

TopSpeed® C++

For IBM® Personal Computers and Compatibles

Language Reference

TopSpeed Corporation

Copyright© 1990-1991, by TopSpeed Corporation. All rights reserved.

TopSpeed® is a registered trademark of TopSpeed Corporation.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Printed in the United Kingdom.

10 9 8 7 6 5 4 3 2 1

Contents

CHAPTER 1

INTRODUCTION 10

Notation 11

CHAPTER 2

BASIC CONCEPTS 12

Introduction 12

Names 12

Declarations and Definitions 13

Scopes of Identifiers 13

Linkages of Identifiers 14

Storage Durations of Objects 16

Fundamental Types 16

Derived Types 18

Lvalues 19

Typedef Names 19

Program Startup 20

CHAPTER 3

C++ LEXICAL TOKENS 22

Tokens 22

Literals 25

Floating Constants 25

Character Constants 28

CHAPTER 4

CONVERSIONS 32

Introduction 32

Arithmetic Operands 32

CHAPTER 5

DECLARATIONS 37

| | |
|----------------------------------|----|
| Introduction | 37 |
| Declaration Specifiers | 38 |
| Storage-Class Specifiers | 39 |
| Function Specifiers | 40 |
| Typedef Declarations | 40 |
| Elaborated Type Specifiers | 42 |
| Asm Declarations | 44 |
| Linkage Specifications | 44 |

CHAPTER 6

DECLARATORS 47

| | |
|--|----|
| Introduction | 47 |
| Pointer Declarators | 48 |
| Reference Declarators | 49 |
| Pointer-to-Member Declarators | 50 |
| Array Declarators | 51 |
| Function Declarators | 52 |
| Declarators with Special Keywords | 55 |
| Abstract Declarators and Type Names | 57 |
| Function Definitions | 58 |
| Initialization | 59 |
| Initializing Aggregates | 60 |
| Initializing Multidimensional Arrays | 61 |
| Initializing Strings | 63 |
| Initializing References | 63 |
| Initializing Functions | 64 |
| Declaration Ambiguities | 65 |

CHAPTER 7

EXPRESSIONS 66

| | |
|-------------------------------|----|
| Introduction | 66 |
| Precedence of Operators | 68 |

| | |
|--|----|
| Primary Expressions | 69 |
| Postfix Operators | 70 |
| Explicit Type Conversion (Function Notation) | 73 |
| Member Selection Operators | 73 |
| Postfix Increment and Decrement Operators | 74 |
| Unary Operators | 75 |
| Prefix Increment and Decrement Operators | 75 |
| Unary Arithmetic Operators | 77 |
| The sizeof Operator | 78 |
| The new Operator | 79 |
| The delete Operator | 81 |
| Cast Expressions | 82 |
| Pointer-to-Member Operators | 84 |
| Multiplicative Operators | 85 |
| Additive Operators | 86 |
| Bitwise Shift Operators | 86 |
| Relational Operators | 87 |
| Equality Operators | 88 |
| Bitwise AND Operator | 89 |
| Bitwise Exclusive OR Operator | 89 |
| Bitwise Inclusive OR Operator | 90 |
| Logical AND Operator | 90 |
| Logical OR Operator | 90 |
| Conditional Operator | 91 |
| Assignment Operators | 92 |
| Simple Assignment | 92 |
| Compound Assignment | 93 |
| Comma Operator | 94 |
| Constant Expressions | 95 |

CHAPTER 8

STATEMENTS 97

| | |
|--------------------------|----|
| Introduction | 97 |
| Labeled Statements | 97 |

| | |
|--------------------------------------|-----|
| Compound Statement, or Block | 98 |
| Expression and Null Statements | 98 |
| Jump Statements | 99 |
| The goto Statement | 99 |
| The continue Statement | 99 |
| The break Statement | 100 |
| The return Statement | 101 |
| Selection Statements | 101 |
| The if Statement | 102 |
| The switch Statement | 103 |
| Iteration Statements | 104 |
| The while Statement | 105 |
| The do Statement | 105 |
| The for Statement | 105 |
| Declaration Statements | 106 |
| Expression Ambiguity | 107 |

CHAPTER 9

| | |
|---|------------|
| CLASSES | 108 |
| Class Names | 108 |
| Class Members | 109 |
| Member Functions | 110 |
| Const and Volatile Member Functions | 111 |
| The this Pointer | 112 |
| Inline Member Functions | 112 |
| Static Members | 113 |
| Unions | 115 |
| Anonymous Unions | 115 |
| Bit Fields | 116 |
| Nested Class Declarations | 117 |
| Local Class Declarations | 118 |
| Local Type Names | 118 |

CHAPTER 10

DERIVED CLASSES 119

| | |
|----------------------------|-----|
| Introduction | 119 |
| Derived Classes | 119 |
| Multiple Inheritance | 120 |
| Virtual Base Classes | 121 |
| Ambiguities | 122 |
| Virtual Functions | 124 |
| Abstract Classes | 126 |

CHAPTER 11

MEMBER ACCESS CONTROL 127

| | |
|--|-----|
| Member Access Specifiers | 127 |
| Base Class Access Specifiers | 128 |
| Access Declarations | 130 |
| Access Rules for Virtual Functions | 132 |
| Multiple Access | 132 |

CHAPTER 12

SPECIAL MEMBER FUNCTIONS 133

| | |
|--|-----|
| Constructors | 133 |
| Destructors | 135 |
| Calling Constructors and Destructors | 136 |
| Temporary Objects | 138 |
| Conversions | 140 |
| Conversions by constructor | 140 |
| Conversion Functions | 140 |
| Initialization | 141 |
| Explicit Initialization | 142 |
| Initializing Bases and Members | 142 |
| Copying Class Objects | 144 |
| The new and delete Operators | 144 |

CHAPTER 13

OVERLOADING 146

| | |
|--|-----|
| Introduction | 146 |
| Declaration Matching | 147 |
| Argument Matching | 147 |
| Conversion Ordering | 148 |
| Overloaded Function Address Resolution | 151 |
| Overloaded Operators | 152 |
| The Assignment Operator | 152 |
| The Function Call Operator | 152 |
| The Subscript Operator | 153 |
| The Class Member Access Operators | 153 |
| The Increment and Decrement Operators | 154 |
| Unary Operators | 155 |
| Binary Operators | 155 |

CHAPTER 14

THE PREPROCESSOR 156

| | |
|-----------------------------------|-----|
| Introduction | 156 |
| Phases of Translation | 156 |
| Trigraph Sequences | 157 |
| Preprocessing Directives | 158 |
| Defining a Macro | 159 |
| Conditional Inclusion | 164 |
| Source File Inclusion | 166 |
| Line Control | 168 |
| Error Directive | 168 |
| Pragma Directive | 168 |
| Null Directive | 169 |
| Predefined Macro Names | 169 |
| Preprocessor Syntax Summary | 169 |

APPENDIX A

REFERENCES 172

| | |
|---|------------|
| APPENDIX B | |
| SUMMARY OF SCOPE RULES | 173 |
| APPENDIX C | |
| NAME ENCODING | 175 |
| APPENDIX D | |
| IMPLEMENTATION DEFINED BEHAVIOR..... | 178 |
| APPENDIX E | |
| UNDEFINED BEHAVIOR | 185 |
| APPENDIX F | |
| TOPSPEED C++ EXTENSIONS. | 187 |
| APPENDIX G | |
| COLLECTED SYNTAX. | 188 |
| APPENDIX H | |
| COMPILER ERRORS AND WARNINGS | 198 |
| Introduction | 198 |
| C++ Language Warnings | 198 |
| C++ Language Errors | 202 |
| INDEX | 246 |

CHAPTER 1

INTRODUCTION

This manual documents the C++ language as implemented in TopSpeed C++. The implementation adheres to that defined in “The Annotated C++ Reference Manual”, by Margaret Ellis and Bjarne Stroustrup (Addison Wesley 1990), referred to as the Base Document.

The manual is intended as a guide for knowledgeable programmers; it is not an introduction or tutorial on the language. The *C++ Language Tutorial* supplied with TopSpeed C++ provides a more elementary and extensive summary of the language.

The C++ language is a general purpose programming language, based upon ANSI C. The C language was designed to allow efficient programs to be written, machine specific or portable as required. In addition to this, C++ provides support for an object-oriented programming paradigm, a user-extensible type system via the class mechanism, inline functions, overloaded functions and operators, references, and free-store management.

This manual discusses the following:

- The syntax of the C++ language.
- The constraints on text written in the C++ language.
- The semantic rules for interpreting C++ programs.
- The restrictions and limits imposed by the Base Document.
- The restrictions and limits imposed by the TopSpeed C++ implementation.
- The extensions to the Base Language that are provided by the TopSpeed C++ implementation.

Notation

The following notational conventions are used throughout this manual:

| | |
|----------------------------|--|
| <i>Italic</i> | is used to emphasize a particular concept or point. |
| Courier | is used for C++ program examples and for C++ keywords and program fragments within text. |
| <i>Oblique</i> | is used for C++ syntax listings. |
| Bold | for C++ keywords and tokens in syntax listings. |
| <i>Thing_{opt}</i> | is used to indicate that ‘Thing’ is optional. |

CHAPTER 2

BASIC CONCEPTS

Introduction

A C++ program consists of one or more source files. These source files will contain C++ code and may contain material (for example, macro definitions) for processing by the preprocessor. Each source file can be independently processed by the compiler.

A *translation unit* consists of a source file after preprocessing, together with all the headers and source files included via the preprocessing directive `#include`. However, the translation unit does not include any source lines skipped as a result of conditional inclusion preprocessing directives.

One or more translation units may be compiled separately and joined together with a linker to produce an executable program.

The process of turning C++ source text into an executable program occurs (conceptually) in a series of stages. These are enumerated in detail in Chapter 14, 'Phases of translation'.

Names

In a C++ translation-unit, *names*, or *identifiers*, may be given to any of the following:

- Objects
- Types
- Functions or sets of functions
- Enumerators
- Class members
- Labels

In C++, there are rules for deciding each of the following:

- Which, if any, of these names are usable at a point in the translation unit (visibility and access rules).
- Whether it is possible to temporarily reuse a name (scope and name space).
- How identical names may relate to each other (linkage).
- The period of time (lifetime) during which storage is reserved for names of particular objects.
- Which of a set of functions with the same name should be used (overloading).

Details of these rules are discussed in the relevant sections.

Declarations and Definitions

A name is introduced into a translation-unit by means of a *declaration*. An object or function that is used anywhere in a program must also have exactly one *definition*, which causes the storage required to be set aside.

A declaration is also a definition, except for the following cases :-

- A function declaration for which no body is specified.
- An declaration with the extern keyword for which no initializer is specified.
- A typedef declaration.
- The declaration of a static member in a class declaration.
- A class name declaration.

Scopes of Identifiers

There are constraints on identifiers, with respect to where in the program each identifier can be used (i.e., is *visible*). In particular, an identifier is visible only within a region of program text called the identifier's *scope*.

The scope of all identifiers is determined by the location of the identifier's declaration. There are four kinds of scope:

| | |
|-------|---|
| Block | If a declaration or type name appears inside a block (such as a function body), or if it is one of the param- |
|-------|---|

| | |
|--------------------|---|
| | <p>eters in a function definition, the name has <i>block scope</i>, and may be used within the block and any blocks enclosed by it, after the point of declaration. The scope terminates at the } that closes the associated block.</p> |
| File | <p>If a declaration or type name appears outside any block or list of parameters, the declaration has <i>file scope</i>, and can be used from the point of declaration to the end of the translation unit.</p> |
| Function | <p>The only identifier that has <i>function scope</i> is a label name. A <i>label name</i> is declared implicitly by its syntactic appearance, which must be within a function. The label name can be used in a goto statement anywhere in the function in which it appears. Names with function scope do not hide and are not hidden by names in any other scope.</p> |
| Class | <p>A name declared as a member of a class (See Chapter 9) has class scope. Such a name may be used in a member function of the class, after a member selection operator . or -> applied to a suitable object, or after the scope resolution operator :: applied to its class name or a class derived from it. The visibility of class member names is also affected by ambiguity rules and access rules - see chapters 9-13.</p> |
| Function prototype | <p>A <i>function prototype</i> is a declaration of a function that also declares the types of the function's parameters. If a declaration or type name appears within the list of parameter declarations in a function prototype (i.e., not as part of a function definition) or friend declaration, the identifier has <i>function prototype scope</i>. This scope terminates at the end of the function declarator.</p> |

The scope rules are described fully in Appendix B.

Linkages of Identifiers

.i.linkage;

Linkage is the name given to the process whereby two or more lexically identical identifiers from different translation units or scopes are made to refer to the same object or function in a program. Names that are not mentioned below have no linkage. A name may not have more than one linkage in the same scope.

Internal Linkage

The name of an object or function with file scope is said to have *internal linkage* if any of the following is true:

- It is explicitly declared static.
- It is explicitly declared inline.
- It is explicitly declared const, and not explicitly declared extern.

A class name has internal linkage if it has no static members and no member functions that are not inline, and is not used in the declaration of any object or function that has external linkage.

A name with internal linkage does not refer to the same object or class as the same name in any other translation unit.

External Linkage

The name of an object, function or class that has file scope, and does not have internal linkage as described above, has external linkage.

Static members and non-inline member functions have external linkage. For the purposes of external name equivalence, a member name consists of its name qualified by the name of its class.

A local name that is declared extern, and not already declared static, has external linkage.

All uses of a particular name with external linkage in a program refer to the same object, function (or set of functions) or class. All declarations of a name which has external linkage must specify identical types (apart from the use of typedefs).

If a name with external linkage is used in an expression, then somewhere in the entire program there must be exactly one external definition for the identifier. This definition must be a declaration that has external linkage and must be one for which storage is allocated. An exception is when the identifier is part of the operand of a sizeof operator, which is evaluated at compile time.

Other Linkage

A name may be declared using a *linkage-specification*, to indicate a special type of linkage that is required (for example, to link with functions or objects that are not defined in a C++ translation-unit). This is described further in Chapter 5: ‘Linkage specifications’.

Storage Durations of Objects

An object has a *storage duration* that determines the object's lifetime. There are two classes of storage duration that can be declared:

| | |
|-----------|--|
| static | <p>An object has static storage duration if it is declared with either the storage class specifier <code>static</code> or with external or internal linkage.</p> <p>Such an object is created and initialized only once, prior to program startup. The object exists and retains its most recently stored value throughout the execution of the entire program.</p> |
| automatic | <p>An object has automatic storage duration if it is declared in a function with no linkage and without the static storage class specifier.</p> <p>An object with automatic storage duration is created every time its definition is executed, and destroyed on exit from its block. It is illegal to jump past a definition statement unless the entire block containing it is skipped.</p> |

Fundamental Types

Each object in C++ has an associated type, which is specified when the object is declared. A *type* defines the set of values that an object may take, and the set of operations that may be performed on its values.

Types in C++ can be divided into the fundamental types, described below, and the derived types derived from them.

The character, integer and enumeration types are known collectively as *integral types*. The character, integer, enumeration and floating types are known collectively as *arithmetic types*.

Character Types

The storage allocated for an object of type `char` is one byte. All values of the source character set are positive.

A `char` may be signed or unsigned.

An unsigned `char` occupies the same amount of storage as a plain `char`, but can represent only nonnegative values.

A signed `char` occupies the same amount of storage as a plain `char`, but can represent both positive and negative character values. In TopSpeed C++, a plain `char` is by default interpreted as a signed `char`. However, the pragma

`option(uns_char=>on)` can be used to specify that plain char should be regarded as unsigned.

Together, the char, signed char, and unsigned char types make up the *character types*. In C++, these are three distinct types, so it is possible, for example, to declare three overloaded functions which differ only in whether the argument is a char, a signed char or an unsigned char, and a reference to signed char will require a temporary if initialized with a plain char.

Integer Types

C++'s *integer types* are used to represent whole numbers. The range of valid values depends on which integer type is being used. Integer types may be signed or unsigned.

A plain int has a storage size of two bytes. The valid range for this type falls within the bounds `INT_MIN` and `INT_MAX`, whose values are given in the header file `<limits.h>`.

There are four kinds of *signed integer types*:

- signed char (one byte)
- short int (two bytes)
- int (two bytes)
- long int (four bytes)

The range of values for each signed type in the list depends on the type's size. The range for each signed type is a subrange of the values for the types below it in the list. For example, values for signed char are a subrange of values for any other signed integer type, values for short int are a subrange of values for int and long int.

There is an unsigned type corresponding to every signed integer type. This unsigned type utilizes the same amount of storage (including the sign flag).

All nonnegative values that can be represented by a signed integer type can be represented by its corresponding unsigned type, and the representation of the same value is the same in each type.

Computations involving unsigned operands will not cause an overflow error. A result that cannot be represented by the resulting unsigned type is reduced as follows:

$$\text{reduced} = \text{large} \bmod N$$

where N is the number one greater than the largest value that can be represented by the unsigned type.

Floating Point Types

Real numbers are represented using *floating point types*. Representations of real values will be approximations to the actual value, due to limitations imposed by representation in a finite storage area.

There are three floating point types:

- float (four bytes)
- double (eight bytes)
- long double (ten bytes)

The set of values of a float is a subset of the set of values of a double, which is a subset of the set of values of a long double.

See <float.h> for definitions of the size limits for floating point types in TopSpeed C++.

Enumeration Types

An *enumeration* consists of a collection of named constant values. Integer constants are associated with each name, beginning within 0 for the first name in an enumeration, and incrementing by 1. It is possible to specify other values to be associated with names.

Enumerations are described in Chapter 5: ‘Enumeration Specifiers’.

The void type

The void type is an incomplete type, and specifies an empty set of values. An expression may be explicitly converted to void to discard its value, and a function that does not return a value is declared as returning type void.

Derived Types

Derived types can be derived from the fundamental types, and recursively from other derived types, in the following ways:

| | |
|-----------------------|---|
| <i>array types</i> | are used to specify a contiguously allocated set of members of any one type of object, called the <i>base</i> , or <i>element, type</i> . Array declarations are discussed in Chapter 6: ‘Array Declarators’. |
| <i>class types</i> | are used to specify user-defined types. Class declarations are discussed in detail in Chapters 9-12. |
| <i>function types</i> | are used to specify a function. A function is character- |

| | |
|--------------------------------|--|
| | ized by its parameters and return type. Function declarations are discussed in Chapter 6: ‘Function Declarators’. |
| <i>pointer types</i> | are used to point to functions or to objects of any type. Pointers and arithmetic types are collectively called <i>scalar types</i> . Pointer declarations are discussed in Chapter 6: ‘Pointer Declarators’. |
| <i>reference types</i> | are used to specify a name which refers to an object or function. Reference declarations are discussed in Chapter 6: ‘Reference Declarators’. |
| <i>structure types</i> | are class types without access restrictions. See Chapter 9. |
| <i>union types</i> | are structures that can hold only one value at a time. See Chapter 9: ‘Unions’. |
| <i>pointer-to-member types</i> | are used to identify a particular member within a class object. A pointer to member is derived both from the type of the objects it can point to, and the class into which it can point. In general, where this manual refers to “pointers”, it does not include pointers-to-members, unless explicitly stated otherwise. Declarations of pointers-to-members are described in Chapter 6: ‘Pointer-to-member Declarators’. |

The methods for constructing derived types can be applied recursively, although there are restrictions, as discussed in Chapter 6.

Lvalues

An lvalue is an expression that denotes a storage location. For example, the name of an object is an lvalue that denotes the storage location of that object; an integer constant is not an lvalue, as it has no storage associated with it.

A modifiable lvalue is an lvalue that denotes a storage location that can be modified, without violating any language or implementation constraint. For example, the identifier of an object declared as `const int` is an lvalue, but it is not a modifiable lvalue as there is a language constraint specifying that a such an object may not be modified.

Expressions requiring lvalue operands or returning lvalue results are described in Chapter 7.

Typedef Names

A programmer may give names to fundamental or derived types by means of the typedef mechanism, described in Chapter 5: ‘Typedef Declarations’. A

typedef name behaves like a reserved word within its scope, but can be redeclared in other scopes.

Program Startup

The TopSpeed C++ compiler and linker create executable C++ programs from source files. Executable programs generated by TopSpeed C++ run in a hosted environment, i.e., they run under the control of an operating system.

An executable program must contain a function called `main` within a translation unit used to create the executable program. This function is called when the program starts executing. When a program starts up, all objects in static storage are initialized.

The `main` function must have been defined as one of the following:

```
int main() {...}
int main(void) {...} // equivalent form
int main(int argc, char *argv[]) {...}
int main(int argc, char *argv[],
        char *env[]) {...}
```

If present, the arguments passed to `main` have the following significance:

| | |
|-------------------|---|
| <code>argc</code> | (for <i>argument count</i>) represents the number of command line arguments when the program is called. This value is nonnegative. |
| <code>argv</code> | (for <i>argument value</i>) represents the individual arguments. If <code>argc</code> is greater than 0, then <code>argv[0]</code> through <code>argv[argc-1]</code> will contain pointers to strings; <code>argv[argc]</code> is a null pointer. <code>argv[0]</code> points to the program name. If <code>argc</code> is greater than 1, subsequent array members (i.e., <code>argv[1]</code> through <code>argv[argc-1]</code>) represent additional command line arguments. |
| <code>env</code> | is a TopSpeed C++ extension, and represents the DOS (or OS/2) environment strings that have been defined. |

The `main` function may not be overloaded, and may not be called from within a program, nor may its address be taken. The function `main` may not be declared as `inline` or `static`, and its linkage is always external.

When the function `main` returns, the program termination takes place. The return value is used to specify the return code that the program returns to the operating system. A call to the function `exit()` also causes program termination to take place, as if `main` had returned the value specified in the argument.

At program termination, destructors are called for all constructed static objects, in the reverse order to that of construction. If any functions have been registered to be executed at program termination using the library function `atexit()`, they are executed before destructors for static objects are called.

The library function `abort()` may be called to terminate a program without the normal program termination occurring. In this case, destructors are not called for static objects, and functions passed to `atexit()` are not executed.

CHAPTER 3

C++ LEXICAL TOKENS

A *token* is the minimal lexical element of the language. Different tokens are accepted during the preprocessing and the compilation phases. Chapter 14 provides more details about the preprocessor.

Tokens

The following kinds of token are recognized during compilation:

- keyword
- identifier
- literal
- operator
- punctuator

During compilation, white space and comments are ignored except when they serve as token separators. White space may appear within a token only as part of a character constant, string literal or header name. In such cases, the white space is significant.

If the input stream has been parsed into tokens up to a given character, the next token is the longest sequence of characters (not separated by white-space) that could constitute a token. Thus, the program fragment `x+++y` is parsed as `x ++ + y`, because `x` and `++` are both valid tokens. In contrast, `x++y` is parsed as `x + ++ y`.

The program fragment `0xG` is parsed as an invalid token, even though a valid parse might be obtained if the identifier `x`, or `G`, previously defined as a macro name, were replaced by its macro definition (e.g. if `x` were defined as `+`). Similarly, the program fragment `0xF` is parsed as a (valid) hexadecimal constant token, regardless of whether either `x` or `F` is a macro name.

Keywords

The following tokens are reserved in C++. These are called keywords, and they may not be used as identifiers or otherwise redeclared in the program.

| | | | |
|--------------------|---------------------|-----------------------|-----------------------|
| <code>asm</code> | <code>double</code> | <code>new</code> | <code>switch</code> |
| <code>auto</code> | <code>else</code> | <code>operator</code> | <code>template</code> |
| <code>break</code> | <code>enum</code> | <code>private</code> | <code>this</code> |

| | | | |
|----------|--------|-----------|----------|
| case | extern | protected | throw |
| catch | float | public | try |
| char | for | register | typedef |
| class | friend | return | union |
| const | goto | short | unsigned |
| continue | if | signed | virtual |
| default | inline | sizeof | void |
| delete | int | static | volatile |
| do | long | struct | while |

The keywords `catch`, `throw`, `try` and `template` are concerned with exception handling and templates. The definition of these language features is still experimental, and they are not supported by TopSpeed C++. However, the keywords are treated as reserved in order to minimize upheaval to existing code when these features are supported.

TopSpeed C++ has the following additional keywords, which may be disabled by means of the pragma option(`ansi=>on`):

```
cdecl    pascal
near     far
huge     interrupt
```

Identifiers

```
identifier:
  non-digit
  identifier non-digit
  identifier digit
non-digit: one of
  _ a b c d e f g h i j k l m
    n o p q r s t u v w x y z
    A B C D E F G H I J K L M
    N O P Q R S T U V W X Y Z
digit: one of
  0 1 2 3 4 5 6 7 8 9
```

Identifiers can contain

- Lowercase and uppercase letters
- Digits
- The underscore character

The first character in an identifier must be a letter or underscore.

The case of letters is significant in determining whether two identifiers are the same. For example, `abc`, `ABC`, `AbC` are treated as different identifiers.

Names may be internal or external. There is a limit of 1024 characters on the length of both internal or external names.

The following are all valid identifiers:

```
L001    _bufptr    WHILE
array    integer    Number_Of_Elements
```

Identifiers containing a double underscore and identifiers starting with an underscore should not be used by the programmer, as these are reserved by the TopSpeed C++ system for internal use.

In C++, an *identifier* can denote any of the following:

- An object
- A function
- A tag or a member of a structure, union, or enumeration.
- A typedef name.
- A label name.
- A macro name.
- A macro parameter.

Macro names and parameters are identifiers only during the preprocessing phase. A macro name may have the same spelling as a keyword. No other identifier may have the same name as a keyword.

Operators

```
operator: one of
[ ] ( ) . ->
+ - ~ ! / % < > ^ |
? : = , # sizeof
++ -- & *
<< >> <= >= == != && ||
*= /= %= += -= <<= >>= &= ^= |=
.* ->* ::
##
```

An *operator* specifies an operation to be performed on one or more *operands*. This operation yields a value. See Chapter 7 for more information about operators.

Punctuators

```
punctuator: one of
[ ] ( ) { } * , : = ; ... #
```

A *punctuator* is a symbol that has independent syntactic and semantic significance in a particular context. Unlike an operator, however, a punctuator does not specify a value-producing operation to be performed. Depending on context, the same symbol may represent a punctuator, an operator, or part of an operator.

Comments

In C++, a *comment* begins with `/*` and ends with `*/`, or starts with `//` and ends with a newline character. All characters between these two delimiters are treated as part of the comment. Comments are recognized anywhere in a program except in the following places:

- Within a character constant,
- Within a string literal,
- Within a comment

The contents of a comment are examined only to find the characters `*/` or newline that terminate the comment. Because of this, comments ordinarily cannot be nested.

As an extension, TopSpeed C++ provides an option to allow comments to be nested.

Literals

```
literal:
    floating-constant
    integer-constant
    enumeration-constant
    character-constant
    string-literal
```

Literals (or “*constants*”) are used to represent specific values that will not change during program execution. The type of a constant is determined by its form and value.

Floating Constants

```
floating-constant:
    fractional-constant exponent-partopt floating-suffixopt
    digit-sequence exponent-part floating-suffixopt
fractional-constant:
    digit-sequenceopt . digit-sequence
    digit-sequence .
exponent-part:
    e signopt digit-sequence
    E signopt digit-sequence
sign:
    +
    -
digit-sequence:
    digit
    digit-sequence digit
```

```
floating-suffix:
f
l
F
L
```

A *floating constant* consists of a value part, possibly followed by an exponent or a suffix or both. The *value*, or *significand*, part may include the following:

- *Whole number part*: a sequence of digits.
- *Fractional part*: a sequence of digits following a period.
- *Period*: if present, follows the whole number part and precedes the fractional part, if that is present.

Either the whole number part or the fractional part must be present.

The exponent part consists of e or E, followed by a digit sequence, which may be signed.

Either the period or the exponent part must be present.

Digit sequences are interpreted as decimal integers. The exponent indicates the power of 10 by which the value part is to be scaled.

A floating constant has type double, unless it is suffixed. A constant with f or F as a suffix has type float; a constant with l or L as a suffix it has type long double.

The following are all valid floating constants. The second example has type long double, and the fifth example has type float; the others have type double.

```
1.0e36    12e6L    .618    01e-9
34159f    42.
012e4
```

Integer Constants

```
integer-constant:
decimal-constant integer-suffixopt
octal-constant integer-suffixopt
hexadecimal-constant integer-suffixopt
decimal-constant:
non-zero-digit
decimal-constant digit
octal-constant:
0
octal-constant octal-digit
hexadecimal-constant:
0x hexadecimal-digit
0X hexadecimal-digit
hexadecimal-constant hexadecimal-digit
```

```

non-zero-digit: one of
1 2 3 4 5 6 7 8 9
octal-digit: one of
0 1 2 3 4 5 6 7
hexadecimal-digit: one of
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F
integer-suffix:
unsigned-suffix long-suffixopt
long-suffix unsigned-suffixopt
unsigned-suffix:
u
U
long-suffix:
l
L

```

An *integer constant* is a sequence of one or more digits. It may not contain a period or exponent part.

The constant may have a suffix that specifies its type. If the suffix is `u` or `U`, the constant is unsigned; if the suffix is `l` or `L`, the constant is a long int.

The constant may also have a prefix that specifies the constant's base as octal, decimal, or hexadecimal. Octal or hexadecimal constants begin with zero (`0`); otherwise the constant is considered a decimal constant.

| | |
|----------------------|--|
| Octal constant | Starts with the prefix <code>0</code> optionally followed by a sequence of the digits, <code>0</code> through <code>7</code> only. Such a value is computed base <code>8</code> . |
| Decimal constant | Starts with a nonzero digit optionally followed by a sequence of decimal digits. Such a value is computed base <code>10</code> . |
| Hexadecimal constant | Starts with the prefix <code>0x</code> or <code>0X</code> followed by a sequence of one or more decimal digits and the letters <code>a</code> (or <code>A</code>) through <code>f</code> (or <code>F</code>) representing values <code>10</code> through <code>15</code> , respectively. Such a value is computed base <code>16</code> . |

The type of an integer constant is the first type from a candidate list that can accurately represent the constant. The candidate list depends on the constant's base and suffixes. The following table shows the candidate lists for the possible base and suffix combinations.

```

Combination Candidate list (searched left to right)
Unsuffix decimalint, long, unsigned long
Unsuffix octalint, unsigned, long, unsigned long
Unsuffix hexadecimalint, unsigned, long, unsigned long
U or u suffix onlyunsigned, unsigned long
L or l suffix onlylong, unsigned long
U or u and L or l suffixesunsigned long

```

The following list shows some example values. An `int` is represented in `16` bits and a `long` is represented in `32` bits.

| Value | Type |
|--------------|---|
| 80000 | long int |
| 300000000 | unsigned long int |
| 040000 | unsigned int |
| 0100000 | long int |
| 0x0 | int |
| 0xfeed | unsigned int |
| 2u | unsigned int |
| 0xfacedU | unsigned long int |
| 30000000000l | unsigned long int |
| 034l | long int |
| 0lu | unsigned long int |
| 0x3ful | unsigned long int |
| Note: | Negative values such as -38 actually consist of the unary minus applied to a positive constant. The calculation is done at compile time. However, the type of the constant is decided before the unary minus is applied. Thus -32768 is actually a long int because 32768 is a long int (see unsuffixed decimal constants above). However, -0x8000 is an unsigned int because 0x8000 is unsigned int. |

Character Constants

character-constant:

‘ *c-char-sequence* ‘
L ‘ *c-char-sequence* ‘

c-char-sequence:

c-char
c-char-sequence c-char

c-char:

any character in the source character set except the single-quote ‘,
backslash \, or newline character
escape-sequence

A *character constant* is generally a single character within single quotes, but may consist of multiple characters, also within single quotes. The character may be any character in the source character set (except a single quote, a backslash, or a newline), or it may be a character specified by an escape sequence.

A wide character is the same except that it is prefixed by the letter L. Wide characters have type `wchar_t`, which is defined as unsigned char in TopSpeed C++.

Certain characters require special handling:

- A single quote must be represented by an escape sequence in which the quote is preceded by a backslash (`'\''`).
- A backslash must be prefixed with a backslash (`'\\'`).
- A double quote may be represented by the escape sequence (`'\"'`) or by using the character itself (`'\"'`).
- A question mark may be represented by the escape sequence (`'\?'`) or by using the character itself (`'?'`).

For example:

```
't' /* letter t */
'Q' /* letter Q */
'\'' /* single quote */
'\\' /* backslash */
'\"' /* double quote */
'\'' /* double quote */
```

Escape sequences can also consist of octal or hexadecimal values prefixed by a backslash (octal) or a backslash and an x (hexadecimal). For example:

```
'\0' /* character 0 */
'\222' /* character 146 */
'\xFE' /* character 254 */
```

Finally, the following escape sequences can be used to represent nongraphic character constants:

| Escape Sequence | Effect |
|----------------------------|---|
| <code>\a</code> (ASCII 7) | (Alert) Produces an audible alert (a bell). |
| <code>\b</code> (ASCII 8) | (Backspace) Backs up one character. |
| <code>\f</code> (ASCII 12) | (Form feed) Moves to start of next logical page. |
| <code>\n</code> (ASCII 10) | (Newline) Moves to start of next line. |
| <code>\r</code> (ASCII 13) | (Carriage return) Moves to start of current line. |
| <code>\t</code> (ASCII 9) | (Horizontal tab) Moves to next horizontal tab. |
| <code>\v</code> (ASCII 11) | (Vertical tab) Moves to next vertical tab. |

A character constant has type `char`. The value of a single-character constant is the numerical value of the character's representation.

TopSpeed C++ also allows multiple-character constants. The number of characters allowed is equal to the number of bytes used to represent a long int. Thus, up to four characters are allowed in character constants for

TopSpeed C++. TopSpeed C++ treats a character constant containing two characters as of type `int`, and one containing more than two characters as of type `long int`.

The values of individual characters in a multiple-character constant are placed in bytes from left to right in the object being used to store the value. For example, the constant `'abcd'` would be stored in four bytes as follows:

```
01100001 01100010 01100011 01100100
97 98 99 100
'a' 'b' 'c' 'd'
```

By default the type `char` is treated as a signed char. I.e., the high-order bit position of a single-character constant is treated as a sign bit. If the option `(uns_char=>on)` pragma is specified, a `char` is treated as an unsigned char.

String Literals

```
string-literal:
    " s-char-sequenceopt "
    L " s-char-sequenceopt "
s-char-sequence:
    s-char
    s-char-sequence s-char
s-char:
    any character in the source character set except the
    double-quote " , backslash \ , or newline
    escape-sequence
```

A *string literal* is a sequence of zero or more characters within double quotes. For example:

```
"39*23"
"a sample string literal"
""
```

are all string literals. The third one contains no characters.

A wide string literal is the same, except that it is prefixed by `L`. Each element in a wide string literal has type `wchar_t`, which is unsigned char in TopSpeed C++.

The rules that apply to each character in a string literal are almost the same as those for a character constant. In a string literal, a single quote can be represented by itself (`'`) or by an escape sequence (`\'`). The double quote, on the other hand, must be represented by an escape sequence (`\''`).

A string literal has static storage duration and is of type *array of char*. The string literal is initialized with the given characters. String literals that are adjacent tokens are concatenated into a single string literal during preprocessing. For example:

```
"One" "|" "Two"
```

is concatenated to the following string literal:

`"One|Two"`

A null character is appended onto the end of all string literals during compilation. Thus, the string literal,

`"TopSpeed"`

actually has nine characters.

A string literal containing a single character is *not* the same as a single-character constant. The value of the character constant is of type `char` (note that C++ differs from ANSI C in this respect - in ANSI C the type is `int`) and is the value of the character; the value of the string literal is the letter itself plus the null character at the end. The string literal is of type *array of char*, which may be converted to a *pointer to char*. Thus, the value of the string literal is actually the location of the first character in the string.

`TopSpeed` C++ will store a single copy of identical string literals. Thus, string literals should not be modified, as this may cause string literals in some other part of the system (even in another translation unit) to be modified.

Escape sequences are converted into single characters in the execution character set just prior to adjacent string literal concatenation.

The following are examples of string literals:

```
"Hello\n"  
"The bell\\a tolled"  
"0\\1\\2"  
"\\
```

CHAPTER 4

CONVERSIONS

Introduction

Many expressions can involve different types of operands. In such cases, *conversions* will need to be made, to bring operands into conformity with each other. For many operators, such conversions will be done automatically.

For example, when you want to add a short int and a long int, the former will automatically be converted to a long int before the addition is carried out. This process is known as *implicit conversion*, since it occurs without any explicit instructions. Nevertheless, implicit conversion does follow explicit rules, which are summarized in the following pages.

It is also possible to produce an *explicit conversion* by using a cast, as described in Chapter 7: ‘Cast Expressions’, or using a function style conversion expression - see Chapter 7: ‘Explicit Type Conversion’.

Regardless of whether a conversion is implicit or explicit, the conversion rules will always try to leave the operand’s value unchanged. If it is necessary to change a value, high bits of quantities will be discarded when converting to smaller types.

The compiler will sign extend when converting from a smaller signed type to a larger one. That is, the sign bit from the smaller type will become the sign bit in the larger value.

Arithmetic Operands

Conversions are very common when arithmetic operands are involved because such operands will often involve different types of values. For example, several types can be involved in expressions containing integer operators. Similarly, all these types as well as floating types can be involved in expressions using floating point operators.

Integral Promotions

In C++, characters are treated as an integral type. As a result, the following types can all be used in any context in which an int or unsigned int is allowed:

- A char
- A short int
- An int *bit-field*
- The signed or unsigned forms of the preceding types
- An object of enumeration type
- An enumerator

If an int can represent all values of the original type, the value is converted to an int. Otherwise, a value is converted to an unsigned int. This conversion is called *integral promotion*. The integral promotions preserve value, including sign. Whether a plain char is treated as signed depends on the `option(uns_char)` pragma.

Signed and Unsigned Integers

The value of an unsigned integer is unchanged when it is converted to another integral type, provided the value can be represented by the new type.

The value of a nonnegative signed integer is unchanged when it is converted to an unsigned integer of equal or greater size. Otherwise, the following two steps are taken:

1. If the unsigned integer is bigger, the signed integer is first promoted to a signed integer of the same size as the unsigned integer.
2. The value is converted to unsigned by adding to it one greater than the largest number that can be represented in the unsigned integer type.

In a two's-complement representation (as is used in TopSpeed C++), the only change in the bit pattern is that the high-order bits are filled with copies of the sign bit if the unsigned integer is longer.

When an integer is demoted to a shorter unsigned integer, the result is the nonnegative remainder on division by $1 + \text{maxrep}$, where `maxrep` is the largest unsigned number that can be represented in the shorter type.

When an integer is demoted to a shorter signed integer, or when an unsigned integer is converted to a signed integer of equal length, the result depends on whether the original value can be represented within the range of the target

type. If so, the value remains unchanged. If not, the least significant bits of the value are used to produce the result:

For example, since `int` is a 16-bit type, when `0x3fffff` (a signed long) is converted to a signed `int`, the result is `0xffff` (`= -1`); when this value is converted to an unsigned `int`, the result is also `0xffff` (`= 65535`).

Floating and Integral Values

The value of the fractional part is discarded when an object of floating type is converted to integral type. If the value of the resulting integral part cannot be represented in the storage space assigned to the integral type, the result is undefined..

When a value of integral type is converted to floating type, the value being converted will be within the range for the floating type, but will not necessarily be representable by an exact value. In that case, the result will be the nearest higher or lower value.

Floating Point Values

The value of a float is unchanged when it is promoted to double or long double. Similarly, the value of a double is unchanged when it is promoted to long double.

When a double is demoted to float, or a long double to double or float, the result depends on whether the original value is within the range of representable values for the smaller type.

If the value being converted is outside this range, the result is undefined. If the value is within in the range, but cannot be represented exactly, the result is either the nearest higher or nearest lower value, chosen according to the IEEE Standard.

Usual Arithmetic Conversions

The term *usual arithmetic conversions* refers to the process of converting the operands of an operator to the same, defined type. Most operators that work on integral operands first apply these conversions to the operands. The purpose of these conversions is to reduce the possible combinations of operands on which an operator might actually have to operate. The type produced after performing the usual arithmetic conversions is also the type of the resulting value for most operators. (See the discussion about these operators in Chapter 7 for exceptions to this rule.)

In general, conversions are to the higher operands, using a hierarchy with long double at the top, and `int` at the bottom.

IF either operand has type long double, the other operand is converted to long double.

| | |
|---------|---|
| ELSE IF | either operand has type double, the other operand is converted to double. |
| ELSE IF | either operand has type float, the other operand is converted to float. |
| ELSE | the integral promotions are performed on both operands, then ... |
| IF | either operand has type unsigned long int, the other operand is converted to unsigned long int. |
| ELSE IF | either operand has type long int, the other operand is converted to long int. |
| ELSE IF | either operand has type unsigned int, then the other operand is converted to unsigned int. |
| ELSE | both operands have type int. |

For example:

```

/* given the following definitions: */
long double   ld1 = 1.0;
double        d1 = 1.0;
float         f1 = 1.0;
unsigned long ul1 = 1;
long          l1 = 1;
unsigned      u1 = 1;
int           i1 = 1;
// the following conversions will be made
// when the specified expressions are evaluated:
// f1 is converted to long double;
// l1 is converted to double;
// quotient is converted to long double.
ld1 = f1 * ld1 + d1 / l1;
// i1 is converted to unsigned;
// product is converted to long;
// sum is converted to unsigned long
ul1 = u1 * i1 + l1;

```

Pointer Conversions

An operator that requires an operand of pointer type may apply any of the implicit conversions described below to the type of its operand. Other pointer conversions may be made explicitly, as described in Chapter 7: ‘Explicit Type Conversion’.

