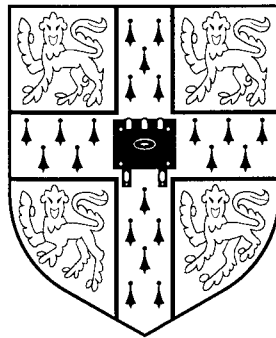


S/1

Programming Data Book

for Part I of the
Engineering Tripos

2003 Edition



Cambridge University Engineering Department

Contents

C++ Summary	1
1 Overview	1
2 Comments	1
3 Identifiers & declarations	1
4 Constants	1
5 Built-in types	2
6 Expressions	2
7 Predefined functions	2
8 Assignment statements	3
9 Blocks	3
10 Control statements	3
11 Text input and output	6
12 Enumerated types	7
13 Arrays	7
14 Strings	7
15 Functions	8
16 Structures	10
17 Classes	10
18 Files	12
Fortran Summary	13
Simple VHDL	19

C++ Summary

1 Overview

A C++ program has the following form:

```
#include <iostream>
#include "myheader.h"
using namespace std;

const int    c1 = ...; // c1, c2 are the names of
const float c2 = ...; //                constants
struct      s1 {...}; // s1, s2 are the names of
struct      s2 {...}; //                user-defined structures
class       x1 {...}; // x1, x2 are the names of
class       x2 {...}; //                user-defined classes

int main()
{
    int      v1, v2;    // v1, v2 are variables of predefined type int
    s2       v3;       // v3 is a variable of user-defined type s2

    cin >> v1;         // program body is a sequence of
    .....;            // statements separated by semicolons
    .....;
    v2 = v1 + c1;
    .....;
    cout << v1 << " becomes " << v2 << endl;

    return 0;         // main program always returns 0 if successful
}
```

2 Comments

Comments begin with double-slashes // and continue for the rest of the line. Comments are ignored by the compiler.

3 Identifiers & declarations

Identifiers are a sequence of letters, digits and underscores starting with a letter, and with no restriction on length. They are used to give a name to a user-defined object, e.g. variable, struct, function, etc. Every name in a C++ program must be declared before it is used. `int v1, v2;` in the program above declares two integer variables with identifiers `v1` and `v2`.

4 Constants

Symbolic constants are declared and initialised as variables, but the declaration is preceded with the `const` keyword. `const float pi = 3.1415;` declares a floating point constant `pi`.

5 Built-in types

int	for whole numbers	e.g. 3276, -1, 0, 4, 20
float	for real numbers	e.g. 2.703, 1.2e10, -4.0e-2
double	for real numbers, greater precision	e.g. 3.141592654, 6e23, 1.6022e-19
char	for character values	e.g. 'A', 'b', 'X', '9', '??'
bool	for logical values true and false	

Note that `short int` and `long int` are also available, and may be represented by fewer or more bits than `int`. `unsigned int`, `short unsigned int` and `long unsigned int` can be used to hold zero-or-positive whole numbers.

6 Expressions

Expressions are used to compute values by applying operators to variables, constants and literal values. The operators for the predefined types include:

integer ops	+, -, *, /, %(modulus)	e.g. (13+4)/5 is 3
real ops	+, -, *, /	e.g. (13.0+4.0)/5.0 is 3.4
relational ops	==, !=, >=, <=, >, <	e.g. (6+2)<9 is true
logical ops	&& (and), (or), ! (not)	e.g. (6>8) (3<4) is true
bit-shift ops	>>, <<	e.g. 2<<10 is 2048

Precedence order:

1. ! (logical not)
2. *, /, %
3. +, -
4. >>, <<
5. <, <=, >, >=
6. ==, !=
7. && (logical and)
8. || (logical or)

Increment and decrement operators (see section 8 on assignment statements):

- `count++` is equivalent to `count = count + 1;`
- `count--` is equivalent to `count = count - 1;`

Operators can be defined for user types. See the use of `<<` and `>>` in section 11, and the `--` member function in the example class in section 17.

7 Predefined functions

You need to have the `#include <cmath>` directive at the top of your program to use these. There are versions of these functions for parameters and return values of both type `float` and of type `double`.

