Chapter VII

Summary

7.1 Identifiers

Identifiers are arbitrary names of any length given to classes, objects, functions, variables, user-defined data types and so on. Identifiers can contain the letters a to z and A to Z, the underscore character "_", and the digits 0 to 9. There are only two restrictions:

- The first character must be a letter or an underscore.
- C++ recognizes only the first 32 characters.

7.2 Line Splicing with \

When you have to begin a new line in a string and wish to just continue from where you left use the back slash \. For example:

```
cout << "C++ is an easy \
 language to learn for some.";
```

7.3 C++ Comments

C++ uses // for single line comments and /* .....*/ for multi-line comments.

7.4 Case Sensitivity

C++ identifiers are case sensitive, so that Sum, sum and SUM are distinct identifiers.

7.5 Braces

```
if (condition)
{
c = a + b;
x + +;
}
```

← compound statement

```
if (condition)
  c = a + b;    ← executed if the condition of if is true;
  x + +;        ← executed irrespective of if
```

7.6 Operators

Operators are tokens that trigger some computation when applied to variables and objects in an expression. These can be categorized as follows:

- *Arithmetic*
- *Assignment*
- *Bitwise*
- *C++ specific* :: .* ->* :
- *Comma*
- *Conditional* ?:
- *Equality*
- *Logical* (results in true or false)
- *Postfix Expression* (), [], { }, . , ->, (*cast*)
- *Primary Expression* literal (sometime referred to as constants), expression including declaration expressions, **this**, scope resolution operator ::, name of a class or enumerator or structure, and the class destructor ~ (tilde).
- *Preprocessor*
- *Reference/Indirect*
- *Relational* (<, > , ==, etc)

7.7 Precedence of operators

There are 17 precedence categories, some of which contain only one operator. Operators in the same category have equal precedence with each other.

Operators	**Associativity**
( ) [ ] -> :: .	Left to right
! ~ + - ++ – & * (typecast)	Right to left
sizeof new delete typeid	Right to left
.* ->* (Either operator is for dereferencing pointer to a class member)	Left to right
* / %	Left to right
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
== !=	Left to right
&	Left to right

Operators	Associativity
^	Left to right
|	Left to right
&&	Left to right
||	Left to right
?: (conditional expression)	Right to left
= *= /= %= += -= &= ^= |= <<= >>=	Right to left
,	Left to right

The operator "!" is for logical negation and '~' is for bitwise complement. **typeid** is for run-time type identification which lets you write portable code that can determine the actual type of a data object at run time.

Depending on the context, the same operator can have more than one meaning. For example, the ampersand (&) can be interpreted as:

- a bitwise AND (A&B)
- an address operator (&A)
- a reference modifier

Note No spaces are allowed in compound operators. Space changes the meaning of the operator and will generate an error.

7.8 Structure of if and if-else statements

The syntax for an **if** statement

```
if( condition)
    statement;
```

or for a compound statement

```
if(condition)
   {
statement1;
statement2;
 ...
    }
```

For **if-else** statement

```
if(condition)
   statement1;
else
   statement2;
```

or for compound statements

```
if(condition)
  {
 statement1;
 statement2;
 ...
   }
else
  {
 statement3;
 statement4;
    }
```

Multiple if-else

```
if(condition1)
     {
       ...
       }
else if(condition2)
      {
      ...
        }
else if(condition3)
       .
       .
       .
else
       {
      ...
          }
```

With the multiple if-else we can use either single or compound statements.

7.9 switch statement

```
switch( variable)
        {
    case value1:
        statements;
        break;
    case value2:
        statements;
        break;
    case value3:
        statements;
        break;
    default:
        statements;
         }
```

7.10 Loops

```
for(<counter with initial value>; <condition>; <expression for adjusting initial
        value>)
    {
    statements;
    }
```

You can also use an empty **for** loop for an infinite loop:

```
for( ; ; )
   {
 statements + condition to exit loop using break;
    }

do {
    statements;

    }while(condition)

 while(condition)
    {
    statements;
    }
```

For an infinite loop

```
 while(1)
   {
  ...
```

```
    if(condition) break;
   ...
        }
```

7.11 Type definition

```
e.g.    typedef unsigned char BYTE;
        typedef double MATRIX[3][3];

void main()
   {
BYTE ch;      // same as unsigned char ch;
MATRIX A, B; // same as double A[3][3], B[3][3];
 ...
       }
```

7.12 Enumerated Data Type

```
e.g     enum BOOL {F, T};
        enum WeekDay{ Mon=1, Tues, Wednes, Thurs, Fri, Sat, Sun};
```

7.13 Conditional Assignment

```
variable = (condition)? value1 : value2;
```

7.14 Structures

```
e.g.    struct  _Complex {
                    double Real;
                    double Imag;
                        };


void main( )
    {
_Complex a, *b;

a.Real = 5.9;
a.Imag = 7.3;

b->Real = 6;
b->Imag = 8.9;
```

```
    ...
}
```

7.15 Pointers

References

e.g.
```
int i;
int & y = i;        // y is i
```

Also if a function has its arguments by reference:

```
float   f( float & a, float &b)
```

then the calling program can simply call f as follows:

```
f(c,d);
```

Any changes carried-out on c and d by f will be reflected in the calling program.

Pointers to simple variables

e.g.
```
int *i;     // a pointer to an integer
int j;

*i = 4; //dereferencing a pointer using *
i = &j;   //assigning the address of j to i
```

Arrays

e.g.
```
float a[3][5];   //3 rows by 4 columns
```

Arrays can be declared and assigned values as follows:

```
float a[][3] = { { 1, 2, 3},
                 { 3, 4, 5}};

float v[] = { 4, 6, 7, 8, 9 };
```

Allocating and Deallocating a Dynamic Array

One -Dimensional array

e.g.
```
double *p;
int n;
p=new double [n];
...
delete [] p;
```

Two-Dimensional Array

e.g.
```
float **p;
int NR, NC, i;
```

```
p = new float * [NR];
for( i=0; i<NR; i++)
  p[i] = new float [NC];

...

for(i=0; i<NR; i++)
  delete [] p[i];
delete [] p;
```

Pointers to functions

e.g.
```
double factorial(int x);
        ...
void main()
 {
 double (*f)(int);
 f=factorial;
    ...
  }
```

7.16 Classes

e.g.
```
class _Complex {
        private:
          member variables;
          member functions; // for functions used internally
                            // by the class
        public:
          member functions;
          member variables; // rarely are variables declared
                            // public
      };
```

Public member functions contain constructors, destructors, getter functions, setter functions and functions for manipulating the variables.