C++ Language Summary

Version 4.1 (ANSI/ISO Standard/STL Version/with Classes)
Last updated 2/25/04

This document provides neither a complete nor rigorous description of the C++ language. It does, however, describe the features of the language that are most useful to engineers and scientists. These frequently used aspects of the language are described below:

Program Structure

Any text on a line after a `//` symbol is ignored (used for comments). Long (multi-line comments can be placed between `/*` and `*/` symbols.

Use consistent indentation to indicate intended program structure.

Functions must be declared before use but can be defined after use. Modern C/C++ style is to put all function definitions after *main()* with all declarations before *main()* as prototypes. An alternate style eliminates the need for separate function declarations by placing all definitions before *main()* and their first use. Function definitions can not be nested. Modern style also involves placing declarations in header files (.h or no extensions) and definitions (implementations) in source code (.cpp) files.

Return to Table of Contents

Built in Types and References

Identifiers (names) are case sensitive and can be of any length but typically only the first 31 characters are significant. They must start with a letter (including _) and may contain letters and numbers. Objects names are generally all lower case. Class names generally start with an upper case letter. Constants are generally in all upper case.

Variables can be declared anywhere before they are used use. I usually collect all declarations at the top of each function so that they are easy to find. Some C++ programmers declare variables just before they are used. At any rate comments should be included with all non-trivial variable declarations describing significance, use and units (if any).

Use square brackets to indicate arrays. Arrays are declared with the number of storage locations indicated, but array element references start at zero. Modern C++ compilers support the string and vector container classes (declared in the **`string`** and **`vector`** include files, respectively). These container classes provide significant advantages over the use of arrays.

Return to Table of Contents

Sample Circle/Cylinder Class Syntax

```
// A derived class declaration

Class Cylinder: public Circle
{
protected:
  double length; // need data member.
public:
  Cylinder(double r = 1.0, double l = 1.0): Circle(r),length(l){}
  virtual double calcVol() const; // Returns the volume.
};

// Partial class definition (implementation)

double Cylinder::calcVol(void) const
{
  return (length*Circle::calcArea()); // Call base class function.
}
```

Return to Table of Contents

Microsoft MFC Notes

Visual C++ Application = Visual Part (user interface) + Functional Part (procedural code)

Application types: Dialog, Single Document Interface (SDI), Multiple Document Interface (MDI).

Common MFC Control Types: Check Box, Command Button, Edit Box, Group Box, Label (static text), Line and Radio Button

Message Box syntax:

```
MessageBox("Content: Hello World!", "Title: Sample",
MB_ICONQUESTION);
```

Message box icon types: *MB_ICON… QUESTION*, *EXCLAMATION*, *INFORMATION* and *STOP*. Other message box named constants: *MB_... OK*, *OKCANCEL*, *YESNO*, *YESNOCANCEL*, etc. Message box return values: *IDOK*, *IDYES*, *IDNO*, *IDCANCEL*, etc.

Useful MFCWnd Class Member Functions

`UpdateData()`	`UpdateData(TRUE)` retrieves the data from each control by copying its value into the control's associated *Value* category member variable. `UpdateData(FALSE)` copies values from each *Value* category member variable to their corresponding control for display.
`EnableWindow()`	`EnableWindow(TRUE)` enables the corresponding control. The control is specified by prefixing a *Control* category member variable to the function. `EnableWindow(FALSE)` disables the corresponding control.
`SetFocus()`	`SetFocus(TRUE)` gives the corresponding control input focus. The control is specified by prefixing a *Control* category member variable to the function. `SetFocus(TRUE)` removes input focus from the corresponding control.

Return to Table of Contents

Built in Types -

`void`	A generic "nontype."
`bool`	Boolean type (usually 1 byte), i.e., true (usually non-zero) or false (usually 0).
`char`	Characters (usually 1 byte).
`int`	Integers (2 or 4 byte).
`float`	Single precision real (floating point) numbers. Usually 4 bytes and not typically used.
`double`	Double precision real (floating point) numbers. Usually 8 bytes and typically used.

