.A

- Detailed View - Annotation - Members of class derived1

This Detailed View shows all the members in the object.
Both declared members and inherited members are shown.

- Private
There is only data in the Private Section. This data is not accessible from outside the object. It must be accessed through a method that has access to the Private Section.
The private data can be accessed by the following:
the methods in the Public section

- Public
There are methods but no data in the Public Section. All the methods are accessible from outside the object type.
The public methods can also be accessed by the following:
the methods in the Public section
The public methods can access the following:
the data and the methods in the Inherited Public section
the data in the Private section
the methods in the Public section

Figure E.3 (cont'd) Detailed View with N1C++ - Example 4

- Inherited Public
There are methods and data in the Inherited Public Section. All are accessible from outside the object type. They also may be accessed through a method that has access to the Inherited Public Section.
The inherited public data and methods can be accessed by:
  the methods in the Public section
The inherited public methods can not access other data or methods declared in this object type. They can only access the inherited data or methods that they accessed in their base object type.
The inherited public methods cannot access the following:
  the data in the Private section
  the methods in the Public section

■ Method Definitions

The method definitions of class derived1 are shown in Figure 2.1.1.1.B.

```
Class derived1 Definitions
  Public Constructor Methods
    derived1::derived1(void)
      strcpy(name, "derived1");

  Public (Regular) Methods
    void derived1::data_show(void)
      base1::data_show();
      puts(name);
```

Figure 2.1.1.1.B

Figure E.3 (cont'd) Detailed View with C++ - Example 4

2.1.2 Structs

Detailed View - Structs - Introduction

This section documents both the Members and the Method Definitions for each struct object type.

The user may choose to view only one of these for each struct object type.

The Members are subdivided into general categories: private members, protected members, public members, friends, and four categories of inherited members. For all of these categories except friends, the members are categorized further into data, constructor methods, destructor methods, operator methods, and regular methods (i.e. not in one of the other three categories of methods).

The Method Definitions are categorized into the these general categories: private, protected and public. There are no Method Definitions for the friends and inherited members since they are defined elsewhere. The three general categories used are further subdivided into constructor methods, destructor methods, operator methods, and regular methods. Data is not defined.

In struct object types data and methods are by default in the public category.

2.1.2.1 struct base1

■ Members

The members of struct base1 are shown in Figure 2.1.2.1.A.

Members of STRUCT	base1
of the Program:	example4

Public	Data		char name[80]
	Methods	Constructor	base1(void)
		(Regular)	void data_show(void)

All method definitions are implemented

Figure 2.1.2.1.A

Figure E.3 (cont'd) Detailed View with C++ - Example 4

■ Detailed View - Annotation - Members of struct base1

This Detailed View shows all the members in the object.
Both declared members and inherited members are shown.
No inherited members are found in this object type.

- Public
There are methods and data in the Public Section. All are accessible from outside the object type.
The public data and methods can also be accessed by the following:
  the methods in the Public section
The public methods can access the following:
  the data and the methods in the Public section

■ Method Definitions

The method definitions of struct base1 are shown in Figure 2.1.2.1.B.

```
┌──Struct base1 Definitions──────────────────────────────────┐
│  ┌──Public Constructor Methods──────────────────────────┐  │
│  │  ┌──base1::base1(void)──────────────────────────┐    │  │
│  │  │ strcpy(name, "base1");                       │    │  │
│  │  └──────────────────────────────────────────────┘    │  │
│  └──────────────────────────────────────────────────────┘  │
│  ┌──Public (Regular) Methods────────────────────────────┐  │
│  │  ┌──void base1::data_show(void)─────────────────┐    │  │
│  │  │ puts(name);                                  │    │  │
│  │  └──────────────────────────────────────────────┘    │  │
│  └──────────────────────────────────────────────────────┘  │
└────────────────────────────────────────────────────────────┘
```

Figure 2.1.2.1.B

Figure E.3 (cont'd) Detailed View with N1C++ - Example 4

2.1.2.2 struct base2

- Members

The members of struct base2 are shown in Figure 2.1.2.2.A.