- A constant expression with value 0 can be implicitly converted to any pointer type. This is called a null pointer, and will not compare equal to any pointer that points to an object.
- A pointer to any object type that is neither const nor volatile can be implicitly converted to type void*. Note that in C++ (unlike in ANSI C) the reverse conversion is not made implicitly, and must be made explicit if required.
- A pointer to a function type can be implicitly converted to type void*.

- A pointer to a derived class can be implicitly converted to a pointer to an accessible base class provided that the conversion is unambiguous. The converted pointer will point at the sub-object within the derived class that represents the base class. The null pointer converts to itself.
- An expression of array type can be implicitly converted to a pointer to the first element of the array.
- An expression of function type is always implicitly converted to the corresponding pointer to function type, except when used as the operand of the & or () operators.

Reference Conversions

When a reference is initialized (including initialization by argument passing or function return), a reference to a derived class can be implicitly converted to a reference to an accessible base class, provided that the conversion is unambiguous. The converted reference will be to the sub-object within the derived class that represents the base class.

Other reference conversions may be made explicitly - see Chapter 7: 'Explicit Type Conversion'.

An operator that requires an operand of pointer-to-member type may apply any of the implicit conversions described below to the type of its operand.

- A constant expression with value 0 can be implicitly converted to any pointer-to-member type. The result of the conversion will not compare equal to any pointer-to-member that points to a member.
- A pointer to a member of an accessible base class can be implicitly converted to a pointer to a class derived from it, provided that a pointer to the derived class can be implicitly converted into a pointer to the base class.

Note that a pointer to member is not a pointer to an object, and the rules described earlier for conversions that are applied implicitly to pointers do not apply.

CHAPTER 5

DECLARATIONS

Introduction

In C++, a declaration is used to introduce a name into a scope, and indicates how the name is to be interpreted in future. A name may be declared to be associated with a type, a variable, a function, or a constant.

declaration:

```
decl-specifiersopt declarator-listopt ;  
asm-declaration  
function-definition  
linkage-specification
```

The *asm-declaration*, *function-definition*, and *linkage-specification* forms of a declaration are described in later in this chapter.

The *decl-specifiers* part of a declaration indicates the type, linkage, and storage class of the objects contained in the *declarator-list* part - the declarators forming the declarator list contain the names being declared, and may further modify the type for each name. The *decl-specifiers* may only be omitted for a function declaration. The *declarator-list* may only be omitted for a class or enumeration declaration. Declarators are described in Chapter 6.

A declaration specifies several things about the entity being declared.

type An object's type determines the amount of storage required for an object or for a function's return value. By specifying the type, a declaration also specifies how to interpret an identifier in an expression.

linkage An object's linkage determines where an identifier can be referenced - i.e., whether only within a translation unit or throughout the program. The linkage can have values *extern* or *static*.

lifetime An object's lifetime, or storage duration, specifies whether an object continues to exist after its containing function or block finishes executing, or

whether the object must be created each time its declaration is executed. The lifetime can have values *auto* or *static*.

scope The location of a declaration determines whether the declaration is valid just for the function or for the entire module within which the declaration occurs. An object's *scope* can be file, block, class, function, or function prototype (see Chapter 2: 'Scopes of Identifiers').

initial value It is possible to specify an initial value for an object, by including this value as part of the object's declaration.

A *definition* is a declaration that also causes storage to be reserved for the object or function named by an identifier. A declaration with initialization is always a definition. A *declarator-list* is a sequence of declarators, separated by commas. Each of these declarators may have associated type information or an initializer, or both.

Linkage and storage duration make up the *storage class* for the declaration. Either a linkage or a type must be provided for an identifier. Thus, a declaration must specify either the storage class (i.e., the linkage) or the type - or both - for an identifier.

If an identifier has no linkage, there may be no more than one declaration of that identifier within a given scope and name space. There is an exception to this rule, where an identifier may be declared as a class name in the same scope in which it is declared as a variable or type. This exception is provided for compatibility with ANSI C - see also Chapter 9.

All declarations that refer to the same object or function within the same scope must specify compatible types.

Declaration Specifiers

```
decl-specifiers:
    decl-specifier
    decl-specifiers decl-specifier
decl-specifier:
    storage-class-specifier
    type-specifier
    fct-specifier
    friend
    typedef
```

The longest sequence of *decl-specifiers* that could form the *decl-specifiers* part of a declaration is taken. This resolves some ambiguities which can otherwise arise between declaring an object of a typedef name and redeclaring a typedef name.

The *friend* specifier is used to control access to class members, and is described in Chapter 11: 'Friends'.

Storage-Class Specifiers

```
storage-class-specifier:  
    extern  
    static  
    auto  
    register
```

The *storage-class specifier* determines the lifetime and linkage of a declared object. At most one storage-class specifier may be given in the declaration specifiers for a declaration. A storage class specifier may only be used in a declaration of an object or function.

auto The object has local lifetime - i.e., within a function - and no linkage. This specifier is not allowed outside function definitions. As **auto** is the default inside a function definition, this specifier is usually redundant, but may be useful to explicitly distinguish a declaration from an expression statement - see Chapter 8: ‘Expression Ambiguity’.

register This specifier follows the same rules as **auto**. Its purpose is to give a hint from the programmer to the code generator that the object being declared is frequently used, and that an attempt should be made to keep the object’s value in a register for as long as possible. TopSpeed C++ does its own optimal register allocation, so the register specifier is always equivalent to **auto**.

Note that in C++ there is no restriction on taking the address of a register variable, unlike ANSI C.

extern The object has static storage duration. The linkage may be internal or external, as described below. A class member may not be declared **extern**.

static The object has static storage duration and internal linkage. A formal argument may not be declared **static**. A function declaration within a block may not be **static**. A non-member function declared with the **inline** specifier and not declared **static** behaves as if it was declared **static**.

The linkage for an object declared **const** is internal, unless it has already been given external linkage. The linkage for an object declared **extern** is external, unless it has already been given internal linkage. An object declared without a storage class specifier in file scope, and not declared **const** or **inline**, has external linkage unless it has already been given internal linkage.

If a name is declared with both internal and external linkage in the same scope, TopSpeed C++ will report an error.

Function Specifiers

fcn-specifier:

virtual
inline

The specifiers virtual and inline may only be used in a function definition or declaration.

The virtual specifier may only be used for a non-static member function. Its meaning is described in Chapter 10: ‘Virtual Functions’.

The inline specifier is used to indicate that the function should be substituted inline where possible rather than being called. A non-member function declared inline has internal linkage.

A member function that is defined within the class declaration is always inline, and the inline specifier is redundant here. A member function that is not defined within the class declaration, and not declared inline in the class declaration, has external linkage, unless a definition with the inline specifier appears before the first call of the function. An inline member function must have the same definition in every translation unit in which it appears in a program.

Typedef Declarations

A declaration in which the typedef declaration specifier appears is a *typedef declaration*. Any identifiers in the declarator list are declared as *typedef-names* rather than as object - the identifier becomes equivalent to a keyword within its scope, and names the type that would be given to a variable so declared had the typedef specifier been absent.

A typedef-name may be declared more than once in the same scope, provided that the type to which it refers is identical in each declaration.

If a typedef is used to declare a typedef-name for an unnamed class, the typedef-name is also given to the class. For example:

```
typedef struct {
    // ...
} mystruct;
```

is equivalent to

```
struct mystruct {
    // ...
};
```

A typedef-name that is declared as a synonym for a class-name, for example:

```
typedef mystruct tname;
```

is itself a class name. However, it may not be used in an elaborated type specifier (i.e., after the class, struct, or union keywords), nor may it be used to name constructors or destructors within the class declaration. It may, however, be used to name constructors and destructors when they are explicitly called, or when they are defined outside the class declaration.

A typedef-name that is declared as a synonym for an enumeration may not be used in an elaborated type specifier (i.e., after the enum keyword).

```

Type Specifiers
type-specifier:
    simple-type-name
    class-specifier
    enum-specifier
    elaborated-type-specifier
    const
    volatile

simple-type-name:
    void
    char
    short
    int
    long
    float
    double
    signed
    unsigned
    complete-class-name
    qualified-type-name
qualified-type-name:
    typedef-name
    class-name :: qualified-type-name
complete-class-name:
    qualified-class-name
    :: qualified-class-name
qualified-class-name:
    class-name
    class-name :: qualified-class-name

```

At most one *type-specifier* may appear in the *decl-specifiers* in a declaration with the following exceptions:-

- The type-specifiers `const` and/or `volatile` may be added to any legal type specifier in a declaration of an object (but not of a type).
- The type-specifier `int` may appear with the specifiers `short` or `long`. The type-specifier `double` may appear with the specifier `long`. If either `short` or `long` appear alone, `int` is assumed.
- The type-specifiers `signed` or `unsigned` (but not both) may appear with the specifiers `char`, `short`, `int`, or `long`. If either `signed` or `unsigned` appear alone, `int` is assumed. Any of the types `short`, `int` or `long` appearing without `unsigned` is assumed to be signed. Whether `char` specified alone is assumed to be signed or unsigned can be controlled by the option `(uns_char)` pragma.

An object declared with the type specifier `const` may be initialized, but the value cannot subsequently be changed. Unless specified extern, a `const` object must be initialized, and will not have external linkage.

Every element of an array declared `const`, and every non-static data member of a class declared `const`, is itself `const`.

An object of `const` integer type which is initialized by a constant expression can itself be used in constant expressions.

A `const` object that has no constructor or destructor may be placed in read only memory by TopSpeed C++.

If an object is declared with the type specifier `volatile`, TopSpeed C++ will always keep it in memory rather than registers, and will not perform any optimizations that assume the object is not modified between one reference to it and the next.

For a description of the *class-specifier* type specifier, see Chapter 9. For a description of the *enum-specifier* type specifier, see later in this chapter. A *qualified-type-name* may be used to specify a nested type - see Chapter 9: 'Local Type Names'.

Elaborated Type Specifiers

```
elaborated-type-specifier:
  class-key class-name
  class-key identifier
  enum enum-name
class-key:
  class
  struct
  union
```

An *elaborated-type-specifier* can be used to indicate explicitly that a name is being used as a *class-name* or *enum-name* in a declaration, and consists of the name in question prefixed by one of the keywords `class`, `struct`, `union` or `enum`.

An *elaborated-type-specifier* must be used when the class name has also been declared as a non-class entity in the same scope (the extension to the scope rules to allow overloading of class names with non-class names is necessary for compatibility with ANSI C, where tags occupy a different name space from other identifiers).

An elaborated type-name may also be used to declare an identifier to be a class name, without declaring the class itself. Objects of the named class cannot be declared until the class has been declared, but pointers or references to the class may be declared. This is useful to declare mutually referential structures. For example:

```

struct A;

struct B {
    struct A * Aptr;
}

struct A {
    B * Bptr;
}
Enumeration Specifiers
enum-specifier:
    enum identifieropt { enum-listopt }
enum-list:
    enumerator
    enum-list , enumerator
enumerator:
    identifier
    identifier = constant-expression

```

An *enumeration* consists of a list of named integer constant values. The identifiers in an enumeration list are declared as constants, and may be used wherever a constant is required. The name of an enumeration becomes an *enum-name*, and may not be redeclared within its scope. The names of enumerators and enumerations occupy the same name space as other identifiers - in particular, an enumeration does not introduce a new scope, so that the names of all enumerators must be distinct from other identifiers declared in the same scope.

Each enumeration is a distinct integral type (unlike in ANSI C). A value of enumeration type (such as an enumerator) may be implicitly converted to an integer, but there are no standard implicit conversions to enumeration types. No standard operators other than simple assignment are defined for enumeration types, though many operators will accept enumerations as operands after (implicitly) converting them to int.

Each enumeration constant is associated with a particular integer value. In the default case the enum constants are assigned values starting from zero in increments of 1.

It is possible to reset this sequence by explicitly assigning a particular integer value to one or more of the enum constants. This is done by specifying = and the integer value after the enumeration identifier. An expression used to give a value to an enumeration identifier in an enum list must be an integral constant expression of type int.

For example:

```

enum numbers {
    zero, one, two, three
};
enum roman_enum {
    I=1, V=5, X=V+V, L=V*X, C=L+L, D=C*V, M=X*C
} date;

```

The values of the enumeration constants declared above are as follows: zero is 0, one is 1, etc. In the second declaration, X is 10, L is 50, etc.

The values of the constants need not be ordered or unique within the specifier. As illustrated in the example above, an enumerator is defined immediately after it is encountered, and can be referred to in the initialization expressions of subsequent enumerators in the list.

An enumeration defined in a class, and its enumerators, can only be accessed in the scope of that class (i.e., within member functions, or after explicit qualification with the class name).

Asm Declarations

An *asm declaration* is provided in the Base Language to permit implementations in which assembly takes place as a separate phase from compilation to pass information through to the assembler. This feature is not relevant to TopSpeed C++, so any asm declarations encountered are simply ignored. The syntax of an asm declaration is as follows :-

```
asm-declaration:
    asm ( string-literal ) ;
```

TopSpeed C++ provides an extension whereby a function may be specified as machine-code - see Chapter 6: ‘Initializing Functions’.

Linkage Specifications

A *linkage specification* is used to control linkage to languages other than C++. A *linkage specification* takes the form

```
linkage-specification:
    extern string-literal { declaration-listopt }
    extern string-literal declaration
declaration-list:
    declaration
    declaration-list declaration
```

The string literal indicates the linkage convention required, and applies to all declarations within the *declaration-list*. In TopSpeed C++, the following linkage specifications are supported. Any string not in the list below will be reported as an error by TopSpeed C++.

```
.i.linkage:"C++";
```

“C++”

This is the default linkage convention. Declarations made within a “C++” linkage specification behave as they would if the linkage specification were not present, except as described below.

| | |
|-----------|--|
| “C” | This provides linkage to functions and objects declared in C object modules. This has the effect of suppressing the name-encoding of function arguments. |
| “Pascal” | This provides linkage to functions and objects declared in Pascal object modules. Objects declared with “Pascal” linkage have a different prefix applied to their names, as described below, and the name is converted to upper case. Name encoding of function arguments is suppressed. |
| “Modula2” | This provides linkage to functions and objects declared in Modula-2 object modules. Objects declared with “Modula2” linkage have a different prefix applied to their names, as described below, and name encoding of function arguments is suppressed. |

As functions declared with linkage other than “C++” do not have their argument profiles encoded into their names, at most one of a set of overloaded functions may have non-C++ linkage.

A “Pascal” or “Modula2” linkage specification will suppress the current name prefix (by default, in C++, this is a single underbar, but can be changed by means of the prefix pragma). In order to link to names declared in a given Modula or Pascal file, the prefix should be specified to be the name of that file, followed by a \$ for function names or a @ for data names. In order to simplify this process, a Modula or Pascal linkage specification can be specified as “Modula2.modname” or “Pascal.modname”, where modname names the Modula or Pascal file in which the object’s definition is to be found.

An object or function declared with the static specifier within a linkage specification uses the default (i.e. “C++”) linkage, and the linkage specification is ignored for this object. A function declared within a linkage specification, if not explicitly declared static, behaves as if it was explicitly declared extern.

Linkage specifications are only permitted in file scope. A linkage specification does not introduce a new scope, and linkage specifications may be nested. A linkage specification for a function or class also applies to non-member functions or objects declared within it.

No function or object may be declared with more than one linkage specification unless the linkage specifications are identical. If a function declared with a linkage specification is subsequently redeclared with no linkage specification, the linkage of the function is not affected. It is an error for a function already declared without a linkage specification to be redeclared with one.

Functions and objects declared within a linkage specification may be defined, either at the point of declaration or subsequently. For example, a

C++ function might be defined with the “C” linkage specification if it was to be called from a C program, whereas a C function that was to be called from C++ would be declared (but not defined) with a “C” linkage convention in any C++ translation unit that called it. A function that is declared and defined with a linkage other than “C++” can still be called from C++ code in the normal way.

For example:

```
extern "C" printf(const char * ...);
extern "Modula2.test" {
    modfunc1(int, int);
    modfunc2(unsigned long);
}
```

CHAPTER 6

DECLARATORS

Introduction

As described in Chapter 5, a declaration consists of two parts; the *decl-specifiers* and the *declarator-list*. The *decl-specifiers* define the storage class, linkage, and base type of the entities being declared. The *declarator-list* names the entities being declared, and may supply additional type information and/or initial values for each entity. A *declarator-list* is a comma-separated list of declarators, each with an optional initializer :

```

declarator-list:
    init-declarator
    declarator-list , init-declarator
init-declarator:
    declarator initializeropt
declarator:
    ptr-operatoropt direct-declarator
direct-declarator:
    modifiersopt dname
    declarator [ constant-expressionopt ]
    declarator ( argument-declaration-list ) cv-qualifier-listopt
    ( declarator )

ptr-operator:
    modifiersopt * cv-qualifier-listopt
    modifiersopt & cv-qualifier-listopt
    modifiersopt complete-class-name :: * cv-qualifier-listopt
cv-qualifier-list:
    cv-qualifier
    cv-qualifier-list cv-qualifier
cv-qualifier:
    const
    volatile
dname:
    name
    class-name
    ~ class-name
    typedef-name
    qualified-type-name
modifiers:
    modifier
    modifiers modifier

```

```

modifier:
    cdecl
    far
    huge
    interrupt
    near
    pascal
    < expression >

```

Each declarator in a declaration defines a single object, function, or typedef name. A declarator will contain a single *dname*, which specifies the name of the identifier being declared - in most cases this is a simple identifier. Other special function names are used for declaring special member functions or overloaded operator functions - these are described in Chapters 12 and 13.

The type specifiers in the *decl-specifiers* part of a declaration apply to every name declared in the declarator-list.

Consider a declaration of the form

T D

where T represents the decl-specifiers and D is a declarator.

If D is an identifier, then the identifier is declared to be of the type specified by T.

If D is of the form (D1), the type of D is the same as D1.

Pointer Declarators

In a declaration T D, a declarator D of the form

```

modifiersopt * cv-qualifier-listopt D1

```

declares D1 to be of type *modifiers cv-qualifier-list pointer to T*. If the *cv-qualifier-list* contains the qualifiers const and/or volatile, these attributes apply to the pointer (rather than the object pointed to). Any modifiers specified in *modifiers* also apply to the pointer rather than the object pointed at. For example:

```

int i;
int *const cpi=&i; // const pointer to int
int far *f;      // far pointer to int

*cpi = 3;        // ok - object pointed at is not const
cpi++;           // error - cpi is const

```

A pointer to a reference may not be declared, nor may a pointer to a bit-field.

TopSpeed C++ supports the following *modifiers* which may be used in pointer declarators :-

| | |
|-----------------------|--|
| near | indicates that the pointer is a two-byte pointer, containing just an offset, which can only point to objects in the default data segment, or the default code segment for a pointer to a function. |
| far | indicates that the pointer is a four-byte pointer containing both a segment and an offset, and can point to objects outside or inside the default data or code segment. |
| huge | also indicates a four-byte pointer, but the object pointed to may occupy more than one segment, so the pointer must be normalized after any arithmetic is performed with it. |
| cdecl | for a pointer to function, indicates the function pointed at uses the standard C calling conventions. |
| pascal | for a pointer to function, indicates that the function pointed at uses the Pascal calling conventions. |
| interrupt | for a pointer to function, indicates that the function pointed to is an interrupt function. |
| < <i>expression</i> > | indicates the declaration of a <i>based pointer</i> . Based, or relative, pointers are two-byte pointers, but rather than using the current default segment (i.e., the contents of the DS register) to provide the segment part, the segment is calculated from the nominated expression each time such a pointer is dereferenced. The expression must be an lvalue of type int or unsigned. |

The effects of all these modifiers depend upon the memory model and the current pragma settings. See the *Developer's Guide* for further details. All the modifiers mentioned above are TopSpeed C++ extensions to the Base Language, and can be disabled by means of the pragma option(ansi=>on).

Reference Declarators

In a declaration T D, a declarator D of the form

*modifiers*_{opt} & *cv-qualifier-list*_{opt} D1

declares D1 to be of type *modifiers cv-qualifier-list reference to T*.

References to references, bit-fields, or the type void may not be declared. An array of references cannot be declared, nor can a pointer to a reference.

A declaration of a reference must contain an initializer, except where the reference is

- Explicitly declared extern
- A class member declared within a class declaration
- An argument or return type of a function

TopSpeed C++ supports the following *modifiers* which may be used in reference declarators :-

| | |
|-----------|--|
| near | indicates that the reference is a two-byte reference, containing just an offset, which can only reference objects in the default data segment. |
| far | indicates that the reference is a four-byte reference containing both a segment and an offset, and can reference objects outside or inside the default data segment. |
| cdecl | for a reference to a function, indicates the function referenced uses the standard C calling conventions. |
| pascal | for a reference to a function, indicates that the function referenced uses the Pascal calling conventions. |
| interrupt | for a reference to a function, indicates that the function referenced is an interrupt function. |

See the *Developer's Guide* for further details of the effects of different memory models and pragmas on these modifiers. All the modifiers mentioned above are TopSpeed C++ extensions to the Base Language, and can be disabled by means of the pragma option(ansi=>on).

Pointer-to-Member Declarators

In a declaration T D, a declarator D of the form

***modifiers*_{opt} *class-name* :: * *cv-qualifier-list*_{opt} D1**

declares D1 to be of type *modifiers cv-qualifier-list pointer to member of class class-name of type T*.

TopSpeed C++ supports the following *modifiers* which may be used in pointer-to-member declarators :-

| | |
|--------|--|
| cdecl | for a pointer to function, indicates the function pointed at uses the standard C calling conventions. |
| pascal | for a pointer to function, indicates that the function pointed at uses the Pascal calling conventions. |

See the *Developer's Guide* for further details of the effects of different memory models and pragmas on these modifiers. All the modifiers mentioned above are TopSpeed C++ extensions to the Base Language, and can be disabled by means of the pragma option(ansi=>on).

Array Declarators

In a declaration $T\ D$, a declarator D of the form

D1 [*constant-expression*_{opt}]

declares $D1$ to be of type *array of T*. The constant expression, if present, must be integral, and greater than zero. This specifies the number of elements in the array. Array elements in C++ are numbered from zero to $N-1$, where N is the value specified in the array declaration.

The type T from which the array is derived may be a fundamental type other than void, a pointer type (including pointers to functions), a pointer-to-member type, a class, an enumeration, or another array type. It may not be type void or a reference type, nor may it be a type (such as an undeclared class) whose size is unknown.

For example:

```
double dvals [35]; // 35-element array of double
complex cvals [5]; // 5-element array of complex
int fvals []; // array of unspecified # of int
```

.i.array:multidimensional;

When several *array of* specifications are adjacent, a multi-dimensional array is declared.

```
double two_d[2][3];
```

declares a two-dimensional array of double. The declaration

```
double *ptr_array[5];
```

specifies a 5-element array of *pointers to* double values.

For two arrays to be compatible their element types must be compatible. If both array sizes are given they must be equal.

In the following contexts (only) an array bound may be omitted:

.i.array bounds;

- When an array is being declared as a parameter of a function.
- When the array declaration has storage-class specifier *extern* and the definition that actually allocates storage is given elsewhere.
- In a multi-dimensional array declaration, only the first size may be omitted.

- When the declarator is followed by initialization. In such a case, the size is determined by the number of initializers supplied.
- In an incomplete type, such as `int a[]`. In such a case, the size must be given later in a declaration.

If the size is not present, the array type is an incomplete type. In the declaration

```
extern int b[];
```

`b` is declared as an incomplete array of `int`.

Function Declarators

In a declaration `T D`, a declarator `D` of the form

```
D1 ( argument-declaration-list ) cv-qualifier-listopt
```

declares `D1` to be of type *cv-qualifier-list function taking arguments of type argument-declaration-list returning type T*. Note that both the argument types (other than default arguments - see below) and the return type are part of the functions type - a function taking an `int` parameter and returning `int` is not the same type as a function taking a `short` parameter and returning `int`.

The syntax of the *argument-declaration-list* is as follows:

```
argument-declaration-list:
    arg-declaration-listopt ...opt
    arg-declaration-listopt , ...
arg-declaration-list:
    argument-declaration
    arg-declaration-list , argument-declaration
argument-declaration:
    decl-specifiers declarator
    decl-specifiers declarator = expression
    decl-specifiers abstract-declaratoropt
    decl-specifiers abstract-declaratoropt = expression
```

An empty argument list, or an argument list consisting simply of `(void)`, specifies that a function takes no arguments. An argument list terminated by an ellipsis `...` specifies that a function may take an unspecified number of arguments in addition to those specified. The presence or absence of an ellipsis is part of the function type. The optional comma preceding an ellipsis has no effect on the semantics. Additional arguments passed using an ellipsis may be accessed using the functions and macros defined in `<stdarg.h>` - see the Library Reference Manual for details.

A *cv-qualifier-list* is permitted in a function declaration for non-static member functions (only). The meaning is described in Chapter 9: ‘Member Functions’. The qualifiers in the list form part of the function type.

The function return type is specified by the type-specifiers in T. If no type is specified, `int` is assumed. A function may be declared to return no value by specifying the return type `void`. A function may not return an array or function type (though it can return a reference or pointer to such a type).

An array of functions may not be declared, although an array of pointers to functions can.

It is not legal to define a new type in the argument types or return types of a function, as such a type would not be in scope except within the function itself, and so the function could never be called.

If an expression of function type (i.e. an identifier declared using a function declarator) is used other than as the operand of the function call operator `()` or the address-of operator `&`, it is implicitly converted to a pointer to the function.

The argument-declaration-list is used when a function is called to convert the supplied arguments into the correct types. It is an error in C++ to call a function for which no declaration is in scope. The argument list is also used to distinguish overloaded functions (functions with the same name) - see Chapter 13.

The decl-specifiers in an argument declaration may contain type-specifiers, or the storage class specifier `register`. Other specifiers will be reported as an error.

Each argument in the argument-declaration-list may optionally be named. If the function declaration forms part of a function definition, these names are used to access the formal arguments. Argument names in a function declaration may not be used other than for documentation - they go out of scope at the end of the declaration. There is no requirement that the names used to name function arguments match if the function is declared more than once. Argument names may be omitted in a function definition if the formal argument is not used within the function body.

The following examples illustrate some function declarations:

```
int i();           // no parameters, returning int
int *pi(int);     // int parameter, returning int*
int (*pf)(int);   // pointer to function with int
                  // parameter returning int
cplex cf(void);   // no parameters, returning cplex
int pf(int, ...); // int and optional other
                  // parameters, returning int
```

Default Arguments

In a function declarator, an argument declaration which includes an expression specifies a *default argument*. A default argument may only be specified if all subsequent arguments in the list also have default arguments,

in this or a previous declaration of the function. A default argument may not be specified for an argument where a previous declaration of the same function has already specified a default for that argument, even if the expressions specified are identical.

A function declared with default arguments is similar to a set of overloaded functions, as in the following example :-

```
int h(int = 4, double = 0);
```

is roughly equivalent to

```
int h(int, double);
inline int h(int i) { return h(i, 0);
inline int h() { return h(4); };
```

except that in the first case the address of h may not be assigned to a variable of type *pointer to function taking no arguments returning int*, while in the second it can.

Note that the function could have been declared

```
int h(int, double);
int h(int, double=0);
int h(int=4, double);
```

but not

```
int h(int, double);
int h(int=4, double); // error: default not at end
int h(int, double=0);
```

nor

```
int h(int, double);
int h(int, double=0);
int h(int=4, double=0); // error: default redeclared
```

The expressions specified in a default argument are evaluated each time the function is called without the corresponding argument being specified. Default expressions are evaluated in the scope at which they are defined, rather than that where they are called. A local variable may not be used in a default expression, nor may a formal argument of a function, even though these may be in scope at the point of definition. A default expression in a class member function may not refer to non-static class members.

The presence of default argument values for a function does not affect the type of the function, which is the same as if all arguments were specified without default values.

Overloaded operator functions may not have default arguments.

Declarators with Special Keywords

TopSpeed C++ provides several extensions in the form of a variety of additional C++ keywords. These may be used to modify declarations of variables, pointers and functions. (Such keywords can be disabled by the pragma option(ansi=>on).)

cdecl, far, huge, interrupt, near, pascal

Here we only focus on the syntax of how to specify the modifiers in declarations. For a complete discussion of the effect of the various modifiers see the discussions of mixed model programming (near, far, huge), mixed language programming (cdecl, pascal) and interrupt functions in the *Developer's Guide*.

Generally, a modifier affects the entity immediately to the right. More than one modifier can modify the same entity.

- The near, far and pascal keywords can modify variable declarations.
- The near, far, huge, interrupt, cdecl and pascal keywords can modify pointer declarations.
- Function declarations can be modified with the near, far, interrupt, cdecl and pascal modifiers.
- The effect of all the keywords (except huge) can also be achieved using pragmas. See the *Developer's Guide* for more information about pragmas.

Variable Declarations

The near and far keywords can be used to override the memory model. The near modifier forces a variable to be allocated in the default data segment regardless of the memory model used. The far keyword forces a variable to be allocated in a segment of its own.

For example, if you are using the small model, you can do the following to make sure a variable is not allocated in the default data segment:

```
int far store[10000]; /* far modifies store */
```

If you are using one of the large data models, the near keyword can force a variable to be allocated in the default data segment, even if the size of the variable is larger than the data threshold. For example:

```
int near table[18000]; /* near modifies table */
```

A modifier affects only the variable immediately to the modifier's right. For example, in the following declaration only x is declared to be far.

```
long far x, y;
```

The pascal modifier can also be used in variable declarations. For example:

```
extern unsigned pascal x;
```

The effect is that x has (traditional) Pascal linkage. I.e., the external name is in upper case (since Pascal is not case sensitive), no `_` is put in front of x.

Note: the near and far modifiers cannot be used for declarations of function parameters and local variables (unless declared as static or extern) because these are always allocated on the run-time stack.

Pointer Declarations

There are two kinds of pointers, near (16-bit) pointers and far (32-bit) pointers. The size of the pointers depends on which memory model is being used.

The default pointer size can be modified by the near and far modifiers:

```
char near *npc;
```

This declares npc to be a near pointer - i.e., it points to a character in the default data segment. Note that near modifies the `*`.

```
int far *fpi;
```

Here fpi is a far pointer to an int

```
long * near * nppl;
```

The near keyword modifies the `*` to the right, so nppl is a near pointer to a pointer to a long.

```
int far * near fpi;
```

The pointer variable fpi itself is allocated in the default data segment because of the near modifier; it is a far pointer to an int.

```
int far *fp1, far *fp2, near *np;
```

Several declarators with modifiers can be given in the same declaration, fp1 and fp2 are far pointers, np is a near pointer.

```
long huge *hp;
```

This declares hp to be a pointer to a huge object - i.e., an object that can have a size large than 64K. Such an object must be allocated using the halloc function (see the TopSpeed C Library Manual).

Declarations of pointers to functions can have the following additional modifiers: cdecl, pascal and interrupt. There are no huge pointers to functions.

```
int (near *npf)(char);
```

This declares npf to be a near pointer to a function. For example, the call

```
(*npf)(a)
```

is done with a near function call.

```
void (interrupt *pif)();
```

This declares pif to be a pointer to an interrupt function

```
int (far pascal *pf)()
```

Here the * has two modifiers far and pascal. The order is not significant. pf is a far pointer to a function using the Pascal calling convention.

Function Declarations

Function declarations can be modified by the near, far, cdecl, pascal and interrupt keywords.

```
unsigned near func();
```

This declares func to be a near function - i.e., it is called with a near call.

```
double pascal far func2(int);
```

func2 is a far function; it has Pascal calling convention and linkage. The order of the pascal and far modifiers are insignificant.

```
int cdecl func3();
```

This declares func as using the traditional C calling convention,.

```
void interrupt far MyIntr();
```

This declares MyIntr to be an interrupt function,.

Abstract Declarators and Type Names

```
type-name:
    type-specifier-list abstract-declaratoropt
type-specifier-list:
    type-specifier type-specifier-listopt
abstract-declarator:
    ptr-operatoropt direct-abstract-dtor
```

direct-abstract-dtor:

```
( abstract-declarator )
direct-abstract-dtoropt [ constant-expressionopt ]
direct-abstract-dtoropt ( argument-declaration-list )
cv-qualifier-listopt
```

In C++, a type name is required in three situations:

- In cast expressions.
- When applying sizeof to a type.
- In a new expression.

A type name is not a typedef name (see below). Rather, a type name is basically a declaration that omits the identifier being declared. The type specified by the type name is the type that would be given to the identifier, had it been present.

Empty parentheses in a type name are interpreted as a function with no parameters (not as redundant parentheses).

Several cases are shown in the following list of constructions and what they name illustrate.

`char`

a `char`.

`unsigned *`

pointer to unsigned int.

`double [10]`

an array of 10 doubles.

`int *[5]`

an array of five pointers to int.

`float *()`

a function taking no parameters returning a pointer to type float.

`long (*)[20]`

a pointer to an array of 20 longs.

`void (*)(int)`

a pointer to function with int parameter and no return value.

`char (*const [])(double, ...)`

an array with an unspecified number of constant pointers to functions, each with one parameter that has type double and an unspecified number of other parameters. These functions each return a char.

Function Definitions

```
function-definition:
    decl-specifiersopt declarator ctor-initializeropt fct-body
fct-body:
    compound-statement
```

A function definition consists of a function declaration followed by a *fct-body* which specifies the code executed when the function is called. The *ctor-initializer* is only used when defining a constructor function - see Chapter 12: ‘Initializing bases and members’.

On entry to a function, the value of each supplied argument expression is used to initialize the corresponding parameter. Arguments that are array expressions and function designators are converted to pointers before the call. Thus, a declaration of a parameter as *array of type* will be adjusted to *pointer to type*, and a declaration of a parameter as *function* will be adjusted to *pointer to function*.

Each parameter for a function is treated as having automatic storage duration, and is in the scope of the outermost block of the *fcn-body*. The parameter's identifier is an lvalue. In effect, a parameter is declared at the head of the compound statement that constitutes the function body. As a result, the parameter may not be redeclared in the function body (except in an enclosed block).

Initialization

An object can be given an initial value when the object is declared, either by means of a constructor call (implicit or explicit), or, for a type for which no constructor is declared, by specifying an *initializer* in the declarator for that object. Initialization by constructors is discussed in Chapter 12.

```

initializer:
    = assignment-expression
    = { initializer-list ,opt }
      ( expression-list )
initializer-list
    expression
    initializer-list , expression
    { initializer-list ,opt }
```

When this is done, the value is assigned immediately after storage is allocated to the object. This means that objects of static storage class are initialized once at program startup. Objects of non-static storage class are initialized every time they start a new lifetime.

The initializer for a scalar must be a single expression, optionally enclosed in braces. The value of the expression becomes the initial value of the object. The same type constraints and conversions apply as for simple assignment.

In C++, all variables may be initialized using general expressions using any objects or functions already declared. The order of initialization for static objects is described in Chapter 12. (Note that this is in contrast to C, where only constant expressions are permitted in the initializers for static objects.) Any static object for which no initializer is specified is initialized to all bits zero.

The value of an auto object declared without initialization is undefined until it is explicitly assigned a value.

Initializing Aggregates

An *aggregate* is an object of array type, or of a class type (struct, union or class) with no constructors, protected or private members, base classes or virtual functions. An aggregate may be initialized using a brace-enclosed *initializer-list*. Each initializer in the list is used to initialize the next member or element of the aggregate, applied recursively to any members or elements of subaggregates.

An aggregate that is a class may also be initialized with an object of its class or a class publicly derived from it.

An aggregate that is a union may be initialized by a single value, optionally enclosed in braces, that specifies the initial value for the first member.

If an initializer-list specifies fewer values than there are items to be initialized, subsequent values are taken to be zero. If the initializer-list specifies more values than there are items to be initialized, TopSpeed C++ will report an error. If an array of unknown size is initialized, the number of elements initialized by means of the initializer-list is taken to be the size of the array, and the type of the array is no longer incomplete.

An array of characters may be initialized by a string literal, optionally enclosed in braces. Successive characters of the string literal initialize corresponding elements of the array. The null character terminating the string literal is always included in the initialization (unlike in ANSI C, where the null is omitted if the size of the array is insufficient). For example, the following initialization makes the `_word` a 6-element array, including the terminating null character.

```
char the_word [] = "hello";
```

The following is an error in C++:

```
char fivechars[5] = "hello";
```

as too many characters (six, including the null) have been supplied.

If the initializer of a subaggregate begins with a left brace, the succeeding initializers initialize the members of the subaggregate. Otherwise - that is, if the subaggregate is not being initialized by using braces - only enough initializers from the list are taken to process the members of the first subaggregate. Any remaining initializers are left to initialize the next member of the aggregate of which the current subaggregate is a part.

The declaration

```
int p[] = { 2, 16, 256 };
```

defines and initializes `p` as a one-dimensional array object that has three members. The declarations

```
struct cnum {
    double re, im;
    int info [3];
} cvals = { 3.9, 6.7, {1, 5, 9}};

struct cnum more_cvals = {6.3, 5.8, 2, 12, 36};
```

initialize two basic type members of a structure and also an aggregate type that is a member of the structure.

Initializing Multidimensional Arrays

Arrays whose elements are themselves aggregates (for example, structures or arrays) can be initialized cell-by-cell or by initializing an entire subaggregate at a time.

Braces may be used to delimit subaggregates. For example, if the initializer for cell [0][0] of a two-dimensional array begins with a left brace, then subsequent values are used to initialize the first subaggregate. The declaration,

```
double matrix[4][3] = {
{ 1,    2.0,  3    },
{ 1.0,  4,    9    },
{ 1,    8,    81.0 },
};
```

defines and initializes a two-dimensional array. The declaration contains three groups of bracketed values. Each of these initializes one row of matrix. Thus, the elements of matrix[0] are assigned the values:

```
matrix[0][0] = 1
matrix[0][1] = 2.0
matrix[0][2] = 3
```

Similarly, matrix[2] is assigned the values contained in the third group - so that matrix[2][2] would have the value 81.0 after initialization.

Because the initializer ends early, the elements of matrix[3] are initialized with zeroes. Precisely the same effect could have been achieved by

```
double matrix[4][3] = {
1.0, 2, 3, 1, 4, 9.0, 1, 8.0, 81
};
```

In this case, the initializer for matrix[0] does not begin with a left brace, so as many items from the list are used as are needed to initialize the entire first row (in this case, 3). Subsequent groups of three are taken, successively, for matrix[1] and matrix[2].

Braces can also be used to limit the number of values used for initializing a particular subaggregate. For example, the declaration

```
double matrix[4][3] = {
    { 3 }, { 5 }, { 7.0 }, { 11 }
};
```

initializes the first element of each row of matrix (that is, the first column of the matrix) as specified. The remaining elements in each row are (implicitly) initialized with zeros.

The fi array in the following declaration

```
struct {
    float f;
    int i;
} fi[] = { 1.2, 3, { 7 }, 0.4, 14 };
```

has 3 elements. The initializers for this array are inconsistently bracketed. As a result, field fi[1].i is implicitly assigned the value 0, since the initialization for this middle element is incomplete.

The following three declarations all accomplish the same thing - initializing a three-dimensional array of integers.

```
int cube[4][3][2] = {
    { 1 },
    { 2, 3, 4, 5, 6 },
    { 7, 8, 9 }
};
int cube[4][3][2] = {
    1, 0, 0, 0, 0, 0,
    2, 3, 4, 5, 6, 0,
    7, 8, 9
};
int cube[4][3][2] = {
    {
        { 1 },
    },
    {
        { 2, 3 }, { 4, 5 }, {6}
    },
    {
        { 7, 8 },
        { 9 },
    }
};
```

Each of these declarations accomplishes the following assignments:

```
cube[0][0][0] is 1
cube[1][0][0] and cube[1][0][1] are 2 and 3, respectively
cube[1][1][0] and cube[1][1][1] are 4 and 5, respectively
cube[1][2][0] is 6, and cube[1][2][1] is (implicitly) 0
cube[2][0][0] and cube[2][0][1] are 7 and 8, respectively
cube[2][1][0] is 9, and cube[2][1][1] is (implicitly) 0
```

the remaining elements are implicitly initialized to 0.

In the first declaration, the groupings are as described because the initializer for cube[0][0][0] does not begin with a left brace. As a result, up to six items from the current list may be used - to initialize the entire matrix cube[0].

There is only one value, however, so the values for the remaining five members are initialized with zero.

Similarly, neither the initializers for `cube[1][0][0]` nor `cube[2][0][0]` begin with a left brace, so each uses up to six items, initializing their respective two-dimensional subaggregates.

