Informatics

GianAndrea Müller

May 13, 2018

Contents

1 How to...

1.1 ... use this summary

This summary is an overview of the functionality of `C++` in connection with the informatics course for mechanical engineers. It covers the content of the lectures but also contains additional information.
To emphasize the connection to the lecture all chapters containing purely additional information are marked in blue.

1.2 ... correct compilation errors

Read error messages, review basic syntax, look for the additional semicolon.

1.3 ... correct runtime errors

Use a debugger.

1.4 ... approach problems

1. Define your problem.
2. Find your algorithm.
3. Code feature.
4. Compile.
5. `goto` 3.

1.5 ... find more information

- Comprehensive Tutorial
- User friendly documentation
- Extensive technical documentation

Terms

Algorithm An algorithm is a set of rules that defines a sequence of operations to get to the solution of a problem.

Language A programming language is a set of instructions for a computer that can be used to write programs that implement algorithms.

Syntax The syntax of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language. Is a program grammatically correct?

Semantics The semantics of a computer language define how the language has to be interpreted. What is the meaning of a certain program?

Editor A program that allows writing code. There exist powerful editors that can check syntactical correctness on the fly.

Compiler A compiler translates a program written in a programming language to machine code, such that it can be executed by the machine.

Computer A computer is a device that is capable of executing machine code.

Comments Comments document the implemented algorithm within the program for the reader. They are ignored by the compiler.

Include Directives Include directives specify the additional libraries used in a program.

The main function The main function exists in every cpp-program. It is unique and contains the all instructions necessary to execute the program.

Statement Statements are the building blocks of a program. They are executed sequentially and end with a semicolon.

Declaration A declaration introduces a new name to the program.

Definition A definition introduces a body to a name within the program.

Initialization An initialization introduces a value to a defined name and body.

Literals Literals represent constant value within the program. They have a defined type and value.

Variables Variables represent possibly changing values within the program. They have name, type, value and address.

Objects Objects represent values in the computer memory. They have type, adress and value. They can be named, but can also be anonymous. Described less generally an object can be a variable, a data structure, a function, or a method.

Expressions Expressions represent calculations. They are a combination of values, literals, operators and functions. They are primary if they consist of a single name/literal. Otherwise they are compound. They have type and value.

Lvalue An lvalue is a changeable expression that has an address.

Rvalue An rvalue is an expression that is not an lvalue. An rvalue cannot be changed. Every lvalue can be used as an rvalue but not vice-versa.

Operator An operator connects expressions to compound expressions. It specifies the expected operand in type and if it is an rvalue or an lvalue. Operators have an arity.

Arity Arity is the number of arguments or operands an operator or a function takes. For example there exist unary and binary operators.

Block A block in c++ is a number of lines of code enclosed by curly brackets.

2 Nice to know

2.1 Preprocessor directives

Preprocessor directives are lines preceded by a `#`. These lines are not program statements but directives for the preprocessor, thus are evaluated before the program is compiled. These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is ends. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

2.1.1 Macro Definitions

```
#define identifier replacement
```

When the preprocessor encounters this directive it replaces any occurrence of `identifier` in the rest of the code by `replacement`. A macro lasts until it is undefined with `#undef`. As seen in the following example it is also possible to define functions:

```
//A simple constant
#define TABLE_SIZE 100
int table1 [TABLE_SIZE];
#undef TABLE_SIZE //lasts until here

//A function
#define getmax(a,b) a>b?a:b

int main(){
  int x = 5, y;
  y = getmax(x,2); //replaced as: y = x>2?x:2
}

}
```

This would replace any occurrence of `getmax` followed by two arguments by the replacement expression, but also replace each identifier by its respective argument.

```
#define str(x) #x

cout<<str(test); // replaced as: cout<<"test";
```

As seen above an identifier preceded by `#` will be replaced by the argument in double quotes.

```
#define glue(a,b) a ## b
glue(c,out) << "test"; //replaced as: cout << "test";
```

The operator `##` concatenates two arguments leaving no white space between them.