Members of STRUCT	base2
of the Program:	example4

Public	Data		char name[80]
	Methods	Constructor	base2(void)
		(Regular)	void data_show(void)

All method definitions are implemented

Figure 2.1.2.2.A

- Detailed View - Annotation - Members of struct base2

This Detailed View shows all the members in the object.
Both declared members and inherited members are shown.
No inherited members are found in this object type.

- Public
There are methods and data in the Public Section. All are accessible from outside the object type.
The public data and methods can also be accessed by the following:
  the methods in the Public section
The public methods can access the following:
  the data and the methods in the Public section

Figure E.3 (cont'd) Detailed View with C++ - Example 4.

■ Method Definitions

The method definitions of struct base2 are shown in Figure 2.1.2.2.B.

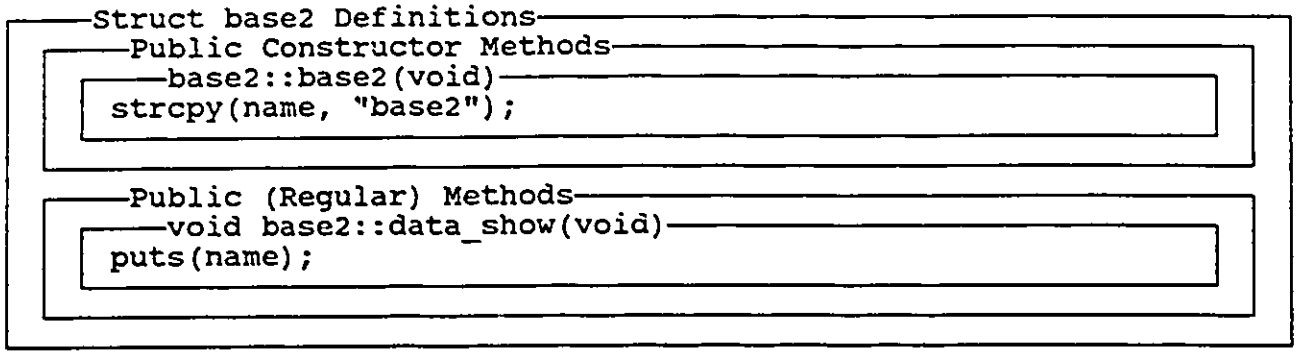


Figure 2.1.2.2.B

Figure E.3 (cont'd) Detailed View with N1C++ - Example 4

2.1.2.3 struct derived2

■ Members

The members of struct derived2 are shown in Figure 2.1.2.3.A.

Members of STRUCT	derived2
of the Program:	example4

Inherited Buried	Data		from class derived1::char name[80]
Inherited Private	Data		from struct base1::char name[80]
	Methods	Constructor	class derived1::derived1(void) struct base1::base1(void)
		(Regular)	class derived1::void data_show(void) struct base1::void data_show(void)
Public	Data		char name[80]
	Methods	Constructor	derived2(void)
		(Regular)	void data_show(void)
Inherited Public	Data		from struct base2::char name[80]
	Methods	Constructor	struct base2::base2(void)
		(Regular)	struct base2::void data_show(void)

All method definitions are implemented

Figure 2.1.2.3.A

Figure E.3 (cont'd) Detailed View with C++ - Example 4

■ Detailed View - Annotation - Members of struct derived2

This Detailed View shows all the members in the object.
Both declared members and inherited members are shown.

- Inherited Buried
There is only data in the Inherited Buried Section. This data is not accessible from outside the object It must be accessed through a method that has access to the Inherited Buried Section. The Inherited Buried Section is accessed by an indirect call to a member that had access in the base object to the now-buried data or methods.

- Inherited Private
There are methods and data in the Inherited Private Section. None are accessible from outside the object type. They must be accessed through a method that has access to the Inherited Private Section.
The inherited private data and methods can be accessed by:
  the methods in the Public section
The inherited private methods can not access other data or methods declared in this object type. They can only access the inherited data or methods that they accessed in their base object type.
The inherited private methods cannot access the following:
  the data and the methods in the Public section

- Public
There are methods and data in the Public Section. All are accessible from outside the object type.
The public data and methods can also be accessed by the following:
  the methods in the Public section
The public methods can access the following:
  the data and the methods in the Inherited Private section
  the data and the methods in the Inherited Public section
  the data and the methods in the Public section
The public methods cannot access the following:
  the data in the Buried section

- Inherited Public
There are methods and data in the Inherited Public Section. All are accessible from outside the object type. They also may be accessed through a method that has access to the Inherited Public Section.
The inherited public data and methods can be accessed by:
  the methods in the Public section
The inherited public methods can not access other data or methods declared in this object type. They can only access the inherited data or methods that they accessed in their base object type.
The inherited public methods cannot access the following:
  the data and the methods in the Public section

Figure E.3 (cont'd) Detailed View with C++ - Example 4.

