

The Bracelet

C++ Quick Reference

For Arab Team Forum → C and C++ Forum

Raghad A.A

The Bracelet is a C++' Quick Reference for those who have finished studying C++ Programming Language or they've already got a great knowledge about this language. It may be useful for the beginners in which they may avoid memorizing the words and syntax. Nicely, the student may print this text and keep it near by his/her computer.

Before you read, you should familiarize yourself with this text conventions and its approach, so read them carefully. They are included within the following pink-bordered box, and then enjoy.

Some conventions involved in this text:

- It is divided into three columns read from left → right and vice versa because it doesn't have a specific arrangement. Just catch the red-colored topic and keep track of it downward until you encounter another red-colored topic.
- red-colored denotes main topic
- Brown-colored text denotes sub-topic and issue.
- Syntax: colored with blue—whatever immediately followed this word is treated syntactically.
- **Bold-text** style denotes a keyword, an operator, and delimiting of scope {...}
- *Italic-text* style fits to be substituted with something.
- [...]—everything between them is optional except that one which has **bold-style** and treated as array subscript operator[...].
- Another colored-text serves as highlight within the code structure, such as the comments and the preprocessing directives.
- : it is found in a perfect line by itself so it denotes multi-statements to be continued.

Authored and Collected by Raghad A.A in 2008-01-11
Presented to Arab Team Forum → C and C++ Forum

Typical Header File Structure ".h":

```
....
Syntax:
#ifndef HEADER_NAME
#define HEADER_NAME
//Include Headers
#include <HeaderFile>
#include "HeaderFile.h"
//Using namespaces
//Global Declaration (file scope)
:
:
//Class or Classes Definition
:
:
#endif
```

The Primitive Data Types:

- Integral Data Types:

int , char, wchar_t, bool

note: bool type has two discrete values:

false , true aka(0, 1 respectively)

- Floating-Number Data Types:

float , double

- None-Data Type (Empty)

void

Structurs:

- Normal Structure:

```
Syntax:
struct strucName
{
    type elem1;
    type elem2;
    type elem3;
    :
    :
}[structures-variables];
```

The ".cpp" implementation related to a specific header file:

```
....
Syntax:
//Include Headers
#include <HeaderFile>
#include "HeaderFile.h"
//Using namespaces
//Global Scope (file scope).
:
:
//Functions Implementations of
//the related Header file.

ClassName::Const([param...list])
:BaseConstruct([param...list])
{
    //Constructor's body
    :
}

type ClassName::func([param...list])
{
    //Function's body
    :
    [return value;] //if needed
}

}
```

Some specifiers, qualifiers, and modifiers:

- Class-Storage Specifiers:

auto , extern, register, static, mutable

- The Qualifiers:

const , volatile

- The Modifiers:

signed , unsigned , long , short

Functions:

-Function Prototypes:

```
Syntax:
type FunctionName([param...list]);
```

-Function Definition:

```
Syntax:
type FunctionName([param...list])
{
    //function body
    :
    :
    [return value;] //if needed
}

}
```

The main Program Structure:

```
...
Syntax:
//Include Headers
#include <HeaderFile>
#include "HeaderFile.h"
//Using namespaces
//Global Scope (file scope).
//Functions Prototypes.

//The main function
type main([param...list])
{
    //the main program
    //part goes here
    :
    :
    [return value;] //if needed
}
//Global Functions Definitions
```

Comments:

- C++ supports two comment styles:

Syntax:
//In-line Comment...

/*Multi-line Comment
as long as you need...*/

Pointers:

- Define a Pointer:

```
Syntax:
type *pName;
```

- Pointer set to address of variable:

```
Syntax:
pName = &var;
```

- NULL Pointer: points to nothing:

```
Syntax:
type *pName = 0;
```

- We access the structure's element by using the dot '.' Operator.

Syntax:

Structures-variable.elem1;

- A Bit-Fields:

It is a bit-based structure member.