Function	Description
<code>sqrt(x)</code>	square root
<code>sin(x)</code>	trigonometric sine of x (in radians)
<code>cos(x)</code>	trigonometric cosine of x (in radians)
<code>tan(x)</code>	trigonometric tangent of x (in radians)
<code>exp(x)</code>	exponential function
<code>log(x)</code>	natural logarithm of x (base e)
<code>log10(x)</code>	logarithm of x to base 10
<code>fabs(x)</code>	absolute value (unsigned)
<code>pow(x,y)</code>	x raised to power y

8 Assignment statements

Assignment statements assign a value to a variable. The general form is:

```
<variable> = <expression>;
```

In general the type of the expression need not be the same as the type of the variable, but there are complex rules covering the automatic conversions which occur. A straightforward example is that `int` values can be assigned directly to `floats`. If a `float` value is assigned directly to an `int` variable it will be converted into an integer, usually by truncation *not* rounding.

9 Blocks

```
{
    int i;

    s1;
    s2;
}
```

A block is a group of statements and optionally declarations enclosed in braces. The body of any function (see section 15) is a block. Blocks are also commonly used in control statements (see section 10).

Any variables declared within a block are strictly local to that block (and any enclosed block), and they are created each time the block is executed and destroyed when the block is exited (unless declared `static`).

10 Control statements

Control statements determine the order in which nested statements are executed. In the following, b denotes a boolean expression, i is a variable of any simple type and s , $s1$ and $s2$ are single statements. Where more than one statement is required in a control structure, a block should be used.

- IF

```
if (b) s1;          // if b is true, s1 is executed

if (b) s1; else s2; // if b is true, s1 is executed, but
                  // if b is false, s2 is executed

if (b)
{
    // executed only if b is true
    s1;
    s2;
}

if (b)
{
    // executed only if b is true
    s1;
    s2;
}
else
{
    // executed only if b is false
    s3;
    s4;
}
```

- SWITCH

```
switch (i)
{
    case i1:
        s1;    // s1, s2 executed if i == i1
        s2;
        break;

    case i2:
        s3;    // s3 executed if i == i2
        break;

    // ... more cases as required

    default:
        s4;    // s4 executed only if none of the cases matches
        break;
}
```

i1 and i2 are constants of the same type as variable i.

- **WHILE**

```
while (b) s1; // s1 executed repeatedly while b is true

while (b) // s1, s2 executed repeatedly while b is true
{
    s1;
    s2;
}
```

Note that if b is false, s1 or s1 and s2 will *not* be executed at all.

- **DO...WHILE**

```
do s1; while (b); // s1 executed repeatedly until b is false

do // s1, s2 executed repeatedly until b is false
{
    s1;
    s2;
}
while (b);
```

Note that s1 or s1 and s2 are executed *at least once* (even if b is false).

- **FOR**

```
for (s_initialisation; e_boolean; s_increment)
{
    s1;
    s2;
}
```

is equivalent to (but clearer in intent than):

```
s_initialisation;
while (e_boolean)
{
    s1;
    s2;

    s_increment;
}
```

for example:

```
int i;
for (i = 0; i < 20; i++)
{
    s1; // s1, s2 executed 20 times
    s2; // with i = 0 ... i = 19
}
```

11 Text input and output

To make use of the input and output facilities described here, the header file for the type `iostream` and associated operators must be included by using the compiler directive:

```
#include <iostream>
```

The streams and symbols this describes are contained within the `std` namespace, which can be most easily accessed by using the command:

```
using namespace std;
```

The input stream `cin` and the **extraction operator** (`>>`) are used for reading data from the keyboard and assigning it to variables. `cin` and `cout` (see below) are pre-defined variables of type `iostream`. The `>>` and `<<` operators, when associated with `cin` and `cout`, are examples of user-defined operators, as discussed in section 6.

```
cin >> v;
```

Data typed at the keyboard followed by the *return* or *enter* key is assigned to the variable `v`.

```
cin >> v1 >> v2;
```

The value of more than one variable can be entered by typing their values on the same line, separated by spaces and followed by a return, or on separate lines.

A statement to print the value of a variable or a **string of characters** (set of characters enclosed by double quotes) to the screen begins with `cout`, followed by the **insertion operator** (`<<`). The data to be printed follows the insertion operator.

```
cout << "text to be printed";  
cout << v;  
cout << endl;
```

The symbol `endl` is called a **stream manipulator** and moves the cursor to a new line. It is an abbreviation for *end of line*.

Strings of characters and the values of variables can be printed on the same line by the repeated use of the insertion operator.

```
int total = 12;  
cout << "The sum is " << total << endl;
```

prints out

```
The sum is 12
```

and then moves the cursor to a new line.