■ Method Definitions

The method definitions of struct derived2 are shown in Figure 2.1.2.3.B.

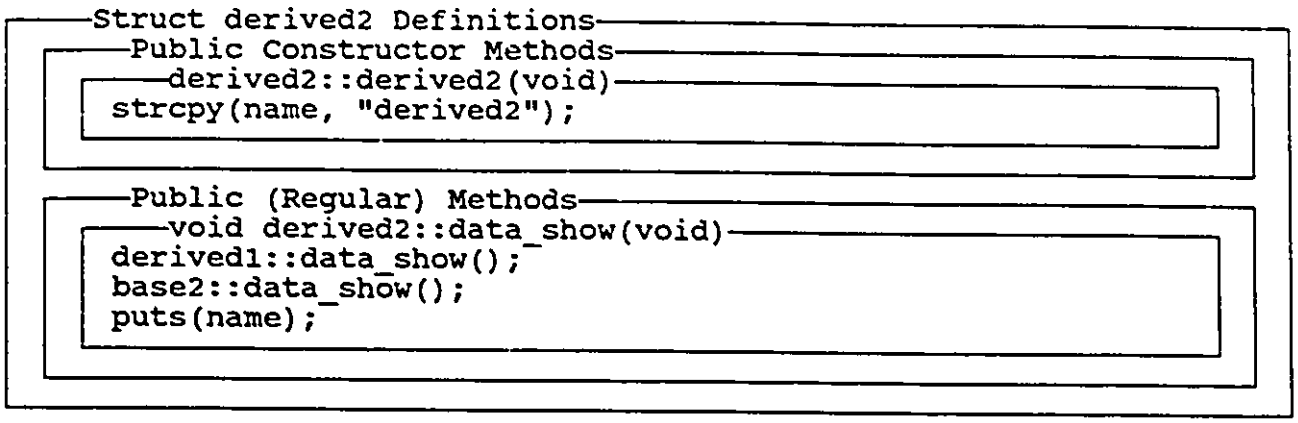


Figure 2.1.2.3.B

Figure E.3 (cont'd) Detailed View with N1C++ - Example 4

2.1.2.4 struct derived3

- Members

The members of struct derived3 are shown in Figure 2.1.2.4.A.

Members of STRUCT	derived3
of the Program:	example4

Inherited Buried	Data		from class derived1::char name[80]
Public	Data		char name[80]
	Methods	Constructor	derived3(void)
		(Regular)	void data_show(void)
Inherited Public	Data		from struct base1::char name[80]
	Methods	Constructor	class derived1::derived1(void) struct base1::base1(void)
		(Regular)	class derived1::void data_ show(void) struct base1::void data_ show(void)

All method definitions are implemented

Figure 2.1.2.4.A

- Detailed View - Annotation - Members of struct derived3

This Detailed View shows all the members in the object. Both declared members and inherited members are shown.

- Inherited Buried
There is only data in the Inherited Buried Section. This data is not accessible from outside the object It must be accessed through a method that has access to the Inherited Buried Section. The Inherited Buried Section is accessed by an indirect call to a member that had access in the base object to the now-buried data or methods.

Figure E.3 (cont'd) Detailed View with C++ - Example 4

- Public
There are methods and data in the Public Section. All are accessible from outside the object type.
The public data and methods can also be accessed by the following:
  the methods in the Public section
The public methods can access the following:
  the data and the methods in the Inherited Public section
  the data and the methods in the Public section
The public methods cannot access the following:
  the data in the Buried section

- Inherited Public
There are methods and data in the Inherited Public Section. All are accessible from outside the object type. They also may be accessed through a method that has access to the Inherited Public Section.
The inherited public data and methods can be accessed by:
  the methods in the Public section
The inherited public methods can not access other data or methods declared in this object type. They can only access the inherited data or methods that they accessed in their base object type.
The inherited public methods cannot access the following:
  the data and the methods in the Public section