Syntax:

```
struct structName
{
    type field1 : length ;
    type field2 : length ;
    type field3 : length ;
    :
    :

    type fieldN : length ;
};
```

Enumerator:

-Use **enum** to create a set of integral constants:

Syntax:

```
enum [ENUM_NAME]
{
    CONST0 = value,
    CONST1 = value,
    CONST2 = value,
    :
    :
    CONSTn = value
};
```

- Inline function: is a small function whose code is expanded in line rather than called:

Syntax:

```
inline type funcName([param...list])
{
    [return value;] //if needed
}
```

- Friend function: is a non-member function which can access the private members of a class:

Syntax:

```
friend type frName([param...list])
{
    [return value;] //if needed
}
```

- Virtual function: Is declared as **virtual** in the base-class and redefined in the derived class(es).

Syntax:

```
virtual type vrFun([param...list])
{
    [return value;] //if needed
}
```

Union:

-Use **union** to create a union which is comprised of two or more variables that share the same memory location:

Syntax:

```
union uName
{
    [modifier] type vName;
    [modifier] type vName;
};
```

- Use * operator to access the value itself:

Syntax:

*anotherVar = *pName;*

- A Constant Pointer: it must be initialized:

Syntax:

*type * **const** pName = &var;*

- A pointer to a constant:

Syntax:

const *type * pName;*

- A constant pointer to a constant: can be initialized to point to a constant or a non-constant value:

Syntax:

const *type * **const** pName = &Iv;*

Arrays:

-Creating an Array:

Syntax:

```
type ArrayName[size];
//2-dimentional array
ArrayName[int-row][int-column];
```

-We can create an array with its element in one shot without specifying the size:

Syntax:

```
type ArrayName[] =
    {elem0, elem1,...elemN};
//2-dimentional array
type ArrayName[row][column] =
    { elem0, elem1,...elemN};
```

-Use index technique to access an individual element:

Syntax:

```
ArrayName[int_index] = value;
//2-dimentional array
ArrayName[row][column]= value;
```

Commonly Used Operators:

-Arithmetic Operators:

- +** Addition
- subtractions.
- *** Multiplication
- /** Division
- %** Modulus(remainder), integral use only

- Increment and Decrement: C++ supports both prefix and postfix:

- ++** Increment
- Decrement

- Rational Operators: always return Boolean value either **false** or **true**:

- ==** Equal to
- !=** Not equal to
- <** Less than
- <=** Less than or equal
- >** Greater than
- >=** Greater than or equal

- Assignment Operators: Used as a short cut for some operations such as `x = x + 20;` `x += 20;`. Except "=" operator it assigns the *rVal* to *lVal* such as assigning a value to a variable:

- =** Assignment
- +=** Addition assignment
- =** Subtraction assignment
- *=** Multiplication assignment
- /=** Division assignment
- %=** Modulus assignment

- Logical Operators: Used in conjunction with the rational operator in boolean expressions and comparison:

- &&** AND
- ||** OR
- !** NOT

- Bitwise Operator: Used to operate in bit(s) (Integral Use Only):

- &** AND
- |** OR
- ^** XOR (Exclusive OR)
- ~** NOT
- <<** Left Shift
- >>** Right Shift

Variables:

- Variable Declaration:

Syntax:
`[specifier, qualifier, modifier] type varName;`

- Variable Initialization:

Syntax:
`[specifier, qualifier, modifier] type varName = value;`

Class:

- Class Prototype:

Syntax:
class *ClassName*
{
 private data and functions

 public:
 public data and functions
}[*object-list*];

- Use dot '.' Operator to access object's data members and member functions via the object name:

Syntax:
obj.*[data,func]*...

- Allocate object(s):

Syntax:
*ClassName**pObj=**new** *ClassName*(*[param..list]*);

- Accessing data through the pointers is done by using the arrow operator (->).

Syntax:
ptrObj->*[data,function]*...

- Deallocate object(s) and free memory:

Syntax:
delete *ptrObj*;
or
delete[*[index]*] *ptrAObj*;//Dealocate array of obj

Another Various Operators:

- Conditional Operator: Used to compare and assign expression:

Syntax:
Var = *cmpExpr* ? *whenTrue* : *whenFalse*;

- The **sizeof** operator: Determines size in bytes:

Syntax:
sizeof(*obj*);

If...else Statement:

- Single Statement Form:

Syntax:
if (*Boolean-expr*)
 statement;
else
 statement;

- Single Statement Form with else if:

Syntax:
if (*Boolean-expr*)
 statement;
else if (*Boolean-expr*)
 statement;
else
 statement;

- Multi-Statement Form: Use { }

Syntax:
if (*Boolean-expr*)
{
 statement1;
 :
}
else{
 statement;
 :
}

References:

The power of reference is shiny in passing by reference technique.