Initializing Strings

In C++, a string is an array of characters. Such arrays can be initialized using string literals. The following declaration

```
char mystring[] = "hello",
```

defines and initializes such an object. `mystring` is defined as an array of `char`, but no size is specified. The string's size is determined implicitly through the initialization. The terminating null string is automatically added to such an array, so that the resulting length of `mystring` is 6.

The contents of such arrays are modifiable. The declaration

```
char *v = "abc";
```

defines a character pointer, `v`. This pointer is initialized to point to a character array object. The members of this character array are initialized with a string literal (four elements, including a trailing `'\0'`).

Initializing References

A variable declared to be of a reference type must have an initializer specified, indicating the object or function to which the reference will refer. A reference cannot be made to point to a different object after it has been defined - any attempt to modify the reference will in fact modify the object referenced!

The initializer for an object of type *reference to type* `T` must be an object of type `T`, or an object that can be converted to type `T`. If the initializer does not specify an lvalue of type `T`, or of a type of which `T` is an accessible base, a temporary is created, and the reference will refer to that temporary. In such cases, if the reference is not to a `const`, TopSpeed C++ will report an error.

A reference to a `const T` can be initialized with a `const T`, a plain `T`, or a temporary created from an object that can be converted to a plain `T`. It may not be initialized from a `volatile T`.

A reference to a `volatile T` can be initialized with a `volatile T`, or a plain `T`. It may not be initialized with a `const T`. A reference to a plain `T` can only be initialized with a plain `T`.

A reference to a const volatile T may be initialized with any of the above.

A reference declared as a function argument or function result need not have an initializer specified - the reference will be initialized from the actual argument or the return value respectively. Other conditions concerning initialization of references still apply to such cases.

For example:

```
void m()
{ int a = 1;
  const int b = 2;

  int &ref1 = a;          // ok
  const int& ref2 = a;    // ok
  const int& ref3 = b;    // ok
  const int &ref4 = 1;    // ok

  int &ref5 = b;          // error - can't initialize
                          // plain int& with const int
  int &ref6 = 1;          // error - can't initialize
                          // plain int& with temporary
}
```

Initializing Functions

Normally you cannot initialize functions in C++. However, TopSpeed C++ provides an extension that allows this for the purpose of defining in-line machine code.

Example 1

The in-line machine code facility is useful if you wish to write code that cannot be expressed in C++ - for example, the 8086 out instruction. Pragma's can be used to control the generation an expansion of in-line code.

```
#pragma save, call(reg_param=>(dx,ax), inline=>on)
// port must be in dx, byte to output must be in ax
static void oport(int port, unsigned char byte) =
{
    0xEE,          /* out dx,al */
};
#pragma restore
```

Example 2

Another reason to use in-line machine code is for writing optimal code. In the example below, calls to the strlen function will be expanded in-line instead of actually calling the function, because of the inline pragma. Such a strategy improves the runtime speed.

```

#pragma save
#pragma call(reg_param=>(di), reg_saved=>(bx,si,ds))
#pragma call(inline=>on)

/* these pragmas set up the correct calling sequence for the function
below, small model */
static unsigned strlen(const char *s) =
{
    0x1E,          // push ds
    0x07,          // pop  es
    0xB9, 0xFF, 0xFF, // mov  cx,-1
    0x2A, 0xC0,     // sub  al,al
    0xF2, 0xAE,     // repnz; scasb
    0xF7, 0xD1,     // not  cx
    0x8B, 0xC1,     // mov  ax,cx
    0x48,          // dec  ax
};
#pragma restore

```

Declaration Ambiguities

The grammar of C++ allows two ambiguous interpretations of certain declarations. For example, the following

```
int x(int(a));
```

could be interpreted either as a declaration of a function called `x`, returning `int`, with a parameter called `a` also of type `int`, or as a declaration of a variable called `x` of type `int`, with a function-style cast of `a` to type `int` as its initializer.

In such cases, the expression is disambiguated by treating the largest construct that can be a declaration to be a declaration. In the case above, the `(int (a))` is considered to be part of the declaration rather than the initializer, and the declaration is therefore of a function (the parentheses around the parameter name `a` are redundant).

In order to explicitly turn the above declaration into an object declaration, the function-style cast `int(a)` can be replaced by a non-function-style cast `(int) a`, or an explicit `=` can be introduced to indicate initialization as follows

```
int x = int(a);
```

Ambiguities may also arise between declarations and expressions - these are discussed in Chapter 8: 'Expression Ambiguities'.

CHAPTER 7

EXPRESSIONS

Introduction

An *expression* is a sequence of operators and operands. This sequence can specify

- An object or function
- How to compute a value (for example, multiplication, addition)
- How to generate side effects (for example, assignment)
- some combination of these..

An expression can contain subexpressions among its elements. The order of evaluation of subexpressions, and the order in which side effects take place are both unspecified. Factors that affect the order of evaluation include:

- The order of precedence for operators in an expression. This order is specified by the syntax and it applies in the absence of parentheses. Operations with higher order of precedence are carried out first - unless parentheses are present to override this precedence.
- The use of parentheses to regroup subexpressions containing operators of different precedence.
- The ordering of elements in the expression. TopSpeed C++ may regroup any expression that involves more than one occurrence of the same commutative and associative binary operator or of operators that have the same precedence. Parentheses are honored in such a regrouping. The commutative and associative binary operators include (*, +, &, ^, |).
- The presence of unary operators (such as the unary + operator), which restrict regrouping.

If the evaluation of an expression causes an object to be modified more than once the value of the expression is undefined.

Note: In C++, many operators may be *overloaded*, that is, the programmer may define meanings for them when applied to objects of class type (see Chapter 13). This chapter describes the behavior of the built-in operators, and does not cover programmer-defined operator behavior or programmer-defined conversions..

An *lvalue* is an expression that designates an object. Certain lvalues are said to be *modifiable*. In particular, an lvalue is modifiable if it is *not* any of the following:

- An array type
- An incomplete type
- A type that has been qualified with the `const` qualifier
- A structure or union that contains a member (either directly or in a nested member) that is qualified with the `const` qualifier

In general, an lvalue is converted to the value stored at the object that the lvalue specifies. The exceptions are if the lvalue is any of the following:

- An array
- An operand for the `sizeof`, unary `&`, `++`, or `—` operators
- The left operand of the dot operator (`.`) or of an assignment operator

In general, an object that has type *array of <type>* is converted to an expression that has type *pointer to <type>*. The exceptions are if the expression is one of the following:

- The operand of the `sizeof` operator
- The operand of the unary `&` operator
- A string literal being used to initialize an array of `char`
- A wide string literal being used to initialize an array of a type that is compatible with `wchar_t`

A *full expression* is an expression that is not part of another expression. Each of the following is a full expression:

- An initializer.
- The expression in an expression statement.
- The controlling expression of a selection statement (if or switch).
- The controlling expression of an iteration statement

(while, do, or for).

- The expression in a return statement.

The end of a full expression is a sequence point. At a sequence point, all side effects (for example, assignments) of previous evaluations have been completed and no side effects of later evaluations have yet occurred.

Precedence of Operators

The following table shows the order of precedence for TopSpeed C++'s operators. Primary expressions have the highest precedence, and the comma operator has the lowest. All operators listed with the same number have the same precedence.

```

18R :: unary (global) scope resolution
18L :: binary (class) scope resolution

17L [] subscripting
17L () function call/type conversion
17L . -> member selection
16L ++ -- postfix increment/decrement

15R ++ -- prefix increment/decrement
15R sizeof object size
15R (type-name) type cast
15R ~ ! - +unary arithmetic/logical operators
15R & *unary address and dereference
15R new delete allocation expressions

14L ->* .* pointer-to-member
13L * / % multiplicative
12L + -additive
11L << >>shift
10L <> <= >=relational
9L == !=equality
8L & bitwise and
7L ^ bitwise xor
6L | bitwise or
5L && logical and
4L || logical or
3R ?: conditional
2R = += -= *= /= %=
<<= >>= &= ^= |=simple/compound assignment
1L , comma

```

L indicates left associative operators;

R indicates right associative operators

Primary Expressions

```

primary-expression:
    name
    literal
    :: identifier
    :: operator-function-name
    :: qualified-name
    this
    ( expression )
name:
    identifier
    operator-function-name
    conversion-function-name
    class-name
    ~ class-name
    qualified-name
qualified name:
    qualified-class-name :: name.

```

A *primary expression* is any of the following:

- An identifier that has been declared as naming an object or function. An identifier is an *lvalue* if it specifies an object and a *function designator* if it specifies a function. The value of an identifier depends on the type given when the identifier was declared.
- An operator function name - see Chapter 13: ‘Overloaded Operators’.
- A qualified name - this is a class name (see Chapter 9) followed by the scope resolution operator `::` and the name of a member, constructor or destructor of the class or of a base class. The type is the type of the member, and the result is the member. The result is an lvalue only if the member is.
- Any of the above preceded by the unary (global) scope resolution operator `::`.
- A conversion function name - see Chapter 12: ‘Conversion Functions’.
- A literal. The literal’s type depends on its form, as described in Chapter 3: ‘Literals’.
- The keyword `this`, which may only be used within the body of a non-static member function.
- A parenthesized expression. The type, value and “lvalueness” of a parenthesized expression are identical to those of the unparenthesized expression. Such an expression is a function designator, or void expression if the unparenthesized expression is, respectively, a function designator, or void expression.

The following list shows several examples of primary expressions.

```
Count      2.6
"fred"     (a+b*c)
301u       (0)
::sqrt
```

Postfix Operators

```
postfix-expression:
    primary-expression
    postfix-expression [ expression ]
    postfix-expression ( argument-expression-listopt )
    simple-type-name ( argument-expression-listopt )
    postfix-expression . name
    postfix-expression -> name
    postfix-expression ++
    postfix-expression --
argument-expression-list:
    assignment-expression
    argument-expression-list , assignment-expression
```

Postfix expressions group left to right.

Array Subscripting

Array subscripting is used to access individual elements of an array. This is accomplished by using the [] operator. For example, the array object specified by

```
int iarray[5]
```

is a 5-element *array of* int. The elements of such an array have indexes 0 through 4. The [] operator can be used to specify the third element of such an array as:

```
iarray[2]
```

One of the expressions in an array specification must have type *pointer to T*. The other expression (i.e., the element number) must have integral type. The result of applying the array subscription operator then has type *T*.

If *a* is an *n*-dimensional array with dimensions *i***j**...**k*, then *a* (when not used as an lvalue) is converted to a pointer to an (*n*-1)-dimensional array with dimensions *j**...**k*. This is consistent with the conversion of an array type to a pointer (see above)

Applying the unary * operator to this pointer results in the target (*n*-1)-dimensional array. This target array is, itself, converted into a pointer (when not used as an lvalue). The unary * operator can be called implicitly (through the subscript operator), with the same effect.

For example, in the array object given by the declaration:

```
int b[7][3];
```

`b` is a 7×3 array of ints. Actually, `b` is an array of seven member objects, each of which is an array of three ints.

In the expression

```
b[i]
```

`b` is first converted to a pointer to the initial 3-element array of ints. Then `i` is multiplied by the size of the pointer's target object (i.e., `sizeof(int) * 3`). The results are added and indirection is applied to yield the specified 3-element array. When used in the expression `b[i][j]`, that array, in turn, is converted to a pointer to the first of the ints, so `b[i][j]` yields an int.

Thus, arrays are stored in *row-major order*, in which the last subscript varies fastest.

Note: `b[i, j]` is not a two dimensional reference. It is equivalent to `(*(b)+(i, j))` i.e., the subscript is a expression.

Function Calls

A function call is a postfix expression followed by parentheses, `()`. The postfix expression denotes the function called. The parentheses will contain zero or more expressions, separated by commas. The expressions within the parentheses represent the arguments to the function.

For example, the following expressions contain four function calls. These calls are to functions `getchar`, `putchar`, `printf`, and `abs`.

```
i = getchar();
putchar('\E n');
printf("%d items\E n", abs( Count))
```

There is a sequence point just before a function call.

The expression that denotes the function called must have type pointer to function returning `T` or reference to function returning `T`. A function name is converted to a pointer to the function. The type of the result has type `T`. The result is an lvalue only if the result type is a reference.

A function may only be called if has been declared in the current scope. The declaration will specify the number and types of the parameters. If default parameters are specified, the function may be called with fewer arguments. If the ellipsis is used in the function declaration, the function may be called with additional arguments, and no matching parameter is available for these arguments.

An argument may be an expression of any object type. Prior to the call to a function, the arguments are evaluated, and each parameter is initialized using the value of the corresponding argument. Standard and programmer-defined conversions are applied if required to convert the arguments to the types of the corresponding formal parameters. If there is no corresponding formal

parameter (i.e., the ellipsis has been used), the default argument promotions are applied - see below.

A formal argument of class type is initialized from the actual argument using a constructor call. If there is no corresponding formal argument for an actual argument of class type, the argument is passed as a data structure, without a constructor being called.

A function may alter the value of any parameter that is not declared `const`. However, unless the parameter is of type *reference to T*, the value of the actual argument is unchanged. Parameters are passed by value (i.e., call by value is used). Note that a parameter of type *pointer to T* may be used to alter the value of the object pointed to, and that array and function arguments are implicitly converted to pointer types.

A temporary may be required to initialize a parameter of type *reference to T*. In such cases, TopSpeed C++ will report an error if the parameter is not `const` (see Chapter 6: 'Initializing References').

The order of evaluation of the function designator and arguments in TopSpeed C++ is such that arguments are evaluated from left to right, followed by the function designator.

Recursive function calls are permitted. The maximum depth of recursion is limited only by the size of the available stack.

The following listing illustrates some function calls:

```
void max_and_min(int [], int, int &, int &);
char contains(int [], int, int);
short process(int [], int, short (int));
short print_item(int);
int list[LIST_SIZE],
    max_val,
    min_val,
    search_val;
short err;
void call_fns (void)
{
    if (contains(list, LIST_SIZE, search_val)) {
        /* some processing */
    }
    max_and_min(list, LIST_SIZE, max_val, min_val);
    err = process(list, LIST_SIZE, print_item);
}
```

Default Argument Promotions

The default argument promotions are applied to any actual argument for which (because of the use of the ellipsis notation) there is no corresponding formal parameter. Such actual arguments are converted as follows :-

- An actual argument of type float is converted to type double.

- An actual argument of type `char`, `short`, or enumeration is converted to `int` or unsigned by the integral promotions (see Chapter 3: ‘Integral Promotions’).
- An actual argument of class type is passed by value as a data-structure (i.e., a simple copy), rather than being used in a constructor for the formal argument.

Explicit Type Conversion (Function Notation)

A postfix expression of the form

simple-type-name (*argument-expression-list*_{opt})

constructs a value of the named type. If the named type has constructors, the *argument-expression-list* is used as the arguments to a call of a constructor - a suitable constructor must be available.

If the named type does not have constructors, the argument expression list must be missing, or contain a single value. The value, if present, is converted to the named type in the same way as a cast expression (see below). If no value is specified, the result of the expression is an undefined value of the named type. For example:

```
main()
{ int i;
  i = int(3.0); // assigns 3 to i;
  i = int();    // assigns an undefined value to i
  complex c(0,0); // initialize c to 0,0 via
                  // constructor
  c = complex(1,0); // assigns 1,0 to c
  c = complex();   // calls default complex
                  // constructor
}
```

Member Selection Operators

The *dot* (.) and *arrow* (->) operators are used to select a member of a structure or union

A postfix expression followed by a dot . and a name selects a member of the class object preceding the dot. The first operand of the . operator must have a class type, which may be qualified. The second operand must be a member of the specified class.

For example:

```
struct complex {
    double re;
    double im;
} cval;
```

```
/* ... */
cval.re = 2.9;
cval.im = -7.6;
```

The value of the expression is the value of the named member. This is an lvalue if the member is an lvalue.

A postfix expression followed by an arrow `->` and an name also selects a member. The first operand of the `->` operator must have type *pointer to* class X. The second operand must be a member of class X. In this case, the member selected by the arrow operator is in the pointer's target class object.

The value of the expression is the value of the member. This is an lvalue if the member is.

The following two expressions are equivalent:

```
S->field and (*S).field
```

Postfix Increment and Decrement Operators

The postfix increment and decrement operators change the values of objects. Such changes are made only after the value is obtained. The operand for these operators

- Must be of arithmetic type, or pointer to an object type
- Must be a modifiable lvalue

The *postfix increment operator* (`++`) returns the value and type of the operand as the operator's result. After the result is returned, the value of the operand is incremented by 1. The side effect of updating the stored value of the operand occurs between the previous and the next sequence point.

For pointers, the value of each increment is the `sizeof` the type pointed at. For other scalars, the increment is the value 1 of the appropriate type.

The *postfix decrement operator* (`--`) is analogous to the postfix `++` operator, except that the value of the operand is decremented rather than incremented.

The result of the postfix increment and decrement operators is not an lvalue.

Note: The increment and decrement operators may not be used for variables of an enumeration type in C++ (unlike in C).

For example:

```

i = 0;
while (*s)// find end of string and length
    i++, s++;
s--;
while (i) {
    i--;    // reverse string s into string p
    *p++ = *s--;
}
*p = 0;    // 0 terminates p

```

Unary Operators

```

unary-expression:
    postfix-expression
    ++ unary-expression
    -- unary-expression
    unary-operator cast-expression
    sizeof unary-expression
    sizeof ( type-name )
    allocation-expression
    deallocation-expression
unary-operator: one of
    & * + - ~ !

```

Expressions with unary operators group right to left.

Prefix Increment and Decrement Operators

The prefix increment and decrement operators change the value of an object. The difference between these operators and the postfix versions is in the timing of the changes.

The operand of the prefix increment or decrement operator

- must have arithmetic type, or pointer to an object type.
- Must be a modifiable lvalue

The value of the operand of the *prefix increment operator* (`++`) is incremented *before* the value is used in an expression. The result is the new value of the operand (i.e., after being incremented). The expression `++val` is equivalent to changing a value by adding 1. For pointers, the value of the increment is the `sizeof` of the object pointed to.

The *prefix decrement operator* (`--`) is analogous to the prefix `++` operator, except that the value of the operand is decremented rather than incremented. The expression `--val` is equivalent to `(val-=1)`.

```

E.g,
int i;
char *s, *p;

```

```

i = 0;
while (*s) // find end of string and length
    ++i, ++s;
while (i) {
    -i; // reverse string s into string p
    *p++ = *-s;
    /* post-increment p; */
}
*p = 0;

```

Address and Indirection Operators

The unary *address-of operator* (&) produces a pointer to the object or function designated by the operand. The result has type *pointer to T*, where T is the operand's type.

For example:

```

int *i_ptr, i;

i_ptr = &i; // i becomes target object for i_ptr

```

If the operand is *const T*, the type of the result is *pointer to const T*. Similarly, an operand which is *volatile* yields a result of type *pointer to volatile*.

The operand of the unary & operator must be one of the following:

- A function designator. If the function is overloaded, the & operator can only be used in an assignment or initialization (including argument passing and function return), and the type of the target is used to determine which function is referred to.
- An lvalue that designates an object, other than a bit-field.
- A qualified name. In this case, unless the member of class X referred to is static, the type of the result is *pointer to member of class X of type T*. For example, in

```
int myclass::* pml = &myclass::member;
```

Both the variable *pml* and the expression used to initialize it have type *pointer to member of class myclass of type int*.

The unary *indirection operator* (*) produces the value pointed to by its operand. The operand must have a pointer type. For example:

```

ival = *i_ptr;
*i_ptr = ival;

```

If the operand has type *pointer to T*, the result has type *T*. If the operand points to a function, the result is a function designator. If it points to an object, the result is an lvalue designating the object.

If `*P` is an lvalue and `T` is the name of an object pointer type, the cast expression `*(T)P` is an lvalue that has the same type as that to which `T` points.

Under certain conditions, the result of applying the indirection operator will be invalid. In such cases, the result is undefined. Invalid values for dereferencing a pointer by the unary `*` operator result when any of the following is accessed:

- A null pointer constant.
- The address of an object that has automatic storage duration (for example, a parameter or a local object) when execution of the block in which the object is declared has terminated.

For example:

```
void check_bounds(int *val_addr, int lo, int hi)
//    force object into given range lo..hi
{ if (*val_addr < lo)
    *val_addr = lo;
  else if (*val_addr > hi)
    *val_addr = hi;
}

main()
{ int val,
  *valp,
  table[SIZE];
  // ...
  check_bounds(&val, 0, 100);
  for (valp=table; valp<&table[SIZE]; valp++)
    check_bounds(valp, 0, 100);
}
```

Unary Arithmetic Operators

The unary arithmetic operators return or change the numerical values of their operands. They do not return lvalues.

The result of the unary `+` operator is the value of its operand, promoted using the integer promotions. The operand must be of arithmetic or pointer type. The expression `+E` is equivalent to `(0+E)`. The type of the result is the type of the operand after any promotion has occurred.

The result of the unary `-` operator is the negation of its operand, promoted using the integer promotions. The operand must be of arithmetic type. The expression `-E` is equivalent to `(0-E)`. The type of the result is the type of the operand after any promotion has occurred.

The result of the `~` operator is the *bitwise complement* of its operand, promoted using the integer promotions. The operand must be of integer type.

Each bit in the operand is processed. If the bit is on, the ~ operator turns it off; if the bit is off, the operator turns it on. The type of the result is the type of the operand after any promotion has occurred.

The result of the logical negation operator (!) is 1 if the value of its operand is zero, otherwise 0. The operand must be of arithmetic or pointer type. The result has type int. The expression !E is equivalent to (0==E).

For example:

```
#define ERR_FLAG 0x0400
short FLAGS;
/* ... */
FLAGS &= ~ERR_FLAG; // clear error flag
if (!FLAGS)          // FLAGS is zero
    FLAGS = -1;
```

The sizeof Operator

The result of the sizeof operator is an integer constant that represents the size (in bytes) of its operand. The size is determined from the type of the operand, which is not itself evaluated. The result has type size_t, defined in the <stddef.h> header. In TopSpeed C++, a size_t is an unsigned int.

The operand may be an expression or the parenthesized name of a type.

The result of applying the sizeof operator depends on the operand's type.

- For an operand that has character type (i.e., char, unsigned char or signed char), the result is 1.
- For an operand that has type short int or unsigned short int, the result is 2.
- For an operand that has type int or unsigned int, the result is 2.
- For an operand that has type long int or unsigned long int, the result is 4.
- For an operand that has type float, the result is 4.
- For an operand that has type double, the result is 8.
- For an operand that has type long double, the result is 10.
- For an operand that has array type, the result is the total number of bytes in the array - i.e., the number of elements times sizeof (*element-type*).
- For a parameter declared to have array or function type, the sizeof operator yields the size of the pointer obtained by conversion.

- For an operand that has class type, the result is the total number of bytes in such an object, including internal padding. *Internal padding* may occur when the structure includes bit fields. All classes and class objects have non-zero size.

The sizeof operator may not be applied to any of the following:

- An expression that has function type.
- An array whose dimension has not been specified.
- An undefined class.
- The parenthesized type name of a function type or an incomplete type.
- A bit-field object.

For example:

```
int table[TABSIZE],
    table_length,
    no_of_elements;
table_length = sizeof table;
/* == TABSIZE*sizeof(int) */
no_of_elements = table_length / sizeof table[0];
```

The new Operator

```
allocation-expression:
::: opt new placementopt new-type-name new-initializeropt
::: opt new placementopt ( type-name ) new-initializeropt

placement:
( expression-list )

new-type-name:
type-specifier-list new-declaratoropt

new-declarator:
* cv-qualifier-listopt new-declaratoropt

new-initializer:
( initializer-list )
```

The new operator is used to create an object of the type specified by the *type-name* or *new-type-name*. The result is a pointer to the new object, or to the initial element for an object of array type.

An object created using the new operator continues to exist until specifically destroyed, (normally) by means of the delete operator.

The *type-specifier-list* in a *new-type-name* may not contain declarations of enumerations or classes, nor the const or volatile specifiers.

When an array type is specified, all dimensions but the first must be constant expressions. The first dimension can be a general expression. (This is an exception to the general rule that array sizes in type names must be constant expressions.)

The use of the new operator is translated into a call of an operator new() function. The first argument to the operator new() function is sizeof(T), where T is the type specified. If a *placement* is specified, the expressions listed are used as additional arguments to operator new() - a suitably declared function must be available. If T is a class type, the operator new function will be looked up in the scope of that class (using the usual scope rules for finding class members). Otherwise the global operator new() is used. The ::new syntax may be used to ensure that the global operator new() is used for a class type.

If a *new-initializer* is specified, it is used to initialize the object created. For an object of class type with a constructor, the *new-initializer* specifies the arguments for the constructor. Otherwise, the expression list must be empty (in which case the object is uninitialized) or contain a single expression used to initialize the object.

An object of a class with a constructor can only be created by new if a suitable *new-initializer* is specified, or a default constructor exists. Both the operator new() and the constructor must be visible, accessible and unambiguous, using the normal rules.

An array may not have any initial value specified when created using new. An array of objects of class type may only be created if a default constructor is available - this will be called for each element of the array. Arrays (including arrays of class objects) are always allocated using the global operator new() function.

Initialization only occurs if the call of operator new() is successful (i.e., does not return the null pointer).

The type specified in a *new-type-name* may not contain parentheses. If a type specification containing parentheses is required, the whole type name should be parenthesized.

When parsing an *allocation-expression*, TopSpeed C++ will treat the longest possible sequence of *new-declarators* as forming the *new-type-name*. This can resolve ambiguities between the operators *, &, and [] which can be used in declarators or expressions.

For example:

```

void * operator new(size_t, int);

class myclass {
    // ...
public:
    void * operator new(size_t);
    myclass(int, int);
    myclass(); // default constructor
};

void g()
{
    myclass *p1 = new myclass(10, 10);
        // uses myclass::operator new(), and
        // constructor myclass(int,int)
    myclass *p2 = ::new myclass;
        // uses global operator new(size_t),
        // and default constructor
    myclass *p3 = new myclass[100];
        // uses global operator new(size_t),
        // and default constructor for each element
    int *p4 = new int (8); // *p3 = 8
    int *p5 = new int (); // *p4 uninitialized
    int *p6 = new int;    // *p5 uninitialized

    int *p7 = new (1000) int;
        // calls ::operator new(size_t, int)
}

```

The delete Operator

```

deallocation-expression:
    ::opt delete cast-expression
    ::opt delete [ ] cast-expression

```

The delete operator is used to destroy an object created using the new operator. The result has type void. The operand must be a pointer that was previously returned by the new operator, or be zero (in which case the delete operator has no effect).

After being destroyed, the object referenced must not be accessed.

If the object pointed to by the operand has a destructor, the delete operator will call it before releasing the storage allocated. To release the storage, an operator delete() function is used. If a destructor is called, the operator delete() function is looked up in the scope of the destructor, unless the global operator delete() is specified explicitly using the ::delete form. If no destructor is called, the global operator delete() is used. Both the destructor and the operator delete function must be visible, accessible and unambiguous according to the normal rules.

The form delete [] *expression* is used to delete an array. In this case, the expression should point to the first element of the array, as returned by the new operator. A destructor is called for each element of the array if appropriate.

If the delete [] form is used for an object that is not an array, or the plain delete form is used for an object that is an array, the result is undefined.

For example, given the declarations in the previous example, the memory allocated might be freed as follows :-

```
void g()
{
    // ...
    delete p7;    // uses global delete
    delete p6;    // uses global delete
    delete p5;    // uses global delete
    delete p4;    // uses global delete
    delete [] p3; // uses global delete
    ::delete p2;  // uses global delete
    delete p1;
    // uses myclass::operator delete()
}
```

Cast Expressions

```
cast-expression:
    unary-expression
    ( type-name ) cast-expression
```

The *cast operator* converts the value of an expression to a type specified within parentheses. For a type that has a name, the functional form `type-name(expression)` may be used as an equivalent means of type conversion. See Chapter 7: ‘Explicit Type Conversion’.

A cast to a reference type yields an lvalue. In ANSI C, and in standard C++ as defined by the Base Document, other casts do not yield an lvalue. TopSpeed C++ has an extension in which a cast yields an lvalue if the expression is an lvalue. This extension may be disabled by means of the `pragma option(lang_ext)`.

A cast changes the type of its operand. A change of representation may occur or the conversion may be quiet - i.e., no change of representation is involved. However, the new type may cause the code generator to act differently.

The following explicit type conversions may be performed. Other type conversions, unless defined by the programmer (see Chapter 12: ‘Conversions’) will be reported as errors.

- Any conversion for which a standard conversion exists (see Chapter 4) may be made explicitly. The meaning is the same.
- A pointer may be converted to an integral type large enough to hold it. In near pointer models, an int can hold a pointer; in far pointer models, a long int can hold a pointer. In TopSpeed C++, long is the only integral type guaranteed to hold a complete pointer.

- Zero (0) may be converted to the null pointer constant.
- An integer may be converted to a pointer. Unless the integer was converted from a pointer, or is zero, the result is undefined. Converting a pointer to an integer large enough to hold it and back again yields the same pointer.
- A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type, subject to the provisos below.
- A pointer to a base class B may be converted to a pointer to a derived class D of which B is a non-virtual base class, provided that there is an unambiguous conversion from D to B. The conversion assumes that the object pointed to is a sub-object of an object of class D, and returns a pointer to the enclosing object. If this is not a valid assumption, the result is undefined. A pointer to an undefined class may be converted to or from another pointer type without the above restriction.
- A pointer to a function of one type may be converted to a pointer to a function of another type and back again. The resulting pointer will compare equal to the original pointer. The behavior is undefined if a converted pointer is actually used to call a function that is not compatible with the original type.
- A pointer to an object may be converted to a pointer to a function, and vice versa, provided that both pointers are the same size. The result of using such a pointer is undefined. If an object pointer is converted to a function pointer and back, or vice versa, the resulting pointer is unchanged.
- An object of type T may be explicitly converted to a reference type X& if the type T* can be explicitly converted to X*. A cast to a reference does not cause constructors or conversion functions to be called.
- A reference to a base class may be converted to a reference to a derived class if a pointer to the base may be converted to a pointer to the derived class.
- An object or value may be converted to a class object, or vice versa, if an appropriate constructor or conversion operator has been defined.
- A pointer to member may be converted to another pointer to member, provided both types are pointers to members of the same class, or both types are pointers to member functions in classes X and Y, where X is derived unambiguously from Y.

- A pointer to a const type may be explicitly converted to a pointer to the same type without the const attribute. The value of the pointer is unchanged. An object of const type, or a reference to such an object, may be explicitly converted to a reference to the same type without the const attribute, and will refer to the original object. An address exception may occur if the pointer or reference is used.
- A pointer to a volatile type may be explicitly converted to a pointer to the same type without the volatile attribute. The value of the pointer is unchanged. An object of volatile type, or a reference to such an object, may be explicitly converted to a reference to the same type without the volatile attribute, and will refer to the original object.

In any pointer conversion described above, if the value before conversion is the null pointer, the value after conversion will be too.

Pointer-to-Member Operators

```
pm-expression:  
cast-expression  
pm-expression .* cast-expression  
pm-expression ->* cast-expression
```

The pointer-to-member operators bind their second argument, which must be of type *pointer to member of class T*, to the first argument, which must be of type *T* for the `.*` operator, or type *pointer to T* for the `->*` operator. The result is the member or member function from the object referred to by the first operand which is indicated by the second operand. The type is specified by the type of the second operand. The result is an lvalue if the second operand is an lvalue.

If the result of a pointer-to-member operator is a function, it may only be used as the operand for a function call.

For example:

```

class c1 {
public:
    int a;
    int b;
    static int c;
    int fun1();
    int fun2();
    static int fun3();
};

void f(c1 o, c1 *op)
{
    int c1::*pm1 = &c1::a;
    int c1::*pm2 = &c1::b;
    int *p3 = &c1::c;           // NOT a pointer-to-member
    int (c1::* pm4)() = &c1::fun1;
    int (c1::* pm5)() = &c1::fun2;
    int (*p6)() = &c1::fun3; // NOT a pointer-to-member

    o.*pm1 = 5;                // assigns to o.a
    op->*pm2 = 6;               // assigns to op->b
    *p3 = 7;                   // assigns to c1::c
    (o.*pm4)();                // calls o.fun1()
    (op->*pm5)();               // calls op->fun2()
    (*p6)();                   // calls c1::fun3()
}

```

Multiplicative Operators

```

multiplicative-exp:
    pm-expression
    multiplicative-exp * pm-expression
    multiplicative-exp / pm-expression
    multiplicative-exp % pm-expression

```

Interpreted as a binary operator, the `*` symbol represents the *multiplication operator*, which yields the product of its operands. These operands must have arithmetic type. The multiplication operator is commutative and associative.

The binary *division operator* (`/`) yields the quotient from dividing the first operand by the second. Both operands must have arithmetic type. The second operand must be nonzero. The result is the arithmetic quotient, rounded towards zero.

The binary *modulus operator* (`%`) requires two integral operands. This operator yields the whole number remainder after dividing the first operand by the second. The second operand must be nonzero. If both operands have the same sign, the result will be positive or zero. If they have different signs, the result is negative or zero.

If the quotient a/b can be represented, the following expression is true:

$$(a/b)*b + a\%b == a$$

The usual arithmetic conversions are performed on the operands. The multiplicative operators group left to right.

Additive Operators

```
additive-expression:  
    multiplicative-expression  
    additive-expression + multiplicative-expression  
    additive-expression - multiplicative-expression
```

Interpreted as a binary operator, the + symbol represents the *addition operator*, which yields the sum of its operands. The addition operator is commutative and associative.

One of the following type constraints must hold for the addition operator:

- Both operands have arithmetic type. The usual arithmetic conversions are performed, and the result is the sum of the operands.
- One operand is a pointer to an object which is element N in an array, and the other operand b has integral type. The result is a pointer to element N+b of the array.

The binary *subtraction operator* (-) yields the difference resulting when the second operand is subtracted from the first.

One of the following type constraints must hold for the subtraction operator:

- Both operands have arithmetic type. The usual arithmetic conversions are performed, and the result is the difference of the operands.
- Both operands are pointers to elements of the same array. The result is of type ptrdiff_t (defined in <stddef.h>), and represents the number of elements in the array which separate the objects pointed to.
- The left operand is a pointer to an object which is element N in an array, and the right operand b has integral type. The result is a pointer to element N-b of the array.

Additive operators group left to right.

Bitwise Shift Operators

```
shift-expression:  
    additive-expression  
    shift-expression << additive-expression  
    shift-expression >> additive-expression  
    \xr{bit_ shift}
```

The *left shift operator* (<<) takes two integral operands. The result of A << B is A shifted left B bit positions. Vacated bits are filled with zeros.

If A has an unsigned type, the value of the result is A multiplied by (2 raised to the power B)

The *right shift operator* (>>) takes two integral operands. The result of A >> B is A shifted right B bit positions.

The value resulting from such a shift depends on the left operand's type and value. The value represents the integral part of the quotient of A divided by 2 raised to the power B, if either of the following holds:

- A has an unsigned type
- A has a signed type and a non-negative value

If A has a signed type, the vacated bits are filled with the same bit value as the original leftmost bit.

The integral promotions are performed on each of the operands for bitwise shift operators. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width (in bits) of the promoted left operand, the result will be 0 or -1 (depending on whether the original value being shifted was positive or negative).

For example:

```
date_time = (hours<<12) | (minutes<<6) | seconds
            | (((year-1980<<9) | (month<<5) | day)<<17);
//  YYYYMMMMDDDDDDhhhhmmmmmmssssss

US_date = (date_time>>26) + 80           // year
          | ((date_time>>10) & 0x0F80)    // day
          | (((date_time>>17) >> 5) & 0x0F)<<12);
//  MMMDDDDDDYYYYYYYY
```

Relational Operators

```
relational-expression:
  shift-expression
  relational-expression < shift-expression
  relational-expression > shift-expression
  relational-expression <= shift-expression
  relational-expression >= shift-expression
```

The *relational operators* are used to compare two values. Depending on the result of such a comparison, the result from a relational operator will be 0 or 1. When applying relational operators, one of the following must hold:

- Both operands have arithmetic type.
- Both operands are pointers to objects that have compatible types.
- Both operands are pointers to compatible incomplete types.

When two pointers are compared, the result depends on the relative locations of the target objects. If the target objects are members of the same array, pointers to elements with larger subscript values compare higher than pointers to elements with lower subscript values.

If the target objects are not members of the same array, the result is undefined. As an exception, if P points to the last member of an array object, the pointer expression P+1 compares higher than P, even though P+1 does not point to a member of the array.

Each of the following operators yields 1 if the specified relation is true and 0 if it is false. The result has type int.

- *less than operator (<)*
- *greater than operator (>)*
- *less than or equal to operator (<=)*
- *greater than or equal to operator (>=)*

If both of the operands have arithmetic type, the usual arithmetic conversions are performed. The relational operators group left to right.

%%% Comparison with the null pointer constant is allowed if the language extensions are enabled (/e+}).

Equality Operators

```
equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression
```

The *equality operators* are used to compare two values. Depending on the result of such a comparison, the result from an equality operator will be 0 or 1. The == (*equal to*) and the != (*not equal to*) operators behave in a similar way to the relational operators except for their lower precedence.

When applying equality operators, one of the following must hold:

- Both operands have arithmetic type.
- Both operands are pointers to the same type, or pointers to members of the same type.
- One operand is a pointer to an object or incomplete type and the other is a pointer to void.
- One operand is a pointer or pointer to member and the other is a null pointer constant.

The following results apply when using the equality operators:

- If two pointers to objects or pointers to incomplete types compare equal, they point to the same object, they are both null pointers, or they both point to one past the last element of the same array.
- If two pointers to members compare equal, they point to the same member.
- If two pointers to functions compare equal, they point to the same function.
- If one of the operands is a pointer to an object (or pointer to an incomplete type) and the other has type pointer to void (optionally qualified), the pointer to an object (or pointer to an incomplete type) is converted to type *pointer to void*.

Bitwise AND Operator

AND-expression:
equality-expression
AND-expression & equality-expression

Interpreted as a binary operator, the & symbol represents the *bitwise AND operator*. This operator takes two integral operands, and yields the bitwise *AND* of the operands as an integral result. Corresponding bits in each operand are examined, and the resulting bit is set to 1 only if corresponding bits in *both* operands are 1; otherwise the resulting bit is set to 0.

The binary & operator is commutative and associative. However, order of evaluation for the operands is not guaranteed to be from left to right.

The usual arithmetic conversions are performed on the operands.

Bitwise Exclusive OR Operator

exclusive-OR-expression:
AND-expression
exclusive-OR-expression ^ AND-expression

The *bitwise exclusive OR operator* (^) is one of two bitwise *OR* operators in C++. This operand takes two integral operands and yields an integral result. Corresponding bits in each operand are examined, and the resulting bit is set to 1 only if exactly one of the corresponding bits is 1; otherwise the resulting bit is set to 0.

The ^ operator is commutative and associative. However, order of evaluation for the operands is not guaranteed to be from left to right.

The usual arithmetic conversions are performed on the operands.

Bitwise Inclusive OR Operator

```
inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | exclusive-OR-expression
```

The *bitwise inclusive OR operator* (`|`) takes two integral operands, and yields an integral value.

The result of the `|` operator is the bitwise inclusive *OR* of the operands. Corresponding bits in each operand are examined, and the resulting bit is set to 1 if either or both of the corresponding bits is 1; otherwise the resulting bit is set to 0.

The `|` operator is commutative and associative. However, order of evaluation for the operands is not guaranteed to be from left to right.

The usual arithmetic conversions are performed on the operands.

Logical AND Operator

```
logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression && inclusive-OR-expression
```

The *logical AND operator* (`&&`) takes two scalar operands, and yields 1 if both of its operands are nonzero; otherwise the `&&` operator yields 0. The result has type `int`. The `&&` operator groups from left to right.

Unlike the bitwise binary `&` operator, the operands for the `&&` operator are always evaluated from left to right. There is a sequence point after the first operand is evaluated. If the first operand compares equal to 0, the second operand is not evaluated.

For example:

```
int *p

if (p!=NULL) && (p->val != key) ..
```

The order in which the subexpressions for `&&` are evaluated is significant, and assures that a NULL pointer is not dereferenced.

Logical OR Operator

```
logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || logical-AND-expression
```

The *logical or operator* (`||`) takes two operands of scalar type, and yields 1 if *either* of its operands is nonzero; otherwise it yields 0. The result has type `int`. The `||` operator groups from left to right.

Unlike the bitwise binary `|` operator, the operands for the `||` operator are always evaluated from left to right. There is a sequence point after the first operand is evaluated. If the first operand is nonzero, the second operand is not evaluated.

```
E.g.,
i = (m == n) || (o == p);
is equivalent to:
if (m == n)
    i = 1;
else
    if (o == p)
        i = 1;
    else
        i = 0;
```

Conditional Operator

```
conditional-expression:
    logical-OR-expression
    logical-OR-expression ? expression : conditional-expression
```

The *conditional operator* (`?:`) is a ternary operator that selects one of two possible operands, depending on the value of the first operand. The first operand must have scalar type. There is a sequence point after the first operand is evaluated. If the first operand is nonzero, the second operand is evaluated and the result of applying the conditional operator is the second operand's value; otherwise the third operand is evaluated and its value is the result. The conditional operator groups from left to right.