2.1.2 Conditional inclusions

The directives `#ifdef`, `#ifndef`, `#if`, `#endif`, `#else` and `#elif` allow to include or discard part of the code if a certain condition is met.

`#ifdef` allows a section of a program to be compiled only if the macro that is specified as the parameter has been defined, no matter which its value is. `#endif` ends the conditional block.

```
#ifdef TABLE_SIZE
int table[TABLE_SIZE];
#endif
```

`#ifndef` serves for the exact opposite: the code between `#ifndef` and `#endif` directives is only compiled if the specified identifier has not been defined yet.

```
#ifndef TABLE_SIZE
#define TABLE_SIZE 100
#endif
int table[TABLE_SIZE];
```

The `#if`, `#else` and `#elif` directives serve to specify some condition to be met in order for the portion of code they surround to be compiled. The condition that follows `#if` and `#elif` can only evaluate constant expressions, including macro expressions.

```
#if TABLE_SIZE>200
#undef TABLE_SIZE
#define TABLE_SIZE 200

#elif TABLE_SIZE<50
#undef TABLE_SIZE
#define TABLE_SIZE 50

#else
#undef TABLE_SIZE
#define TABLE_SIZE 100
#endif

int table[TABLE_SIZE];
```

Notice that the compelling advantage of preprocessor directives over normal conditional statements is that preprocessor directives are evaluated before the code is compiled. An interesting application of that concept is the making of different versions of a program, for instance one version that has special debugging precautions and a second version which runs faster but omits these safety measures.

2.1.3 Line control

When we compile a program and some error happens during the compiling process, the compiler shows an error message with references to the name of the file where the error happened and a line number, so it is easier to find the code generating the error.

The `#line` directive allows us to control both things, the line numbers within the code files as well as the file name that we want that appears when an error takes place.

```
//#line number "filename"

#line 20 "assigning variable"
int a?;
```

This code will generate an error that will be shown as error in file `"assigning variable", line 20`.

2.1.4 Error directive

The `#error` directive aborts the compilation process when it is found, generating a compilation error that can be specified as its parameter.

```
#ifndef _cplusplus
#error A C++ compiler is required!
#endif
```

2.1.5 Source file inclusion

The `#include` directive is replace by the entire content of the specified header or file. There are two ways to use `#include`:

```
#include <header>
#include "file"
```

In the first case, a header is specified between angle-brackets <>. This is used to include headers provided by the implementation, such as the headers that compose the standard library (iostream, string,...). Whether the headers are actually files or exist in some other form is implementation-defined, but in any case they shall be properly included with this directive.

The syntax used in the second `#include` uses quotes, and includes a file. The file is searched for in an implementation-defined manner, which generally includes the current path. In the case that the file is not found, the compiler interprets the directive as a header inclusion, just as if the quotes (“”) were replaced by angle-brackets (<>).

2.1.6 Pragma directive

This directive is used to specify diverse options to the compiler. These options are specific for the platform and the compiler you use. Consult the manual or the reference of your compiler for more information on the possible parameters that you can define with `#pragma`.

2.1.7 Predefined macro names

`__LINE__`	Integer value representing the current line in the source code file being compiled.
`__FILE__`	A string literal containing the presumed name of the source file being compiled.
`__DATE__`	A string literal in the form “Mmm dd yyyy” containing the date in which the compilation process began.
`__TIME__`	A string literal in the form “hh:mm:ss” containing the time at which the compilation process began.
`__cplusplus`	An integer value. All C++ compilers have this constant defined to some value. Its value depends on the version of the standard supported by the compiler.

3 Positional Notation

4 Syntax

4.1 Basic program

```cpp
#include <iostream>
//#include "local_header_file.h"

/*
 * Function declarations (and definitions)
 */

int main(int argc, char ** argv)
{
    /*
     * Function calls
     */
    std::cout << "Hello World!" << std::endl;
    return 0;
}

/*
 * Function definitions
 */
```

4.2 Identifiers

A valid identifier, i.e. the name of a variable is:

- an arbitrarily long sequence of digits, underscores, lowercase and uppercase Latin letters.
- not starting with a digit.
- not starting with two or more underscores.
- not starting with an underscore followed by a capital letter.
- not a keyword of cpp.