12 Enumerated types

When a small set of values is needed to represent a non-numeric quantity, an enumerated type can be declared, e.g.:

```
enum Days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
Days today = Tue;
```

creates a new variable type with legal values Mon, Tue, Wed, Thu, Fri, Sat, Sun. Variable today is initially assigned the value Tue, and the compiler will ensure that only values in the range Mon to Sun may be assigned to today.

13 Arrays

Arrays are used to construct variables with multiple components of identical type. Components are accessed via one or more array index expressions within brackets []. Some examples of the definition and use of arrays are:

```
float    vector[100];        // elements vector[0] .. vector[99]
float    matrix[100][100];   // 2-D array

for (i = 0; i < 100; i++)    // zero array vector
    vector[i] = 0.0;

matrix[2][4] = vector[3];    // 2-D and 1-D accesses
```

14 Strings

A common use of the one-dimensional array structure is to create character strings. A character string is an array of type char which is terminated by the null terminator character, '\0'. String variables are manipulated using the usual array operations with the addition that string literals (a sequence of characters within double quotes) may be used to initialise char arrays and can be used in text output statements, e.g.:

```
char greeting[6] = "Hello"; // '\0' automatically appended
char name[] = "Robert";    // Length automatically set to 7

cout << greeting << endl;
cout << "My name is " << name << endl;
cout << name[0] << greeting[4] << name[2] << endl;
```

prints out:

```
Hello
My name is Robert
Rob
```

15 Functions

Functions are used to divide a program up into smaller more manageable components. Their basic structure is the same as the main program (which is itself just a function, `main()`). The general form is:

```
<return_type> procname(<parameter_type_1> p1, <parameter_type_2> p2 ...)  
{  
    <type_1>    ...;  
    <type_2>    ...;  
  
    s1;  
    s2;  
    ...  
    return <value of type return_type>;  
}
```

Note that the body of the function is a block, as described in section 9, and that variables declared within the function behave as described in section 9.

Example: definition of a factorial function, and a function to calculate ${}^n C_r$:

```
// Factorial function declaration (prototype)  
int factorial(int n);  
  
// Combinations function definition  
int combinations(int n, int r)  
{  
    return factorial(n) / (factorial(n - r)*factorial(r));  
}  
  
// Factorial function definition  
int factorial(int n)  
{  
    int i;  
    int factorial = 1;  
  
    // Calculate the factorial with a FOR loop  
    for (i = 1; i <= n; ++i)  
    {  
        factorial = factorial * i;  
    }  
  
    return factorial;  
}
```

The value (of type `return_type`) returned by a function can be used as a component of an expression. Thus given the above functions and the variables `total`, `count` and `comb` also of type `int`, the following expression and assignment may be used compute ${}^{total}C_{count}$ and store the result in `comb`:

```
comb = combinations(total, count);
```

The formal parameters of a procedure (e.g. `int n` and `int r` in combinations) specify the name and type for each argument. A formal parameter preceded by `&` indicates call by reference. The corresponding actual parameter must be a variable of the same type, and statements within the function can modify its value.

If the `&` is omitted then call by value is indicated. The actual parameter can be any expression yielding a value of the required type, and a copy is assigned to the corresponding formal parameter, i.e. it is an input-only parameter. The parameters for `factorial` and `combinations` are all passed by value.

Declaring the function to be of type `void` indicates that no value is returned by the call. The following functions illustrate `void` and call by reference:

```
void swap(float &a, float &b)
// Swaps x and y data of calling function.
{
    float tmp;
    tmp = b;
    b = a;
    a = tmp;
}

void sort(float data[], int length)
{
    int    i, j, minj;
    float  min;

    for (i = 0; i < length-1; i++)
    {
        minj = i;
        min = data[i];
        for (j = i+1; j < length; j++)
        {
            if (data[j] < min)
            {
                minj = j;
                min = data[j];
            }
        }
        if (minj > i)
            swap(data[minj], data[i]);
    }
}
```

Arrays are always passed by reference, and the function can access and/or modify all the elements of the array. This avoids the expense of making a copy of each element of the array.