One of the following must hold for the second and third operands:

- Both operands have arithmetic type. The usual arithmetic conversions are applied to bring them to a common type, which is the type of the result.
- Both operands are pointers, or constant expressions evaluating to zero. Pointer conversions are applied to bring them to a common type, which is the type of the result.
- Both operands are references. Reference conversions are applied to bring them to a common type, which is the type of the result..
- Both operands have type `T`, where `T` is a class type. The type of the result is `T`.
- Both operands are void expressions. The type of the result is `void`.

If both operands are of the same type, and both are lvalues, the result is an lvalue, otherwise it is not (unlike in ANSI C, where the result is never an lvalue);

For example:

```
max = a > b ? a : b;
select = (y ? s1 : s2).f;
ptr = count > 0 ? src_ptr : NULL;
```

Assignment Operators

```
assignment-expression:
    conditional-expression
    conditional-expression assignment-op assignment-expression
assignment-op: one of
    = *= /= %= += -= <<= >>= &= ^= |=
```

C++'s assignment operators are used to modify the values of objects in the program. C++ provides both simple and compound assignment operators.

An *assignment operator* stores a value in the object specified by the left operand. An assignment operator's main task (i.e., updating an object's value) actually occurs as a side effect. The left operand for an assignment operator must be a modifiable lvalue.

After applying an assignment operator, an *assignment expression* has the value and the unqualified type of the left operand. This, however, is not an lvalue.

The side effect of updating the stored value of the left operand occurs between the sequence points preceding and following the assignment expression.

Simple Assignment

The *simple assignment operator* (=) assigns the value of the right operand to the left operand. The right operand's value replaces the value stored in the object designated by the left operand. Prior to this, the value of the right operand is converted to the type of the left operand.

One of the following conditions regarding possible operand types must hold:

- Both operands have arithmetic type. (The left operand may be qualified.)
- Both operands are pointers to compatible types. They may be qualified types and, if so, the left operand must have all the qualifiers belonging to the right operand.
- One operand is a pointer to an object or incomplete type and the other is a pointer to void (possibly qualified).

- The left operand is a pointer and the right is a null pointer constant.
- The left operand is of pointer to member type, and the right operand is of compatible pointer to member type, or is a constant expression that evaluates to 0.

An assignment to an object of class *X* uses the function *X::operator=()*. If the programmer has not defined a default *X::operator=()* function, it is generated automatically, allowing an object of class *X* or a class unambiguously and publicly derived from *X* to be assigned to an object of class *X*. Other values may be assigned to an object of class *X* only if a suitable *X::operator=()* function is defined (see Chapter 12: ‘Copying Class Objects’). Assignment and initialization of class objects have different semantics.

An assignment to an object of type *reference to T* modifies the value referenced, rather than the reference itself. The only operation to act on the reference itself is initialization.

If the left and right operand objects overlap in any way the behavior is undefined.

In the program fragment:

```
int i;
char c;

((c = i) == -1)
```

the int value *i* may be truncated when stored in the char. The truncated value will then be converted back to an int prior to the comparison. The resulting value will not necessarily be equal to the original value.

Compound Assignment

A *compound assignment* takes the following form:

```
left_op <op>= right_op;
```

<op> represents a slot for an operator. For example:

```
left_op += right_op;
left_op &= right_op;
left_op /= right_op;
left_op %= right_op;
left_op <<= right_op;
```

are all compound assignments.

A compound assignment is equivalent to the following:

```
left_op = left_op <op> right_op;
```

except that *left_op* is evaluated only once.

For the operators `+=` and `-=` only, either of the following special rules may apply:

- Both operands must have arithmetic type. (The left operand may be optionally qualified.)
- The left operand may be a pointer to an object type and the right must have integral type.

For the other compound assignment operators, each operand must have an arithmetic type consistent with those allowed for the corresponding binary operator

Note: The unary increment operator (`++`) and decrement (`--`) operators are also assignment operators. Thus, `--i` is equivalent to `(i-=1)`.

For example:

```
int *p;
long count;
short FLAG;
while (*p == ENTRY)
    p += ENTRY_LEN;
count /= 4;
FLAG |= err_bits;
```

Comma Operator

```
expression:
    assignment-expression
    expression , assignment-expression
```

Both operands for a comma operator are evaluated, but only one value is returned as a result. The left operand of a *comma operator* is evaluated as a void expression - that is, no value is returned. There is a sequence point after this operand is evaluated.

After this sequence point, the right operand is evaluated. The result of applying the comma operator has the right operand's type and value. A comma operator yields an lvalue if the right operand is an lvalue.

In contexts where a comma is used as a punctuator (such as in argument lists for functions and initializer lists), the comma operator as described here may appear only within matching nested parentheses. For example, in the function call:

```
f(a, (b=3, b+2, b+7), (c=25, c-10), d)
```

the function has four arguments, the second of which has the value 12 and the third of which has the value 15.

For example, after the following statement is processed, `val` will have the value corresponding to `table[index]`.

```
val = (index+=offset, table[index]);
```

Constant Expressions

constant-expression:
conditional-expression

A *constant expression* is any expression that can be evaluated to a single value during translation. A constant expression may appear anywhere a value of the same type as the constant expression may appear. As a result, constant expressions are often used to initialize objects. A constant expression cannot cause side-effects.

In certain situations, a constant expression is required. For example, the following cases require an integral constant expression:

- Specifying the size of an array
- Specifying a value for an enumeration
- Specifying a case value for a switch statement
- Specifying the size of a bit-field

The operands in an *integral constant expression* must have one of the following types.

- integer constant
- character constant
- enumeration
- sizeof expression
- cast of floating constant to integral type

Any operators for which these operands are valid can be used in an integral constant expression. E.g, +, %, -, etc., can all be used. The comma operator may not be used.

On the other hand, operators that require lvalues cannot be used. These operators include ++, —, & ->, . (dot operator). Such operators can be used within the operands for the sizeof operator.

An *arithmetic constant expression* is similar to an integral constant expression, but with arithmetic type substituted for integral type.

Any operators that can be used with arithmetic operands can be used in an arithmetic constant expression - provided the operator is valid for the actual operands that will be passed to the operator.

Note: Because the operands to sizeof are not evaluated it is possible to use non-constant expressions in this context

For example:

```
static int dbl_size = sizeof(double);  
static int a[4] = { 0, 1+1, (char)1.7, 'a'};  
struct m {int f,g;} s = { 1+(1*2), 97%6};  
extern int *p = &a[2];
```

Further constraints that apply to the integral constant expressions used in conditional inclusion preprocessing directives are discussed in Chapter 12: ‘Conditional Inclusion’.

CHAPTER 8

STATEMENTS

Introduction

A *statement* specifies an action to be performed by the program. A statement can be simple or compound. A *compound statement* groups zero or more simple and compound statements together. This group is treated as a single statement.

Statements are executed in the order in which they are encountered in the program, unless this sequence is modified by statement types described below.

```
statement:  
  labeled-statement  
  compound-statement  
  expression-statement  
  jump-statement  
  selection-statement  
  iteration-statement  
  declaration-statement
```

Labeled Statements

```
labeled-statement:  
identifier : statement  
case constant-exp : statement  
default : statement
```

A *labeled statement* serves as a target statement for an unconditional jump (i.e., *goto*) instruction. A statement may be preceded by an identifier that serves as a *label name* for the statement. The label is followed by a colon and then by the statement.

For example:

```
the_label: val *= 7.5;
```

Two labels, *case* and *default* are predefined in C++. These two labels may only appear within the body of a *switch* statement (see below).

Compound Statement, or Block

```
compound-statement:
  { statement-listopt }
statement-list:
  statement
  statement-list statement
```

A *compound statement* (also known as a *block*) consists of a collection of statements treated as one syntactic unit. A compound statement begins with a left curly brace ({}) and ends with the corresponding right brace ({}).

A compound statement causes a new scope to be opened. It is possible to declare new variables, typedefs and classes in this new scope.

TopSpeed C++ calculates the maximum amount of storage required by all blocks (nested blocks adding to the outer enclosing block). This storage is allocated when the function is entered.

For example:

```
for (i = 0; i < NROWS; i++) {
    int row_total = 0;
    for (j = 0; j < NCOLS; j++)
        row_total += matrix[i][j];
    printf("row %d total is %d\n", i, row_total);
}
```

Expression and Null Statements

```
expression-statement:
  expressionopt ;
```

The expression in an *expression statement* is evaluated as a void expression for its side effects. Some typical examples of this are assignments, functions calls, and the use of the ++ and — operators.

A *null statement* does nothing. (Such a statement consists of just a semicolon, or matching braces not containing any statements.)

A function call may be evaluated for its side effects only - that is, the function's return value is not needed. In such a case, it is possible to discard the returned value explicitly, by using a cast to convert the expression to a void expression. (Such a conversion is not mandatory, however.)

For example:

```
int p(int);    /* function declaration */
/*...*/
(void) p(0); /* function call*/
```

In the program fragments:

```
char *s;
/*...*/
while (*s++ != '\\')
    ;
```

and

```
char *s;
/*...*/
while (*s++ != '\\')
    { }
```

null statements are used to supply empty loop bodies to the iteration statements.

Jump Statements

```
jump-statement:
goto identifier ;
continue ;
break ;
return expressionopt ;
```

A *jump statement* causes an unconditional transfer of control to another place within the function. If this causes a block to be exited, destructors are called for all constructed objects of that scope not yet destroyed.

The goto Statement

A goto statement causes an unconditional jump to the named label in the current function. The label must be a prefix for a statement in the same function.

For example:

```
/* ... */
val++;
if ( val > CUTOFF)
    goto alldone;
/* ... */
alldone: printf ( "all done\n");
```

The continue Statement

A continue statement can appear only within an iteration statement. A continue statement causes a jump to the end of the loop body of the enclosing iteration statement. The loop will continue executing from this point. I.e., the loop condition will be tested again, etc.

For example:

```

for (i=1; i<10; i++)
{
    if (a[i+1]==1)
        continue;
    a[i+1]=0;
    // continue arrives here
}
while (i)
{
    switch (i)
    {
        case 1: if (a[i+2]==2)
                    continue;
        case 2: continue;
        default:
    }
    a[i+4]= 0;
    // continue arrives here
}

do {
    if (redo) continue;
    // continue arrives here
} while (i);

```

The break Statement

A break statement can appear only within a switch or iteration statement. When encountered during execution, a break statement terminates execution of the smallest enclosing switch or iteration statement.

For example:

```

switch (k)
{
    case 0: for (i=1; i++; i<10)
    {
        for (j=1; j++; j<10)
            if (j==5)
                break; // terminate j loop
        else
            a[i+j]=0;
        if (a[i+5]==6)
            break; // terminate i loop
    }
    /* fall through */

    case 1: switch (l)
    {
        case 0: a[1]=0; // fall through
        case 1: a[2]=0;
                break; // terminate l switch
        default :
    }
    break; // terminate k switch
}

```

The return Statement

A return statement terminates execution of the function in which the statement is contained. After the return statement is executed, control is returned to the function's caller.

A return statement for a function that returns a value must include an expression. When a return statement with an expression is executed, the expression is evaluated, and its value is returned to the caller. If the expression's value has a different type from the enclosing function's return type, the value is converted, as if the value were used to initialize an object of that type.

Reaching the end of a function is equivalent to executing a return statement without an expression. This is illegal unless the type of the function is void.

A function may contain any number of return statements. A function with return type void may not contain a return statement with an expression.

In TopSpeed C++ the storage for the return value is allocated in the calling function.

For example:

```
int f1()
{
    int i;
    /* ... code ... */
    return i;
}
complex f2()
{
    complex v1;
    /* ... code ... */
    if (a[6]==9)
        return v1;
    /* ... more code ... */
    return v1;
}
```

Selection Statements

```
selection-statement:
    if ( expression ) statement
    if ( expression ) statement else statement
    switch ( expression ) statement
```

A *selection statement* chooses among a set of statements. The selection is dependent on the value of a *controlling expression*. The statement selected may not be a *declaration statement*.

C++ provides two types of selection statements: the if and switch statements.

The if Statement

The if statement has two forms:

- An if statement with no alternative action specified.
- An if statement with an alternative action specified in an else clause.

In each form, the if statement has a controlling expression and a (simple or compound) *substatement* associated with it. The controlling expression of an if statement must have scalar type, and must be enclosed in parentheses.

In both forms, the controlling expression is evaluated. If this result is nonzero, the first substatement is executed.

If the controlling expression evaluates to zero, the action taken depends on which form of the if statement is being used.

- If no alternative action is specified, the program does nothing in the if statement. Execution continues with the statement after the if statement.
- If an alternative action is specified, the substatement associated with the else clause will be executed.

If the first substatement is reached via a label (i.e., a goto is performed into the first branch of the if statement), the second substatement is not executed.

An else is associated with the nearest preceding if that satisfies the following conditions:

- The if must be in the same block (but not in an enclosed block)
- The if must not yet be matched with an else.

For example, compare the following two if statements:

```
if (index < TABSIZE)
    if (table[index])
        zero++;
    else
        puts("Error: index out of bounds");
if (index < TABSIZE) {
    if (table[index])
        zero++;
}
else
    puts("Error: index out of bounds");
```

The first one is probably not what was intended, since it displays an “out of bounds” message if the value of `table[index]` is zero, rather than if `index` is too large.

The switch Statement

The switch statement is used to select from among several possible actions, depending on the value of a *controlling expression*. This value will determine the point in the *switch body* to which control will jump during execution. The jump will be to one of the following:

- To the statement following a case label whose expression has the same value as the controlling expression.
- To the statement following the default label in the switch body, if no case label matches the value of the controlling expression and if the switch body contains a default label.
- To the statement following the end of the switch body, if no case label matches the value of the controlling expression and if the switch body does not include a default label. In this case, none of the statements in the switch body is executed.

The controlling expression for a switch statement must have integral type, and the case labels must be integral constant expressions. The integral promotions are performed on the controlling expression. The constant expression in each case label is converted to the same type as the promoted controlling expression. All case constants in the same switch statement must have different values after conversion. There may be at most one default label in a switch statement.

A case or default label is accessible only within the closest enclosing switch statement.

In TopSpeed C++, the number of case labels in a switch statement is limited only by the amount of memory available.

For example:

```

switch (error_level) {
    case 0 :
    case 1 :
        warnings++;
        /* fall through */
    case 2 :
        errors++;
        printf("Error (level %d) : %s\n",
            error_level, error_msg);
        break;
    default :
        puts("FATAL error");
        exit();
}
switch (value) // untraditional use of switch statement
    default :
        if (value > 5)
            case 0 :
            case 1 :
                value = 0;
        else
            case 9 :
                value = 1;
/* value is set to 1 if it was originally
   2, 3, 4, 5 or 9 (or negative)
   or set to 0 if it was originally
   0, 1, 6, 7, 8 or greater than 9
*/

```

Iteration Statements

```

iteration-statement:
    while ( expression ) statement
    do statement while ( expression ) ;
    for ( for-init-statement expropt ; expropt ) statement
for-init-statement:
    expression-statement
    declaration-statement

```

Iteration statements are used to repeat something, possibly with variations. An iteration statement includes a controlling expression and a loop body.

The *controlling expression* must have scalar type. This expression is evaluated to determine whether to execute another iteration of the loop body. That is, an iteration statement causes the loop body to be executed repeatedly until the controlling expression evaluates to zero. The *loop body* refers to the statements executed as long as the controlling expression is nonzero. The loop body may be a simple or compound statement.

Iteration statements take one of three forms:

- The while statement
- The do, or do-while statement
- The for statement

The while Statement

In a while statement, the loop body executes as long as the controlling expression is nonzero. The evaluation of this expression takes place *before* each execution of the loop body. Thus, the while statement executes zero or more times.

For example:

```
while (index < TABLE_SIZE)
    if (table[index] = key)
        break;
    else
        index++;
```

The do Statement

In a do statement, the evaluation of the controlling expression takes place *after* each execution of the loop body. Thus, the do statement executes at least once.

For example:

```
do {
    ch = getchar();
    switch (ch) {
        /* ... */
    }
} while (ch != 'Q');
```

The for Statement

The for statement makes it possible to specify a while statement more succinctly. Except for the behavior of a continue statement in the loop body, the statement

```
for ( init-statement
      ex-2 ;
      ex-3 )
    statement
```

and the sequence of statements:

```
init-statement
while (ex-2) {
    statement
    // continue in for-loop would arrive here...
    ex-3 ;
    // ...but continue in while-loop arrives here
}
```

are equivalent.

Thus *init-statement* specifies the *initialization* for the loop (note that this always ends in a semicolon). *ex-2* is the controlling expression; it specifies an evaluation made before each iteration. Execution of the loop continues until this expression evaluates to zero. *ex-3* specifies an operation that is performed after each iteration.

The *init-statement* may be an empty statement, and is evaluated only for its side effects. Similarly, the expression *ex-3* may be omitted, and may have any type, including void, and is evaluated only for its side effects. If *ex-2* is omitted, it is treated as if a non-zero constant had been written - i.e., the loop condition is always true.

For example:

```
int count = 10;

// terminates when count reaches 100
for ( ; count < 100; )
    printf ( "%d hello\n", count++);

// executes nine times, for count = 1 through 9
for ( count = 1; count < 10; count++)
    printf ( "%d hello\n", count);

// executes forever
for ( ; ; )
    printf ( "Forever\n");
```

Declaration Statements

```
declaration-statement:
    declaration
```

A declaration statement is used to declare a new identifier in the scope of the immediately enclosing block. If the same name was declared in an enclosing block, the previous declaration is hidden for the duration of the block.

A variable declared in a declaration statement is by default auto. If it is auto or register, it is created each time control flows past the declaration statement which declares it, and destroyed at the end of the block. If it is static, it is initialized only the first time that control flows past its declaration statement.

An auto variable declared in a loop is created and destroyed once per iteration around the loop. A declaration in a *for-init-statement* is in the scope of the block or statement which immediately encloses the for statement.

When a jump is made out of a loop or block, or back past a declaration statement of an initialized auto variable, any variables which are declared at the point jumped from but not at the point jumped to are destroyed.

A jump into or within a block or loop is illegal if, for any variable that is not declared at the point jumped from but is declared at the point jumped to, the declaration involves an implicit or explicit initialization.

A variable declared in a statement under the control of an if statement has a lifetime that is limited to its own arm of the if statement, and cannot be accessed outside it. For example:

```
if (b)
    for(int j=0; j<10; j++)
        // ...
        // j destroyed here
else
    j = 0; // error - j not accessible

j = 0;    // error - j not accessible
```

Expression Ambiguity

The grammar of C++ is ambiguous, in that certain forms of expression statements can be indistinguishable from certain forms of declarations. For example:

```
int (a);
```

could be interpreted as either a declaration of a variable called `a` of type `int`, or a conversion of the variable called `a` to type `int`, the result being discarded.

In such cases, a statement that (at the syntactic level) could be either a declaration or an expression statement is always treated as a declaration. The meanings of any identifiers (beyond whether or not they are type names) are not considered when resolving this ambiguity, so that a statement may be treated as an illegal declaration even if it would be legal if treated as an expression. The use of the `auto` keyword can be useful to explicitly indicate that an ambiguous construct is a declaration.

Note that function declarations within a block require a type to be specified, even though they do not require it when declared outside a block. This removes a possible ambiguity between declarations and calls of functions:

```
main()
{ int foo();      // declaration
  foo();          // function call
}
```

CHAPTER 9

CLASSES

A class is a user-defined type. Once defined, the class-name may be used as a type name (in the same way that `int` is a type name) for the remainder of its scope.

Class Names

A class name is simply an identifier.

```
class-name:
    identifier
```

A class specifier (frequently known as a class declaration, but better described as a class definition) is used to define a new class name.

```
class-specifier:
    class-head { member-listopt }
class-head:
    class_key identifieropt base-specopt
    class-key class-name base-specopt
class-key:
    class
    struct
    union
```

A declaration which uses an *elaborated-type-specifier* (See Chapter 5: ‘Elaborated Type Specifiers’) with a *class-key*, but omits the declarator list, introduces a class name without defining it. Once a class name has been introduced, it may be used in extern declarations and in pointer and reference declarators, provided the definition is completed before any objects so declared are used.

A class in which the name is omitted would normally be used in ANSI C style typedef declarations (see Chapter 5: ‘Typedef Declarations’). For example, the construct

```
typedef struct { float real; float imag } complex;
```

is equivalent to

```
struct complex { float real; float imag };
```

A *typedef-name* used to name a class in this way is a *class-name*.

An optional *base-spec* may be used to declare a derived class. These are described in Chapter 10.

If the member list is empty, the class is an empty class. The size of an object of an empty class is not zero, and it will have a distinct address allocated for it.

The class name is introduced into the current scope as soon as the class name is seen, and may be used in pointer and reference declarators within the member-list. Any object, class etc. declared with the same name in an enclosing scope is hidden. A class may be declared in the same scope as an object, function or enumerator of the same name - in this case, an *elaborated-type-specifier* will be required to refer to the class name. This allows compatibility with ANSI C, where structure tags occupy a separate namespace from other names. For example:

```
struct point { /* ... */ };
point a;      // No need to elaborate here in C++
int point();  // point redeclared
struct point b; // elaborated type required now
```

It is an error to define more than one class with the same name in the same scope, even if the classes are identical.

Class Members

```
member-list:
    member-declaration member-listopt
    access-specifier : member-listopt
member-declaration:
    decl-specifieropt member-declarator-listopt ;
    function-definition ;opt
    qualified-name ;
member-declarator-list:
    member-declarator
    member-declarator-list , member-declarator
member-declarator:
    declarator pure-specifieropt
    identifieropt : constant-expression
```

The member list defines all data, functions, enumerations, bitfields, classes, friends and type-names that make up the members of the class. It may also contain *access-specifiers* controlling what access is permitted to these members - these are described in Chapter 11.

A class member may not be defined more than once in the member list, nor may additional members of a class be defined outside the member list. A member function is characterized by its argument types as well as by its name, so that several member functions with the same name will not constitute a multiple definition provided the types are different. This is called *overloading*, and is described in Chapter 13.

Data member names must be unique within the member list.

Member declarators may not contain initializers (see Chapter 6: ‘Initialization’). Members may not be auto, register or extern.

The *decl-specifiers* are optional only for member function declarations. The *member-declarator-list* is optional only after a *class-specifier* or *enum-specifier*, or after a *decl-specifier* of the form friend *elaborated-type-specifier*.

A *pure-specifier* may only appear when declaring a virtual function - see below.

If any member object is defined as a class object, the definition of the named class must be complete. Thus a class may not contain an instance of itself (except as a static member). It may however have members which are pointers to itself.

A non-static member object which is an array must have all dimensions specified.

For non-static data members, it is guaranteed that where there is no intervening access-specifier, later members are allocated at higher addresses than those declared earlier. In TopSpeed C++, these addresses are contiguous - however, this may not be the case for all C++ compilers on all processors, so for the sake of portability, it is unwise to rely upon this fact.

No static data member, enumerator, or nested type may have the same name as its class. A member function with the same name as its class is a *constructor* - see Chapter 12.

Member Functions

If a function is declared in a class member-list, and does not have the friend specifier, it is a member function. Member functions are called using the class member syntax (see Chapter 7: ‘Member Selection Operators’).

Member functions may be both declared and defined within the class-specifier, or may be declared inside the class-specifier and subsequently defined outside it. The definition is considered to be within the scope of the class, so that it can use the names of members of its class directly, the object so referred to being the members of the object for which the function is called. Static member functions may only refer directly to static data members, enumerators, and nested types.

If a member function is defined after the class declaration, the function name must be qualified by the class-name. For example:

```
struct namelist {
    char name[20];
    namelist *next;
    int search(const char *);
};
int namelist::search(const char *a)
{ if (strcmp(a, name) == 0) return 1;
  return (next != NULL) ? next->search(a) : 0;
}
```

A member function may be named as a friend in subsequent class declarations.

A member function that is called must have exactly one definition in a program. Declarations of inline member functions may be defined in several source files, provided that the definitions are identical.

Non-static member functions must only be called for objects of the appropriate class. Otherwise, the effect is undefined.

Const and Volatile Member Functions

In a member function which is declared with the `const` qualifier, all non-static members of the object for which the function is called are regarded as `const`, and any attempt to modify them will give a compile-time error. A `const` member function may be invoked for objects which are `const` or non-`const`, but a non-`const` member function can only be invoked for objects which are non-`const`.

Analogous rules apply for a member function declared with the `volatile` qualifier. A member function may be declared as both `const` and `volatile`.

As an exception to the above, constructors and destructors may be called for objects which are `const` or `volatile`, even though the constructor or destructor cannot be declared `const` or `volatile`.

The `const` or `volatile` qualifier applied to a member function in this way is in effect modifying the type of the `this` pointer, as described below. In particular, note that it is therefore possible to overload member functions that differ only in the presence or absence of the `const` and/or `volatile` specifiers. For the same reason, if a member function with the `const` and/or `volatile` qualifier is not defined inline in the class declaration, the same qualifiers must be specified on the function definition.

For example:

```

struct X {
    int a;
    static int b;
    void func1() const;
    void func1();          // overloads func1 above
    void func2();
};

void X::func1() const {
    a = 0;    // error - *this is const
    b = 0;    // ok - static member
    func2();  // error - can't call non-const member
              // function for const object *this
}

void X::func2() {
    a = 0;
    b = 0;
}

```

The this Pointer

In the definition of a member function (other than a static member function), the keyword `this` is defined to be a pointer to the object for which the function was called. Members of the class can be referred to as `this->membername`, although this is seldom necessary as member names are directly visible within member function definitions.

The `this` pointer may not be modified within the member function (that is, its type is `classname *const`, unless the member function has been declared as `const` and/or `volatile`, in which case it is `const classname *const`, `volatile classname *const` or `const volatile classname *const` as appropriate).

For example:

```

struct namelist {
    char name[20];
    namelist *next;
    namelist *search(const char *);
};

namelist *namelist::search(const char *a)
{ if (strcmp(a, name) == 0) return this;
  return (next != NULL) ? next->search(a) : NULL;
}

```

Inline Member Functions

If a member function is defined in the class declaration, the effect is equivalent to declaring it as `inline`, with the definition rewritten to appear immediately after the class declaration. This notional rewriting occurs after preprocessing but before syntax analysis.

This has several implications on the scope of any names used in the function definition. For example, the function body may refer to members (including function members) which are defined later in the member list. In addition, the function body may refer to the size of the class, even though not all members have been seen at the point where the function definition appears (before rewriting).

Inline member functions may be defined in nested or local classes, where rewriting as described above would produce syntax errors. (Indeed, member functions of local classes may only be defined inside the class declaration.)

For example:

```
int func1();

struct X {
    int func1() { return func2(); };
    int func2() { return func1(); };
};
```

is legal (if rather useless) and defines two mutually recursive functions, as it is equivalent to

```
int func1();

struct X {
    int func1();
    int func2();
};
inline int X::func1() { return this->func2(); }
inline int X::func2() { return this->func1(); }
```

Static Members

If the declaration of a member of a class contains the specifier `static`, the member is a static member. Static members of a global class have external linkage.

A static data member has only one copy, shared by all objects of the class, rather than a copy in every object. The declaration of a static data object does not define the object - the definition must occur elsewhere. Thus static data members may be declared with incomplete types, provided that the type is fully defined at the time when the definition of the static data member is encountered. Local classes may not have static data members.

Static member functions do not have a `this` pointer, as they can be called independently of any object of the class type. A static member function may not therefore be declared `const` or `volatile`. Static member functions cannot be `virtual`, and must be distinct in name or argument types from all non-static member functions of the same class. Static member functions of local classes have no linkage, and must be defined inline (see above).

Static members can be referred to using the `::` scope resolution operator (i.e., *member::classname*), without reference to any object of the class. They may also be referred to using the member access operators `.` and `->`. In the latter case, only the type of the expression on the left hand side is used to determine the class to which the static member belongs, and the expression is not evaluated.

Static members exist even when no objects of their class have been defined. A static data member not explicitly declared as `const` is modifiable, even if accessed via a `const` object of the class.

Static members are initialized in file scope, in the same way as global objects. The scope resolution operator is required in the definition. Static function members may also be initialized inline. Apart from initialization, static members obey the same access rules as other class members - see Chapter 11.

The type of a static member is independent of its class.

Static data members can be regarded as global variables that are specific to a class. As their names will not clash with any global names or names from other classes, they will not affect other parts of a program.

For example:

```
class Car {
    Car *next;
    static Car *active;
    /* keep a list of all active
       objects of this class */
    // ... other data members
    static Moveall();
    /* Moveall member function does not
       refer to a particular object, so
       needs no this pointer */
};

struct myclass {
    static int a;
    const static int b;
    myclass (); // default constructor
};

void myfunc()
{ const myclass m1;
  myclass m2;
  m1.a = 0; // ok - m1 is const, but
            // static members are not
  m1.b = 0; // error - b is const
  m2.a = 0; // ok
  m2.b = 0; // error
}
```

Unions

A union is akin to a structure with all data members overlapping, rather than being allocated separate memory. The size of the union is sufficient to contain the largest of its member objects. Assignment to any data member of a union affects all others.

Unions may not be derived from other classes, nor may other classes be derived from them.

Unions may have constructors, destructors, and other member functions, other than virtual member functions.

Unions may not have static data members, nor data members which are objects of a class for which a constructor, destructor, or user-defined assignment operator has been defined.

For example:

```
struct long_name {
    long zero;
    char *name_ptr;
};

union name_descriptor {
    char name[8];
    long_name lname;
};

void write_name(name_descriptor &thisname)
{ if (thisname.lname.zero == 0)
    cout << thisname.lname.name_ptr;
  else
    cout << thisname.name;
}
```

Anonymous Unions

An anonymous union is declared as follows

```
union { member-list } ;
```

where no objects or pointers are declared for the union, and no name is specified.

An anonymous union defines an unnamed object of union type. The members of the union are accessed directly, without using the usual *objectname.member* syntax, and all member names must therefore be distinct from other names in the current scope. The members can be used just like ordinary variables declared in the current scope, except that they are all allocated at the same address.

Anonymous unions may not have function members. All data members must be public. A global anonymous union must be declared as static, and all members will have static linkage.

For example:

```
struct long_name {
    long zero;
    char *name_ptr;
};

struct name_descriptor {
    union {
        char name[8];
        long_name lname;
    };
    // other information ...
};

void write_name(name_descriptor &thisname)
{ if (thisname.lname.zero == 0)
    cout << thisname.lname.name_ptr;
  else
    cout << thisname.name;
}
```

Bit Fields

A member declarator of the form

identifier_{opt} : constant expression

declares a bit-field member. The constant expression indicates the number of bits (including the sign bit, if required) to be allocated to the object. This may provide a means of storing more than one member in the same word of storage.

The constant expression that specifies the width of the bit-field must have integral type, must be non-negative, and must not exceed the number of bits in a long int. If the width is zero, the bit-field must not be named.

A plain int bitfield is treated as signed in TopSpeed C++.

The unary & operator cannot be applied to a bitfield, so there are no pointers or references to bit-fields.

If the identifier naming the bit-field is omitted, an unnamed bit-field of the specified size is introduced - the purpose of such declarators is to introduce padding. As a special case, if the bit-field width is specified as zero, this indicates that the next bit-fields is to start at the next word boundary.

The first bit-field encountered is the least significant in the byte, word, or doubleword in which the bit field is allocated. For example:

```

struct {
    int aa : 2;
    int bbb : 3;
    int : 7;
    int c: 1;
    int ddd : 3
}

```

will pack as dddxxxxxxxxbbbaa in a 16 bit int, where the rightmost a is the least significant bit, and x represents an anonymous bit.

Unlike in ANSI C, any integral type may be used for a bit-field. ANSI C permits only int, unsigned int and signed int, while TopSpeed C additionally allows char and unsigned char.

Nested Class Declarations

A class declared within another class is known as a *nested class*. Such a class is in the scope of the enclosing class, and its name is local to that class.

Declarations in a nested class may refer to type names, static members and enumerators from its enclosing scope. Other members may not be referred to except by means of explicit member access operators . or -> using objects or pointers of the appropriate types.

Member or friend functions of a nested class cannot access private members of the enclosing class, nor can member or friend functions of the enclosing class access private members of the nested class - the normal access rules apply (see Chapter 11).

The scope resolution operator :: may be used to define or reference member functions and static data members of nested classes, for example:

```

struct outer {
    struct inner {
        static int x;
        int f(int i);
    }
}

int outer::inner::x = 1;

typedef outer::inner in;
int in::f(int i) { /* ... */ }

```

Note that in the example above, outer is an empty class, as no members are declared within it. The declaration of a nested class does not imply that a member object of that class is created.

Note also that the scope rules for class names in C++ are the same as the rules for any other names. This marks an incompatibility with C, where a struct type declared within another struct would still be visible outside the enclosing struct.

Local Class Declarations

A class that is declared within a function definition is called a local class. The name of such a class is local to the function in which it is declared, and the local class is in the scope of the function.

Declarations in a local class may refer to type names, static variables, extern variables and functions and enumerators from its enclosing scope. Automatic variables may not be referred to.

The enclosing function cannot access private members of the local class - the usual access rules apply (see Chapter 11).

A local class may not have static data members. Any member functions must be defined inline within the class declaration.

Local Type Names

A type name may be introduced within a class declaration using a typedef declaration within the member list. Such type names have the same scope rules as other names, so that a type name declared within a class is local to that class, and cannot be used outside it except within member functions or if explicitly qualified with the class name.

A name which is a *class-name*, a *typedef-name*, or the name of a constant used in a type name, may not be redefined within a class declaration after it has been used in that class declaration. A name which is not a *class-name* or a *typedef-name* may not be redefined as a *class-name* or *typedef-name* after it has been used in a class declaration. These rules limit the context sensitivity of class member declarations, and in particular the rewrite rules for inline member functions. For example, the following is an error :-

```
struct X {  
    int f() { return sizeof (INT); };  
    typedef long INT;    // error - INT has been used  
};
```

CHAPTER 10

DERIVED CLASSES

Introduction

In an object-oriented programming language, it is often useful to be able to express relationships between classes of objects where, for example, classes B and C are varieties of class A. Any object which is a B must by definition also be an A, and so an object of class B has all the properties of class A, as well as some of its own describing the more specific properties of class B. This process is called inheritance, and in C++ is implemented by the mechanism of derived classes.

Derived Classes

A class declaration includes an optional *base-spec* controlling the inheritance :-

```

base-spec:
    : base-list

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

base-specifier:
    class-name
    virtual access-specifieropt class-name
    access-specifier virtualopt class-name

access-specifier:
    private
    protected
    public
  
```

The *class-name* in the *base-specifier* must name a class which has been previously declared (and defined) - see Chapter 9. A class declared with such a *base-spec* is said to be derived from the class named, which is in turn called a base class of the derived class. All members of the base class are *inherited* by the derived class - that is, unless redefined, they behave as if they were members of the derived class.

A class which is named in the *base-list* is called a *direct base*. Other classes which are base classes of classes named in the *base-list*, but are not themselves named in the *base-list*, are called *indirect bases*.

The scope resolution operator `::` may be used to refer explicitly to a member of a base class which is hidden by a member of the derived class.

A pointer to a derived class may be used wherever a pointer to its base class is required, and an implicit conversion will occur, provided that the base class is public, and the conversion is unambiguous (see below). Similarly, a reference to a derived class will be implicitly converted to a reference to a public base type.

If the keyword `virtual` is present in the base specifier, a virtual base class is specified. The difference between a virtual base class and a non-virtual base class is described below.

For example:

```
struct base {
    int a, b;
    int foo();
};

struct derived : public base {
    char *a;          // overrides the a in base
    int fl();
};

void g(derived *dp)
{
    dp->a = "hello";
    dp->b = 0;          // b is inherited
    dp->foo();          // foo is inherited
    dp->base::a = 0;    // refers to inherited a
}
```

Multiple Inheritance

Several base classes may be named in the *base-list* of a derived class - this is known as multiple inheritance. It is not legal to name the same class twice as a direct base class, but it may be an indirect base more than once. For example:

```

class list {
    list *next;
    /* ... */
};
class alist : public list {
    /* ... */
};
class blist : public list {
    /* ... */
};

class ablist : public alist, public blist {
    /* ... */
};

```

An object of class ablist will contain sub-objects of class alist and blist, as well as any additional members defined for it. It will also contain two sub-objects of class list, one in the alist sub-object and one in the blist sub-object. A member of ablist that is inherited from class list cannot be referred to directly without an ambiguity arising - such ambiguities can be resolved using the scope resolution operator `::` - for example `alist::next` or `blist::next`.

Virtual Base Classes

In the example above, the class ablist contains two sub-objects of class list - this would be appropriate if the intention was to maintain two lists, one of alist objects and one of blist objects, and that ablist objects should be found on both lists.

An alternative scheme, where a single list is maintained to contain objects that are of class alist, blist or ablist, can be achieved by using the keyword `virtual` in the base class specifier.

```

class list {
    list *next;
    /* ... */
};

class alist : public virtual list {
    /* ... */
};

class blist : public virtual list {
    /* ... */
};

class ablist : public alist, public blist {
    /* ... */
};

```

A single sub-object of a virtual base class will be shared by all derived classes in which it is declared `virtual`. In this example, an object of class ablist has a single member `next`, and it can be referred to unambiguously without requiring any use of the scope resolution operator.

A derived class may inherit a mixture of virtual and non-virtual base classes of a type T. In such cases, the derived class will have one sub-object of type T for each non-virtual base of type T it inherits, plus one shared copy for all the virtual bases of type T.

Ambiguities

An access to a class member is ambiguous if it could refer to more than one member inherited from different bases, and the compiler cannot determine which. A compilation error is reported in such cases.

Note that ambiguity is checked before access control (see Chapter 11), so that even a private member inherited from a base class can cause an ambiguity to arise if another member of the same name is inherited from a different base class. For example:

```
class base1 {
    int a;    // note: base1::a is private
    // ...
};

class base2 {
public:
    int a;    // base2::a is public
    // ...
};

class derived : public base1, public base2 {
    // ...
};

void foo (derived *dp)
{ dp->a = 0; // ambiguous, base1::a or base2::a
  // even though base1::a is private
  dp->base1::a = 0; // error - member is private
  dp->base2::a = 0; // ok
}
```

Ambiguities can be resolved using the scope resolution operator `::`. The class-name named on the left hand side of the scope resolution operator need not be the class in which the specified member is found - it merely indicates where to start searching.

Where virtual base classes are inherited, it is possible to reach the same sub-object via two routes. This is not an ambiguity. For example:

```

class list {
    list *next, *prev;
    /* ... */
};

class alist : public virtual list {
    /* ... */
};

class blist : public virtual list {
    /* ... */
};

class ablist : public alist, public blist {
    /* ... */
    void unlink();
};

void ablist::unlink ()
{ next->prev = prev;
  prev->next = next;
  // alist::next and blist::next refer to the
  // same member, so no ambiguity
}

```

If a name *dominates* another name then no ambiguity can arise between the two, and the dominant name will be assumed if there is a choice. A name *n1* is said to *dominate* another name *n2* if *n1* is defined in a class which is derived (directly or indirectly) from the class in which *n2* is defined. Note that this can be significant in resolving ambiguities only if virtual base classes are involved, as otherwise the name *n2* would be hidden by *n1* if they were identical.

For example:

```

struct V {
    int i;
    void f(int);
};

struct base1 : virtual V {
    int i;          // dominates V::i
    void f(int);    // dominates V::f()
};

struct base2 : virtual V {
    // ...
};

struct derived : base1, public base2 {
    void foo() { f(i); }; // ok
};

```

In this example, there is no ambiguity between `base1::i` and `base2::V::i`, nor between `base1::f()` and `base2::V::f()`, since in both cases the member in `V` is dominated by the member in `base1`.

A pointer to a derived class may only be explicitly or implicitly converted into a pointer to a base class if the conversion refers unambiguously to a single sub-object.

For example:

```
class vbase { };
class base { };
class d1 : public base, public virtual vbase
{ };
class d2 : public base, public virtual vbase
{ };
class derived : public d1, public d2
{ };

void myfunc(derived *p)
{
    base *bp = p; // error - ambiguous
    vbase *vp = p // ok
}
```

Virtual Functions

A member function defined using the virtual keyword is called a virtual function. If a base class contains a virtual function, and a function with the same name and type is declared in a class derived from this base type, the derived class member function will override the base class member function. Any call of the function for an object of the derived type will then call the derived class member function rather than the base one.