■ Method Definitions

The method definitions of struct derived3 are shown in Figure 2.1.2.4.B.

```
Struct derived3 Definitions
  Public Constructor Methods
    derived3::derived3(void)
      strcpy(name, "derived3");

  Public (Regular) Methods
    void derived3::data_show(void)
      derived1::data_show();
      puts(name);
```

Figure 2.1.2.4.B

Figure E.3 (cont'd) Detailed View with NlC++ - Example 4

2.2 Non-Member Methods

Detailed View - Non-member Methods - Introduction

This section produces Method Definitions for all the methods that are not declared to be part of an object type.

The Method Definitions are shown in Figure 2.2.

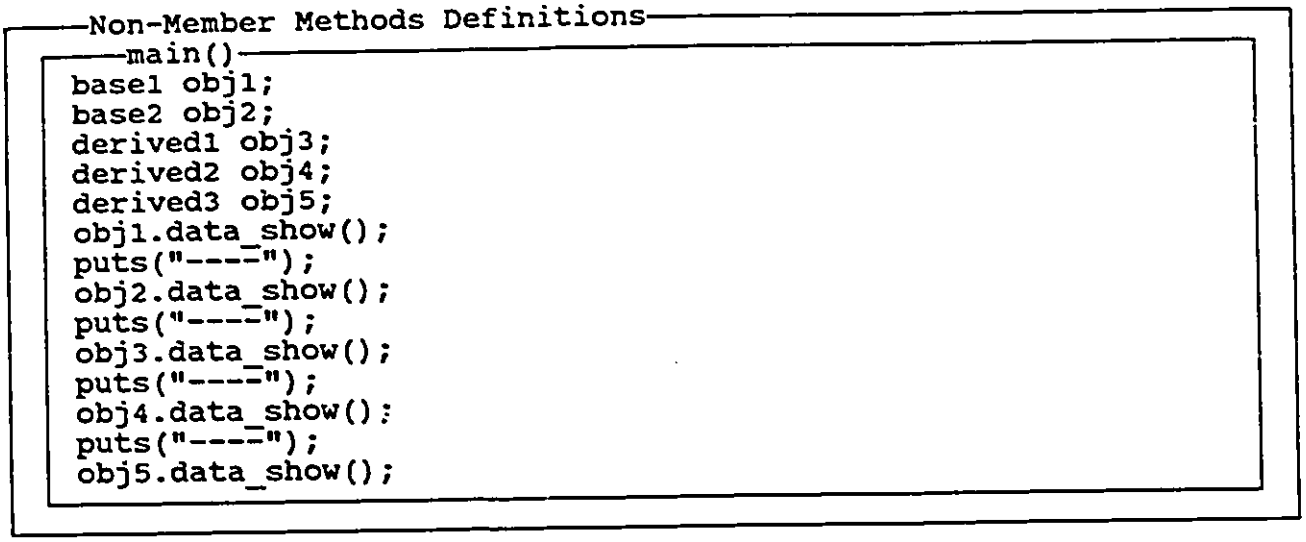


Figure 2.2.

Figure E.3 (cont'd) Detailed View with N1C++ - Example 4

Appendix F

GLOSSARY

Note that the explanations of the terms used in this glossary are done in the context of C++ and DoC++.

Annotation
: additional natural language explanations used to supplement the ORC++ diagrams; three types of annotation exist in DoC++: global overview annotation, inheritance overview annotation and detailed view annotation.

Annotation, Detailed View
: additional natural language comments used to supplement the Detailed View; for each object type, it lists for each section present the details of encapsulation and accessibility.

Annotation, Global Overview
: additional natural language comments used to supplement the Global Overview.

Annotation, Inheritance Overview
: additional natural language comments used to supplement the Inheritance Overview; for each object type, it lists the derived object types and the object types from which the object type inherits.

Buried, Inherited
: a section of members in an object type containing inherited data and/or methods that are not accessible by the member methods of the object; must be accessed through a call to an accessible method in the object from which the buried data and/or methods was inherited.

Class
: an instance of a class object type in which the default level of encapsulation is private; also used for the class object type itself.