Functions may call themselves (and others) recursively, as illustrated in the following version of the `factorial` function:

```

int r_factorial(int a)
{
    int result;

    if (a <= 1)
        result = 1;
    else
        result = a * r_factorial(a - 1);    // recursive call

    return result;
}

```

16 Structures

Structures are used to construct variables with multiple components of possibly differing types. Each component (also known as a *data member* or *field*) is referenced by a name rather than by an index expression as in arrays. A *'.'* is used to indicate data member access. Examples of definition and use:

```

struct Complex {
    float   re;
    float   im;
};

struct Time {
    int     hour;
    int     min;
};

Complex    a,b;
Time       t;

a.re = a.re + b.re;        // a = a + b
a.im = a.im + b.im;
t.hour = 12; t.min = 0;    // t = noon;

```

17 Classes

Classes are an extension of structures to allow a type to contain **member functions** as well as data components. The member functions are typically designed to perform useful operations on the data members of variables of that class.

public member functions and data members are available to any part of a program using the class, and can be used to provide the public interface to the class.

private member functions and data members are only visible to other member functions of the class, and can be used to hide the details of the implementation.

```

class FlexArray {
private:
    int    len;
    float  data[100];

public:
    void    clear();
    void    add(float item);
    float   operator--();    // remove last item and return it.
}

void FlexArray::clear()
{
    len = 0;
}

void FlexArray::add(float item)
{
    if (len < 100)
    {
        data[len] = item;
        len++;
    }
    else
    {
        // handle array overflow error
    }
}

float FlexArray::operator--()
{
    if (len > 0)
    {
        len--;
        return data[len];
    }
    else
    {
        // handle array underflow error
    }
}

FlexArray fa;

fa.clear();
fa.add(x);
float y = fa--;

```

The FlexArray:: prefix denotes that the function is a member of the FlexArray class.

18 Files

Files are used for permanent retention of large amounts of data. To perform file processing in C++ the header file `<fstream>` must be included. The latter includes the definitions of the `ifstream` and `ofstream` classes (contained within the `std` namespace). These are used for input from a file and output to a file. This is a more general form of the mechanisms described in section 11, allowing named files and different types.

The following statements open the file called `input_data`, and read in data which is stored sequentially in variables `a`, `b` and `c`:

```
ifstream fin;
fin.open("input_data");
if (fin.good())
    fin >> a >> b >> c;
else
{
    // error opening file
}
```

In a similar way data can be written (in this example, the values of an array) to a file called `my_results` with:

```
ofstream fout;
fout.open("my_results");
if (fout.good())
{
    for(i=0; i<N; i++)
    {
        fout << array[i] << endl;
    }
}
else
{
    // error opening file
}
```

Note that `fin` and `fout` are arbitrary names chosen by the programmer. The `good()` member function of `ifstream` and `ofstream` objects returns false if there is an error.

After finishing reading from and writing to files they must be **closed** with the statements:

```
fin.close();
fout.close();
```

and the two variables `fin` and `fout` can be re-used for other files.

Fortran Summary

N.B. This section deals with standard Fortran 77. Many compilers allow considerable extensions.

Layout - Columns 1 to 5 are reserved for labels. Column 6 is reserved for a continuation character. Columns 7 to 72 are available for statements. Spaces are ignored and are recommended as an aid to program readability.

Variables - up to 6 characters in name, first character a letter.

By default a variable with first character A...H or O...Z inclusive is of real type (approx 7 significant figures).

By default a variable with first character I...N is of integer type (up to approx 2.15×10^9).

Alternatively declare variable REAL or INTEGER at start of program.

Other variable types COMPLEX, DOUBLE PRECISION, LOGICAL and CHARACTER*n must be declared separately.

IMPLICIT NONE as first statement turns off default rules for variable types.

Subscripted Variables - For subscripting use brackets, X(I), A(20,2*J) etc. and declare at start of program. Any variable may be subscripted, but must not appear in both subscripted and unsubscripted form in the same program. Subscripts by default start from 1 and may appear as integer constants or expressions.

Constants - may be real or integer, with same limits as variables. Use E for exponent of 10, i.e. 3.43E-6. Complex constants have form (creal, cimag).

Library Functions - Arguments in brackets. Standard supplied ones are SIN, COS, TAN, LOG (=log_e), EXP, SQRT, ATAN, ACOS, ASIN, ATANH, ABS, INT, REAL, CMPLX. Arguments for SIN, COS, TAN are in radians.

Function SIGN(A,B) computes |A| . (sign of B).

ATAN2 has 2 arguments:- ATAN2 (Y,X) computes arctan(Y/X).