For a virtual function, the function which is called is determined solely by the type of the object for which the function is invoked, even if the expression that references it has been converted to some other type. This contrasts with non-virtual functions, where the function called is determined by the type of the expression which denotes the object for which the function is called. For a non-virtual function call, the function to be called can be determined at compile-time, whereas for a virtual function call, it cannot in general be determined until run-time. For example:

```

struct base {
    virtual void vf();
    void f();
};

struct derived : public base {
    void vf(); // vf() is virtual
    void f();
};

void test (derived *dp, base *bp)
{ dp->vf(); // derived::vf() called
  dp->f();  // derived::f() called
  bp->vf(); // derived::vf() called
  bp->f();  // base::f() called
}

main()
{ derived d;
  test (&d, &d);
}

```

A virtual function can only be declared as a member of a class. A virtual function may not be a static member. A member function in a derived class which overrides a virtual function is itself virtual, whether the virtual keyword is present or not. It is an error if a member function in a derived class differs from a virtual function in a base class only in its return type.

A derived class that inherits a virtual function need not declare an overriding function. In this case, the inherited function is used by calls for objects of the derived type.

A virtual function may be declared as a friend in another class (see Chapter 11: ‘Friends’).

If a virtual function in a base class is not defined, and is not declared pure (see below), a link error will result.

If the scope resolution operator is used to refer to a virtual function, the virtual call mechanism is suppressed, and the function referred to is called directly. This is commonly used within the definition of a virtual member function of a derived class, in order to call the virtual member function of the base class which it overrides. For example, given the definitions above, we could have

```

derived::vf()
{ base::vf(); // calls base::vf explicitly
  vf();      // recursive call
}

```

Abstract Classes

A class in which one or more pure virtual functions are defined is called an *abstract class*. A pure virtual function is a virtual function whose declarator contains a *pure-specifier* :-

```
member declarator:  
    declarator pure-specifieropt  
pure-specifier:  
    = 0
```

Abstract classes may not be used as the type of an object, argument or function return value. Pointers and references to an abstract class may be used, however.

An abstract class may only be used as a base class for other classes.

If a derived class has an abstract class as a direct base class, it will inherit one or more pure virtual functions. Unless the derived class defines overriding functions for all inherited pure functions, the derived class will itself be abstract.

If a pure virtual function is called, directly or indirectly, TopSpeed C++ will report a run-time error.

For example:

```
class abstract {  
    virtual void func1() = 0;  
};  
  
class also_abstract : abstract {  
    int a, b;  
    void func2();  
    // inherited function func1 is pure,  
    // so this class is abstract too  
};  
  
class not_abstract : also_abstract {  
    virtual void func1();  
    // overrides pure virtual function,  
    // so this class is not abstract  
};
```

CHAPTER 11

MEMBER ACCESS CONTROL

An important feature of classes is their use for data-hiding. By disallowing access to members except from within member functions, the programmer is restricted to using a well-defined interface for all accesses to the class. Thus all parts of a program which rely on a particular implementation of the class are localized.

Member Access Specifiers

In the member-list of a class declaration, an access specifier may be given:

```
member-list:  
  access-specifier : member-listopt  
  
access-specifier:  
  public  
  private  
  protected
```

An *access-specifier* determines the access control applied to all members which follow it in the *member-list*, until either another *access-specifier* or the end of the class declaration is reached.

Members defined before the first access specifier in a class declared using the class keyword are private by default - in a class declared using struct or union they are public by default.

Access specifiers may occur in any order in the member list. The same access specifier may occur several times in the member list.

A member which is private can only be used by member functions and friends declared in the member list of the class to which it belongs.

A member which is protected can be used by member functions and friends declared in the member list of the class to which it belongs, and by member functions and friends of any class derived from this class.

A member which is public can be used anywhere where it is in scope.

For example, given the class declarations

```
class myclass {
private:
    int priv;
protected:
    int prot;
public:
    int pub;

    void memberfunc();
    friend void friendfunc(myclass *);
};

class derived : public myclass {
    void derivedmember();
};
```

the following access rules would apply :-

```
void myclass::memberfunc()
{ priv  = 0; // ok
  prot  = 0; // ok
  pub   = 0; // ok
}

void friendfunc (myclass *p)
{ p->priv  = 0; // ok
  p->prot  = 0; // ok
  p->pub   = 0; // ok
}

void derived::derivedmember ()
{ priv  = 0;    // error - not accessible to derived
  prot  = 0;    // ok
  pub   = 0;    // ok
}

void anyfunc (myclass *p)
{ p->priv  = 0; // error - private
  p->prot  = 0; // error - private
  p->pub   = 0; // ok
}
```

Base Class Access Specifiers

Each base class declared in the *base-list* of a derived class declaration may have an *access-specifier* given:

```
base-specifier:
    virtual access-specifieropt class-name
    access-specifier virtualopt class-name
```

If the keyword `public` is used, the base class is called a public base class. In this case, all public members of the base class are inherited as public members of the derived class, and all protected members of the base class are inherited as protected members of the derived class.

If the keyword `private` is used, public and protected members of the base class are inherited as private members of the derived class.

In either case, private members of the base class are not accessible to a member function of a derived class, unless access is specifically granted using friend declarations in the base class.

If no *access-specifier* is used to qualify the base class, public is assumed for derived classes declared using `struct` or `union`, and private for derived classes declared using `class`. TopSpeed C++ will issue a warning in such cases.

The access rules for members inherited from base classes only apply when these members are accessed as members of the derived class. In the following example, a member function of the derived class `d2` cannot access the members `a` and `b`, as they are private members of its base class `d1`. However, these fields can still be accessed via a pointer to a base class in which they are public. As `b` is a public static member of base, any function can access it by qualifying it with its class.

The expression `base::b` in `d2::f2()` is interpreted as accessing the static member of base class `b`, rather than as `this->base::b`, which would look up the member of `d2` called `b`, in the base class sub-object `base`. The latter would not be legal, as when accessed as a member of `d2`, `b` is inaccessible.

For example

```

class base {
public:
    int a;
    static int b;
};

class d1 : private base {
    void f1(base *);
};

class d2 : public d1 {
    void f2(base *);
};

void f0(base * bp)
{ bp->a = bp->b; // ok - these are public
  base::b++;   // ok - public static member
}

void d1::f1(base * bp)
{ bp->a = bp->b; // ok - these are public
  base::b++;   // ok - public static member
  a = b;       // ok - member function can
               // access private members
  b++;         // ok - member function can
               // access private members
}

void d2::f2(base * bp)
{ bp->a = bp->b; // ok - these are public
  base::b++;   // ok - any function can
               // access public static
  a = b;       // error - inaccessible
  b++;         // error - inaccessible
}

```

A member or friend of a derived class X can implicitly convert a pointer to X into a pointer to a private immediate base class of X.

Access Declarations

If a derived class has a private base class, the public and protected members of the base class become private members of the derived class, as described above. The access control of individual members inherited in this way may be altered by means of *access declarations*.

An access declaration is made by mentioning the name of the imported member, qualified by the base name and a scope-resolution operator, in the derived class member-list. No type may be specified in an access declaration. The derived class may not declare a member of the same name.

Access declarations may not restrict nor relax the access that the member had in the base class. Access declarations may therefore only occur in the public or protected part of a derived class declaration.

If an access declaration is used to adjust the access control of a member which is an overloaded function, all functions of the given name inherited from the base class are affected.

For example:

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

class derived : private base {
public:
    base::a;
    int d;
};
```

In this example, class `derived` has two public members, `a` and `d`, and two private members `b` and `c`.

Friends

A function declared in the member-list of a class using the friend declaration specifier declares a *friend* function. A friend function is not a member of the class being defined, but has access to the private and protected members of the class.

The name of a function declared as a friend is not in the scope of the class, and is cannot be accessed by using a member-selection operator with an object of the class.

A friend declaration that refers to an overloaded name or operator function only causes that function whose type matches the friend declaration to become a friend. It is possible to declare all functions of a class as friends using an *elaborated-type-specifier*:

```
class one {
    friend class two;
    /* ... */
}
```

A class that is declared as a friend in this way can use private and protected names from the class that grants the friendship.

A friend declaration can be the first declaration of a class or function. In this case, the name of the friend is declared at the same scope as the class in which the friend is declared. A function first declared as a friend is equivalent to an extern declaration (see Chapter 2: ‘Linkage of Identifiers’).

A friend function may be defined in a class declaration - this is inline, and is treated as if it was rewritten immediately after the class declaration in the same way as an inline member function (see Chapter 5: ‘Storage-Class Specifiers’).

Access specifiers do not affect friend specifiers.

A derived class does not inherit friends declared by its base classes. A friend of a friend class is not a friend.

Access Rules for Virtual Functions

Access rules for virtual functions are checked statically, based on the type of the expression used to select the object for which the function is called, rather than dynamically based on the actual class of the object denoted. For example:

```
class base {
public:
    virtual myfunc();
};

class derived : public base {
private:
    myfunc();
};

void test(base *bp, derived *dp)
{ bp->myfunc();    // legal, even though it will call
                  // (private) derived::myfunc
  dp->myfunc();    // illegal - member is private
}

main()
{ derived d;
  test(&d, &d);
}
```

Multiple Access

If virtual base classes are used, it is possible that a member may be accessible via more than one inheritance route. In this case, the access allowed is the maximum access granted by any inheritance route. For example:

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

class B : private virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
    foo() { a = 1; }; // ok - B::a is private,
                    // but C::a isn't
};
```

CHAPTER 12

SPECIAL MEMBER FUNCTIONS

This chapter describes the semantics of special member functions - these are not (in general) called explicitly by the programmer, but allow the programmer to control how the use of objects of a given class are treated by the compiler.

Special member functions include

- *constructors*, which specify what action is required to turn raw memory into an object of a class whenever such an object comes into being.
- *destructors*, which specify any action required to clean up before an object of a class ceases to exist.
- *conversions*, which specify what actions are required to convert an object of a different class or type into an object of the owner class.
- Special operator functions, including the assignment operator function `operator=()` and the free-store management functions `operator new()` and `operator delete()`. These are special, in the sense that if the programmer does not define member-functions for these operators, the default operator behavior is applied. Other operator functions for which no default behavior is defined are described in Chapter 13.

Constructors

A constructor is defined by declaring a member function whose name is the same as its class. Constructors may not be declared as virtual, static, const or volatile. A constructor may, however, be called for an object which is const or volatile - this is an exception to the general rule that only a const member function can be called for an object that is declared const - see Chapter 9: 'Const and Volatile Member Functions'. A constructor may not have a return type, not even void, and no return value may be specified.

Whenever an object of a class that has a constructor is created, a constructor is implicitly called - this process is described further below.

A constructor that can be called without supplying an argument is referred to as a *default constructor*. A constructor in which all arguments have default values is a default constructor, as well as one with no arguments.

A constructor for a class X that can be called with a single argument of type X is referred to as a *copy constructor*. The type of the corresponding formal argument must be *reference to X*, optionally qualified by *const* and/or *volatile*. As with default constructors, if additional arguments are present, the constructor is a copy constructor only if they all have default values.

If no constructor is specified for a class, a default constructor is generated automatically by TopSpeed C++ if required. Note that if any constructor has been declared, this automatic generation does not occur. In such cases, if a default constructor is required, and none of the declared constructors is suitable, TopSpeed C++ will report an error.

If no copy constructor is specified for a class, a copy constructor is generated automatically if required.

Constructors obey the usual access rules, so that a constructor declared as *private* may only be used to create objects from within a member or friend function. Constructors are not inherited.

A generated copy or default constructor is public. A generated copy constructor for class X will take an argument of type *const X&* only if all base classes and members have copy constructors taking a *const* argument, otherwise it takes an argument of type *X&*.

The address of a constructor may not be taken.

A constructor may be called explicitly, to create a temporary object of a given class, whose lifetime is the expression in which the constructor is called. The syntax for this construct is

class-name (***expression-list***_{opt})

For example:

```

class myclass {
    int a;
    int b;
public:
    myclass();           // default constructor
    myclass(int=0,int=0); // also a default constructor
    myclass(myclass &);  // acts as copy constructor
};

class myclass1 {
    int a;
    int b;
public:
    myclass1(int, int = 0);
};

class myclass2 {
    int a;
    int b;
public:
};

void foo()
{
    myclass a(1);        // myclass::myclass(1,0)
    myclass b;           // ambiguous - 2 default ctors
    myclass c = a;       // calls copy constructor

    myclass1 a1(1);      // myclass1::myclass1(1,0)
    myclass1 b1;         // error - no default ctor
    myclass1 c1 = a1;    // ok - uses generated copy ctor

    myclass2 b2;         // uses generated default ctor
    myclass2 c2=b2;      // uses generated copy ctor
}

```

Destructors

A destructor is defined by declaring a member function whose name is the class name prefixed by `~`. Destructors may not be declared as static, const or volatile. A destructor may, however, be called for an object which is const or volatile - this is an exception to the general rule that only a const member function can be called for an object that is declared const - see Chapter 9: 'Const and Volatile Member Functions'. A destructor may not have arguments or a return type, not even void, and no return value may be specified.

Whenever an object of a class that has a destructor is destroyed, a destructor is called - this process is described further below.

If no destructor is specified for a class, a default destructor will be generated automatically by TopSpeed C++ if any base class or member has a destructor.

Destructors obey the usual access rules, so that if a destructor is declared as private, an error will be reported if an object of the class is destroyed outside

a member or friend function. A destructor that is generated by the compiler is public. Destructors are not inherited.

A destructor may be declared as virtual. If a destructor for class X is declared as virtual, then all destructors in classes derived from class X will also be virtual. In these cases, when an object is destroyed using the operator delete, the destructor, the operator delete() function and the size passed to the operator delete() function will all be determined by the actual type of the object (i.e. late binding) rather than the type of the expression to which the delete operator is applied. If a destructor is not virtual, the destructor, the operator delete() function and the size are determined by the type of the expression (early binding). This can lead to errors, as in the following example :-

```
struct base {
    base();
    ~base();
};
struct derived : base {
    derived();
    ~derived();
};

void foo()
{ base *bp = new derived;
  /* ... */
  delete bp    // uses ~base()
}
```

The address of a destructor may not be taken.

Calling Constructors and Destructors

Whenever an object of a class for which a constructor is defined is created, a call to the constructor for the object is implicitly made. An object may be created in the following ways

- A global object is created during program startup, before main() is called, and is destroyed at program termination. The order of creation of global objects is the order in which they appear in the source file. The order of destruction is the reverse of the order of creation.
- A local static object is created the first time that control reaches its declaration. Such objects are destroyed at program termination, in the reverse of the order in which they were created, and before global objects are destroyed.
- An auto object is created every time control reaches its definition statement, and destroyed at the end of its block.

- A temporary object may be created, and destroyed when no longer required - see below.
- An object may be created using the new operator, in which case it is not destroyed until the delete operator is applied to it.

When an array is created, a default constructor is called for each element, in ascending order. When destroyed, a destructor is called for each element in descending order.

When an object of a class is created, so are its members and sub-objects for any base classes. The order of creation of these is as defined below; the order of destruction is the reverse:

1. The sub-objects for any virtual base classes are created.
2. The sub-objects for non-virtual base classes are created (including their non-virtual base classes and members).
3. The data members are created, in the order of their declaration.
4. The object itself is created, and its constructor (if any) called.

For multiple base classes, the creation order is the order in which they are declared in the base-list. If a class has more than one direct base class, all base classes and members of the first direct base are created first, then the first direct base itself, before the base classes of the second direct base are considered (in other words, a depth-first search order is used).

A virtual base is created only the first time it is reached by the depth-first scan described above.

See below for a description of how arguments may be specified for the constructors of base classes and members.

In all the above cases, constructors and destructors will be called implicitly by the compiler where required, and the programmer should not in general attempt to make explicit calls to them.

An explicit call to a constructor may be used to create a temporary object of a given class (see below).

An explicit call of a destructor may occasionally be useful when a new operator has been used to place an object at a specific address, so that use of the delete operator to destroy the object is inappropriate. To make an explicit call of a destructor for class X, the fully qualified form `ptr->X::~~X()` must be used, where ptr is a pointer to the object of class X to be destroyed. An explicit destructor call of the above form may be made for any simple type

(including predefined types). If no destructor exists for the named type, the call has no effect.

Constructors and destructors may call other member functions. If a virtual member function is called from within a constructor or destructor, the virtual calling mechanism is disabled, so that the function that is called is the one defined in the class of the constructor or destructor, or a base class, rather than any overriding function declared in a more derived class.

Temporary Objects

The TopSpeed C++ compiler will generate temporary, unnamed objects of a class X in the following circumstances

- A constructor is called explicitly, using the syntax `X(expression-listopt)`. In many circumstances, TopSpeed C++ will optimize away the creation of the temporary, as described below.
- A function which returns type X is called. Such an object is initialized using a copy constructor when the function's return statement is executed.

Whenever a temporary object is created in one of the above ways, TopSpeed C++ will ensure that it is destroyed by calling the appropriate destructor (if any). Unless a reference is bound to the temporary object, TopSpeed C++ will destroy all temporary objects at the end of the statement in which they are created. If a reference is bound to the temporary, the temporary object is destroyed after the reference is destroyed - this will be before exit from the scope in which the temporary was created.

When a constructor for class X is explicitly called, TopSpeed C++ will not create a temporary object if

- The explicit constructor call is used as an initial value for an object of class X.
- the explicit constructor call is used as an actual argument. (Not currently true, but might become so)
- The explicit constructor call is used to specify the return value for a function returning class X.

In these circumstances, the object is constructed directly in the required destination object, rather than being constructed in a temporary before initializing the destination using a copy constructor.

For example, in the program fragment

```

class complex {
    double re,im;
public:
    complex (const complex &c);
    complex (double a=0, double b=0);
    ~complex ();
};

void g(complex) {}

complex f()
{
    complex c1 = complex(1,2); // no temp created
    g(complex(0,-1));          // temp created
    return complex(-1,-1);     // no temp created
}

void main()
{
    complex c2 = f(); // temporary created
}

```

the order of events is as follows :-

1. A temporary variable is created in main()'s local variable space, and its address passed to f(). It is not initialized yet.
2. f()'s local variable c1 is constructed using the constructor complex(1,2).
3. A temporary variable is created in f()'s local variable space, and initialized using the constructor complex(0,-1). This temporary is then copied to the stack to be passed as a parameter to g().
4. After g() has returned, the temporary created in 3 is destroyed using ~complex().
5. The return statement causes the constructor complex(-1,-1) to be used to initialize the temporary created in 1.
6. The local variable c1 is destroyed using the destructor ~complex(), and function f() returns.
7. main()'s local variable c2 is created, and initialized using a copy constructor from the temporary created in 1.
8. The temporary created in 1 is destroyed using ~complex().
9. When main() returns, the local variable c2 is destroyed using ~complex().

Conversions

Classes in C++ may be used to extend the base C++ language by adding additional types to it. Just as implicit conversions are applied to the built in types as described in Chapter 4, the programmer can define conversions that will be implicitly applied to objects of user-defined classes. Such conversions can be specified in two ways :-

- A constructor for class X which accepts a single argument specifies a conversion from the type of the argument to class X.
- A member function for class X whose name is of the form operator *type-name* specifies a conversion from class X to the named type.

At most one user-defined conversion is implicitly applied to any expression, so that an expression of type A would not be implicitly converted to type C, even if conversions from A to B and from B to C existed.

If an ambiguity arises when a conversion is required, so that the required conversion could be achieved in more than one way, TopSpeed C++ will report an error. Ambiguities are checked before access control rules are checked.

Conversions by constructor

If an object of class X is initialized or assigned from an expression of some other type T, a constructor of class X which accepts a single argument of type T will be implicitly called (provided that this conversion is unambiguous, and that the access rules for the constructor are not violated).

Note that function arguments and return values use the semantics of initialization, so that implicit conversions may be applied in these circumstances as described above.

Conversion Functions

A member function defined with a name of the form

```
conversion-function-name:
operator conversion-type-name
conversion-type-name:
type-specifier-list ptr-operatoropt
```

is a conversion function, and defines a conversion from its class to the type named in the *type-specifier-list*.

No argument types or return type may be specified. A conversion function may be virtual, and is inherited in the normal way. A conversion function declared in a derived class will hide a conversion function to the same type in a base class.

If an object of class X is used in a context where type T is expected, and a conversion function from X to T is defined, the value will be converted to type T by an implicit call of `X::operator T()`, provided that the conversion is unambiguous, and the access control rules for the conversion function are not violated.

If an object of class X is used in a context where several types could be used, and conversion functions exist from X to more than one of these types, the conversion is ambiguous, and TopSpeed C++ will report an error.

For example:

```
struct complex {
    double re, im;
    operator int()    { return int(re); };
    operator double() { return re; };
    complex(double r, double i=0)
        { re=r; im=i; };
};

main()
{ complex a(2);
  int i = a + 1;
    /* ambiguous: int(a)+1 or
     int(double(a) + 1) ? */
}
```

Initialization

.i.initialization:constructor;

When an object of a type that has no constructors, private or protected members, virtual functions or base classes, an initial value may be specified using an initializer list - see Chapter 6: 'Initialization'. If no initializer is specified, the initial value is undefined if the object is auto or is created using the new operator. Static objects are cleared to zero before the program starts if no initial value is specified.

When an object of a type that has a constructor is created, if it is not explicitly initialized as described below, the default constructor is used to initialize the object. If there is no default constructor, TopSpeed C++ will report an error. Note that default constructors are generated automatically only if no constructors are declared.

Explicit Initialization

An object of a class X which has a constructor may be explicitly initialized in one of two ways :-

1. A parenthesized parameter list may be given, to be used as the arguments to a constructor. For example:

```
complex c(20,30);
```

2. A single value can be specified as the initial value using the = operator. This value is used as the argument to a constructor. For example:

```
complex c = 0;
```

The compiler selects which constructor to call in the same way as with overloaded member functions. If no suitable constructor is found, or an ambiguity arises between more than one constructor, TopSpeed C++ will report an error.

Note that the = operator used to specify an initial value is distinct from the = operator used to specify assignment. For the assignment operation, the operator=() function is used - see below.

Function arguments and return values are initialized in the same way as if their value was specified using the = initialization form.

Objects created using the new operator with a parenthesized parameter list, and base classes and member objects whose initializers are specified in a *ctor-initializer*, are initialized in the same way as if a parenthesized parameter list was given.

An array of objects of class X can be initialized with an *initializer-list* (see Chapter 6: 'Initialization') that specifies a constructor call for each element of the array. If the number of initializers in the list is fewer than the number of elements in the array, the default constructor is used for the remaining elements.

For example:

```
complex a[3] = {  
    0,           // calls complex::complex(0, 0)  
    complex(1,2) // a[2] uses default ctor  
};
```

Initializing Bases and Members

When a class is created, the constructors for its base classes and non-static data members are automatically called, if they exist. If any base class or member object with a constructor does not have a default constructor, then

the arguments to be used for the call of its constructor must be specified in the definition of every constructor of the owner class, using a *ctor-initializer*:

```

ctor-initializer:
    : mem-initializer-list
mem-initializer-list:
    mem-initializer
    mem-initializer , mem-initializer-list
mem-initializer:
    complete-class-name ( expression-listopt )
    identifier ( expression-listopt )

```

If the class-name or identifier in a *mem-initializer* names a base class or a member with a constructor, the expression list specifies the arguments for the constructor, and will determine which constructor is used according to the usual rules. Otherwise, the *expression-list* must consist of a single expression, which specifies the value to which the named member is to be initialized.

A *mem-initializer* is the only way to specify initial values for non-static const and reference members.

The *expression-list* in a *mem-initializer* is evaluated in the scope of the constructor of which it forms a part.

The order in which base classes and members are given in the *mem-initializer-list* does not affect the order in which initialization occurs.

A non-virtual base class specified in a *mem-initializer* must be a direct base class. For virtual base classes, a mem initializer can be specified for direct or indirect base classes. The *mem-initializer* in the most derived class (the complete object) is used to initialize a virtual base class - a *mem-initializer* for a virtual base classes in any base classes of the most derived class is ignored. If the most derived class does not specify a *mem-initializer* for a virtual base class, then there must be a default constructor for that virtual base class.

For example:

```

struct V {
    V(int a) { cout << a; };
};

struct A : virtual V {
    A() : V(1) {};
};

struct B : virtual V {
    B() : V(2) {};
};

struct C : A, B {
    C() : V(3) {};
};

C c; // V::V(3) called
B b; // V::V(2) called
A a; // V::V(1) called

```

Copying Class Objects

An object of class *X* may be copied to another object of class *X*, or a base class of class *X*, either by initialization (using a copy constructor) or by assignment. Assignment to a class object uses the operator=() function.

The programmer may define an operator=() function for a class, by defining a member function with the name operator=. The operator=() function may be overloaded, and is subject to the usual access rules, but is not inherited.

Unless the programmer defines an operator=() function for class *X* that takes an argument of class *X*, such a function is declared implicitly, and will be defined (i.e., code will be generated for it) if required. The generated operator=() function performs member-wise assignment from its argument to the object for which the function is called, and returns a reference to the object for which it is called.

The generated operator=() function is public. If all members and base classes of class *X* have operator=() functions taking a const argument, the generated operator=() function will be X& X::operator=(const X&), otherwise it will be X& X::operator=(X&). The generated operator=() function will ensure that sub-objects of virtual base classes are assigned only once.

The new and delete Operators

A member function declared with the name operator new is called implicitly by the compiler when the new operator is used to create an object of the class.

An operator new function is always static, whether declared with the static keyword or not. The first argument must be of type size_t. The return type must be void*.

If additional arguments are specified, the operator `new()` function may be overloaded (see Chapter 13).

An operator `new()` function is inherited, and follows the usual rules for scope, ambiguity resolution and access control. It may not be virtual.

A member function declared with the name `operator delete` is called implicitly by the compiler when the `delete` operator is used to destroy an object of the class created using `new`.

An operator `delete()` function is always static, whether declared with the `static` keyword or not. The first argument must be of type `void *`; the second, if present must be of type `size_t`. No additional arguments may be specified. The return type must be `void`, and no return value may be specified.

An operator `delete()` function may not be overloaded, so there may only be one operator `delete()` member function in a class.

An operator `delete()` function is inherited, and follows the usual rules for scope, ambiguity resolution and access control. It may not be virtual.

The syntax and usage of the `new` and `delete` operators, which are implicitly translated into calls of operator `new()` and operator `delete()` functions, are discussed further in Chapter 7.

If no operator `new()` function has been declared or inherited for a particular class, the global operator `new()` function is used to allocate storage for objects of that class. If no operator `delete()` function has been declared or inherited for a particular class, the global operator `delete()` function is used to deallocate storage for objects of that class.

For example:

```
#define P00LSIZE 500
typedef unsigned size_t;

class myclass {
    static myclass pool[P00LSIZE];
    static int nextpool;
public:
    void* operator new(size_t)
    { return &pool[nextpool++]; };
    void operator delete(void *)
    { --nextpool; }
};

void f()
{ myclass *a = new myclass;
  // ...
  delete a;
}
```

CHAPTER 13

OVERLOADING

Introduction

In TopSpeed C++, it is in general possible to define several functions in the same scope with the same name. The compiler will then determine which particular version of the function is to be used by comparing the parameters of the function with the supplied arguments. This process is known as *overloading*.

In order to be able to resolve which to use from a set of overloaded functions, it is necessary that the functions are sufficiently distinct. This gives rise to the following rules for overloading function names :-

- Functions that differ only in the return type may not be overloaded.
- Functions that differ only in that one is a static member function and the other is a non-static member function may not be overloaded.
- Functions that differ only in the use of typedefs to refer to the same fully specified argument type may not be overloaded.
- Functions that differ only in having argument types with the const or volatile attributes may not be overloaded. This rule only applies to the type of the arguments themselves - it is possible, for example, to define overloaded functions where one takes an argument of type `const T*`, and the other an argument of type `T*`.
- Functions that differ only in respect of an argument of array versus pointer type may not be overloaded.
- Functions that differ only in respect of the first dimension of an argument of array type may not be overloaded. The sizes of the second and subsequent dimensions only of array type arguments are significant for resolving overloading.

Declaration Matching

If two functions are declared in the same scope with the same name and identical argument types (according to the rules above), then they refer to the same function. TopSpeed C++ will report an error if there is more than one definition of a function, if a member function is declared more than once in a class declaration, or if a redeclaration of a function specifies a different result type.

If a function is defined in a nested scope, functions of the same name in enclosing scopes are hidden, regardless of their argument types. Thus a member function of a derived class hides rather than overloads a function of the same name in a base class. If a member function in the base class is declared as virtual, a function of the same name in a derived class with the same argument types *overrides* the virtual function - see Chapter 10.

Note that all member functions declared within the member-list of a class declaration are in the same scope, regardless of whether they are declared as static or virtual, or whether they are public, private or protected. Scope resolution always takes place before overloading resolution, which in turn takes place before access rules are checked.

For example:

```
typedef int INT;

int func1(int);
INT func1(INT);    // ok - redeclaration
int func1(short);  // overloaded function
short func1(int);  // error - redeclaration with
                  //          different result type

class myclass {
    virtual short func1(int); // hides global func1()
};

class derived : myclass {
    short func1(int); // overrides myclass::func1()
};
```

Argument Matching

When an overloaded function is called (directly), all matching functions of the specified name that are in scope are potential candidates to be called. A matching function is a function for which conversions exist to make each supplied argument type match the declared argument type.

TopSpeed C++ will select the matching function whose argument list best matches the supplied arguments to the function call. In order to be the best matching function, a function must have a better or equal match for each argument than all other matching functions, and must have a strictly better

match for at least one argument. The criteria for comparing argument matches are described below.

If no matching function is found, or if no matching function meets the criteria above to be the best match, TopSpeed C++ will report an error.

For the purposes of determining argument matches, a function that has *n* default arguments is considered to be equivalent to *n*+1 separate functions, one with all default arguments supplied, and the others taking fewer parameters. For example, for the purposes of argument matching, the declaration

```
int t(int a, float b=0.0, int c = 0);
```

is equivalent to

```
int t(int a, float b, int c);
inline int t(int a, float b)
{ return t(a, b, 0); }
inline int t(int a)
{ return t(a, 0.0, 0); }
```

For the purposes of argument matching, non-static member functions of a class *X* are considered to take an extra argument which specifies the object for which the member function is called. This argument must be matched by the object or pointer specified if the explicit member selection operator is used, or by the first operand for an overloaded operator. No user-defined conversions, nor conversions requiring a temporary, will be applied to achieve a match for this argument.

The type of this extra argument is determined by how the function is called: if called for a pointer using the `->` or `->*` operator, the type is *X**, qualified by `const` and/or `volatile` if the member function is itself so qualified; if called for an object using the `.` or `.*` operator, or for an object that is the first operand of an overloaded operator, the type is *X&*, again qualified by `const` and/or `volatile` if the member function itself is.

An ellipsis in a function declaration matches an actual argument of any type, and all subsequent actual arguments.

Note that the function which has the best match is selected before the access rules are checked, so that an error will be reported if the best matching function is `private`, even though another matching function may be `public`. The result type of the function is not considered in overloading resolution, except when the address of an overloaded function is taken.

Conversion Ordering

Whether an argument match is considered better or worse than another depends upon the sequence of conversions that are required to convert from the actual argument type to the formal argument type. Any sequence of

conversions that includes more than one user-defined conversion is not considered. Any sequence of conversions that can be transformed by deleting conversions into a shorter sequence of conversions which matches the given argument in any matching function is not considered.

Only the worst conversion of any sequence is significant in argument resolution. If two sequences of conversions both contain a user-defined conversion, then they will both be considered to provide an equally good match, even if one also requires a standard conversion while the other provides an otherwise exact match.

A trivial conversion, from the table below, does not affect whether one sequence of conversions is better than another (except that `const` and `volatile` may operate as tie-breakers, as described below).

Trivial Conversions

| From | To |
|---------|-------------|
| T | T& |
| T& | T |
| T[] | T* |
| T(args) | (*T) (args) |
| T | const T |
| T | volatile T |
| T* | const T* |
| T* | volatile T* |

The conversions that may be applied, from the best to the worst, are described below.

1. A sequence containing only trivial conversions is better than all others. A sequence that does not convert a pointer or reference to its `const` or `volatile` form is better than one that does.
2. A sequence containing only integral promotions, promotions from `float` to `double`, and trivial conversions is better than all others except those described above.
3. A sequence containing only standard conversions or trivial conversions is better than all others except those described above. If A is a public base class of B, and B is a public base class of C, pointer conversions from B* to A* are better than conversions to `void*`, and pointer conversions from C* to B* are better than pointer conversions from C* to A*. Analogous rules apply for conversions of references to class types and pointers to members.

4. A sequence involving a user-defined conversion is worse than all others except a match with an ellipsis.
5. A match with an ellipsis is worse than any other sequence. A function with an ellipsis will only be called if it is the only matching function.

Note that a temporary may be required in order to pass an argument to a function with a reference argument. This does not affect argument matching. In particular, given

```
void o1(int &);
void o1(const int&);
```

a call of `o1(3)` will select `o1(int &)`, since both require only trivial conversions, so the tie-break rule is applied to prefer the sequence without the conversion from `int&` to `const int&`. However, as a temporary is required to initialize the reference, and it is illegal to initialize a non-const reference with a temporary, TopSpeed C++ will report an error.

These rules are best illustrated by means of examples. Given the following declarations of overloaded functions,

```
void f1(char);
void f1(int);
void f1(long);

void f2(long);
void f2(double);

class A { public: operator int(); };
class B : public A {};
class C : public B {};

void f3(void *);
void f3(A*);
void f3(B*);

void f4(int, double);
void f4(double, int);
int f5(char);
void f5(long);
```

the following argument resolution would occur :-

```
f1('c'); // calls f1(char) - exact match
f1(1);   // calls f1(int) - exact match
f1(1L);  // calls f1(int) - exact match
unsigned char s = 1;
f1(s);   // f1(int) matches with promotions only
          // f1(long) requires standard conversion
          // f1(char) requires standard conversion
          // f1(int) called.
```

```

f2(0); // ambiguous - standard conversions int to
      // long and int to double compare equal.
C *cp = 0;
f3(cp); // Standard conversion from C* to B* is
      // better than standard conversion from
      // C* to A* or to void*.
      // f3(B*) called.
f3(&s); // standard conversion of unsigned char*
      // to void* is only match.
      // f3(void *) called.
f1(*cp); // Ambiguous: all versions of f1 match with
      // a user-defined conversion of A to int.
      // That f1(int) requires one fewer standard
      // conversion does not affect argument
      // matching.
f4(1,1); // ambiguous; f4(int, double) matches
      // better on first argument,
      // f4(double, int) on second.
int i=f5(0); // ambiguous: f5(char) and f5(long)
      // match equally well. That the result
      // type of f5(long) is unsuitable does
      // not affect argument matching.

```

Overloaded Function Address Resolution

When the address of an overloaded function is taken, explicitly or implicitly, no argument list is supplied, and therefore the above rules cannot be used to determine which function is referred to. In this case, the function is selected by determining which function of the given name that is in scope exactly matches the type of the target - this will be an object being assigned to or initialized (including the initialization that occurs in passing arguments to functions and user-defined operators, and function return values). If no function provides an exact type match, TopSpeed C++ will report an error.

For example:

```

int f1(...);
int f1(int);
int f1(short);
int f1(int, char *);

int (* fp1)(int) = f1; // uses f1(int);
void (* fp2)(int) = f1; // error: no match
int (* fp3)(float) = f1; // error: no match

```

Note that if an overloaded function name is used as a parameter to a function that is itself overloaded, all combinations of the function being called and the function being passed are considered to find the best match. In any matching function that is considered to be called, the match for the function argument must be exact. For example:

```

int g (int (*)(int));
int g (int (*)(...));

void foo()
{ g(f1); // error: ambiguous
  // g(int (*)(int)) with f1 (int) or
  // g(int (*)(...)) with f1 (...)
}

```

Overloaded Operators

Most standard operators can be overloaded by declaring a function with the name `operator??`, where `??` is one of the following

```

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

```

The operators `., .*, ::` and `?:` cannot be overloaded, nor can the preprocessing symbols `#` and `##`. Both unary and binary forms of the `+` `-` `*` and `&` operators can be overloaded.

The use of operator `new()` and operator `delete()` is described in Chapter 12: ‘The new and delete Operators’. The remainder of this section does not apply to them.

An operator function may not have default arguments. Apart from `operator=()`, operator functions are inherited. Operator functions obey the same scope rules, overloading resolution rules, and access rules as normal functions.

The Assignment Operator

The assignment operator may be overloaded by declaring a non-static member function called `operator=`. It is not inherited, but is generated automatically if required for any class for which no suitable `operator=()` function is defined. This is described further in Chapter 12: ‘Copying Class Objects’.

The Function Call Operator

The function call operator may be overloaded by declaring a non-static member function called `operator()`. For a class object `x`, `x(expression-list)` is equivalent to `x.operator()(expression-list)`.

For example, a string class might want to overload the `()` operator to indicate subscripting, using declarations like

```
class string {
    // ... suitable data & functions
public:
    string& operator()(int start, int end);
};

int foo(string &s)
{ s(1, 3) = "ff"; // calls s.operator()(1, 3)
}
```

The Subscript Operator

The subscript operator may be overloaded by declaring a non-static member function called `operator[]`, taking a single parameter. For a class object `x`, `x[y]` is equivalent to `x.operator[](y)`.

For example, a class might be declared as follows to provide run-time array bounds checking. (Note that TopSpeed C++ can automatically perform run-time array bounds checking for all arrays, using the pragma `check(index=>on)`.)

```
class careful {
    int data[10];
public:
    int& operator[](int a)
    { if (a >= 0 && a < 10)
        return data[a];
        error("index out of bounds");
    };
};

void myfunc(careful &s, int j)
{ s[j] = s[j+1]; // indices automatically checked
}
```

The Class Member Access Operators

The class member access operator `->` may be overloaded by declaring a non-static member function called `operator->`. For a class object `x`, `x->y` is equivalent to `(x.operator->())->y`. The `operator->` function must take no arguments, and return either a pointer to a class, or an object or reference of a class for which an `operator->` function can be called.

The class member access operator `->*` behaves as a standard binary operator. The class member access operators `.` and `.*` may not be overloaded.

Overloading the `->` operator is most useful for using “smart” pointers. For example, the class `paged_pointer` in the example below can be used as a pointer to objects of type `info`, but on every dereference a check is made that the object pointed at has not been paged out to disk.

```

struct info {
    char name[20];
    // ...
};

class paged_pointer {
    int diskpage;
    int offset;
    static info buffer[INFOBUFSIZE];
public:
    info * operator->()
    { if (diskpage!=0)
      pagein();
      return &buffer[offset];
    };
};

void myfunc(paged_pointer pp)
{ // ...
  cout << pp->name;
}

```

The Increment and Decrement Operators

The prefix increment and decrement operators may be overloaded by defining functions called `operator++` or `operator--` respectively, taking a single argument of a class or reference to class type, in the same way as any other unary operator. For a class object `x`, `++x` is equivalent to `x.operator++()` or `::operator++(x)` (selected by the usual argument selection process).

The postfix increment and decrement operators may be overloaded by defining functions called `operator++` or `operator--` respectively, taking two arguments. The first must be a class type or a reference to class type. The second must be of type `int`. For a class object `x`, `x++` is equivalent to `x.operator++(0)` or `::operator++(x,0)` (selected by the usual argument selection process). If calling the `operator++` or `operator--` function explicitly rather than by using the `++` or `--` operator, the programmer may pass a value other than zero as the `int` parameter.

For example, the paged pointer type declared above might need the postfix and prefix increment operators defined as follows :-

```

class paged_pointer {
    // ...
public:
    paged_pointer &operator++() // prefix operator
    { ++offset;
      return this;
    }
    paged_pointer &operator++(int) // postfix operator
    { paged_pointer temp = *this;
      offset++;
      return temp;
    }
};

```

Unary Operators

Except for those operators described above, an overloaded operator function for a unary operator must either be a non-static member function taking no arguments, or be a global or static member function taking a single argument which is a class or a reference to a class.

For example, continuing the example of the paged pointer class, the unary indirection operator could be declared as follows :-

```
class paged_pointer {
    // ...
public:
    info &operator*()    // unary * operator
    { if (diskpage!=0)
        pagein();
        return buffer[offset];
    }
};
```

Binary Operators

Except for those operators described above, an overloaded operator function for a binary operator must either be a non-static member function taking one argument, or a non-member or static member function taking two arguments, the first being a class or a reference to a class.

For example, the class `complex` provided in the libraries supplied with TopSpeed C++ has a number of overloaded binary operator functions associated with it. The binary operator functions are generally declared as friends rather than as member functions, as this allows user-defined conversions from simpler types to the complex type to be performed for either operand. Had the binary arithmetic operators for class `complex` been declared as member functions, the first operand would always have to be a complex number.

When overloading operators to deal with user-defined types, it is normally a good idea to try to preserve the expected characteristics of operators that apply when the operators are used for simpler types. For example, by declaring the `operator+()` for complex numbers as a friend function, the commutative property of the `+` operator is maintained. The class `complex` has also been designed so that the identities that apply for arithmetic types (for example, `i+=1` means the same as `i=i+1`, except that the operand `i` is only evaluated once) also hold true for the complex type.