More information on Identifiers.

4.3 Comments

C++ allows masking code such that it is not interpreted as part of the program. This enables documenting the program. There are different possibilities:

```cpp
// normal comment

/*
multi
line
comment
*/
```

Both versions can be nested:

```cpp
///*comment in a comment*/

/*
cout<<"Hello World!<<endl; //comment in a comment
*/
```

4.4 Data Types

4.4.1 Primitive Types

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void

4.4.2 Type modifiers

There exist a number of type modifiers:

Modifier	Effect
signed	variable interpreted as signed
unsigned	variable interpreted as unsigned
short	half number of allocated bits if possible
long	double number of allocated bits if possible

Based on the primitive types and their modifiers the spectrum of available types can be established. Their sizes differ depending on compiler and environment.

Modifier	Typical Bit Width	Typical Range
char	1byte	-127 to 127
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4byte	-2'147'483'648 to 2'147'483'647
unsigned int	4bytes	0 to 4'294'967'295
signed int	4bytes	-2'147'483'648 to 2'147'483'647
short int	2bytes	-32'768 to 32'767
unsigned short int	2bytes	0 to 65'535
signed short int	2bytes	-32'768 to 32'767
long int	4bytes	-2'147'483'648 to 2'147'483'647
signed long int	4bytes	-2'147'483'648 to 2'147'483'647
unsigned long int	4bytes	0 to 4'294'967'295
float	4bytes	+/- 3.4e +/- 38 (7 digits)
double	8bytes	+/- 1.7e +/- 308 (15 digits)
long double	8bytes	+/- 1.7e +/- 308 (15 digits)

4.4.3 Find type sizes on your system

```cpp
#include <iostream>
using namespace std;

int main() {
    cout<< "Size: char : "<<sizeof(char)<<endl;
    cout<< "Size: int : "<<sizeof(int)<<endl;
    cout<< "Size: short int : "<<sizeof(short int)<<endl;
    cout<< "Size: long int : "<<sizeof(long int)<<endl;
    cout<< "Size: float : "<<sizeof(float)<<endl;
    cout<< "Size: double : "<<sizeof(double)<<endl;

    return 0;
}
```

4.4.4 Type qualifiers: const volatile restrict

`const`	Objects of type `const` cannot be changed by the program during execution.
`volatile`	The modifier `volatile` tells the compile that a variable's value may be changed in ways not explicitly specified by the program. (Imagine for instance manually changing the position of a switch on a microprocessor.)
`restrict`	A pointer qualified by `restrict` is initially the only means by which the object it points to can be accessed.

4.4.5 Storage classes

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify.

auto The auto storage class is the default storage class for all local variables.

register The register storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).
The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

static The static storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

extern The extern storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.
When you have multiple files and you define a global variable or function, which will be used in other files also, then extern will be used in another file to give reference of defined variable or function. Just for understanding extern is used to declare a global variable or function in another file.
The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

mutable The mutable storage class only applies to non-static class members of non-reference and non-const type. When a class member is declared mutable it can be changed by const member functions.

4.4.6 Variable Definition

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and name. Type has to be either a primitive type or a user defined object. Multiple names can be introduced at once if separated by commas.

```
int i,j,k;
```

This line both declares and defines the variables i,j and k. Direct initialization is also possible:

```
int d = 3, f = 2;
```

If a variable is left uninitialized its value is undefined in general. However, variables with static storage duration are implicitly set to 0.

4.4.7 Variable Declaration

It is possible to declare a variable without defining it. The declaration is accepted during compilation but has to be fitted with a definition at the time of linking of the program. This means that if a program consists of multiple files you can declare your variable wherever you need it but only define it once, since multiple definitions of the same variable are prohibited.

```
//Variable declaration
extern int a,b; //Compiler knows that the variable exists

//Variable definition
int a,b; //Compiler knows that the variable exists
         //And allocates the storage space needed
```

4.4.8 typedef and using

You can create a new name for an existing type with `using` or `typedef`:

```
//typedef type newname;

typedef unsigned int uint;

//using newname = type;

using uchar = unsigned char;

uint a = 3; //a is an unsigned int

uchar b = '2'; //b is an unsigned char
```

4.4.9 union

A union is a special class type that can hold only one of its non-static data members at a time. The union is only as big as necessary to hold its largest data member. The other data members are allocated in the same bytes as part of that largest member. The details of that allocation are implementation-defined, and it's undefined behavior to read from the member of the union that wasn't most recently written.