Class Object Type
: an object type containing data and/or methods in which the default level of encapsulation is private; may also contain protected, public and friend sections; may inherit data and/or methods from other methods.

Constructor Method
: a member method of object types that is called automatically whenever an instance of the object type is declared; used typically to allocate memory and initialize variables.

Data
: one of the two general types of members in an object type, the other being the method; data is manipulated by methods.

Declaration
: statement of the existence of data or a method in a program or the statement of the existence of an object type in a program; a method declaration must precede its definition.

Definition, Method
: the code that details the action taken by a method; with C++ objects, the method definition may follow the method declaration or follow the declaration of the object type.

Destructor Method
: a member method of object types that is called automatically whenever an instance of the object type passes out of scope; used typically to deallocate memory.

Detailed View
: one of the graphical representations of a C++ program produced by the ORC++ system; provides the user with a list of categorized members and method definitions for each object type in a program.

DoC++
: Documentation for C++ programs, consisting of ORC++ and NlC++.

Friend
: special member methods of an object type that have access to the private data and/or methods in the object despite friends being declared and defined outside the object type; friends may be the methods of complete class or struct object types, or individual methods that may or may not be part of another object type.

Inheritance Attribute
: the specification that controls the inheritance between one object type and another; may be private or public.

Inherited
: description of members of an object type that are not declared in the object.

Members
: data and/or methods that are associated with an object type; some members are declared and others are inherited.

Members, Declared
: members that are declared in the declaration of the object type; members that are not inherited.

Members, Inherited
: members that are inherited from other object types; members that are not declared in the declared.

Method
: one of the two general types of members in an object type, the other being data; methods operate on data and call other methods; member methods may be further classified into constructor, destructor, operator and regular methods.

NlC++
: Natural Language for C++ programs; the component of DoC++ that supplements the ORC++ diagrams by providing a table of contents, section numbering, section introductions and annotation.

Non-member Methods
: methods that are not declared to be part of any object type.

Object
: an instance of an object type.

Object Type
: either a class object type, a struct object type or a union object type; a collection of logically connected data and methods.

Operator Method
: a member method that defines the use of an operator symbol.

ORC++
: Organized Representation for C++ programs; provides the following graphical representation diagrams: the Global Overview, the Inheritance Overview and the Detailed View.

Overview
: a term used for two of the graphical representation diagrams in ORC++: the Global Overview and the Inheritance Overview.

Overview, Global
: one of the graphical representation diagrams in the ORC++ Overview; used to provide a broad view of the components of a C++ program.

Overview, Inheritance
: one of the graphical representation diagrams in the ORC++ Overview; used to provide information on the inheritance characteristics in a C++ program.

Private
: describes data and methods in an object type that cannot be accessed from outside the object type; also used to describe the default inheritance attribute for a class object type.

Private, Inherited
: describes data and methods in an object type that are inherited from another object and cannot be accessed from outside the object type.

Protected
: describes data and methods in an object type, that upon being inherited are private to the inheriting object type.

Protected, Inherited
: describes data and methods in an object type, that are inherited from another object and possess the same properties as declared protected members.

Public
: describes data and methods in an object type that are accessible from outside the object type.

Public, Inherited
: describes data and methods in an object type that are inherited from another object and also are accessible from outside the object type; also used to describe the default inheritance attribute for a struct object type.

Regular Method
: a member method of an object type that does not fit into the following categories of methods: constructor, destructor or operator.

Section Introduction
: one of the features of the N1C++ methodology that introduces the graphical representations of ORC++.

Section Numbering
: one of the features of the N1C++ that provides section numbers and titles for the ORC++ graphical representations.

Struct
: an instance of a struct object type in which the default level of encapsulation is public; also used for the struct object type itself.

Struct Object Type
: an object type containing data and/or methods in which the default level of encapsulation is public; may also contain private, protected, and friend sections; may inherit data and/or methods from other methods.

Table of Contents
: a N1C++ documentation feature giving complete section numbers and titles used in the rest of the DoC++ output when the section numbering option is used.

Union
: an instance of a union object type that has only public data and/or members and cannot be involved in inheritance; the data members occupy the same area of memory.

Union Object Type
: an object type containing only public data and/or methods and not involved in inheritance; the data members in the instance occupy the same area of memory.