- An Aliase: a reference provides another name for a variable in the same scope. It must be initialized:

Syntax:
type &*refName* = *varName*;

- Assigning Value:

Syntax:

```
varName = value;
```

- Use “,” operator to Declare more than one variable of the same type:

Syntax:

```
[specifier, qualifier, modifier] type v1, v2,...v[= v];
```

For Loop

- Single Statement:

Syntax:

```
for ([initialization] ; [test] ; [update])  
    Statement;
```

- Multi-statement: Use {...} :

Syntax:

```
for ([initialization] ; [test] ; [update])  
{  
    Statement1;  
    :  
}  
}
```

Switch Statement:

Syntax:

```
switch ( Integer-expr )  
{  
    case integer-constant:  
        statement(s) ;  
        [break;]  
    case integer-constant:  
        statement (s);  
        [break;]  
        :  
        :  
    [default:  
        statement(s);  
        break;]  
}
```

While Loop

- Single Statement:

Syntax:

```
While ( Boolean-expr )  
    Statement;
```

- Multi-statement: Use {...} :

Syntax:

```
while ( Boolean-expr )  
{  
    Statement1;  
    :  
}  
}
```

Exception Handling: It is a technique in which your program can manage run-time errors.

Syntax:

```
try  
{  
    //try block  
    [throw eType;]  
}  
catch( type1 e )  
{  
    //catch block  
}  
catch( type2 e )  
{  
    //catch block  
}  
catch( typeN e )  
{  
    //catch block  
}  
}
```

- You can throw any exception object from within the try block using **throw** keyword:

Syntax:

```
throw eType;
```

- Passing by reference:

Syntax:

```
(type &...)
```

- Passing constant reference:

Syntax:

```
(const type &...)
```

Do...while Loop

Syntax:

```
do  
{  
    Statement(s);  
    :  
} while ( Boolean-expr );
```

C++ Operators Precedence

Operator	Type	Associativity
::	binary scope resolution	left to right
::	unary scope resolution	
()	parentheses	left to right
[]	array subscript	
.	member selection via object	
->	member selection via pointer	
++	unary postfix increment	
--	unary postfix decrement	
typeid	runtime type information	
dynamic_cast < type >	runtime type-checked cast	
static_cast < type >	compile-time type-checked cast	
reinterpret_cast < type >	cast for nonstandard conversions	
const_cast < type >	cast away const -ness	
++	unary prefix increment	right to left
--	unary prefix decrement	
+	unary plus	
-	unary minus	
!	unary logical negation	
~	unary bitwise complement	
sizeof	determine size in bytes	
&	address	
*	dereference	
new	dynamic memory allocation	
new[]	dynamic array allocation	
delete	dynamic memory deallocation	
delete[]	dynamic array deallocation	
(type)	C-style unary cast	right to left
.*	pointer to member via object	left to right
->*	pointer to member via pointer	
*	multiplication	left to right
/	division	
%	modulus	
+	addition	left to right
-	subtraction	
<<	bitwise left shift	left to right
>>	bitwise right shift	
<	relational less than	left to right
<=	relational less than or equal to	
>	relational greater than	
>=	relational greater than or equal to	
==	relational is equal to	left to right
!=	relational is not equal to	
&	bitwise AND	left to right
^	bitwise exclusive OR	left to right
	bitwise inclusive OR	left to right
&&	logical AND	left to right
	logical OR	left to right
?:	ternary conditional	right to left
=	assignment	right to left
+=	addition assignment	
-=	subtraction assignment	
*=	multiplication assignment	
/=	division assignment	
%=	modulus assignment	
&=	bitwise AND assignment	
^=	bitwise exclusive OR assignment	
=	bitwise inclusive OR assignment	
<<=	bitwise left-shift assignment	
>>=	bitwise right-shift assignment	
,	comma	left to right