CHAPTER 14

THE PREPROCESSOR

Introduction

The C++ preprocessor processes tokens from the source text file before passing these tokens on for syntax checking. This makes it possible to do certain things before compiling the program. In particular, the preprocessor makes it possible to:

- Include other files. These may contain definitions and declarations that can be used in the program.
- Selectively compile certain parts of the code, depending on specific conditions (such as the hardware on which the program is to run).
- Replace macros with other tokens.

Except for the additional C++ tokens `::`, `->*` and `.*`, and the C++ style comments introduced by `//`, the TopSpeed C++ preprocessor is identical in behavior to that specified by the ANSI C Standard.

Phases of Translation

Tokens are built, processed and then analyzed for syntax in the following conceptual translation phases, as specified by ANSI C:

1. Physical source file characters are mapped to the source character set. During this phase,
 - A newline character is substituted for each end-of-line character sequence.
 - Trigraphs are replaced by their corresponding single-character representations.
2. Physical source lines are spliced to form logical source lines, by deleting each backslash-newline pair. Thus, lines that extend over two lines are combined into a single line. (A non-empty source file must end in a

- newline character, but cannot end with a newline immediately preceded by a backslash character - that is, the file cannot end in the middle of a continued line.)
3. The source file is decomposed into preprocessing tokens and white space character sequences. Newline characters are retained. Each comment is replaced by a single white space character, and multiple white space characters are replaced by a single white space character. White space characters in strings and character constants are not affected.
 4. Preprocessing directives are executed and macro invocations are expanded. Any header or source files specified with an `#include` preprocessing directive are processed recursively, from phase 1 through phase 4.
 5. Escape sequences in character constants and string literals are converted to single characters in the execution character set.
 6. Adjacent string literals are concatenated.
 7. Preprocessing tokens are converted into (normal) tokens. White space characters separating individual tokens are no longer needed, and are discarded. The tokens remaining are analyzed and translated, according to syntactic and semantic rules of the language. If a token cannot be translated, a syntax error results.
 8. All external object and function references are resolved. I.e., the object code is linked to produce an executable program.

For example, the following sequence of characters:

```
01< <h3/1.2>=x+++b
#include <2/1.3x>
#define struct.field $
```

forms the following sequence of preprocessing tokens:

```
{01} {<} {<} {h3} {/} {1.2} {>=} {x} {++} {+} {b}
{#} {include} {<2/1.3x>}
{#} {define} {struct} {.} {field} {$}
```

Trigraph Sequences

Trigraph sequences allow the input of characters that may not appear on some keyboards. Use of trigraph sequences makes it possible to input characters not defined in the “ISO 646-1983” Invariant Code Set, which is a subset of the seven-bit ASCII code set.

The only valid trigraph sequences are represented by the three-character sequences in the following table. Such sequences are replaced with the

corresponding single character. A ? is altered only if it begins one of the trigraphs listed.

| Trigraph | Replaced by |
|----------|-------------|
| ??= | # |
| ??(| [|
| ??) |] |
| ??/ | \ |
| ??< | { |
| ??> | } |
| ??! | |
| ??' | ^ |
| ??- | ~ |

E.g, after replacement of the trigraph sequence ??' and ??/, the following source line

```
printf("??'??/?/?/");
```

becomes

```
printf("^?\\");
```

Preprocessing Directives

A *preprocessing directive* is a command to the preprocessor. A directive begins with a #. This must be the first character on the source line except for white space characters. The directive is terminated by a newline character.

The # may be followed by one of the following preprocessor directives:

```
define
undef
if
ifdef
ifndef
elif
else
endif
include
line
error
pragma
```

Space and horizontal tab characters may appear between the # and the directive. Preprocessor directives do not end with a semicolon.

If the # is the only token on a line, it has no effect, and is discarded.

After preprocessing, all preprocessing tokens must have the form of a lexical token.

Defining a Macro

The `#define` directive is used to associate a meaningful name with a token sequence, usually a number, identifier, or expression. It may also be used to aid portability and enhance program readability without sacrificing performance.

There are two types of macros, those without parameters (object-like macros) and those with parameters (function-like macros).

Object-Like Macros

```
# define identifier replace-listnewline
```

An *object-like macro* directive defines a macro whose name corresponds to the identifier after the `#define`. This identifier is called the *macro name*. Each subsequent instance of the macro name is replaced by the *replacement list* of tokens that forms the remainder of the directive. After this substitution, the replacement list is rescanned for more macro names to replace.

White space characters preceding or following the replacement list of preprocessing tokens are not considered part of the replacement list for a macro.

For example:

```
#define BUFLen 512
#define TRUE 1
#define forever while (TRUE)
```

In the first example, all occurrences of `BUFLen` (after the one in the definition) are replaced by `512`. In the third example, all occurrences of `forever` are replaced by `while (TRUE)`.

Function-Like Macros

```
# define ident lparen ident-listopt ) replace-list newline
```

A directive with a macro name followed by a left parenthesis, an optional identifier list, and a right parenthesis defines a *function-like macro* with arguments.

Such a macro invocation is syntactically similar to a function call. The parameters for such a macro are specified in the optional list of identifiers. The scope of these parameters extends until the newline character that terminates the `#define` preprocessing directive.

The left parenthesis in a function-like macro definition must come immediately after the macro name. That is, there can be no intervening white space or tokens between the macro name and the left parenthesis.

Each subsequent instance of the function-like macro name - followed by a (preprocessing token and terminated by a) preprocessing token - is replaced by the replacement list given in the macro definition.

The actual arguments specified in the macro occurrence are substituted for the appropriate identifiers in the replacement list.

For example, given the following two macros:

```
#define square(x)      ((x) * (x))
#define setbit(x, y)  ((x) |= 1 << (y))
```

If a fragment such as

```
square(7.5)
setbit(123, 3)
```

is encountered in the source, the fragment is replaced by the following:

```
((7.5) * (7.5))
((123) |= 1 << (3))
```

Redefining a Macro Name

It is an error to redefine as a macro a name already defined as a macro name unless the replacement lists are identical. In the case of a function-like macro the parameters must also be identical in spelling and number.

Two replacement lists are identical if they differ only in the white space in the source. All tokens in one list must be in the second list - with the same spelling and in the same sequence. The amount of white space in the lists is not taken into account when comparing the lists.

For example, the following redefinitions are valid:

```
#define BUFLen 512
#define max(a, b) ((a)>(b) ? (a) : (b))
#define BUFLen      512
#define max(a,b) ((a)>(b) ?(a) :(b))
```

but the following ones are invalid:

```
#define BUFLen 256
#define max(x,y)((x)>(y) ? (x) : (y))
```

Scope of Macro Definitions

A macro definition remains in scope until an #undef directive for the macro name is encountered. If no such #undef is encountered then the scope extends until the end of the translation unit.

A preprocessing directive of the form:

```
# undef identifier newline
```

removes the specified identifier from the list of identifiers recognized as a macro name. If the specified identifier is not currently defined as a macro name, the `#undef` directive is ignored.

Macro Replacement

During translation phase 4, each identifier is checked to see if it is identical to a defined name. If the token matches a macro name, it is replaced by the body of that macro name as follows:

Object-like macros The token is replaced by the tokens making up the replacement list for the macro name.

Function-like macros In this case, the macro name must be followed by a `(` token. A function-like macro name token that is not followed by a left parenthesis is not replaced.

The tokens between the matching opening and closing parenthesis constitute the *actual parameters* of the macro invocation. Individual arguments between the outermost parenthesis are separated by comma tokens. (Comma preprocessing tokens bounded by nested parenthesis do not separate arguments.) Within the sequence of preprocessing tokens making up an invocation of a function-like macro, a newline is considered a white space character.

TopSpeed C++ gives an error if the number of actual arguments in the macro invocation does not match the number of formal parameters in the macro definition.

Given the following legal macro definitions,

```
#define TRUE      1
#define BUFLen    512
#define forever while (TRUE)
#define square(x)  ((x) * (x))
#define setbit(x, y) ((x) |= 1 << (y))
```

the following fragments:

```
char buffer[BUFLen];
forever {
    receive(buffer);
    send(buffer);
}
num = square(num);
setbit(flag, 4);
```

expand as follows:

```
char buffer[512];
```

```

while (1) {
    receive(buffer);
    send(buffer);
}
num = ((num) * (num));
((flag) |= 1 << (4));

```

Argument Substitution

Argument substitution is used to replace the arguments identified when a function-like macro is invoked. With certain exceptions (listed below), a parameter in the replacement list is replaced by the corresponding argument *after* all contained macros have been expanded. All macros in the argument's preprocessing tokens are completely replaced before such tokens are substituted for the parameters.

A parameter in the replacement list is replaced with the corresponding argument. There is no expansion if that parameter is preceded by a # or ## preprocessing token or is followed by a ## preprocessing token (see below).

Rescanning and Further Replacement

After the required parameter substitutions have been made in the replacement list, the resulting preprocessing token sequence is scanned again, for more macro names to replace.

Not all macro names are replaced in subsequent scannings. No substitution is made if the name of the macro being replaced is found during this rescan of the replacement list. This name is also left if encountered by any nested replacements. Once left, such unreplaced macro name tokens are not replaced, even if they are later processed in contexts in which they would otherwise be replaced.

For example:

```

#define A   A B C
#define B   B C A
#define C   C A B

A
// expands to...
A B C
// rescan: A is not expanded...
A { B C A } { C A B }
// rescan B: A and B not expanded...
A { B { C A B } A } { C A B }
// no further expansion possible in B
// rescan of C: A and C not expanded...
A B C A B A { C A B C A }
// no further expansion possible in C
A B C A B A C A B C A
// ...which is the result

```

To illustrate the rules for substitution and reexamination, consider the following sequence:

```

#define pair(x,y)    (x->next = y)
#define last(q)      (q->next ? last(q->next) : q)

pair(a,b);
pair(a, pair(b, c));
a = last(c);
a = last(last(b));

pair(a, last(c));
b = last(pair(a, c));

```

Macro replacement produces the following:

```

(a->next = b);
(a->next = (b->next = c));
a = (c->next ? last(c->next) : c);
a = ((b->next ? last(b->next) : b)->next ?
    last((b->next ? last(b->next) : b)->next) :
    (b->next ? last(b->next) : b));
(a->next = (c->next ? last(c->next) : c));
b = ((a->next = c)->next ? last((a->next = c)->next)
    : (a->next = c));

```

The sequence of tokens resulting from a macro replacement is not available for preprocessing as a preprocessor directive.

The name following the # preprocessing token at the start of a preprocessing directive is not subject to macro replacement, even if the name has been defined as a macro name.

For example:

```

#define include error
#include "io.h" // still includes io.h

```

The second line still includes the specified file, because the macro replacement specified in the first line is not applied. The preprocessing token sequence that results from macro replacement is not processed as a preprocessing directive even if it resembles one.

The # Operator

It is possible to replace a preprocessing token parameter with a string literal token. If the *stringize operator* (#) is found in the replacement list, it must be followed by a parameter token. The parameter is replaced by a single string literal, which contains the spelling of the preprocessing token sequence for the argument.

Note: The tokens for the argument are not rescanned for macro substitutions, as is normally the case.

Every sequence of white space between the argument's preprocessing tokens is reduced to a single white space character in the string literal. Any white space is removed before the argument's first or after its last preprocessing token. Beyond that, the original spelling of each preprocessing token in the

argument is retained in the string literal, except that a \ character is inserted before each “ and \ character within the created literal.

The ## Operator

Before the replacement list for a macro invocation is re-examined for more macro names to replace, each instance of a *glue operator* preprocessing token (##) in the replacement list is deleted and the token before the glue is concatenated with the token after the glue. (Parameters are handled differently, as described in the next paragraph.) The result of this concatenation must be a valid token. The resulting token is available for further macro replacement. Glue is handled in this way in both object-like and function-like macros

If, in the replacement list, a parameter is immediately preceded or followed by a ## preprocessing token, the parameter is replaced by the corresponding argument's preprocessing token sequence (which is not rescanned).

The order of evaluation is left unspecified in the Base Document; in TopSpeed C++ the order is left to right.

To illustrate the rules for creating string literals and concatenating tokens, consider the following sequence:

```
#define str(x) #x
#define xstr(x) str(x)
#define VERSION 2
#define header(v) xstr(incv ## v)
#define incfile(v) header(v.h)
str(hello) str(!) str(str(?)) xstr(str(?))
#include incfile(VERSION)
```

This results in:

```
“hello” “!” “str(?” “\”?\””
#include “incv2.h”
```

(See below for a full description of the lexical rules governing the characters in a filename in a #include directive.)

Space around the # and ## tokens in the macro definition is optional.

A ## preprocessing token may not occur at the beginning or at the end of a replacement list for either form of macro definition.

Conditional Inclusion

The preprocessor can be used to selectively include/exclude lines of input source from further processing by the compiler.

```
# if const-expr
  group-of-lines1
# else
  group-of-lines2
# endif
```

If the `const-expr` is zero, the text `group-of-lines1` is skipped. Instead, the text `group-of-lines2` is processed and passed on to the translator. If the `const-expr` has a nonzero value, the text `group-of-lines1` is processed while `group-of-lines2` is skipped. The `#else` part is optional.

It is possible to nest `#if` directives. The preprocessor matches `#elses`, `#elifs` and `#endifs`.

For nested `#if`, `#else` and `#endif` directives, the `#elif` directive may be used:

```
# if const-expr newline
  groupopt
# else
# if const-expr newline
  groupopt
# endif
```

can be replaced by:

```
# if const-expr newline
  groupopt
# elif const-expr newline
  groupopt
# endif
```

Preprocessing directives of the forms:

```
# ifdef identifier newline
  groupopt
# ifndef identifier newline
  groupopt
```

check whether or not the identifier is currently defined as a macro name. Their conditions are equivalent to `#if defined identifier` and `#if !defined identifier`, respectively.

Each directive's condition is checked in order. If the condition evaluates to zero (false), the group that the condition controls is skipped. When skipping, directives are processed only as far as the name that determines the directive, in order to keep track of the level of nested conditionals. The rest of the directives' preprocessing tokens are ignored. Only the first group whose control condition evaluates to nonzero (true) is processed.

If none of the conditions evaluates to true, and there is an `#else` directive, the group controlled by the `#else` is processed. If there is no `#else` directive, all the groups until the `#endif` are skipped.

The expression must be an integral constant expression, and may not contain a `sizeof` operator or a cast. However, the expression may contain unary expressions of the form:

defined identifier

or

defined (identifier)

These evaluate to 1 if the identifier is currently defined as a macro name and to 0 if it is not. An identifier is defined as a macro name if the identifier has been the subject of a #define preprocessing directive without an intervening #undef directive.

For example:

```
#ifdef CHEMISTRY
    #define AVOGADRO 6.024E23
#else
    #define PI 3.1415926
#endif
```

Evaluating Constant Expressions

Prior to evaluation, identifiers currently defined as macro names are replaced in the list of preprocessing tokens just as in normal text. (The exceptions are identifiers modified by defined). Any remaining identifiers are replaced with 0L and each integer constant not already suffixed with l or L is considered to be additionally suffixed with L.

During the evaluation of the expression the usual arithmetic conversions apply, except that int and unsigned int act as if they had the same representation as long and unsigned long, respectively. This includes interpreting character constants, which may involve converting escape sequences into characters.

E.g, the following will compare equal in TopSpeed C++ (or any preprocessor that uses the ASCII character set):

```
#define LOWER_A 'a'
#define UPPER_A 'A'
#if (UPPER_A - LOWER_A) == 32
    blah_blah
#endif
if ((UPPER_A - LOWER_A) == 32)
    blah_blah;
```

Source File Inclusion

A preprocessing directive of the form:

```
# include < h-char-sequence > newline
```

is replaced with the entire contents of the source file identified by the specified character sequence between the < and > delimiters and by a set of pathnames. (In TopSpeed C++, this set is determined by the contents of the TS.RED file.) For example:

```
#include <stream.hpp>
```

is replaced with the entire contents of the file `stream.hpp`. This file is sought in whatever directories are specified for `.HPP` files in the `TS.RED` file. The search continues until the file has been found or until all the directories have been checked.

TopSpeed C++ allows `#include` files to be nested to a depth limited by the maximum number of simultaneously open files that is allowed by the operating system

A preprocessing directive of the form:

```
# include " q-char-sequence " newline
```

causes that directive to be replaced with the entire contents of the source file identified by the specified character sequence between the “ delimiters. In TopSpeed C++, this case is treated in the same manner as the previous one: the `TS.RED` is used to determine which directories to search.

Consider a preprocessing directive of the form:

```
# include pp-tokens newline
```

In this directive, `pp-tokens` is not delimited by “ or `<` and `>`. As a result, the preprocessing tokens found after `include` in this directive are processed just as in normal text. That is, each identifier currently defined as a macro name is replaced by its replacement list of preprocessing tokens. The directive resulting after all replacements have been made must match one of the two previous forms.

The sequence of preprocessing tokens between a `<` and a `>` preprocessing token pair is combined into a single header name preprocessing token.

Tokens are not implicitly concatenated. Therefore, the glue operator `##` must be used to form a single token between the `<` and `>` delimiters.

For example:

```
#if __STDC__
#define header stdio ## . ## h
#else
#define header extio ## . ## h
#endif
#define incname < header >
#include incname
```

For more complex examples of macro-replaced `#includes`, see the sections on `#` and `##` operators above.

Line Control

The source file is processed beginning at line 1. The *line number* of the current source line is one greater than the number of newline characters read or introduced up to that point in the first translation phase.

A preprocessing directive of the form:

```
# line digit-sequence newline
```

causes TopSpeed C++ to behave as if the line number of the next and subsequent source lines is based from the digit-sequence (interpreted as a decimal integer). The range of possible values is 1..32767.

A preprocessing directive of the form:

```
# line digit-sequence string-literal newline
```

sets the line number as above, and also changes the presumed name of the source file. The new name is specified by the characters contained within the string-literal, which must be a character string literal.

A preprocessing directive of the form:

```
# line pp-tokens newline
```

allows the digit-sequence and string-literal to be generated via macro names. The preprocessing tokens after line on the directive are processed just as in normal text. The directive resulting after all replacements must match one of the two previous forms.

Error Directive

A preprocessing directive of the form:

```
# error pp-tokensopt newline
```

generates an error message of the form:

```
# error directive: pp-tokens
```

This error is treated like any other compilation error.

Pragma Directive

A preprocessing directive of the form:

```
# pragma pp-tokensopt newline
```

provides a mechanism for passing information to the compiler. TopSpeed C++ has numerous pragmas, which are described in the *Developer's Guide*.

Null Directive

A preprocessing directive of the form:

```
# newline
```

has no effect.

Predefined Macro Names

The following ANSI macros are predefined:

| | |
|--------------------------|--|
| <code>__LINE__</code> | This represents the line number of the current source line, a decimal constant. |
| <code>__FILE__</code> | This is the name of the source file, a string literal. |
| <code>__DATE__</code> | This represents the date of compilation of the source file. The replacement is a string literal of the form “Mmm dd yyyy”, where the names of the months are the same as those generated by the <code>asctime</code> function, and the first character of <code>dd</code> is a space character if the value is less than 10. |
| <code>__TIME__</code> | This represents the time of compilation of the source file. The replacement is a string literal of the form “hh:mm:ss”, as in the time generated by the <code>asctime</code> function. |
| <code>__STDC__</code> | This is defined as the decimal constant 1 if the pragma option(<code>ansi=>on</code>) is specified; otherwise the macro is undefined. |
| <code>__cplusplus</code> | This is defined as the decimal constant 1. |

In addition, TopSpeed C++ has other predefined macros. See the *Developer's eGuide* for information about these.

None of these macro names, nor the identifier defined, may be the subject of a `#define` or a `#undef` preprocessing directive.

Preprocessor Syntax Summary

The following listing summarizes the syntax for preprocessor directives.

```
preprocessing-file:
    groupopt
group:
    group-part
    group group-part
```

```

group-part:
    pp-tokensopt newline
    if-section
    control-line
if-section:
    if-group elif-groupsopt else-groupopt endif-line
if-group:
    # if const-expr newline groupopt
    # ifdef identifier newline groupopt
    # ifndef identifier newline groupopt
elif-groups:
    elif-group
    elif-groups elif-group
elif-group:
    # elif const-expr newline groupopt
else-group:
    # else newline groupopt
endif-line:
    # endif newline
control-line:
    # include pp-tokens newline
    # define identifier replace-list newline
    # define ident lparen ident-listopt ) replace-list newline
    # undef identifier newline
    # line pp-tokens newline
    # error pp-tokensopt newline
    # pragma pp-tokensopt newline
    # newline
lparen:
    the left-parenthesis character without preceding white space
replace-list:
    pp-tokensopt
pp-tokens:
    preprocessing-token
    pp-tokens preprocessing-token
preprocessing-token:
    header-name (only within a #include directive)
    identifier (no keyword distinction)
    pp-number
    character-constant
    string-literal
    operator
    punctuator
    each non-white space character that cannot be one of the above
header-name:
    < h-char-sequence >
    " q-char-sequence "
h-char-sequence:
    h-char
    h-char-sequence h-char
h-char:
    any character in the source character set,
    except the newline character and >
q-char-sequence:
    q-char
    q-char-sequence q-char
q-char:
    any character in the source character set,
    except the newline character and "
newline:
    the newline character

```

pp-number:
digit
. digit
pp-number digit
pp-number nondigit
pp-number e sign
pp-number E sign
pp-number .

APPENDIX A

REFERENCES

“The Annotated C++ Reference Manual (ANSI Base Document)”, Margaret A. Ellis & Bjarne Stroustrup, Addison Wesley (1990).

“American National Standard for Information Systems - Programming Language C, ANSI X3.159-1989”.

“American National Dictionary for Information Processing Systems,” Information Processing Systems Technical Report ANSI X3/TR-1-82 (1982).

“ISO 646-1983 Invariant Code Set.”

“IEEE Standard for Binary Floating-Point Arithmetic,” (ANSI/IEEE Std 754-1985).

APPENDIX B

SUMMARY OF SCOPE RULES

A name used in a program is first checked to ensure that it is not ambiguous (that is, there is only one object of that name in scope, apart from overloaded functions - see Chapter 13).

If the name can be accessed without errors, the type of the expression in which the name is found is used to resolve any overloading. Overloading is described in Chapter 13.

If the name is not ambiguous, the access rules are applied to check that access to the name is permitted at this point in the program. Access rules are described in Chapter 11.

A name which is specified after `X::`, where `X` is a class name, must be a member of the class `X` or of a base class of class `X`, or the name of a nested type, nested class or enumerator defined within class `X` or a base class of class `X`.

A name which is specified after `exp.`, where `exp` is an expression of type `X`, must be a member of class `X` or of a base class of class `X`.

A name which is specified after `exp->`, where `exp` is an expression of type *pointer to X*, must be a member of class `X` or of a base class of class `X`. A name specified after `exp->`, where `exp` is an expression of type `Y`, and `Y` is a class in which `operator->` has been redefined so that `exp.operator->()` returns a pointer to class `X`, must be a member of class `X` or a base class of class `X`.

A name not qualified as above which is used outside any function or class, or which is prefaced by the unary `::` operator, must name a global object, function or enumerator.

A name not qualified as above, and not prefaced by the unary `::` operator, which is used in a function (other than a member function) must be declared in the current block or in an enclosing block, or must be a global name. A local name hides declarations of the same name in enclosing blocks or global - overloading does NOT occur between different scopes.

A name not qualified as above, and not prefaced by the unary `::` operator, which is used in a non-static member function of class `X` must be declared in

the current block or in an enclosing block, or must be a member of class X or a base class of class X, or must be a global name. A local name hides declarations of the same name in enclosing blocks, member names, and global names. A member name hides base class members and globals of the same name.

A name not qualified as above, and not prefaced by the unary :: operator, which is used in a static member function of class X, must be declared in the current block or in an enclosing block, or must be a static member of class X, or must be a global name.

A name used as a function argument name introduces a new name in the scope of the outermost block of a function definition, or in a local scope which disappears at the end of a function declaration which is not also a definition. A default argument is evaluated in the scope current at the point of its declaration - it may not access local variables or non-static class members. A default argument is re-evaluated every time its function is called.

A *ctor-initializer* is evaluated in the scope of its constructor's outermost block. (Note that the arguments to the constructor are therefore in scope).

APPENDIX C

NAME ENCODING

In order to implement function overloading, and the scope rules of C++ which allows identical external names to be declared in different scopes, all function names and external data names are encoded by TopSpeed C++ to allow them to be distinguished by the linker. Although the TopSpeed system will decode these names when they are displayed (in MAP files and by VID, for example), it may be useful to know the encoding algorithm used in order to interface to other products. This appendix documents how TopSpeed C++ encodes external names.

The encoding algorithm used is based on that described in the Base Document, with some adjustments to facilitate mixed language programming between the TopSpeed family of languages.

Each encoded name consists of up to four parts.

1. The prefix - this is added to the front of all names after encoding is complete. By default, this consists of a single underbar, but this can be adjusted using the pragma prefix, and is also adjusted within Pascal or Modula linkage specifications.
2. The owner qualification - this indicates the class inside which the name or type is declared. For each level of nesting, the owner name is encoded as its length (in ASCII) followed by the characters of its name. The owner qualification is terminated by a double underbar. The owner qualification is only present for member functions and static data members.
3. The name - this appears exactly as it appears in the program, except for the following special encodings :-

Special function encodings

```
aa__ operator &&  
ad__ operator &  
as__ operator =  
cl__ operator ()
```

```

cm__ operator ,
co__ operator ~
ct__ constructor
dl__ operator delete
dt__ destructor
dv__ operator /
eq__ operator ==
er__ operator ^
ge__ operator >=
gt__ operator >
ls__ operator <<
le__ operator <=
lt__ operator <
md__ operator %
mi__ operator -
ml__ operator *
mm__ operator -
ne__ operator !=
nt__ operator !
nw__ operator new
oo__ operator ||
opX__ conversion function operator <type>, where
      X is the <type> encoded as described below.
or__ operator |
pl__ operator +
pp__ operator ++
rf__ operator ->
rm__ operator ->*
rs__ operator >>
vc__ operator []

```

Compound assignment operators of the form `<op>=` are encoded as for the corresponding operator above, prefixed by ‘a’, so for example the operator `+=` function is encoded as `apl__`.

4. The argument profile. This is only present for function names, and is separated from the name by an ‘@’ symbol. Each argument is encoded in turn, using a sequence of characters from the table below to indicate the type. The this pointer for member functions is not encoded - however, any qualifiers applied to a non-static member function are encoded at the start of the argument profile. After these qualifiers (if present) comes an ‘F’ to introduce the argument profile proper.

Type encodings

```

c char
d double
e ellipsis
f float
i int
l long
s short
r long double
v void

```

A class type is encoded as its length in ASCII followed by its name, so myclass is encoded as 7myclass. A qualified name is encoded as Q, followed by the number of names to follow as a single digit, followed by each name prefixed by its name as usual, so that the type outer::inner would be encoded as Q25outer5inner.

The following modifiers may be applied to each of the above types. Modifiers are always specified in alphabetical order

```
Modifiers
C const
S signed
U unsigned
V volatile
```

Encodings for derived types are built up from the encodings for simple types by combining them with the encodings for the declarators :-

```
Declarators
P<type> pointer to <type>
R<type> reference to <type>
F<args>_<type> function taking specified arguments returning <type>
A<size>_<type> array of specified size of <type>
M<type-1><type-2> pointer to member of class <type-1> of type <type-2>
```

TopSpeed C++ does not use special encodings for repeated arguments of the same type, as described in the Base Document. TopSpeed C++ encodes a function without arguments as simply F, rather than Fv (to indicate the void argument list).

A few examples of C++ functions and their encodings are provided below, to illustrate how they are encoded. Note that the return type is not encoded in a function name.

```
Complex::Complex(double) _7Complex__ct__@Fd
operator+(Complex&,double) _p1__@FR7Complexd
myclass::foo(int) const _7myclass__foo@CFi
myfunc() _myfunc@F
f(volatile const int&)_f@FCVRi
outer::inner::f() _5outer5inner__f@F
f(outer::inner &) _f@FRQ25outer5inner
```

APPENDIX D

IMPLEMENTATION DEFINED BEHAVIOR

Implementation-defined behavior is behavior which depends on the characteristics of the implementation. It is assumed that the program construction and data are correct.

The implementation-defined features for TopSpeed C++ are described here.

1. Translation

- Diagnostics are reported either as warnings or errors as follows:
(File.c line,col) Error:message
(File.c line,col) Warning:message

The messages are documented in Appendix H of this manual.

2. Environment

- If the main function is declared as
`int main(int argc, char *argv[])`

The array elements `argv[0]` to `argv[argc - 1]` contain pointers to strings. These strings contain the command line arguments for the program. The case of the arguments is preserved. The command line arguments are separated by one or more space characters, except for text enclosed in double quotes (which will count as a single argument)

`argv[0]` is a pointer to the program name, except under version 2 of DOS where this element is not available. `argv[0]` is a NULL pointer in that version.

The program invocation

```
myprog First "Second plus more"
```

results in the following values for `argv`:

```
argv[0] points to "myprog"  
argv[1] points to "First"  
argv[2] points to "second plus more"
```

env, an optional third argument to main, is an array of pointers to strings containing the operating system environment variables. The last pointer in env is a NULL pointer.

- The function main() has C linkage.
- The operating system determines what is to be considered as an interactive device. Thus under DOS and OS2 the console, printer and serial port (if present) are treated as interactive devices, but disc files are not.

3. Identifiers

- In TopSpeed C++, all characters in an identifier are significant. This is true for all identifiers, both during compilation and linking.

Note The maximum token length (and thus identifier length) is 1024 characters.

- Case is significant for external linkage.

4. Characters

- The translation and execution character sets consist of the ASCII character set (character values 0-127) plus the IBM character set (character values 128-255)
- TopSpeed C++ treats multibyte characters in a simple way: a multibyte sequence consists of only one character (ie there is only an initial shift state) Thus the “extended” character set has the range 0 to 255.
- There are 8 bits in a character in the execution character set
- Source characters are mapped directly without change onto execution characters. Thus any string literal will be the same in the source character set as in the execution character set.
- Any integer character constant that contains a character or escape sequence not represented in the basic execution character set, or the extended character set for a wide character constant, will be represented as an int if possible, otherwise the value has overflowed. A warning is issued if an escape sequence cannot be represented as a char.
- An integer character is a sequence of 1 to 4 characters. If the sequence has one character, it is mapped to its corresponding ASCII value. If there are more characters, they are mapped to an integer type that can represent the value. Each character is mapped to a byte in this integer type.

For example the constant 'abcd' is represented as a long with the hexadecimal value 0x61626364. If a character constant contains an escape sequence whose value is not representable as a char, a warning is issued and the value of the escape sequence is truncated to a char value. The same applies to wide character constants.

- The current locale used to convert multibyte characters into corresponding wide character (codes) for a wide character constant is the C locale.
- A plain char is signed by default. This can be changed to an unsigned char using the pragma option (uns_char=>on).

The char type is signed. All characters in the source character set have positive values when stored in a char variable; all other values (that is, those in the range 128 to 255) will have negative values.

5. Integers

- The sets of values represented by various integer types are described in the limits.h file. Integer types are represented using the two's complement format.
- If a value with integral type is converted to a signed integer with smaller size, the high-order part of the converted value is discarded. There is no change in the actual bit pattern. If a value of unsigned type is converted to a signed integer of the same size, the bit pattern is unchanged.
- Bitwise operations for signed integers are performed as for unsigned integers except for '<<': see below
- The remainder has a negative sign when only one of the operands is negative. The remainder is positive when neither or both operands are negative.
- The right shift on a negative-valued signed integral type preserves the sign: that is, the sign bit is propagated to the right.

6. Floating point

- The representation of floating point numbers conforms to the IEEE standard. The set of values is defined in float.h include file.
- When a value of integral type is converted to floating type, and the value being converted cannot be represented exactly, the results is the nearest value smaller than the original value.

- When a double (or long double) is converted to a float, and the value cannot be represented exactly, the results is the nearest value smaller than the original value.

7. Arrays and pointers

- The sizeof operator yields a constant value that has type `size_t`. This type is defined in the header file `<stddef.h>`. In TopSpeed C++, `size_t` has type unsigned int.
 - In TopSpeed C++ pointers can have two sizes. A pointer can be a far (4 byte) or a near (2 byte) pointer. The size depends on the memory model used, as described in greater detail in the *Developer's Guide*. In a conversion between an integer and a pointer, the pointer is seen as having an unsigned integer type corresponding to its size
 - A near pointer can be converted to an unsigned int or to an int without loss of information. No change takes place in the bit pattern. A far pointer can be converted to an unsigned long or to a long without loss of information. If a far pointer is converted to an int or to an unsigned int, the segment part is discarded.
 - An integer can be converted to a pointer. An int or unsigned int can be converted to near pointer without change in the bit pattern. If the integer is converted to a far pointer, the segment part will contain the default data segment (or default code segment if a code pointer)
- A long or unsigned long can be converted to a far pointer without change in value. If a long is converted to a near pointer, the high-order word is discarded.
- The result of subtracting two pointers has the type `int`, which is also defined as the typedef `ptrdiff_t` in the header file `<stddef.h>`

Note The result of subtracting two huge pointers has type long int.

8. Registers

- TopSpeed C++ is an optimizing compiler that does its own optimal register allocation for variables. Therefore the register storage class does not have any effect.

9. Structures, Unions, Enumerations and Bit-fields

- If a member of a union (`m1`) is accessed after a value has been assigned to a different member (`m2`), then the assigned value will be interpreted according to the type of `m1`. Note that no conversion takes place; rather, the bit pattern is simply interpreted according to the type of `m1`.

- Except for bit-fields, all members of a struct are byte aligned. That is, the members are packed as tightly as possible.
- The high-order bit in an int bit-field is treated as a sign bit.
- Bit-fields are allocated within a byte, word or doubleword from the least significant bit.
- Several bit-fields can be packed into a byte, word or doubleword. A char bitfield will never overlap a byte boundary; an int bitfield will overlap at most one byte boundary, and a long bitfield will overlap at most three byte boundaries. If insufficient space remains in the current byte to fulfil the above requirements, the allocation starts at the next byte boundary.
- All enumeration types are represented as the type int.

10. Qualifiers

- An object of volatile qualified type is considered to be accessed when its value is read or a new values is written into it.

11. Declarations

- An asm declaration has no meaning defined in this release of TopSpeed C++. TopSpeed C++ will report a warning when an asm declaration is encountered.

12. Declarators

- There is no explicit limit to the maximum number of declarators that can modify an arithmetic, structure or union type

13. Statements

- TopSpeed C++ has no explicit limit for the number of case values in a switch statement. Keep in mind, however, that the compilers can run out of member for the internal representation of a very large number of cases.

14. Classes

- TopSpeed C++ allocates storage for base classes in the order of their occurrence in a depth-first left-to-right recursive scan of the base classes. An exception to this is made for virtual base classes, which are allocated after all other base classes, in the order of their first occurrence in the aforementioned scan.

15. Constructors and New

- In an allocation expression, operator `new()` allocates the memory itself before invoking the constructor (if any) for the object being created.

16. Temporaries

- TopSpeed C++ will create a temporary object under a number of circumstances - these are described in detail in Chapter 12: 'Temporary Objects'.
- TopSpeed C++ will destroy a temporary object at or before the end of the expression in which it is generated, except where it is bound to a reference. In this case, it is destroyed at the point at which the reference is destroyed.

17. Linkage specifications

- The linkage specifications which are supported, and their semantics, are described in Chapter 5: 'Linkage Specifications'.

18. Preprocessing Directives

- Character constants occurring in the conditional expression for a `#if` command are treated in a manner similar to the way in which character constants are handled outside the preprocessor.
- Such character constants may have negative values
- The file name specified in a `#include` directive is searched for via the redirection file (`ts.red`). (See the *Developer's Guide* for a description of the redirection file.)
- TopSpeed C treats both the double quote form and the angle bracket form of file name specification in the same way: they both use the redirection file.
- The file name given in both the double quote form and the angle bracket form must be a valid DOS or OS/2 file name (full name, partial name or tail name). I.e., no "translation" of the specified name takes place.
- For pragma directives see the *Developer's Guide*.
- The date and time of translation are always available

Additional

The following additional implementation-dependent features should also be noted:

- In the preprocessing phase, sequences of whitespace characters (except for newline) are replaced by a single space character.
- The characters ‘ and “ cannot form preprocessing tokens on their own. A compilation error is issued if this occurs.
- #include directives can be nested up to a limit set by the operating system. The operating system (DOS or OS/2) has an upper limit as to how many files can be open at the same time.
- The macro __STDC__ is defined (with value 1) if the pragma option(ansi=>on) is specified. Otherwise, it is undefined.
- An error message is issued if any character is encountered other than the characters in the source character set (specified below), unless such a character occurs in a character constant, string constant or in source that is skipped because of conditional compilation. The source character set consists of the following characters :-

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I | J | K | L | M |
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
| a | b | c | d | e | f | g | h | i | j | k | l | m |
| n | o | p | q | r | s | t | u | v | w | x | y | z |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | |
| ! | “ | # | % | & | ‘ | (|) | * | + | - | . | . |
| : | ; | < | > | = | ? | [|] | / | ^ | _ | { | } |
| | | | | | | | | | | | | ~ |

plus the horizontal tab, vertical tab, form feed, and end-of-line characters.

APPENDIX E

UNDEFINED BEHAVIOR

The base document imposes no requirements on the behavior of a C++ implementation in the following cases. The behavior in these cases is assumed to involve one or more of the following:

- Use of a non-portable or erroneous program construct.
- Use of erroneous data.
- Use of indeterminately-valued objects.

The behavior in the TopSpeed C++ environment when such cases arise is summarized in this appendix.

1. If the main function returns without specifying a value, the termination status given to DOS or OS/2 is undefined.
2. If the source contains a character not contained in the source character set, an error message will be given - unless the source is skipped by conditional compilation.
3. If an identifier has both external and internal linkage within the same translation unit, the identifier has internal linkage.
4. The compiler checks as far as possible that all declarations referring to the same object or function have compatible types. Incompatible declarations not found by the compiler (for example if the declarations occur in different translation units) will be found by TopSpeed linker which has type-safe linking (i.e., the object files contain type information that is checked by the linker).
5. If the `\` in an escape sequence is followed by a character that is not defined as an escape sequence in the base document, the `\` character is ignored. For example the escape sequence `\J` yields the character J.
6. Adjacent string literals are concatenated. This is also true if one is a character string literal and the other is a wide character string literal.

7. Identical string literals are distinct. A string literal can be modified; however, this is not recommended since future versions could put string literals in read only memory.
8. If a pointer has an invalid value assigned to it, and the indirection operator * is applied to it:
 - Under OS/2 a General Protection error will result.
 - Under DOS some random memory location (corresponding to the pointer value) will be referenced. If a value is assigned to the specified location, the result is unpredictable
9. If the second operand of a division or modulus operator is of integral type and has the value zero, an error is reported.
10. If the right hand operand of a shift left operator is negative or has a value greater than the word width (16), the result will be 0.

If the right hand operand of a shift right operator is negative or has a value greater than the word width, the result will be 0 if the left operand is unsigned, or else signed with sign bit 0; if the left operand is signed and the sign bit is 1, the result is -1.
11. If an actual argument for a macro call consists of no tokens, then an empty token-list is substituted for the formal argument in the replacement list.
12. If a change is made to a const object, under OS/2 a protection violation may occur. Under DOS, in general, the value of the const object will be modified without any exception occurring. The same is true of string literals.
13. If a pure virtual function is called, a run-time error will be reported.

APPENDIX F

TOPSPEED C++ EXTENSIONS.

This appendix summarizes the extensions to the language defined in the Base document that are provided by the TopSpeed C++ implementation

1. TopSpeed C++ allows env as an optional third parameter for main. env is an array of pointers to strings. The strings contain environment variables.
2. TopSpeed C++ provides the pragma option(nest_cmt) to get around the restriction on nested comments.
4. TopSpeed C++ provides several extensions in the way in which declarators work for various types and in the flexibility declarators have. These take the form of additional modifiers, using the keywords described below. Such language extensions can be disabled by using the pragma option(lang_ext=>off).
5. TopSpeed C++ has a special pointer type, called a relative (or based) pointer. A *relative pointer* is a near (16-bit) pointer; however, instead of having DS or CS as the segment value, you can specify the segment yourself in the pointer declarator.
6. TopSpeed C++ provides a variety of additional C++ keywords. These may be used to modify declarations of variables, pointers and functions. These keywords can be disabled by the pragma option(ansi=>on).

cdecl pascal near far huge interrupt
7. TopSpeed C++ allows you to initialize functions, for the purpose of defining in-line machine code.
8. TopSpeed C++ has an extension in which a cast yields an lvalue if the expression is an lvalue. This extension can be disabled by specifying the pragma option(lang_ext=>off).
9. TopSpeed C++ allows as an extension a mixture of pointer and integer operands for the relational operators. As a result, an expression such as

p > 0

(where p is a pointer) is valid.