Type Modifiers -

`unsigned`	Doesn't use sign bit (assumes *int* if base type is omitted).
`long`	May have twice as many bytes as base type (assumes *int* if base type is omitted).
`short`	May have half as many bytes as base type (assumes *int* if base type is omitted).
`const`	Constant (values can't be changed during execution).

Return to Table of Contents

Common Operators

Assignment: =
Increment and decrement: ++ (pre or post fix) and -- (pre or post fix)
Arithmetic: +, -, *, / and % (integer remainder)
Relational: == (equality), != (inequality), <, >, <= and >=
Boolean: && (and), || (or) and ! (not) (*and*, *or* and *not* are used in ANSI/ISO C++)
Bitwise: & (and), | (or), ^ (xor), ~ (not), << (shift left) and >> (shift right)

Return to Table of Contents

Control Constructs

Zero is considered false and nonzero is considered true in *conditions*. Statements end with semicolons, i.e. ;'s. A block is a statement or two or more statements enclosed in braces, i.e. { and }. A block can be used anywhere a statement can be used. Statements and blocks can be spread across multiple lines.

Selection (if and switch constructs):

```
conditional expression ? expression1 : expression2

if (condition) block1 [else block2]

if (condition) block1 else if (condition) block2 ... [else
block3]

switch (expression)
{  case value1: [block1 [break;]]
   ...
   case valuen: [blockn [break;]]
   default: [blockn+1 [break;]]
}
```

Repetition (while and for constructs):

```
while (condition) block

do (block) while (condition);

while {…; if (condition) break; …;}

for (initialize; test; update) block;
```

which is equivalent to:

```
initialize;
while (test)
{   block;
    update;
}
```

Return to Table of Contents

Standard Libraries

Many commonly used features of C and C++ are defined in the standard libraries. There is massive overlap between the C libraries (declared in include files with names like <*libname.h*>) and C++ libraries (declared in include files with names like <*libname*>). The C++ libraries generally contain the same functions as the corresponding C libraries but with these functions placed in the `std` namespace. As a result, the following line generally should be placed immediately following the inclusion of the C++ libraries:

```
using namespace std;
```

The following table lists the new C++ names, the old C/C++ names and some commonly used functions of the most popular standard libraries.

New C++ Name	Old C/C++ Name	Use and functions
iostream	iostream.h	Defines insertion (<<) and extraction (>>) operators and creates the global stream objects like `cin` and `cout`. See also, Stream Member Functions
iomanip	iomanip.h	Provides a variety of steam formatting and manipulation tools. See also, Stream Manipulators section.
fstream	fstream.h	Required for file I/O operations. See also, Stream Member Functions.
cmath	math.h	Provides a wide range of special math functions. These include the trigonometric functions (with angles expressed in radians), `exp(double` *`x`*`)`, `log(double` *`x`*`)`, `log10(double` *`x`*`)`, `pow(double` *`base`*`, double` *`power`*`)`, `sqrt(double` *`x`*`)`, `fabs(double` *`x`*`)` and `fmod(double` *`numerator`*`, double` *`denominator`*`)`.
cstdlib	stdlib.h	Miscellaneous stuff. Including the `void srand(int` *`seed`*`)` and `int rand()` pseudo-random random number functions, the `int system(const char` *`command`*`[])` system command function and the exit(int *exit_code*) program exit function. Using the exit function in main generates a warning message in MSVC++ 6.0.
cassert	assert.h	Provides the assert error handling mechanism. This has largely been replace by the exception mechanism in bigger programs, but is still useful in smaller ones.
string	None	Provides the string class. Note that `<string>` is completely different than the old `<string.h>` library (now <cstring.h>. See also, String Class Member Functions and Operators

vector	vector.h	Provides the Standard Template Library (STL) implementation of a 1-dimensional, random access sequence of items. Generally replaces the use of 1-dimensional C/C++ arrays. See also, Vector Class Member Functions and Operators and the MSVC++ 6.0 <valarray>> include file.
ctime	time.h	Types and functions associated with calendar and time operations. Includes both processor and actual time and date functions.
complex	None	Provides a template class for storing and manipulating complex numbers.