CMPLX can have 1 or 2 arguments:- CMPLX(X,Y) gives X+iY, CMPLX(X) gives X+i0.

Arithmetic Operations

+, -, *, /, ** (for exponentiation).

Multiplication sign must appear explicitly, i.e. (A+B)*C, not (A+B)C.

A/B*C is interpreted (A/B)*C.

Expressions such as (I/J) within integer expressions are of type integer and hence truncated (towards zero) before further processing.

Assignment Statement - General form

variable (simple or subscripted) = expression
E.g. Y = Y + 2

Logical IF

IF(logical expression) statement

Logical expressions use .GT. .LT. .EQ. .GE. .LE. .NE. and .AND. .OR.

Block IF

```
IF( logical expression ) THEN  
.....statement(s).....  
ENDIF
```

or

```
IF( logical expression ) THEN  
.....statement(s).....  
ELSE  
.....statement(s).....  
ENDIF
```

or

```
IF( logical expression ) THEN  
.....statement(s).....  
ELSE IF( logical expression ) THEN  
.....statement(s).....  
ELSE IF.....  
.....  
ELSE  
.....statement(s).....  
ENDIF
```

Labels - integers up to 99999 at start of line. Same value must not appear more than once in a program or subprogram.

Jumps

```
GOTO label
```

DO loops

```
DO label, I=M1,M2,M3
```

I is a variable (usual integer).

Causes statements up to and including the labelled one to be executed for each value of I from I=M1 up to I = M2, in steps of M3 (M3 defaults to 1). If M2 is less than M1, then the loop is skipped.

Labelled statements must not be GOTO, IF, DO, END. If necessary use CONTINUE.

Input

```
READ *, VAR1,VAR2,...  
READ(n, label) VAR1,VAR2,  
label FORMAT( Iw, Fw.d, Aw,..)
```

Usually n = 5 or * for keyboard input, n = any positive integer for a data file.

I for integers, F for reals, Aw for reading w characters.

Output

```
PRINT *, VAR1, VAR2,...  
WRITE(n,label) VAR1,VAR2,...  
label FORMAT( lw, Fw.d, Ew.d, Dw.d, Aw, ' text ', nX, /,... )
```

Usually $n = 6$ or $*$ for the screen, $n =$ any positive integer for a data file.
I for integers (w places), F for reals (total no of places w , d decimal places), E for reals in decimal-exponent form, D for double precision reals in decimal-exponent form, Aw for w characters, nX gives n spaces, / gives an extra line.
Lists may be written in the form

```
WRITE(n,label) (A(I),B(I),I=1,6),X,Y
```

Comment line - Any line beginning with a C in the first column.

Line continuation - An (otherwise ignored) character in the 6th column indicates a line is a continuation of the preceding one.

STOP - This statement must be somewhere in the program.

END - Indicates the end of a program and must be the last statement.

FUNCTION subprograms. General form

```
FUNCTION name (dummy argument list)  
.....declaration statements.....  
.....program statements.....  
RETURN  
END  
E.g.  
FUNCTION SUM(A,N)  
REAL A(N)  
C  
C Sum the first n elements of array A  
SUM = A(1)  
DO 10, I=2,N  
10 SUM = SUM + A(I)  
C  
RETURN  
END
```

Calling program has:

```
variable = name( actual argument list)  
or variable = expression with name(list)  
E.g.  
DIMENSION BLOCK(100)  
C Find sum of first 20 elements  
.....  
Y = SUM( BLOCK, 20 )
```

Notes: The values of the variables in the dummy argument list (in the example A and N) must not be changed within the function subprogram. A FUNCTION subprogram has the same default type conventions as variables (See the section on variables). So that for

example a name beginning with the letters I,J,K,L,M, or N is assumed to set an integer value. This can be overridden by having the word **REAL** precede the word **FUNCTION**. The actual argument list may include constants, expressions and/or subscripted variables, but the number of arguments and type of each in the calling statement must agree with those expected by the subprogram.