APPENDIX G

COLLECTED SYNTAX.

```

identifier:
    non-digit
    identifier non-digit
    identifier digit
non-digit: one of
    _ a b c d e f g h i j k l m
      n o p q r s t u v w x y z
      A B C D E F G H I J K L M
      N O P Q R S T U V W X Y Z
digit: one of
    0 1 2 3 4 5 6 7 8 9
operator: one of
    [ ] ( ) . ->
    + - ~ ! / % < > ^ |
    ? : = , # sizeof
    ++ -- & *
    << >> <= >= == != && ||
    *= /= %= += -= <<= >>= &= ^= |=
    .* ->* ::
    ##
punctuator: one of
    [ ] ( ) { } * , : = ; ... #
literal:
    floating-constant
    integer-constant
    enumeration-constant
    character-constant
    string-literal
floating-constant:
    fractional-constant exponent-partopt floating-suffixopt
    digit-sequence exponent-part floating-suffixopt
fractional-constant:
    digit-sequenceopt . digit-sequence
    digit-sequence .
exponent-part:
    e signopt digit-sequence
    E signopt digit-sequence
sign:
    +
    -
digit-sequence:
    digit
    digit-sequence digit
floating-suffix:
    f
    l
    F
    L

```

```

integer-constant:
    decimal-constant integer-suffixopt
    octal-constant integer-suffixopt
    hexadecimal-constant integer-suffixopt
decimal-constant:
    non-zero-digit
    decimal-constant digit
octal-constant:
    0
    octal-constant octal-digit
hexadecimal-constant:
    0x hexadecimal-digit
    0X hexadecimal-digit
    hexadecimal-constant hexadecimal-digit
non-zero-digit: one of
    1 2 3 4 5 6 7 8 9
octal-digit: one of
    0 1 2 3 4 5 6 7
hexadecimal-digit: one of
    0 1 2 3 4 5 6 7 8 9
    a b c d e f
    A B C D E F
integer-suffix:
    unsigned-suffix long-suffixopt
    long-suffix unsigned-suffixopt
unsigned-suffix:
    u
    U
long-suffix:
    l
    L
character-constant:
    ' c-char-sequence '
    L ' c-char-sequence '
c-char-sequence:
    c-char
    c-char-sequence c-char
c-char:
    any character in the source character set except the single-quote ' ,
    backslash \ , or newline character
    escape-sequence
string-literal:
    " s-char-sequenceopt "
    L " s-char-sequenceopt "
s-char-sequence:
    s-char
    s-char-sequence s-char
s-char:
    any character in the source character
    set except the double-quote " ,
    backslash \ , or newline
    escape-sequence
declaration:
    decl-specifiersopt declarator-listopt ;
    asm-declaration
    function-definition
    linkage-specification
decl-specifiers:
    decl-specifier
    decl-specifiers decl-specifier

```

```

decl-specifier:
    storage-class-specifier
    type-specifier
    fct-specifier
    friend
    typedef
storage-class-specifier:
    extern
    static
    auto
    register
fct-specifier:
    virtual
    inline
type-specifier:
    simple-type-name
    class-specifier
    enum-specifier
    elaborated-type-specifier
    const
    volatile

simple-type-name:
    void
    char
    short
    int
    long
    float
    double
    signed
    unsigned
    complete-class-name
    qualified-type-name
qualified-type-name:
    typedef-name
    class-name :: qualified-type-name
complete-class-name:
    qualified-class-name
    :: qualified-class-name
qualified-class-name:
    class-name
    class-name :: qualified-class-name
elaborated-type-specifier:
    class-key class-name
    class-key identifier
    enum enum-name
class-key:
    class
    struct
    union
enum-specifier:
    enum identifieropt { enum-listopt }
enum-list:
    enumerator
    enum-list , enumerator
enumerator:
    identifier
    identifier = constant-expression
asm-declaration:
    asm ( string-literal ) ;

```

```

linkage-specification:
    extern string-literal { declaration-listopt }
    extern string-literal declaration
declaration-list:
    declaration
    declaration-list declaration
declarator-list:
    init-declarator
    declarator-list , init-declarator
init-declarator:
    declarator initializeropt
declarator:
    ptr-operatoropt direct-declarator
direct-declarator:
    modifiersopt dname
    declarator [ constant-expressionopt ]
    declarator ( argument-declaration-list ) cv-qualifier-listopt
    ( declarator )

ptr-operator:
    modifiersopt * cv-qualifier-listopt
    modifiersopt & cv-qualifier-listopt
    modifiersopt complete-class-name :: * cv-qualifier-listopt
cv-qualifier-list:
    cv-qualifier
    cv-qualifier-list cv-qualifier
cv-qualifier:
    const
    volatile
dname:
    name
    class-name
    ~ class-name
    typedef-name
    qualified-type-name
modifiers:
    modifier
    modifiers modifier
modifier:
    cdecl
    far
    huge
    interrupt
    near
    pascal
    < expression >
argument-declaration-list:
    arg-declaration-listopt ...opt
    arg-declaration-listopt , ...
arg-declaration-list:
    argument-declaration
    arg-declaration-list , argument-declaration
argument-declaration:
    decl-specifiers declarator
    decl-specifiers declarator = expression
    decl-specifiers abstract-declaratoropt
    decl-specifiers abstract-declaratoropt = expression
type-name:
    type-specifier-list abstract-declaratoropt
type-specifier-list:
    type-specifier type-specifier-listopt

```

```

abstract-declarator:
    ptr-operatoropt direct-abstract-dtor
direct-abstract-dtor:
    ( abstract-declarator )
    direct-abstract-dtoropt [ constant-expressionopt ]
    direct-abstract-dtoropt ( argument-declaration-list )
    cv-qualifier-listopt
function-definition:
    decl-specifiersopt declarator ctor-initializeropt fct-body
fct-body:
    compound-statement
initializer:
    = assignment-expression
    = { initializer-listopt }
    ( expression-list )
initializer-list
    expression
    initializer-list , expression
    { initializer-listopt }
primary-expression:
    name
    literal
    :: identifier
    :: operator-function-name
    :: qualified-name
    this
    ( expression )
name:
    identifier
    operator-function-name
    conversion-function-name
    qualified-name
    class-name
    ~ class-name
    qualified-name
qualified name:
    qualified-class-name :: name
postfix-expression:
    primary-expression
    postfix-expression [ expression ]
    postfix-expression ( argument-expression-listopt )
    simple-type-name ( argument-expression-listopt )
    postfix-expression . name
    postfix-expression -> name
    postfix-expression ++
    postfix-expression --
argument-expression-list:
    assignment-expression
    argument-expression-list , assignment-expression
simple-type-name ( argument-expression-listopt )
unary-expression:
    postfix-expression
    ++ unary-expression
    -- unary-expression
    unary-operator cast-expression
    sizeof unary-expression
    sizeof ( type-name )
    allocation-expression
    deallocation-expression
unary-operator: one of
    & * + - ~ !

```

```

allocation-expression:
    ::opt new placementopt new-type-name new-initializeropt
    ::opt new placementopt ( type-name ) new-initializeropt
placement:
    ( expression-list )
new-type-name:
    type-specifier-list new-declaratoropt
new-declarator:
    * cv-qualifier-listopt new-declaratoropt
new-initializer:
    ( initializer-list )
deallocation-expression:
    ::opt delete cast-expression
    ::opt delete [ ] cast-expression
cast-expression:
    unary-expression
    ( type-name ) cast-expression
pm-expression:
    cast-expression
    pm-expression .* cast-expression
    pm-expression ->* cast-expression
multiplicative-exp:
    pm-expression
    multiplicative-exp * pm-expression
    multiplicative-exp / pm-expression
    multiplicative-exp % pm-expression
additive-expression:
    multiplicative-expression
    additive-expression + multiplicative-expression
    additive-expression - multiplicative-expression
shift-expression:
    additive-expression
    shift-expression << additive-expression
    shift-expression >> additive-expression
relational-expression:
    shift-expression
    relational-expression < shift-expression
    relational-expression > shift-expression
    relational-expression <= shift-expression
    relational-expression >= shift-expression
equality-expression:
    relational-expression
    equality-expression == relational-expression
    equality-expression != relational-expression
AND-expression:
    equality-expression
    AND-expression & equality-expression
exclusive-OR-expression:
    AND-expression
    exclusive-OR-expression ^ AND-expression
inclusive-OR-expression:
    exclusive-OR-expression
    inclusive-OR-expression | exclusive-OR-expression
logical-AND-expression:
    inclusive-OR-expression
    logical-AND-expression && inclusive-OR-expression
logical-OR-expression:
    logical-AND-expression
    logical-OR-expression || logical-AND-expression

```

```

conditional-expression:
    logical-OR-expression
    logical-OR-expression ? expression : conditional-expression
assignment-expression:
    conditional-expression
    conditional-expression assignment-op assignment-expression
assignment-op: one of
    = *= /= %= += -= << >>= &= ^= |=
expression:
    assignment-expression
    expression , assignment-expression
constant-expression:
    conditional-expression
statement:
    labeled-statement
    compound-statement
    expression-statement
    jump-statement
    selection-statement
    iteration-statement
    declaration-statement
labeled-statement:
    identifier : statement
    case constant-exp : statement
    default : statement
compound-statement:
    { statement-listopt }
statement-list:
    statement
    statement-list statement
expression-statement:
    expressionopt ;
jump-statement:
    goto identifier ;
    continue ;
    break ;
    return expressionopt ;
selection-statement:
    if ( expression ) statement
    if ( expression ) statement else statement
    switch ( expression ) statement
iteration-statement:
    while ( expression ) statement
    do statement while ( expression ) ;
    for ( for-init-statement expropt ; expropt ) statement
for-init-statement:
    expression-statement
    declaration-statement
declaration-statement:
    declaration
class-name:
    identifier
class-specifier:
    class-head { member-list opt }
class-head:
    class_key identifieropt base-specopt
    class-key class-name base-specopt
class-key:
    class
    struct
    union

```

```

member-list:
    member-declaration member-listopt
    access-specifier : member-listopt
member-declaration:
    decl-specifieropt member-declarator-listopt ;
    function-definition ;opt
    qualified-name ;
member-declarator-list:
    member-declarator
    member-declarator-list , member-declarator
member-declarator:
    declarator pure-specifieropt
    identifieropt : constant-expression
base-spec:
    : base-list

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

base-specifier:
    class-name
    virtual access-specifieropt class-name
    access-specifier virtualopt class-name
member declarator:
    declarator pure-specifieropt
pure-specifier:
    = 0
member-list:
    access-specifier : member-listopt

access-specifier:
    public
    private
    protected
base-specifier:
    virtual access-specifieropt class-name
    access-specifier virtualopt class-name
conversion-function-name:
    operator conversion-type-name
conversion-type-name:
    type-specifier-list ptr-operatoropt
ctor-initializer:
    : mem-initializer-list
mem-initializer-list:
    mem-initializer
    mem-initializer , mem-initializer-list
mem-initializer:
    complete-class-name ( expression-listopt )
    identifier ( expression-listopt )
preprocessing-file:
    groupopt
group:
    group-part
    group group-part
group-part:
    pp-tokensopt newline
    if-section
    control-line
if-section:
    if-group elif-groupsopt else-groupopt endif-line

```

```

if-group:
    # if const-expr newline groupopt
    # ifdef identifier newline groupopt
    # ifndef identifier newline groupopt
elif-groups:
    elif-group
    elif-groups elif-group
elif-group:
    # elif const-expr newline groupopt
else-group:
    # else newline groupopt
endif-line:
    # endif newline
control-line:
    # include pp-tokens newline
    # define identifier replace-list newline
    # define ident lparen ident-listopt ) replace-list newline
    # undef identifier newline
    # line pp-tokens newline
    # error pp-tokensopt newline
    # pragma pp-tokensopt newline
    # newline
lparen:
    the left-parenthesis character without preceding white space
replace-list:
    pp-tokensopt
pp-tokens:
    preprocessing-token
    pp-tokens preprocessing-token
preprocessing-token:
    header-name (only within a #include directive)
    identifier (no keyword distinction)
    pp-number
    character-constant
    string-literal
    operator
    punctuator
    each non-white space character that cannot be one of the above
header-name:
    < h-char-sequence >
    “ q-char-sequence “
h-char-sequence:
    h-char
    h-char-sequence h-char
h-char:
    any character in the source character set,
    except the newline character and >
q-char-sequence:
    q-char
    q-char-sequence q-char
q-char:
    any character in the source character set,
    except the newline character and “
newline:
    the newline character

```

pp-number:
 digit
 . digit
 pp-number digit
 pp-number nondigit
 pp-number e sign
 pp-number E sign
 pp-number .

APPENDIX H

COMPILER ERRORS AND WARNINGS

Introduction

This Appendix explains the errors and warnings given by the TopSpeed C++ compiler. The chapter consists of two sections, each containing an alphabetical or numerical list of messages and explanations. The sections are:

1. C++ language warnings. These report doubtful C++ code that is, nevertheless, legal. You need not fix these warnings to make an executable file.
2. C++ language errors. These report violations of the language syntax, constraints and semantics. You must fix these errors before making an executable file.

In this chapter, we use a name surrounded in angle brackets <>, to indicate a name substituted by the compiler. For example, when TopSpeed C++ displays the error described as: Parameter <parameter name> is already defined in function., it will substitute the parameter name in error such as Parameter MyParameter is already defined in function.

C++ Language Warnings

Warnings suggest problems with your C++ code but do not prevent the compiler from producing an object file. The compiler issues warnings for several reasons, mostly to inform you about doubtful C++ constructs. All warnings, or individual warnings can be switched ON or OFF, or can be promoted to errors using the Options Compiler C options Warnings menu, the command line, or #pragmas in your source files. See the *Developers's Guide* for more information.

<variable name> declared but never used

You have declared a local variable in a function or a block statement but never used the variable.

<variable name> is assigned a value that is never used

You have declared and assigned a value to a variable but have never used the value in the function.

Address for local variable not in DGROUP

This warning is generated if the address of a local variable is taken when the pragma `data(ss_in_dgroup=>off)` is used in Small model. This will occur when making a Windows DLL because the pragma `data(ss_in_dgroup=>off)` is used to indicate that the DLL's data segment is not the same as that of the stack segment. In this case, a pointer to a local variable with storage on the stack will not address that variable correctly. Either a variable with static storage or the pointer initialized with a value returned from a Windows memory allocation function should be used.

Assignment in test expression

This warning is primarily useful to warn against potential mistyping of the C equal operator, `==`. A Pascal or beginning C++ programmer might make the mistake of using the assignment operator, `=` instead of the equality operator, `==`. For example:

```
if (x = y) printf("x is equal to y");
```

This warning is generated when a class definition occurs as a function return type specification. Such usage, while valid C++, is very unusual, and almost always indicates that the semicolon that should terminate a class definition has been omitted. This warning can be controlled using the pragma `warn(wcrt)`; the default setting of this pragma is on.

Code has no effect

This warning is issued for an expression statement or for the left operand in a comma expression if the expression has no side effects. For example:

```
f;          // expression statement has no side effects,
            // maybe f(); was intended
a[x,y];    // left operand of , has no side effects,
            // maybe a[x][y] was intended
```

const object in code-segment has non-constant initializer

This warning is generated when the `const_in_code` pragma is specified, and a variable with the `const` attribute has a non-constant initializer. Such variables will be placed in a code segment, and this may result in protection violations under OS/2 or Windows 3 protected mode. This warning can be controlled using the pragma `warn(wcic)`; the default setting of this pragma is on.

Constant has long type

This warning is given if an integral constant without an `L` suffix has type `long` because of its value. This happens for decimal constants when the value

is greater than 32767 and for octal and hexadecimal constants when the value is greater than 65535.

Constant in code segment requires initialization.

This warning is issued if a constant placed in the code segment requires run-time initialization, as may be the case for an object declared `const` in C++, whose initializer is an expression, when the `const_in_code` pragma is set on. This situation will lead to a protection violation in OS/2 and Window 3 protected mode applications, so `const_in_code` should be set off if this warning is encountered. The default setting is on.

Conversion may lose significant digits

This warning is issued if an expression is converted from long or unsigned long to int or unsigned int. In TopSpeed C++, the long integer types are represented in 4 bytes and int and unsigned integers are represented in 2 bytes. However, some C and C++ implementations use the same for all these types; programs ported from such implementations may not work correctly if they contain implicit conversions of this nature.

Declaration has no effect

You have a declaration that doesn't declare anything, for example `long int;`. A declaration must declare either a variable, a typedef or function name, a class or enumeration name, or the members of an enumeration. This warning may also be generated for an asm declaration, which has no effect in TopSpeed C++.

Default access specifier used for base class <name>

If the access specifier for a base class is omitted, `public` is assumed if the derived class is declared `struct`; `private` is assumed if it is declared `class`.

```
class bse {
    int b;
};

class c_drv : bse {    // base class is private
    int d;
};

struct s_drv : bse {   // base class is public
    int d;
};
```

Expression in '[']' in 'delete' is ignored

The number of elements in an array allocated by `new` is now stored in a reserved location (in earlier versions of C++, it had to be supplied by the programmer). Any expression used in `delete` will therefore be ignored.

Far pointer to near pointer conversion

This warning is reported when a 32-bit far pointer is converted to a 16-bit near pointer. When doing the conversion, the compiler discards the segment value. This can make the pointer point into a wrong segment.

Near pointer to far pointer conversion

When a 16-bit near pointer is converted to a 32-bit far pointer, the segment value of the far pointer will be the default data segment (DS) for data pointers, which could be the wrong segment.

'overload' is not needed and is obsolete

C++ version 2.1 no longer requires the use of the overload keyword. For example:

```
overload func; // obsolete
void func(char *);
void func(char *, int);
```

Parameter <parameter name> is never used

The specified parameter has not been used in the function. Some functions may intentionally have dummy parameters, in which case you can ignore the warning. C++ allows parameters which are not used to be unnamed, in which case this warning is not produced.

Pointer conversion

This warning is issued when the compiler has done a conversion between incompatible pointer types or a conversion between a pointer and an integral type. Conversion between two pointer types is usually harmless, while conversion between a pointer and an integer is more doubtful and may not be portable to other implementations. By inserting a suitable typecast in the offending code, the warning will vanish.

Possible use of <variable name> before being assigned a value

You have used a local variable that may not yet have been assigned a value. This condition is checked with a simple scan through the function, so this warning may not necessarily be correct if you use goto statements to change the flow of control.

Returns the address of a local variable

You have a return statement that returns the address of a local variable. This causes a problem because TopSpeed C++ reclaims the variable storage on completion of the function. The pointer, therefore, points to an invalid address.

Unknown pragma

You have a pragma directive that is for a foreign compiler, or you have mistyped a TopSpeed C++ pragma. The compiler, therefore, ignores the pragma.

Unexpected text in preprocessor command

A source line that includes a preprocessor command must be terminated without any further text, except comments, after the command. For example:

```
#ifdef MYMAC
/* .... */
#endif MYMAC
```

The `#endif` command contains extra text `MYMAC`, which is not allowed by the C++ syntax. However, comments are allowed, so the following is legal:

```
#endif /* MYMAC */
```

Unknown preprocessor command encountered while skipping

In a section of a file that is being skipped due to conditional inclusion, a source line that starts with a `#` but is not followed by a valid preprocessor command. This is not an error, but may be useful for locating misspelt `#endif` commands, for example:

Value of constant is out of range

You have specified the value of an integral constant outside the range of an unsigned long, or a floating point constant outside the range of a long double.

Value of escape sequence is too big

You have specified the value of an octal or hexadecimal escape sequence that cannot be represented as an unsigned char.

C++ Language Errors

These messages report violations of the C++ language syntax, constraints and semantics. You must correct any errors in the source code before the compiler will produce executable code. Even though an error has occurred, the compiler will continue to compile the rest of the file to attempt to find all the errors in the source.

<function name> redeclared with different return type

You have declared the function with a return type different from the one in the original declaration.

<identifier> is redeclared as a function

This error reports that the identifier in a function declaration has been previously defined as a variable or an enumeration constant in the same scope, as in:

```
int f;
int f(void); // Error: f is redeclared as a function
```

<modifier name> modifier not allowed here

Not all modifiers make sense in all declarations. See the following list of illegal cases:

1. Pointers to functions cannot have the huge modifier.

```
int (huge *fp)(); // Error
```
2. Class members cannot have modifiers themselves, though a member of pointer type can have modifiers.

```
struct {
    int near x;    // Error: near on a member
    int near *np; // OK, the pointer type is modified
                // not the member np
}
```
3. Function arguments cannot have modifiers themselves, though parameters of pointer type may contain modifiers, for example:

```
int f(int far x, int far * fp);
/* the parameter x cannot be modified; the second
   parameter is ok, far modifies the pointer type */
```
4. Function declarations cannot be modified by the huge modifier.

```
int huge f(); // Error: huge function is not allowed
```
5. Variables may not be modified by huge or interrupt.

```
int huge x; // Error
```
6. In addition to #5, local variables may not be modified by near and far, since local variables are always allocated on the stack.
7. A pointer that does not have pointer to function type cannot be modified by the interrupt, pascal and cdecl modifiers.

<variable name> is already initialized

You have declared an external variable more than once with an initializer. For example:

```
extern int i1 = 1;
extern int i1 = 1; // Error, initialized twice

extern int i2 = 1;
extern int i2;    // Ok, no initializer second time
```

<variable name> is already defined

You have declared the same local variable more than once. You can, however, declare a global variable any number of times and all declarations will refer to the same variable. For example:

```
extern unsigned j;
extern unsigned j; // OK, j has external linkage,
                  // refers to previous

{
    int i;
    int i;          // Error: local variable defined
                  // more than once
}
```

is not followed by a parameter

You must follow the # operator with a macro parameter, for example:

```
// the following is correct use of the '#' operator

#define idebug(a) printf("%s = %d\n",#a,a)
```

at beginning or end of replacement list

You cannot have the glue operator ## for macro definitions at the beginning or end of the macro replacement list.

#elif command has no matching #if command

You have an #elif preprocessor command without a previous matching #if, #ifdef, or #ifndef command.

#else command has no matching #if command

You have an #else preprocessor command without a previous matching #if, #ifdef, or #ifndef command.

#endif command has no matching #if command

You have an #endif preprocessor command without a previous matching #if, #ifdef, or #ifndef command.

#error directive: <text>

You have placed an #error preprocessor directive in the file that has been executed by the preprocessor. <text> is the user-specified text immediately following the #error command. For example:

```
#ifndef __MSDOS__
    #error This program will only run under MS-DOS
#endif
```

will issue an error when the macro `__MSDOS__` is not defined, thus preventing compilation on the program.

& operator not allowed on bit-field variable

You cannot apply the address operator `&` to a bit-field variable.

#if commands too deeply nested

`#if` preprocessor directives can only be nested up to a level of 32.

#pragma save commands too deeply nested

You can only nest `#pragma save` commands up to a level of 10.

Access declaration in 'private' section of class

An access specifier may be used in the protected or public part of a derived class declaration only.

Allocated storage on stack > 64K

You have exhausted the space for local variables. The compiler allocates local variables on the run-time stack, which can be up to 64K bytes. The stack has a default size that is dependent on the memory model. The default stack size can be changed with the data pragma. You have two alternatives when you get this error:

1. Change some of the largest variables into external variables by moving their declarations out of the function.
2. Use the heap allocation functions to allocate the memory needed.

Ambiguity between user-defined conversion and operator overloading

It is not possible for the compiler to resolve the ambiguity between two or more user-defined conversions and/or operators.

```

class b {
public:
    int i;
    operator int ();
    int operator *(int);
};

void foo()
{
    b x;
    int i;
    i = x * 2; // error - convert x to 'int' or use
               // x.operator*(int) ??
}

```

Ambiguous call

It not possible for the compiler to decide which function to call. Argument matching cannot distinguish between the two overloaded functions since both require standard conversions of equal merit, for example from int to char and from int to signed char.

```

class c{
    int func(char);
    int func(signed char);
    void f(int i) {
        func(i); // func(char) or func(signed char) ?
    }
};

```

Ambiguous class member reference

There are multiple declarations of a base class member. The member name must be qualified to be used.

```

class a {
    int f();
};

class b {
    int f();
};

class c : public a, public b {
    int i;
    c() { i=f(); } // ambiguous, a::f or b::f ?
};

```

Ambiguous conversion

It is not possible for the compiler to resolve the ambiguity between two or more user-defined conversions.

```

struct a {
    operator int();
};

struct c : public a {
    operator void *();
};

void func(c& rc) {
    if (rc) // which conversion, void * or int ?
        return;
}

```

Ambiguous pointer conversion (base class is not unique in derived class)

A pointer must unambiguously refer to the same object. In this example, both classes c and d contain a bse. The base class bse of c and d should be virtual if it is the intention that it is the same object.

```

class c : public bse {
    // ...
};

class d : public bse {
    // ...
};

class e: public d, public c {
    // ...
};

void f(e *pe) {
    bse* pb=pe; // error - which bse ?
}

```

Ambiguous user-defined conversions in conditional expression

It is not possible for the compiler to resolve the ambiguity between two or more user-defined conversions in a conditional expression.

```

class b {
public:
    operator int ();
    operator int *();
};

int foo(int i, b &x1, b &x2)
{ return i ? x1 : x2; // error - convert x1 and x2
                      // to 'int' or 'int *' ?
}

```

Argument list not allowed in destructor declaration

A destructor may not have a return type or argument list.

Array index is out of range

This error is only given if the array index checking option is enabled, and an array is indexed by a constant expression that is larger than the number of array elements.

```
int a[10];
a[10] = 0; // error
```

The run-time system can check arrays indexed by variables.

Array object created by new cannot have an initializer

Objects allocated from the free store may not be initialized by this method.

```
int ia[] = new int [4] (1, 2, 3, 4); // error
```

Array of functions not allowed

The element type of an array may not be a function type, for example:

```
int af[5](); // Error, af is an array of functions

int (*apf[5])();// OK, array of pointers to function
```

Array or pointer expression expected

This error is issued if none of the operands for the array subscripting operator [] have pointer or array type. The [] operator requires a pointer operand and an integral operand. Note that an expression that has type array of T is converted to a pointer to T in this context.

Array size expression must be greater than zero

The constant expression specifying the number of array elements must have integral type and a value that is greater than zero.

Bad return type for operator

Class member access operator -> must return a pointer to a class, or a type for which an operator -> function is defined.

```
class c {
    int operator ->(); // must return pointer
};
```

Base initializer not allowed here, function is not a constructor

Only constructors may specify base or member initializers. For example:

```
class c {
    int i;
    void f() : i(1) {} // error - f not a constructor
};
```

Bit-field width is too big

A bit-field cannot be wider than the size of the underlying type. For TopSpeed C++, this means that maximum bit-field width is 32 (for a long bit-field), and any value above this is illegal.

Bit-field width must be positive

The constant expression in a bit-field declaration must have a value in the range 0 to 32.

'break' statement appears outside a 'switch'/'loop' body

You have a break statement outside of a switch, while, do-while or for statement.

Cannot access class member <name>

Private class members may only be accessed by member functions and friend classes or functions. Protected class members may also be accessed by members and friends of a class derived from this class. For example:

```
class c {
private:
    int a;
};

void f(c &sc) {
    sc.a=1; // error - a is a private member of c
}
```

Cannot access member without an object

Although a member may be visible to a static member function, it cannot be accessed since the function has no this pointer.

```
class c {
    int a;
    static void f() {
        a=1; // error - cannot access a without an object
    }
}
```

Cannot generate default function

The compiler cannot generate a default assignment operator or copy constructor, because it has been defined as private in a base class.

```

class B {
    B(B&); // copy constructor
public:
    int i;
};

class D : public B {
public:
    int y;
};

void test5()
{
    D d1;
    D d2 = d1; // error - cannot generate D(D&),
               // B(B&) is private
}

```

Cannot take address of a constructor

The address of a constructor cannot be taken.

Cannot take address of a destructor

The address of a destructor may not be taken.

Cannot take the address of a non-lvalue expression

The address of an expression that is not an lvalue cannot be taken , i.e., an expression that does not have a storage location. This is true both for explicit use of the & operator and for implicit array to pointer conversions.

'case' label appears outside a 'switch' statement

You can only have a case label within a switch statement.

'case' label redefined

The expressions for all the case labels within a switch statement must have unique values after conversion to the type of the switch expression.

Cast to abstract class type

It is illegal to cast to the type of an abstract class. Abstract classes may only be used as base classes.

Class <class name> is undefined

The class has not been defined yet, so may not be used in a qualified name. For example:

```

class x;
int x::f() // Error - x is undefined
{ }

```

Class defined in friend declaration

A class must not be defined in a friend declaration.

```
class c {  
    friend class f { int a; }; // error - class defined  
};
```

Class definition in argument list

A class may not be defined in an argument list.

```
void f(class c {int a;}, int); // error
```

Class in base class list is not defined

The class has not been defined yet, so may not be used as a base class.

```
class bse;  
class c : public bse { // not defined  
};
```

Class is not a direct base or member of this class

Constructor-initializers may only be specified for direct base classes and virtual base classes.

```
class B {  
    int i;  
    B(int);  
};  
  
class C : public B {  
    int i2;  
};  
  
class D : public C {  
    int i3;  
    D() : B(1) {} // B is not a direct base class  
};
```

Class member <member name> has incomplete type

You have declared a non-static class member for which the compiler cannot compute the size.

Class name expected

A typedef name that names a class is a class name and a synonym for that class. However, it may not be used after a class, struct or union prefix, nor in the names for constructors and destructors within the class declaration. For example:

```

class X;
typedef X Y;

class X {
    int i;
    ~Y();    // error - class name expected
};

```

Class name expected before ::

A name appearing before :: must be a class name.

Class-specifier for <name> is incompatible with previous declaration

You have used a name with a struct, union or class specifier that is incompatible from its declaration. The class-specifiers struct and class are compatible, but neither is compatible with union. For example:

```

struct Node {           // declares Node a struct
    class Node * next; // ok
};

union Node head;       // Error: union not compatible

```

Const initializer supplied for non-const reference

A non-const reference may not be initialized to refer to a const object. For example:

```

void foo{
    const int a = 0;
    int &ir = a; // error - a is const
}

```

Const member has no initializer

const class members must have initializers in each constructor-initializer.

```

class c {
    const int a;
    c() { } // error - no initializer for a
};

```

'const' or 'volatile' mismatch between object and function

A const object can only be passed to a function declared as taking a const this pointer.

'const' or 'volatile' on constructor

const or volatile storage specifiers on constructors are illegal.

'const' or 'volatile' on destructor

const or volatile storage specifiers on destructors are illegal.

'const' or 'volatile' on non member function

const or volatile on a function relates the this pointer. A non-member function does not have a this pointer.

Constant has no initializer

A constant may only be assigned to at the point of definition. Therefore it is an error not to do so.

```
const int i; // error - no initializer.
```

'continue' statement appears outside a loop body

You have a continue statement outside of a while, do-while or for statement.

Conversion operator is not a member function

A conversion operator must be a member function pointer, because it must use the this pointer.

Converted operand is not an lvalue

A conversion does not yield an lvalue, since it cannot return a value.

```
struct b {  
    operator int();  
};  
  
void foo(b &x)  
{  
    x *= 2; // error - operator int() not lvalue  
    x++;   // error - operator int() not lvalue  
}
```

Declaration of enum <name> is incomplete

You must define the enumeration members when you declare an enumeration name, for example:

```
enum model x; // Error: enum model not defined
```

Declaration of non-static class member <name>

For example:

```
class B1 {  
    int b1;  
    static int b2;  
};  
  
int B1::b1; // error - non static member  
           // cannot be declared outside class  
int B1::b2; // ok
```

Declaration syntax error

You have the type specifier part missing from a declaration. The only time you can leave out the type specifier is when you declare a function.

Default argument not allowed in function

Default arguments are not allowed in operator functions.

```
class c {
    c& operator +=(int=0); // error
};
```

Default argument redefined

Default function arguments may not be redefined, not even if the defaults specified are identical. This error will also occur if it is not possible to merge declarations of default arguments.

```
int f(int x=0);
int f(int x=4); // default argument redefined.
```

Default constructor for class needed in array initialization

If initializers are not supplied for every member of an array of objects, a default constructor is required. If a constructor has been defined for the class a default constructor will not be created by the compiler.

```
class c {
public:
    c(int);
};

c c_array[4]={ 1, 2, 3}; // error - no default
                        // ctor for c_array[3]
```

'default' label appears outside a 'switch' statement

You can only have a default label within a switch statement.

'default' label is already defined

You cannot have two or more default labels in a switch statement.

Destructor is not a member of class

The destructor defined was not declared in its class. While a default destructor will be created by the compiler if one is not declared for the class, it is illegal for the user to do this.

```
class c {
    c();
};

c::~~c() { // error - not member of class c
}
```

Destructor redeclared

A destructor may only be declared once for each class.

```
class c {
    ~c();
public:
    c();
    ~c(); // destructor redeclared
};
```

Division by zero

You have a division expression (‘/’ or ‘%’) where the right-hand operand is the constant zero.

Duplicated class <name> in base class list

A base class may be specified only once in a base class list, regardless of access specifiers and virtual base classes. This error can also occur if a class is simultaneously a direct and an indirect base class.

```
class c : virtual public bse, bse { // error
};
```

End of file in macro call

You must specify the macro name, all arguments, and the terminating) of a macro in the same source file.

Enum <name> is already defined

You must not define the same enumeration name more than once in the same scope. For example:

```
enum status { ok, read_error, write_error };
enum status f; // OK, refers to definition above
enum status { ready, busy };

// Error: the enum name status is defined twice
```

Expression has incomplete type

You have attempted to reference the value of an expression that has incomplete type. For example, you could have used the * or -> operators on an operand of type pointer to an undefined class:

```
struct Node; // forward declaration for Node
struct Node *np, n;

n = *np; // Error: *np has incomplete type
```

Expressions are not assignment compatible

The expression on the right-hand side of an assignment operator is not type compatible with the left-hand side operand. Or, a parameter is not type compatible with the formal argument type.

File not found <filename>

The file could not be found. Check the filename and then check the redirection file (TS.RED). The redirection file must contain the directories you wish the compiler to search for the file.

Filename too long <filename>

You can only have filename specifications of less than 65 characters.

First argument for operator delete() must have type 'void *'

Operator delete must take a first argument of type void *.

First argument for operator new() must have type 'size_t'

Operator new must take a first argument of type size_t.

Function <function name> is redeclared in class declaration

No member may be declared more than once in the member list. This error message may also occur if an attempt is made to overload a function with parameter types that are not sufficiently different. Member functions that differ only in that one is declared static may not have the same name.

```
class c{
    int func(int);
    static int func(int); // error
};
```

Function <function name> is already defined

You have defined the function more than once.

Function argument type is an abstract class

An abstract class is one that contains a pure virtual function and therefore can only be used as a base class.

```

class c {
public:
    c();
    virtual int f()=0;
};

int func(c pc) {    // error - argument is abstract
    return pc.f();
}

```

Function cannot be overloaded, it does not have C++ linkage

Function overloading is only possible using C++ linkage, which allows the encoding of argument profiles.

Function cannot be virtual (it has no 'this' pointer)

Only non-static member functions can be declared virtual.

Function does not return a value

A function that is not defined with return type void must return a value. An exception to this rule is made in the case of the function main.

Function has no match for target type

The compiler cannot distinguish between two or more overloaded functions when taking its address.

```

void f(int, char);
void f(int, signed char);

void (*pfii)(int, int) = f;
    // f(int, char) or f(int, signed char) ?

```

Function member(s) in anonymous union

An anonymous union may not have function members of any type.

```

static union {
    int a;
    int f();    // not allowed
};

```

Function or pointer expression expected

The left-hand operand of the function call operator () must have function or pointer to function type. Note that an expression that has type function returning T is converted to a pointer to function returning T in this context.

Function return type not allowed for destructor

No constructor, destructor or conversion operator may have a return type.

Function return type not allowed for constructor

A constructor may have arguments, but may not have a return type.

Function type result of '.' or '->*' may only be called

Pointers to member functions may only be called.

```
struct B1 {
    int foo();
};

int (B1::*pmf)() = &B1::foo;

void test3(B1 *pb)
{
    int (*p)();
    p = pb->*pmf; // error - pb->*pmf can only be used
                // in a call: (pb->*pmf)()
}
```

Global anonymous union is not static

A global anonymous union must be static.

```
union { // error - not static
    int a;
    char c[2];
};
```

Global operator delete() takes one argument

A declaration of the global operator delete may only have one argument. This must be of type void *.

Identifier expected in #define command

The first argument for the #define preprocessor command must be an identifier.

Identifier expected in #ifdef command

You have not supplied the identifier argument for an #ifdef preprocessor command.

Identifier expected in #undef command

You have not given an identifier as an argument to the #undef preprocessor command.

Illegal 'pointer to member' cast

A pointer to member may be cast to another pointer to member of the same class. If the two types are pointers to member functions of classes, one of which is unambiguously derived from the other, they may also be cast.

Illegal access declaration of member (not a member of a base class)

A member specified in an access declaration must be a member of a base class.

```
class nb {  
    int a;  
};  
  
class c {  
public:  
    nb::a; // error - nb is not a base class.  
};
```

Illegal character

You have a character in the source that is not part of the standard C character set, for example, \$.

Illegal copy constructor

The parameter for a copy constructor must be a reference to an object of the class.

```
class c {  
    c(c); // illegal argument (should be c(&c))  
};
```

Illegal expression in return statement

You have defined a void function that should return nothing but, instead, contains a return expression.

Illegal function return type

A function return type may not be an array or function type. For a function definition, the return type cannot be an incomplete class type (i.e., a class that has not yet been defined).

Illegal initializer for class member (only one expression is allowed)

The initializer of a class data member that has no constructor may only have one expression.

```
class B {
    int i;
    B() : i(1,2) { } // error - two values supplied
}
```

Illegal initializer for member

A member cannot be initialized with an invalid type.

```
int *p;
class B {
    int i;
    B() : i(p) { } // error - invalid type
};
```

Illegal linkage specification

Legal linkage specifiers are C++, C, Modula2, and Pascal. Modula2 and Pascal specifiers may be followed by a period and a module name: “Pascal.PASLIB”.

```
extern “pascal” int pfunc(); // error (wrong case)
```

Illegal operand types in compound assignment

You have conflicting types for the two operands in a compound assignment operator `op= expression`. If the left-hand operand of the `+=` and `-=` operators has a pointer to object type then the right-hand operand must be an `int`. All the other operators take operands of arithmetic type similar to their corresponding binary operator. For example:

```
int *p1, *p2;
p1 += p2; // Error: both operands have pointer type
```

Illegal operator declaration

The wrong type of arguments were supplied for an operator. The type of arguments must correspond to the type of the operator (unary, binary or assignment) and of the operand.

Illegal storage class or function specifier for member

Constructors may not be virtual or static. Destructors may not be static. Data members may not have any storage class specified except `static`.

```
class c {
public:
    static c(); // error - constructor can't be static
    inline int x; // error - inline not allowed here
};
```

Illegal storage class or function specifier for destructor

A destructor may be virtual but not static; it needs a `this` pointer.

Illegal type of pointer operand

You cannot use an expression of type pointer to operand of incomplete type for the * indirection, ++ increment, — decrement, or [] array subscription operators.

Illegal value for enumeration constant <name>

The value of an enumeration constant must be representable as an int value, i.e., it must be in the range -32767 to 32767. For example:

```
enum { a, b = 32767, c }; // Error: c not in range
```

Illegal variable type

You cannot declare a variable of type void.

In calling conventions: <message>

This error reports a number of problems because of incompatible calling conventions for:

- Function calls
- Assigning a pointer to a function expression, to a pointer variable.
- Passing a pointer to a function expression as an actual parameter.

This error will occur only if you have used the call pragma or the modifiers near, far, cdecl or pascal to alter the default function calling convention.

The following are descriptions of the suberrors reported because of function call incompatibilities:

Function in wrong segment for a near call

You have called a function with a near call, but the function is not located in the current code segment CS.

'ds_entry' attributes do not match

The same_ds pragma is on, but the data segment values specified by the ds_entry pragma do not match.

Function does not save the DS register

The lvalue has fixed DS, but the rvalue does not preserve the DS register.

The following are descriptions of the suberrors reported because of function assignments and function parameter incompatibilities. In these descriptions, lvalue refers to the left-hand operand of the assignment or the formal parameter, and rvalue refers to the right-hand operand of the assignment or the actual parameter.

Function in wrong segment for a near call

The rvalue is called with a near call, but the lvalue does not specify the same the same code segment.

'ds_entry' attributes do not match

The same_ds pragma is ON but the data segment values specified by the ds_entry pragma do not match.

Function does not save the DS register

The calling function has fixed DS, but the called function does not preserve the DS register.

'near'/'far' call attributes do not match

The rvalue and lvalue have different near/far call attributes.

'same_ds' attributes do not match

The rvalue and lvalue have different same_ds attributes.

C convention attributes do not match

You have an lvalue and rvalue with different c_conv attributes caused by either the c_conv pragma or the pascal or cdecl keywords.

'reg_param' attributes do not match

You have an lvalue and rvalue with different reg_param attributes caused by the reg_param pragma, or the pascal or cdecl keyword.

Function does not save enough registers

You have an lvalue that does not preserve all the registers required by the rvalue.