Return to Table of Contents

String Class Member Functions and Operators

Selected String Functions -

`string(int size)`	Constructor. Argument *size* is optional but recommended. Note lower case *s*.
`int .length()`	Returns the length of the string.
`string .substr(int start, int size)`	Returns the substring starting at location *start* of length *size*.
`string& .insert(int n, string s)`	Inserts a copy of *s* into string starting at position *n*. The rest of the original string is shifted right.
`string& .erase(int from, int to)`	Removes characters from position *from* to through position *to* from the string. Moves the rest of the string to the left. Returns the modified string.
`int .find(string ss)`	Returns the starting position of the first occurrence of substring *ss*.
`getline(istream is, string s)`	Places next line from *is* into *s*. The string extractor (>>) only gets "words". Not actually a member function.

String operators include: `[]`, `=`, `>>`, `<<`, `+`, `==`, `!=`, `<`, `<=`, `>` and `>=`.

String elements are numbered starting at 0. Constructor/assignment example: `string name = "John Doe";`

Return to Table of Contents

Vector Class Member Functions and Operators

Selected Vector Functions -

`vector(int size)`	Constructor. Argument *size* is optional but recommended. Note lower case *v*.
`int .size()`	Returns the number of elements in the vector.
`bool .empty()`	Returns true only if there are no elements in the vector.
`void .push_back(Vector_type value)`	Puts *value* into a new storage location created at the end of the vector.
`void .pop_back()`	Removes the last element from the vector and discards it.
`void .resize(int newsize, Vector_type value)`	Resizes the vector. If *newsize* is less than the current size the vector is truncated. If *newsize* is larger than the current size the vector is enlarged by adding new elements after the last existing elements. These new elements are set to *value* if one is provided.
`iterator .begin()`	Returns an iterator that points to the first element of the vector.
`iterator .end()`	Returns an iterator that points immediately beyond the last element of the vector.
`int .insert(iterator location, Vector_type value)`	Inserts *value* into the vector at the specified location and returns the location.
`int .erase(iterator location)`	Removes the element at *location* from the vector and returns the position of the removal.
`void .clear()`	Removes all elements from a vector.
`void .swap(vector v)`	Interchanges the elements of the current vector and *v*. This operation is generally more efficient than an individual swapping of elements.

Vector operators include: `[]`, `=`, `==`, `!=`, `<`, `<=`, `>` and `>=`.

Vector elements are numbered starting at 0. Iterators can be created by adding element numbers to the result of the .begin() member function. Constructor example: `vector<double> a(MAX_SIZE, 0.0);`

Return to Table of Contents

Stream I/O

Stream Operators (defined in <iostream>)

`ostream& << const object`	Stream insertion.
`istream& >> object&`	Stream extraction.

Stream Objects Created and Opened Automatically

`istream& cin`	Standard console input (keyboard).
`ostream& cout`	Standard console output (screen).
`ostream& cprn`	Standard printer (LPT1?).
`ostream& cerr`	Standard error output (screen?).
`ostream& clog`	Standard log (screen?).
`ostream& caux`	Standard auxiliary (screen?).

Stream Classes (requires <fstream> and/or <strstream>)

`fstream`	File I/O class.
`ifstream`	Input file class.
`istrstream`	Input string class.
`ofstream`	Output file class.
`ostrstream`	Output string class.
`strstream`	String I/O class.

Return to Table of Contents

Stream Manipulators (defined in <iomanip>)

`dec`	Sets base 10 integers.
`endl`	Sends a new line character.
`ends`	Sends a null (end of string) character.
`flush`	Flushes an output stream.
`fixed`	Sets fixed real number notation.
`hex`	Sets base 16 integers.
`oct`	Sets base 8 integers.
`ws`	Discard white space on input.
`setbase(`*`int`*`)`	Sets integer conversion base (0, 8, 10 or 16 where 0 sets base 10).
`setfill(`*`int`*`)`	Sets fill character.
`setprecision(`*`int`*`)`	Sets precision.
`setw(`*`int`*`)`	Sets field width.
`resetiosflags(`*`long`*`)`	Clears format state as specified by argument.
`setiosflags(`*`long`*`)`	Sets format state as specified by argument.