SUBROUTINE subprogram - similar to functions but may reset any number of the argument list.

```

SUBROUTINE name (dummy argument list)
.....declaration statements.....
.....program statements.....
RETURN
END
e.g
SUBROUTINE ORDER (A,N)
REAL A(N)
C
C Order elements of A
DO 20, I=2,N-1
DO 10,J=1,N-I
IF (A(J+1).LT.A(J)) THEN
TEMP = A(J)
A(J) = A(J+1)
A(J+1) = TEMP
ENDIF
10 CONTINUE
20 CONTINUE
C
RETURN
END

```

Calling program has:

```

CALL name( actual argument list)
e.g.
DIMENSION COST(50)
.....
M = 50
CALL ORDER( COST, M)

```

Notes:- The actual argument list may include constants, expressions and/or subscripted variables or subprogram names, but the number of arguments and type of each in the calling statement must agree with those expected by the subprogram. An actual argument which is a subprogram name must be included in an **EXTERNAL** statement (See that section).

Further Notes on Subprograms

Argument lists are optional - a subroutine may have no argument.

Each **FUNCTION** or **SUBROUTINE** subprogram is compiled as a separate unit. The numbers used for labels and local variable names (those not included in the argument list) may be duplicated without confusion. The variable **I** in the examples is completely separate from any variable **I** in the calling program.

Values can also be passed back and forth between subprograms and calling programs by the use of a **COMMON** statement (See that section).

Statement Functions - for single line functions. These come after declaration statements and before the first executable program statement. They take the form

```

name( dummy argument list) = expression
e.g
FN(X,Y) = (X*X+Y*Y)*0.5

```

Subscripted variables may not be used in statement function definitions, but they may be used when the function is later included in an arithmetic expression

```

Z = A + FN(C,D(J))

```

Statement functions are local to the program unit in which they are defined.

External statement - used when the name of any function subprogram is passed as an argument to another subprogram. It must be placed with the declaration statements in the calling program, preceding statement function definitions and the first executable statement:

```

EXTERNAL list of function names
E.g.
EXTERNAL FT
.....declaration statements....
.....
Y = ...
C
C FT is a function to be used in subroutine INTG
CALL INTG( FT, Y, ANS)
.....
END
C-----
SUBROUTINE INTG( FUNC, X, ANS)
.....
C FUNC is a dummy function name
ANS = FUNC(X)*0.5 + ...
.....
RETURN
END
C-----
FUNCTION FT(X)
.....
FT = .....
.....
RETURN
END

```

COMMON statement - used to pass arguments implicitly between a calling program and a subprogram. Space allocated in the **COMMON** area is shared by one or more subprograms, and the statement is used to specify names of variables and arrays to occupy that area. It must be placed with the declaration statements in the calling program and in any subprogram which expects arguments in **COMMON**.

<p>COMMON list of variable and array names (with array size if not already declared)</p> <p>E.g. REAL INDEX(3) COMMON LIST(200), COUNT, INDEX</p>

DATA statement - for setting initial values

<p>DATA list of names / initial values /</p> <p>E.g. CHARACTER*4 TEST DATA B, I, TITLE / 100.5, 24, 'TEST' /</p>
--

This statement must come before the first executable statement. It cannot be used to set a variable in **COMMON**. It only sets variables at the time a program is first started. It will not for example reset variables each time a subprogram is entered.

Simple VHDL

VHDL stands for “very high speed integrated circuit hardware description language.” It describes circuits in terms of design entities. Each entity consists of two parts. The interface and the architecture specification.

OUTLINE

```
entity component_name is
    list of input and output ports
end component_name;

architecture arch_name of component_name is
    declarations of internal signals;
begin
    description of what the the entity does
    and how it is implemented
end arch_name;
```

EXAMPLE

```
entity NAND2 is
    port(A, B :in BIT;
          Y   :out BIT);
end NAND2;

architecture NANDARCH of NAND2 is
begin
    Y <= not (A and B);
end NANDARCH;
```

To include a predefined gate in a larger design you use the following syntax.

OUTLINE

```
unique label to refer to this version of the gate
:
entity
name of gate entity
(
name of gate architecture
)
port map
(
'port list' of signals connected to gate
);
```

EXAMPLE

```
N1:entity NAND2 (NANDARCH) port map(P, Q, Y);
```

Operators (incomplete list)

<pre><= signal assignment = relational equality < less than <= less than or equal to and logical and nand logical nand xor logical exclusive or + addition * multiplication</pre>	<pre>:= variable assignment /= relational inequality > greater than >= greater than or equal to or logical or nor logical nor not logical negation - subtraction / division</pre>
--	---

VHDL is not case sensitive. Identifiers must begin with a letter. Two dashes “--” are used to begin a comment line.