Incompatible pointer types (const, volatile qualifiers are different)

A pointer to const may not be assigned to a pointer to non-const. A pointer to volatile may not be assigned to a pointer to non-volatile.

```
const int *pci;
int *pi=pci;      // assigns pointer to const int
                  // to pointer to int
```

Incompatible type of initializer expression

You have given an initializer expression that is not compatible with the object type it is initializing.

Increasing access to base class member

An access specifier may not be used to relax access rules applying to a member accessible in a base class.

```
class a {
protected:
    int f();
};

class c : public a {
public:
    a::f; // error - increases access
};
```

Inherited conversion operator is ambiguous

The compiler cannot distinguish between conversion operators defined in base classes.

```
struct a {
    operator int();
};

struct b {
    operator int();
};

class c : public a, public b {
    void test() {
        if(*this) // which operator a::int() or b::int()
            return;
    }
}
```

Initialization of array member needs default constructor

If no constructor has been declared for a class the compiler will generate a default. However if any constructor with a parameter list differing from the default is declared, a default constructor will not be generated. Since the initialization of an array of objects requires this default constructor, an error will be generated in this case.

```
class c {
    c(int);
};

void f() {
    c *c_array = new c[10]; // no default constructor
}
```

Initialization of non 'const' reference needs a temporary

The use of temporary variables for the initialization of non-constant variables is illegal.

```
int &r=0;    // 0 is not an lvalue, so a
            // temporary is required
```

Initialization of reference to 'T' with a 'const T'

It is illegal to initialize a non-const reference to a type with a const object.

```
const int ci=1;
int &ri=ci; // error

void f(int &r);
void test()
{
    f(ci);           // error
}
```

Initialization too deeply nested

The number of subaggregates in an initializer list can only be nested up to a level of 32.

Initializer list not allowed here, class is not an aggregate

A class that contains a constructor, private or protected members, base classes or virtual functions is not an aggregate, and may not be initialized in this way. A constructor should be used instead.

```
class c {
public:
    int a;
    int b;
    virtual void f();
};

c sc={1, 2}; // error - c is not an aggregate
```

Initializer not allowed here

You cannot specify an initializer for the following types of declarations:

- Local variables declared with the extern storage class.
- Variables that have incomplete class type.
- Function declarations. However, an extension of TopSpeed C++ does allow you to specify inline machine code as **an initializer for functions**.

Initializer not compatible for member

The initializer of a class data member must be compatible with its type.

```
char * p;

class B {
public:
    int i;
    B() : i(p) // error - initializer not compatible
    { }
}
```

Integral constant expression expected

You have not supplied a constant expression of integral type for either:

- An array bound.
- The size of a bit-field member.
- The value of a case constant.
- The value of an enumeration **constant**.

Integral expression expected

You have not supplied an integer expression where the compiler expected one, for example in the test expression of a switch statement.

```
Internal <number>
```

Internal compiler error that should never occur. If you ever get such an error, please report it to JPI Technical Support.

Invalid argument in #line command

The arguments for a #line command must be a digit sequence, optionally followed by a string literal specifying a filename. For example:

```
#line 0x123 // Error, not a digit sequence
#line 123   // Ok
```

Invalid array declaration

You have declared an array element as void, or have forgotten an array bound in a multi-dimensional array. In a multi-dimensional array, you can leave out the first array bound, but you must specify the rest of the array bounds. For example:

```
void v[3]; // Error, element type is void
int a[][]; // Error, array bound missing
int a[][4]; // OK, first array bound can be omitted
           // and given in a later declaration
```

Invalid char constant

You have a character constant that contains no characters, the newline character or the backslash character \. Note the newline and backslash characters may be specified using an escape sequence.

Invalid filename specification

You have an invalid filename specification according to the C++ syntax.

Invalid hexadecimal constant

A hexadecimal constant consists of the base specification 0x or 0X, followed by one or more hexadecimal digits. It is an invalid constant if only the base is specified.

Invalid modifier list

A declaration can be modified by one or more modifiers. Some combinations of modifiers are not meaningful; in a declaration only one of the near, far and huge modifiers may be used in the same list; likewise, only one of pascal and cdecl may be used in the same list. For example:

```
int near far * p;      // Error: near and far
int pascal cdecl f(); // Error: pascal and cdecl
int near pascal f2(); // OK
int far * near * npf; // OK, near and far not in the
                     // same modifier list
```

Invalid storage class

You have declared either a parameter with a storage class other than register or a function or variable outside a function with a storage class other than static or extern.

Invalid string constant

You have a string constant that contains an illegal character. It is most likely that you have a double quote “, backslash \, or newline character in the string. Use an escape sequence to specify these characters.

Invalid token

An invalid token has been found in the source. An example of an invalid token is 1E+x.

Invalid type in bit-field declaration

The type of a bit-field member must be integral.

Invalid type specification

You have a declaration with an invalid combination of type specifiers, such as long short int, or long char. This error often occurs when the semicolon is omitted at the end of a class declaration.

Invalid type specifier in conversion operator

In the declaration of a conversion operator function, an invalid type specifier was given. For example:

```
class f {  
    operator class p { } (); // error  
}
```

Left operand of '->' must be a pointer to a class

The left-hand operand of the member selection operator -> must have type pointer to a class.

Left operand of '.' must be a class

The left operand of the structure member selection operator (.) must be of class type.

Left operand of assignment must be a modifiable lvalue

The left-hand operand of an assignment operator must be a modifiable lvalue. A modifiable lvalue is an expression that does not have an array type, an incomplete type or a const qualifier (including a const member if it is a class type).

```
int func()  
{  
    int const ic = 3;  
    ic = 5;    // Error: ic has a const qualified type  
              // and can only be given a value by  
              // initialization  
}
```

Linkage specification redeclared for function

For example:

```
extern "C" cfunc();  
extern "Pascal" cfunc(); // error
```

Macro definition too big

You have a macro definition replacement larger than 2048 characters.

Macro expansion too big

You cannot have a macro expansion that results in more than 2048 characters.

Macro expansion too deeply nested

You have a macro expansion that has too many nested macro invocations.

Member function has been called before declared inline:

inline functions must be defined before being called. For example:

```
class c {
    int i;
public:
    int ifn();
    c() { i=ifn(); } // ifn declared but not defined
};

inline int ifn() // error - called before defined
{ return 0;
};
```

Member function must be defined within local class

Member functions of local classes must be defined within that class (i.e., the function body must be specified).

```
void func() {
    class cc {
        int f(); // error
        int f2() { return 0; }; // Ok
    };
}
```

Member initializer for enumeration constant <name>

An enumeration constant may not be initialized by this method.

```
class B {
    enum color { red, green, blue };
    int i;
    B() : i(1), red(1) // error: attempt to initialize
                    // an enumeration constant
    { }
};
```

Member initializer for member function <name>

Member functions may not be initialized.

```
class B {
    int f();
    B() : f(0) // error - initializing member function
    { }
};
```

Member initializer for member of array type

Member arrays cannot be initialized by this method.

```
class B {
    int a[3];
    B() : a(1,2,3) // error
    { }
};
```

Member initializer for static member

A static class member must be initialized outside of its class declaration.

```
class B {
    static int i;
    B() : i(1)      // error
    { }
};
```

Member needs initialization

No default constructor exists for class m and no constructor-initializer has been specified in the constructor of c.

```
struct m {
    int data;
    m(int);
};

class c {
    c();
    m i;
};

c::c() { // member i has not been initialized.
}
```

Mismatched '#pragma restore' command

A #pragma restore preprocessor command must have a matching #pragma save command.

Missing #endif

You have one or more #if, #ifdef or #ifndef preprocessor commands without a matching #endif. It is not allowed to split an #if, #endif pair across source files.

Missing comment terminator

One of your comments is missing a terminator */. A comment must be terminated in the same source file.

More than one storage class specifier in declaration

You can only have one storage class specifier, such as extern, in a declaration. For example:

```
int extern register i; // Error
```

'near', 'far' or 'huge' modifier on local variable <variable name>

You cannot use the near, far or huge modifiers when declaring local variables, since the compiler uses the stack for allocation.

'new' on abstract class

An abstract class is one that contains a pure virtual function and therefore can only be used as a base class.

```
class c {
public:
    c();
    virtual int f()=0;
};

void func() {
    c *pc=new c; // error - c is abstract
}
```

No expression in return statement

You have a return statement in a non-void function that does not return a value.

No matching constructor for call

No constructor has been declared that matches that required by an initializer.

```
class c {
public:
    c();
};

c sc(1); // error
```

No matching function for call of <name>

An overloaded function has been called, but none of the declared functions has an argument profile that matches that supplied. In the example below, no conversion exists from int to class X, so no matching member is found for the argument profile (int, int).

```
class X;

class c{
public:
    int f(X, int);
    int f(int);
};

void f() {
    c sc;
    sc.f(1, 1); // no conversion exists from int to X
}
```

No return value in function <function name>

You have a non-void function without a return statement. To make this error vanish, insert a return statement with a suitable value at the end of the function.

No terminating ‘}’ for inline member function

All inline functions must be terminated by a ‘}’ character.

```
class c {
public:
    int f() { return a; // error - no terminating }
    inline int g();
private:
    int a;
};
```

No type specifier for member

A class member must have a type specifier.

```
class c {
public:
    int a;
    b; // error - no type specifier
};
```

Non-class specified as base class

The class specified has not been defined, or is not a class type.

```
int bse;
class c : public bse { // bse not a class
};
```

Object too big

In TopSpeed C++, a single data object cannot occupy more than 64K of storage. This restriction is due to the segmented architecture of the 80x86 family of processor used in the IBM PC (and compatibles), where each segment has a maximum size of 64K. However, you can use the library function `halloc` to allocate objects that do not have this limit. The allocated object can be assigned to a huge pointer, as in this example:

```
int a[50000]; // Error: size of a > 64K
int huge *a;
a = halloc(50000); // OK
```

Operand cannot have type pointer to function

You cannot use the * indirection, ++ increment, — decrement, or [] array subscription operators in an expression containing an operand of type pointer to function.

Operand cannot have type pointer to void

You cannot use an expression of type pointer to void as an operand for the * indirection, ++ increment, — decrement, or [] array subscription operators.

Operand must have scalar type

You have an expression using one of the &&, ||, !, ++, — or ? operators that contains an operand of neither arithmetic nor pointer type.

Operand of '*' must have pointer type

The operand of the indirection operator * must have pointer type.

Operand of '++' or '—' must be a modifiable lvalue

You have the operand of the ++ and — operators containing either an array type, an incomplete type, or a const qualifier (including a const member of a class).

Operand of '~' must have integral type

The operand of the unary bit-wise complement operator ~ must have integral type.

Operand of 'sizeof' has illegal type

You cannot have a function type or incomplete type as the argument for the sizeof operator.

Operand of unary '+' or '-' must have arithmetic type

You have an operand that is neither integer nor floating point when using the unary + or - operator.

Operands of <operand name> have illegal pointer types

You have a binary, relational, or equality operator with illegal pointer types.

Operands of <operand name> have illegal types

You have supplied an illegal type for the conditional (:), the binary addition, subtraction, relational or equality operators.

Operands of <operand name> must have arithmetic types

You must supply arithmetic operands for the binary multiplication * and division / operators.

Operands of <operand name> must have integral types

You must supply integer operands to the binary remainder % and the shift operators << and >>.

Operator <name> must have an argument of class type (or reference to class)

The operator must have an argument of the class type on which it operates.

```
class c {
    friend c& operator-(int); // argument should be a c
};
```

Operator cannot be a static member function

Member operators that modify the object require a this pointer. A static member does not have a this pointer.

```
class c {
    static c& operator+=(c&); // error
};
```

Operator must be a member function

Certain operator function must be a non-static member functions.

```
class c {
    friend c& operator=(c&); // error: must be member
};
```

Overloaded function used as operand

It is illegal to use the address of an overloaded function in an expression.

```
int f();
int f(int);
int f(int, int);

void * pv = &f; // error
```

Overloaded operator delete() takes one or two arguments

Operator delete must take a first argument of type void *. An optional second parameter of type size_t is permitted. It may not be overloaded within a scope.

Overloading cannot distinguish argument types 'T' and 'T&'

Functions with argument types differing only in this respect may not have the same name.

```
void func(int&);
void func(int); // can't distinguish
```

Overloading cannot distinguish argument types that only differ on 'const' and/or 'volatile'

Functions with argument types differing only in this respect may not have the same name.

```
void func(const int);
void func(int); // can't distinguish
```

Overriding virtual function has different return type

There must be an exact match of both return type and arguments between virtual functions in base classes and overriding functions in derived classes.

```
class vb {
protected:
    virtual int f();
};

class c : virtual public vb {
public:
    virtual void f(); // error: return type different
};
```

Parameter <parameter name> is already defined in function

You have declared a function parameter more than once in the same function parameter list. For example:

```
inf f(int c, char *s, char *d, int c);
/* Error: c is declared twice */
```

Parameter has incomplete type

You cannot have a parameter whose type is a class type of unknown size.

Parameter has type 'void'

You cannot have a parameter of type void.

Parameter is not compatible with formal type

Actual parameters must match the type of formal parameters.

```
class B;
int f(B);

void foo()
{
    f(1); // error - actual parameter not compatible
}
```

Pointer conversion from base class to derived (base class is virtual)

It is illegal to convert a pointer from that of a virtual base class to that of a derived class.

```

class B1 {
    int b1;
};

class D1 : public virtual B1 {
    int d1;
};

void test()
{
    D1 * pd1;
    B1 * pb1;
    pd1 = (D1*) pb1; // cannot convert this way
}

```

Pointer conversion to private base class

This pointer conversion from derived class to private base class may only be performed by a friend or member function.

```

class vb {
    // ...
};

class c : private vb {
    // ...
};

void func(c *pc) {
    vb *pvb=pc; // error
}

```

Predefined macro/name cannot be redefined

The predefined macros and the identifier defined cannot be redefined by the `#define` and `#undef` preprocessor commands. The predefined macro names are: `__LINE__`, `__FILE__`, `__DATE__`, `__TIME__`, `__STDC__` and `__cplusplus`.

Private/protected member(s) in anonymous union

An anonymous union may only have public members.

'protected' specifier is not allowed in base class list

Access specifiers for base classes are private or public.

Pure specifier not allowed here, it is not a virtual function

An abstract class may be created by making declaring one or more virtual functions as pure, i.e. no definition is supplied. Non virtual functions, since they are not called via the virtual function table, cannot be declared pure.

```
class c {
public:
    c();
    int f()=0; // not a virtual function
};
```

Qualified function is not a member function

For example:

```
class c {
    int f();
    int g;
};

void c::g() { // not a member function.
}
```

Qualified name in function declaration

For example:

```
class B1 {
    int foo();
};

int B1::foo(); // error - member function cannot be
               // declared outside class
int B1::foo() // ok - member function is defined
{
    // outside class
    return 0;
}
```

Qualified name is not allowed here

Qualified names are not allowed in typedef or parameter declarations. For example:

```
class X {
public:
    int i;
};

typedef int X::i; // qualified name not allowed here
int f1(int X::i); // qualified name not allowed here
```

Qualifier in member declaration

An access specifier may not include the type of the member. For example:

```

class B {
public:
    int i;
};

class D : private B {
public:
    int i2;
    int X::i; // error - qualifier in member declaration
    X::i;     // ok - adjust access to X::i
};

```

Redeclaration of class <name>

You have declared the class-name more than once in the same scope.

Redeclaration of class member <member name>

You have declared a member more than once in the same class.

Redeclaration of identifier <identifier name>

You have declared an enumeration constant in the same scope and with the same name as a variable, an enumeration constant, or a function.

```

int waiting;
enum state { ok, error, waiting };
// Error: waiting is already redefined

```

Redeclaration of type <identifier name>

You have declared the identifier as a typedef name more than once in the same scope.

Redefinition of label <label name>

You have defined a label more than once in the same function.

Redefinition of macro <macro name>

You have redefined a macro, and the new definition is not identical to the previous definition. For two macros to be identical, the tokens in both definitions must be identical (a white space token counts as one space character), both must have the same number of arguments, and argument identifiers must be identical.

```

#define max(x,y)  x >= y ? x : y
#define max(x,y)  x >= y ? /*comment*/ x : y
// the above redefinition is ok, white space,
// including comments is not significant

#define max      x >= y ? x : y
// illegal redefinition, wrong number of arguments

```

Reducing access to base class member

An access specifier may not be used to restrict access to a member accessible in a base class.

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

class c : public a {
protected:
    a::f;    // error - reduces access
};
```

Reference member has no initializer

Reference class members must have initializers specified for on every constructor.

```
class c {
    int &a;
    c() {} // error - no initializer for a
};
```

Reference member needs a temporary in initialization

For example:

```
class B {
    int& f;
    c() : f(1) {} // a initialized to temp
};
```

Reference needs initializer

A variable of reference type must have an initializer.

```
int& r; // error - reference needs an initializer
```

Reference not allowed here

It is illegal to declare an array of references, a pointer to a reference or a reference to a reference.

```
int& ra[10]; // error - array of references
int& *rp;    // error - pointer to reference
int& &rr;    // error - reference to reference
```

Reference to void

A reference to void is not allowed

Return statement in constructor cannot specify a value

A constructor may not have a return type. Therefore it cannot specify a return value.

```
class c {
    c() {
        return 0; // return statement not allowed
    }
}
```

Return statement in function must return a value

A function that is not defined with return type void must return a value. An exception to this rule is made in the case of the function main.

```
int func() {
    return; // error - value required
}
```

Return type for operator delete() must be 'void'

Operator delete must return type void.

```
class c {
public:
    c();
    char *operator delete(void *); // error
};
```

Return type for operator new() must be 'void *'

Operator new must return void *.

```
class c {
public:
    char *operator new(unsigned); // error
};
```

Return type of function is an abstract class

An abstract class is one that contains a pure virtual function and therefore can only be used as a base class.

```
class c {
public:
    c();
    virtual int f()=0;
};

c func() {
    c ac;
    return ac; // returns abstract class
}
```

Return type specified for conversion operator function

A user conversion operator may not specify a return value.

Return value is not compatible with return type

A function must return a type compatible with its declaration.

```
class c;
c& func() {
    return 0; // not a c
}
```

Returning a reference to a local variable

It is illegal to return a reference to a local variable.

```
class c {
    int a;
};

c& func() {
    c ac;
    return ac; // return reference to local
}
```

Scalar expression expected

The test expression in an if, while, do-while or for statement must have integral, floating point or pointer type.

Scope table full

Reports that the compiler's identifier scope table is full. To work around this problem, you can split the source file into two or more source files.

Scope too deeply nested

Block scopes can only be nested up to a level of 32.

Second argument for operator delete() must have type 'size_t'

The optional second parameter to operator delete must be of type size_t, if present.

Second operand does not have type 'Pointer to member'

The second operand of a pointer to member operator must have the type of a pointer to a member of that class.

```

struct c {
    int f();
};

void func() {
    c sc;
    int i;
    int (c::*pf)();
    pf = &c::f;

    i=(sc.*f)(); // error - f not pointer to member
    i=(sc.*pf)(); // ok
}

```

Static data member is not allowed in local class

Static data members are not allowed in classes local to a function, as there is no way to declare them outside the class.

```

void func() {
    class cc {
        static int a; // not allowed
    };
}

```

Static function <function name> has not been defined

You have declared a static function and called the function in the translation unit, but you have not defined the function body. The compiler requires the definition because it cannot resolve the call by external linkage.

Static member has illegal declaration specifier list

For example:

```

class c {
    inline static f(); // error
};

```

Storage class specifier not allowed in member declaration

The member declarations of a structure or union may not contain any storage class specifiers other than static.

```

struct foo
{
    int extern i; // Error: extern not allowed
}

```

Syntax error in #pragma command

You have made a mistake when defining a TopSpeed C++ pragma. See the “Compiler Options and Pragas” section of the *Developer’s Guide* for more information.

Syntax error, expected: <symbols>

You have some source code that violates the C++ syntax. The compiler reports the symbols possible at the position of the error.

'this' appears outside a non-static member function

Only non-static member functions have a this pointer.

Token too long

You have a token, such as a string literal, of more than 1024 characters. The compiler concatenates adjacent string literals so that this error should not occur in practice.

Too few parameters in function call

You have fewer parameters in the function call than formal arguments in the function declaration.

Too many arguments in macro <macro name>

Macro definitions and macro invocations can have up to 32 arguments.

Too many arguments in macro call <macro name>

Macro definitions and macro invocations can have up to 32 arguments.

Too many errors, compilation aborted

The source file contains too many errors, the compilation is aborted.

Too many identifiers in source file

Reports that the compiler's identifier name table is full. To work around this problem, you can split the source file into two or more source files.

Too many initializers

You cannot have more initializers for an array or structure than there are members in the array or structure (this is also true for any subaggregates). For example:

```
int a[3] = { 1, 2, 3, 4};  
// Error: more initializers than array elements
```

Too many parameters in function call

You have more parameters in the function call than formal arguments in the function declaration.

Trailing argument without initializer

No arguments without default initializers may follow an argument supplied with a default initializer.

```
int f(int x=0, int y); // y must have default value
```

Type is incompatible with previous declaration of <variable name>

You have redeclared a variable with a type that is not compatible with the previous declaration.

```
extern int i;
extern short i;    // Error: different types
```

Type of class is an abstract class

An abstract class is one that contains a pure virtual function and therefore can only be used as a base class.

```
class c {
    virtual int f()=0;
};

void func() {
    c ac; // error - abstract class
}
```

Type of first operand does not match type of second operand

The type of the class of the left operand in .* or ->* must match that of the right.

Undeclared class name

The referenced class name has not been declared.

Undeclared identifier <identifier name>

You have an undeclared identifier in an expression.

Undeclared member name <member name>

The identifier used as the right operand for one of the member selection operators . or -> is not a member of the class specified by the left operand. For example:

```
int func()
{
    struct { int i1; int i2 } s;
    int i;
    i = s.i3; // Error, i3 is not a member of s
}
```

Undefined label <label name>

You have not defined the label for a goto statement. For example:

```
int func()
{
    // code
    if (i) goto out;
    // code
} // Error, the label out is never defined
```

Union has base classes

A union may not have any base classes or be a base class. It may however be a member of a class.

Union member has illegal class type

A union may not have as a member any object of a class type that has a constructor or destructor.

Union member is static:

A union cannot have static members

Unknown preprocessor command

You have a source line that starts with a # but is not followed by a valid preprocessor command.

User-defined conversion is ambiguous

It not possible for the compiler to decide which conversion to use. Argument matching cannot distinguish between the conversion operators since both require standard conversions of equal merit, for example from int to char and from int to signed char.

```
class c{
public:
    operator char();
    operator signed char();
};

void f() {
    int i=1;
    c sc1;
    i = i*c; // convert to char or signed char ?
}
```

Value must be in the range '0' to '0xFF'

You have an initialization value for an inline machine code function larger or smaller than a byte.

Variable <variable name> has incomplete type

You have a variable with a type of zero size. An external array declaration could be incomplete if you later, in the same translation unit, complete the declaration. For example:

```
int func()
{
    int a[]; // Error: local array is incomplete
    extern int b[]; // OK, extern incomplete array
}
```

Virtual class has ambiguous virtual function

The compiler cannot distinguish between two or more virtual functions in virtual base classes.

Virtual function declared in a union

A union may not have virtual member functions. It may only have non-virtual or static members functions (including constructors and destructors).

Wrong number of arguments for operator

The wrong number of arguments were supplied for an operator. The number of arguments must correspond to the type of the operator (unary, binary or assignment).

```
class c {
    operator+=(c&, c&); // error: takes one argument.
};
```

Wrong number of arguments in macro call

It is an error if a function-like macro is called with the wrong number of arguments. For example:

```
#define max(x,y) ((x>=y)?(x):(y))
i = max(2); // illegal macro invocation
i = max;    // legal (max is not expanded)
```

INDEX

Symbols

operator 163
 ## operator 164
 #define 159
 #elif 165
 #else 164, 165
 #endif 165
 #if 164, 165
 #ifdef 165
 #ifndef 165
 #include 12, 157, 166
 nesting 167
 #line 168
 #null directive 169
 #pragma 169
 #undef 161
 /* */ comment 25
 // comment 25
 <float.h> 18
 <stdarg.h> 52
 <stddef.h> 78, 181

A

abort() 21
 abstract classes 126
 abstract declarator 57
 access control 127, 130, 148
 base class 128
 member 127
 multiple access 132
 virtual function 132
 access control:constructors 134
 access control:destructor 135
 access declaration 130
 access rules 173
 access specifier 109, 127
 addition operator 86
 address-of operator 53, 75, 76, 116
 aggregate 60
 allocation expression 75, 79
 ambiguity

 allocation-expression 80
 conversion 140, 141
 declaration 38, 65
 expression or declaration 107
 initialization 65
 member access 122, 173
 pointer conversion 120
 anonymous union 115
 argc 20, 178
 argument 71
 argument matching 147
 argv 20, 178
 arithmetic constant expression 95
 arithmetic conversion 34
 arithmetic promotion 34
 arithmetic types 16
 array
 default size 52
 extern 51
 function argument 58
 incomplete initializer 60
 incomplete type 52
 initialization 52
 multi-dimensional 51
 multidimensional 70
 order of storage 71
 pointer to 86
 array declarator 51
 array subscript 70
 array subscript operator 70
 array type 78
 sizeof 78
 array types 18
 arrow operator 70, 73, 153
 asm 22
 asm declaration 44, 182
 assignment
 compound 93
 simple 92, 93
 assignment expression 92
 assignment operator 92, 133, 144, 152
 user-defined 115
 atexit() 20
 auto 22, 59, 106, 107

B

base class 119
 access control 128
 direct 119, 120
 indirect 119, 120
 virtual 120, 121, 122, 132

- base class:initialization 143
- base class:virtual base class 137
- Base Document 172, 175
- Base document 10
- Base Language 44, 49
- Base language 10
- base-spec 109
- based pointer 49, 187
- binary operator 155
- bit-field 79, 116, 182
- bitwise AND operator 89
- bitwise complement operator 75, 77
- bitwise exclusive OR operator 89
- bitwise inclusive OR operator 90
- bitwise shift operator 86
- block 98
- block scope 13
- break 22
- break statement 100

C

- C
 - ANSI C 10, 38, 39, 43, 108, 172
 - Difference from C++ 117
 - difference from C++ 59, 60, 91
- C linkage 44
- C++ linkage 44
- case 23
- case label 97
- cast expression 57
- cast operator 82
- catch 23
- cdecl 23, 48, 49, 50, 55
- char 23, 41
- character constant 28, 166
- character string 30
- character type 16, 78
 - sizeof 78
- class 23, 42, 108, 127
 - abstract 126
 - base 119
 - derived 108, 119, 121, 132
 - friend class 131
 - local 113, 118
 - member list 109
 - member selection 73
 - nested 113, 117
- class declaration 108, 127
- class name 13
- class scope 14
- class type 78
 - sizeof 78
- class types 18
- class-name 40, 42, 108, 118
 - scope 109
- comma operator 68, 71, 94, 95
- comment 25
 - nested comment 25
- comment:nested comment 187
- commutative operator 66
- compatible type 91
- compound assignment 93
- compound statement 98
- conditional compilation 164
- conditional expression 91
- conditional operator 152
- const 23, 41, 47
 - member function 111, 113
- constant 25
 - character 28, 166
 - floating 26
 - integral 78
 - null pointer 77, 83
- constant expression 42, 48, 95, 116, 165
- constructor 40, 42, 80, 83, 110, 111, 115, 133, 138
 - calling virtual function 138
 - default 80
 - order of execution 20
- constructor:copy constructor 134, 138, 142
- constructor:default constructor 134, 141
- constructor:order of execution 136
- continue 23
- continue statement 99
- controlling expression 104
- conversion 32
 - ambiguity 140
 - ambiguous 123, 141
 - argument matching 148
 - arithmetic 34
 - array-pointer 36
 - by assignment 92
 - by constructor 140
 - double-float 34
 - enumeration 43
 - explicit 32, 73, 82
 - float-double 34
 - floating-integer 34
 - function notation 73
 - function-pointer 36
 - implicit 32, 35, 36, 43, 130
 - integer-enumeration 43
 - integer-floating 34
 - integer-unsigned 33

- integral promotions 33
- null pointer 84
- pointer 35
- pointer to base class 120, 130
- pointer to class 83
- pointer to const 83, 84
- pointer to function 83
- pointer-integer 82
- pointer-to-member 36, 83
- quiet 82
- reference 36, 83
- standard 82
- trivial 149
- unsigned-integer 33
- user-defined 140, 150
- usual arithmetic conversions 88, 166
- conversion function 140
- conversion operator 83
- copy constructor 134, 138, 142
- ctor-initializer 142, 143, 174

D

- data-hiding 127
- deallocation expression 75, 81
- decl-specifiers 38
- declaration 13, 37, 41
 - access declaration 130
 - ambiguity 38, 65
 - class declaration 108, 127
 - class name declaration 13
 - function declaration 57
 - matching 147
 - storage-class declaration 39
 - typedef declaration 13, 40
- declaration statement 106
- declarator 48, 182
 - abstract 57
 - array 51
 - function 52
 - pointer-to-member 50
- declarator-list 38, 47
- decrement operator 70, 75, 154
- default 23
- default argument promotion 72
- default arguments 53, 134, 148
- default constructor 80, 134, 141
- default data segment 55
- default destructor 135
- default label 97
- definition 13, 38
 - function definition 58

- delete 23
- delete operator 81, 133, 144
- derived class 108, 119, 121, 127, 128, 132
- derived types 18
- destructor 40, 42, 81, 111, 115, 133, 135
 - access control 135
 - calling virtual function 138
 - default destructor 135
 - explicit call 137
 - order of execution 20
 - virtual destructor 136
- direct base class 120
- division operator 85
- do 23
- do statement 105
- domination 123
- dot operator 70, 73, 152
- double 18, 22, 26, 41, 78
 - sizeof 78

E

- elaborated type specifier 40, 42, 108, 131
- ellipsis 52, 148, 150
- else 22
- empty statement 98
- enum 22, 40, 42
- enum specifier 43
- enum-name 42, 43
- enumeration types 18, 182
- enumerator 43
- env 20, 178, 187
- equality operator 88
- escape sequence 29, 31, 157, 166
- exit() 20
- explicit conversion 32
 - cast 82
- expression 66
 - allocation 79
 - arithmetic constant 95
 - assignment 92
 - conditional 91
 - constant 48, 95, 165
 - controlling 104
 - deallocation 81
 - full 67
 - order of evaluation 66
 - parenthesized 69
 - primary 69
 - unary 75
 - void 69, 91
- expression statement 98

extension

- additional keywords 187
- based pointers 49, 187
- cast of lvalue 82, 187
- env 187
- machine code functions 187
- machine-code functions 44, 64
- modifiers 49, 50, 55, 187
- relative pointers 49, 187

extern 23

extern array 51

extern storage 51

external linkage 15

F

far 23, 48, 49, 50, 55

file scope 14

float 18, 23, 41, 78

sizeof 78

floating constant 26

floating point types 18

for 23

for statement 105

friend 23, 127, 129, 131

friend declaration 111

friend specifier 38

full expression 67

function

- constructor 133
- conversion function 133, 140
- default arguments 53
- destructor 135
- friend 131
- inline 131
- inline member function 111, 112
- member function 52, 110, 133
- name argument 58
- operator = 133
- operator new 133
- overloaded 130, 131
- overloading 53
- pointer to 59
- pure virtual 126
- rewriting 131
- virtual 110, 124, 132

function argument 71

function call 70, 71

function call operator 53, 152

function declarations 57

function declarator 52

function definition 58

function parameter 51

function scope 14

function specifier 40

function types 18

fundamental types 16

G

glue operator 164

goto 23

goto statement 99

greater than operator 87

greater than or equal operator 87

H

huge 23, 48, 49, 55, 56

I

identifier 22, 23, 179

if 23

if-else statement 102

implicit conversion 32

increment operator 70, 75, 154

indirect base class 120

indirection operator 75, 76

inequality operator 88

inheritance 119

initialization 38, 41, 59

aggregate 60

array 80

base classes 143

character array 60

functions 64

members 143

multidimensional arrays 61

references 63

scalar 59

static 59

static members 114

strings 63

union 60

inkage

Pascal 56

inline 23, 39, 40

inline friend function 131

inline member function 111, 112, 131

rewriting 112

int 17, 23, 41, 78

sizeof 78

integer constant 78

integer types 17

- integral promotions 33, 35, 149
- integral types 16
- internal linkage 15
- interrupt 23, 48, 49, 50, 55, 57
- iteration statement 104

J

- jump
 - unconditional 97
- jump statement 99, 106

K

- keyword 22, 24

L

- label 97
 - case 97
- labeled statement 97
- large memory model 55
- left shift operator 86
- less than operator 87
- less than or equal operator 87
- linkage 37
 - "C" 44
 - external 15, 113
 - internal 15
 - "Modula2" 45
 - "Pascal" 45
- linkage specification 15, 44
- literal 22, 25
- local class 113, 118
- local type-names 118
- logical AND operator 90
- logical negation operator 75, 78
- logical OR operator 90
- long 23, 41
- long double 18, 26, 78
 - sizeof 78
- long int 17, 78
 - sizeof 78
- loop body 104
- lvalue 19, 59, 67, 82, 94
 - modifiable 19, 75, 92

M

- macro
 - #define 159
 - #undef 161
 - argument substitution 162

- definition scope 160
- function-like 159, 161
- invocation 157
- object-like 159, 161
- preprocessor 158
- redefining 160
- rescanning and replacement 162
- macro name 24, 159
- macro parameter 24
- macro parameters 159
- main() 20, 136, 178
 - linkage of 179
- member
 - access control 127, 130
 - initialization 143
 - non-static 110
 - static 13, 110, 113, 129
- member access operator 113
- member function 52, 110, 133
 - const 111, 113
 - inline 111, 112
 - pure virtual 126
 - static 112
 - virtual 115, 124
 - volatile 111, 113
- member function:constructor 133
- member function:conversion function 140
- member function:destructor 135
- member list 109
- memory model 56
 - large 55
 - small 55
- modifiable lvalue 19, 75, 92
- Modula2
 - linkage 45
- modulus operator 85
- multidimensional array 70
- multiple inheritance 120
- multiplication operator 85

N

- name encoding 175
- name space 38
- name-encoding 45
- names 12
- near 23, 48, 49, 50, 55
- nested class 113, 117
- nested comments 25, 187
- new 22
- new operator 57, 79, 133, 144
- new-initializer 80

null pointer 35, 77
 null statement 98

O

operator 22, 24
 addition 86
 address-of 75, 76, 116
 array subscript 70
 arrow 70, 73, 153
 assignment 92, 144, 152
 binary 155
 bitwise AND 89
 bitwise complement 75, 77
 bitwise exclusive OR 89
 bitwise inclusive OR 90
 bitwise shift 86
 cast 82
 comma 94
 commutative 66
 conditional 152
 conversion 83
 decrement 70, 75, 154
 delete 81
 division 85
 dot 70, 73, 152
 equality 88
 function call 152
 glue 164
 greater than 87
 greater than or equal 87
 increment 70, 75, 154
 indirection 75, 76
 inequality 88
 left shift 86
 less than 87
 less than or equal 87
 logical AND 90
 logical negation 75
 logical OR 90
 member access 113
 modulus 85
 multiplication 85
 new 57, 79
 pointer-to-member 84
 pointer-to-member selection 152, 153
 post-decrement 74, 154
 post-increment 74, 154
 postfix 74
 pre-decrement 75, 154
 pre-increment 75, 154
 prefix 75
 relational 87
 right shift 87
 scope resolution
 113, 117, 120, 121, 122, 125, 130, 152
 sizeof 57, 75, 78, 95, 165
 stringize 163
 subscript 153
 subtraction 86
 unary 155
 unary minus 75, 77
 unary plus 75, 77
 operator delete () 136
 operator delete() 144
 operator function:operator delete() 133, 136
 operator function:operator new() 133, 137
 operator function:operator= () 133
 operator new() 137, 144
 global 145
 operator precedence 68
 optimizations 42
 overloaded function 130, 131
 overloaded function:address resolution 151
 overloaded function:constructor 142
 overloaded operator 152
 operator delete() 81
 operator new() 80
 overloaded operator:operator delete() 144
 overloaded operator:operator new() 144
 overloaded operator:operator— () 154
 overloaded operator:operator() () 152
 overloaded operator:operator++ () 154
 overloaded operator:operator-> () 153
 overloaded operator:operator->* () 153
 overloaded operator:operator= () 144, 152
 overloaded operator:operator[] () 153
 overloading 53, 109, 146

P

parameter
 function 51
 layout 72
 order of evaluation 72
 storage duration 59
 parenthesized expression 69
 Pascal
 linkage 45
 pascal 23, 48, 49, 50, 55, 56, 57
 phases of translation 156
 placement 80
 pointer
 based 49

- conversion 35, 83
- far pointer 55, 56
- huge pointer 56
- near pointer 55, 56
- null 35
- relative 49
- this 112
- pointer arithmetic 86
- pointer declarations 56
- pointer declarators 48
- pointer types 19
- Pointer-to-Member 50
- pointer-to-member 19, 84
 - conversion 36, 83
- pointer-to-member:selection operator 152, 153
- post-decrement operator 154
- post-increment operator 154
- postfix operator 74
- pragma 49, 50, 64
 - #pragma directive 169
 - call(inline) 64
 - option(ansi) 23, 49, 50, 55, 184, 187
 - option(lang_ext) 82, 187
 - option(nest_cmt) 187
 - option(uns_char) 16, 30, 33, 41, 180
 - prefix 45, 175
- pre-decrement operator 75, 154
- pre-increment operator 75, 154
- precedence 66, 68
- prefix operator 75
- preprocessing directives 158
- primary expression 69
- private 22, 127, 129
- program termination 136
- promotion
 - arithmetic 34
 - default argument 72
 - integral 35
- protected 23, 127
- public 23, 127, 128
- punctuator 22, 24
- pure-specifier 110

Q

- qualified name 130
- qualified-type-name 41
- quiet conversion 82

R

- read-only memory 42

- reference
 - conversion 36, 83
 - initialization 63
 - to temporary 63, 138, 150
- reference declarators 49
- reference types 19, 49
- register 23, 106, 181
- relational operator 87
- relative pointer 49, 187
- return 23
- return statement 101
- right shift operator 87
- row-major order 71

S

- scalar initialization 59
- scope 13
 - block 13
 - class 14
 - file 14
 - function 14
 - function prototype 14
- scope resolution operator
 - 113, 117, 120, 121, 122, 125, 130, 152
- scope rules 173
- selection statement 101
- sequence point 68, 74, 91, 94
- short 17, 23, 41, 78
 - sizeof 78
- signed 23, 41
- signed char 17
- simple assignment 92, 93
- simple-type-name 41
- size_t 78, 144, 181
- sizeof 23
- sizeof operator 57, 75, 78, 95, 165, 181
- small memory model 55
- smart pointer 153
- specifier
 - enum 43
- standard conversions 149
- statement 97
 - break 100
 - compound 98
 - continue 99
 - declaration 106
 - do 105
 - empty 98
 - expression 98
 - for 105
 - goto 99

- if-else 102
- jump 99
- labeled 97
- null 98
- return 101
- switch 103, 182
- while 105
- static 23
- static initialization 59
- static member 13, 129
- storage
 - class members 110
 - extern 51
 - overlapping 93
- storage class 38
- storage duration 16, 38
 - static 16
- storage duration:automatic 16
- storage-class
 - auto 39
 - extern 39
 - register 39
- storage-class declaration 39
- storage-class specifier 39
- string initialization 63
- string literal 30
- stringize operator 163
- struct 23, 42, 127
 - member selection 73
- structure types 19
- subscript operator 153
- subtraction operator 86
- switch 22
- switch statement 103, 182
- syntax error 157

T

- template 22
- temporary objects 138, 183
- this 22, 112, 113
- throw 23
- token 22
- TopSpeed C++
 - 16, 18, 20, 23, 25, 29, 31, 39, 44, 49, 64, 82, 116
- translation phases 156
- translation unit 12
- trigraph sequences 157
- trivial conversion 149
- try 23
- type
 - arithmetic 16

- derived 18
- enumeration 43
- float 18, 78
- int 17
- integer 17
- integral 16
- local 118
- long int 17
- pointer 19
- reference 19
- short 78
- short int 17
- structure 19
- void 18
- void* 35
- type name 57
 - local 118
- type-specifier 41
- typedef 13, 19, 23
- typedef declaration 40
- typedef int INT 118
- typedef-name 41, 108, 118

U

- unary expression 75
- unary minus operator 75
- unary operator 155
- unary plus operator 77
- union 23, 42, 115, 127, 181
 - member selection 73
- union types 19
- unsigned 23, 41
- usual arithmetic conversions 34, 88

V

- virtual 23, 115
- virtual base class 120, 121, 137
- virtual base class:initialization 143
- virtual destructor 136
- virtual function
 - pure 126
- void 23, 41, 82
- void expression 91
- volatile 23, 41, 47, 182
 - member function 113

W

- while 23