Return to Table of Contents

Stream Member Functions

`void .close()`	Closes the I/O object.
`int .eof()`	Returns a nonzero value (true) if the end of the stream has been reached. Use after fail() returns true.
`char .fill(char` *`fill_ch`* `| void)`	Sets or returns the fill character.
`int .fail()`	Returns a nonzero value (true) if the last I/O operation on the stream failed.
`istream& .get(int` *`ch`*`)`	Gets a character as an int so EOF (-1) is a possible value.
`istream& .getline(char*` *`ch_string`*`, int` *`maxsize`*`, char` *`delimit`*`)`	Get a line into the *ch_string* buffer with maximum length of *maxsize* and ending with delimiter *delimit*.
`istream& .ignore(int` *`length`*`[, int` *`delimit`*`])`	Reads and discards the number of characters specified by *length* from the stream or until the character specified by *delimit* (default EOF) is found.
`iostream& .open(char*` *`filename`*`, int` *`mode`*`)`	Opens the *filename* file in the specified *mode*.
`int .peek();`	Returns the next character in the stream without removing it from the stream.
`int .precision(int` *`prec`* `| void)`	Sets or returns the floating point precision.
`ostream& .put(char ch)`	Puts the specified character into the stream.
`istream& .putback(char ch)`	Puts the specified character back into the stream.
`istream& .read(char*` *`buf`*`, int` *`size`*`)`	Sends *size* raw bytes from the *buf* buffer to the stream.
`long .setf(long` *`flags`* `[, long` *`mask`*`])`	Sets (and returns) the specified ios flag(s).
`long .unsetf(long` *`flags`*`)`	Clears the specified ios flag(s).
`int .width(int width | void)`	Sets or returns the current output field width.
`ostream& .write(const char*` *`buf`*`, int` *`size`*`)`	Sends *size* raw bytes from *buf* to the stream.

Return to Table of Contents

IOS Format Flags (ios::*x*)

`dec`	Use base 10.
`fixed`	Output float values in fixed point format (use the resetiosflags(ios::floatfield) manipulator or the unsetf(ios::floatfield) function to reset to default format).
`hex`	Use base 16.
`internal`	Distribute fill character between sign and value.

`left`	Align left.
`oct`	Use base 8.
`right`	Align right.
`scientific`	Outputs float values in scientific format (use the resetiosflags(ios::floatfield) manipulator or the unsetf(ios::floatfield) function to reset to default format).
`showbase`	Encodes base on integer output.
`showpoint`	Include decimal point in output.
`showpos`	Include positive (+) sign in output.
`skipws`	Skip white space (spaces and tabs).
`uppercase`	Forces upper case output.

Return to Table of Contents

IOS File Access Flags (ios::*x*)

`app`	Open in append mode.
`ate`	Open and seek to end of file.
`in`	Open in input mode.
`nocreate`	Fail if file doesn't already exist.
`noreplace`	Fail if file already exists.
`out`	Open in output mode.
`trunc`	Open and truncate to zero length.
`binary`	Open as a binary stream.

Return to Table of Contents

Class and Structure Definitions

class *Name*
{
public:

member_function$_1$ *declaration* [const];
member_function$_2$ *declaration*;
...

private:

data_member$_1$;
data_member$_2$;
...

```
};

[inline] type Class_name::member_function_name(arguments) [const]
{

        // code

};
```

The inclusion of the *const* modifier indicates that the function does not modify the object on which it operates. This restriction is enforced by the compiler.

Return to Table of Contents

Suffixes for Numerical Constants

Integer constants default to the smallest integer type that can hold their value. Otherwise, the following suffixes can be used (alone or together):

`u`	Unsigned
`l` or `L`	Long

Floating point constants default to type *double*. Otherwise, the following suffixes can be used:

`f`	Float
`l` or `L`	Long double

Return to Table of Contents

Character Escape Sequences

`\n`	Newline.
`\t`	Horizontal tab.
`\r`	Carriage return.
`\a`	Alert sound (bell).
`\\`	Outputs a backslash character.
`\"`	Outputs a double quote character.

Return to Table of Contents

Reserved Words

These words can't (or at least shouldn't) be used for programmer defined symbols (names). I've also seen problems with names like *min* and *max* in Visual C++ 6.0. Don't use words with particular meanings like *one* and *two* for names either.

	double	private	throw
asm	else	protected	try
auto	enum	public	typedef
break	extern	register	union
case	float	return	unsigned
catch	for	short	virtual
char	friend	signed	void
class	goto	sizeof	volatile
const	if	static	wchar_t
continue	inlineint	struct	while
default	long	switch	
delete	new	template	
do	operator	this	

Return to Table of Contents

Operator Precedence Chart

This table lists all the C++ operators in order of non-increasing precedence. An expression involving operators of equal precedence is evaluated according to the associativity of the operators.

Operator(s)	**Description(s)**	**Associativity**
`::`	Class scope resolution (binary)	left to right
`::`	Global scope (unary)	right to left
`()`	Function call	left to right
`()`	Value construction	left to right
`[]`	Array element reference	left to right
`->`	Pointer to class member reference	left to right
`.`	Class member reference	left to right
`-, +`	Unary minus and plus	right to left
`>++, --`	Increment and decrement	right to left
`!, ~`	Logical negation and one's complement	right to left
`*, &`	Pointer dereference (indirection) and address	right to left
`sizeof`	Size of an object	right to left
(*type*)	Type cast (coercion)	right to left

Operator(s)	Description(s)	Associativity
`new, delete`	Create free store object and destroy free store object	right to left
`>*`	Pointer to member selector	left to right
`*`	Pointer to member selector	Left to right
`*, /, %`	Multiplication, division and modulus (remainder)	Left to right
`+, -`	Addition and subtraction	Left to right
`<<, >>`	Shift left and shift right	Left to right
`<, <=, >, >=`	Less than, less than or equal, greater than, greater than or equal	Left to right
`==, !=`	Equality and inequality	Left to right
`&`	Bitwise AND	Left to right
`^`	Bitwise XOR	Left to right
`|`	Bitwise OR	Left to right
`&& or and`	Logical AND	Left to right
`|| or or`	Logical OR	Left to right
`? :`	Conditional expression	right to left
`=, *=, /=, %=, +=, =, &=, ^=, |=, >>=, <<=`	Assignment	right to left
`,`	Comma	Left to right

Return to Table of Contents

C++ References

Jones, R. M. Introduction to MFC Programming with Visual C++. Prentice Hall PTR, 1999.

Bronson, G. J. Program Development and Design Using C++, 2nd ed. Brooks/Cole Thomson Learning, 2000.

Bronson, G. J. A First Book of Visual C++. Brooks/Cole Thomson Learning, 2000.

Horstmann, C. S. Computing Concepts with C++ Essentials, 2nd ed. John Wiley & Sons, 1999.

Cohoon, J. P. and J. W. Davidson. C++ Program Design; *An Introduction to Programming and Object-Oriented Design*, 2nd ed. McGraw-Hill, 1999.

Deitel, H. M. and P. J. Deitel. C++ How to Program. Prentice Hall, 1994.

Perry, J. E. and H. D. Levin. An Introduction to Object-Oriented Design in C++. Addison-Wesley, 1996.

Barclay, K. A. and B. J. Gordon. C++ Problem Solving and Programming. Prentice Hall, 1994.

Johnsonbaugh, R. and M. Kalin. Object-Oriented Programming in C++. Prentice-Hall, 1995.

Horstmann, C. S. Mastering Object-Oriented Design in C++. John Wiley & Sons, 1995.