4.4.10 enum

An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. This can improve readability of the code as values are given meaning.

```
//enum enum-name {list of names} var-list;

enum day {mon, tue, wed, thu, fri, sat, sun} today;
today = mon; //today == 0
today = sun; //today == 6
```

The values assigned to the names start with zero and increment by default. But you can also directly assign values:

```
enum color {red, green = 5, blue};
color c;
c = red;    //c == 0
c = green;  //c == 5
c = blue;   //c == 6
```

5 Variable Scope

A scope is a region of the program within which variable definitions persist. There are three types of scopes:

1. Inside a function or a block (local variables)
2. In the definition of function parameters (formal parameters)
3. Outside of all functions (global parameters)

5.1 Local Variables

Variables that are defined within a function or a block can only be used by statements within that same function or block.

```
for(int i = 0; i<3; i++){ //scope of the variable i
  std::cout<<i<<" ";
} //end of scope

// std::cout<<i; will result in error
```

A variable can be redefined within a block. In that case the "closer" definition is used.

```
int i = 5;
{
  int i = 3;
  std :: cout << i; // outputs 3
}
std :: cout << i; // outputs 5
```

5.2 Global Variables

Global variables are defined outside of all the functions, usually on top of the program. They will hold their value throughout the life-time of your program. This also mean that all functions within your program can access these variables.

Including global variables the case of maximum complication, however undesired, is the following:

```
#include <iostream>
using namespace std;

int i = 2;

void fun(){
  cout<<i;
}

int main(){
  int i = 5;
  {
    int i = 3;
    std :: cout << i;    // outputs 3
  }
  std :: cout << i;      // outputs 5
  fun();                 // outputs 2
}
```

6 Literals

6.1 Integer Literals

An integer literal can be a binary, decimal, octal or hexadecimal constant. A prefix specifies the base: `0b` for binary, `0` for octal, `0x` for hexadecimal and nothing for decimal.
An integer literal can also have a suffix that is a combination of u (unsigned) and l (long).

```
212      // Decimal number
212ul    // Long unsigned decimal number
0xFeeL   // Long hexadecimal number
0b101    // Binary representation of 5
011      // Octal representation of 9
078      // Illegal: 8 is not an octal digit
```

6.2 Floating-point Literals

A floating point literal has an integer part, a decimal point, a fraction part and an exponent part. You can represent floating point literals either in decimal form or exponential form. While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

```
3.14195       // Decimal representation of pi
314195E-5L    // Exponential representation of pi
510E          // Illegal: incomplete exponent
210f          // Illegal: no decimal or exponent
.e55          // Illegal: missing integer or fraction
```

6.3 Boolean Literals

There are two Boolean literals and the are part of standard C++ keywords: `true` and `false`.

```
bool a = true;
cout<<a; //outputs 1
a = false;
cout<<a; //outputs 0
bool = some_integer; //true for some_integer != 0
```

The conversion from integer to bool results in `true` for all nonzero values and in `false` for zero.

6.4 Character Literals

Character literals are enclosed in single quotes (’). A character literal can be a plain character (’x’), an escape sequence (’`\t`’) or a universal character (’`\u02C0`’).

\\	\\ character
\’	’ character
\"	” character
\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab

6.5 String Literals

String literals are enclosed in double quotes. A string contains any combination of plain characters escape sequences and universal characters.

7 Operators

7.1 Arithmetic Operators

+	Adds two operands
-	Subtracts the second operand from the first
*	Multiplies both operands
/	Divides the first operand by the second
%	Returns the remainder of an integer division of the two operators
++	Increases an integer value by one
--	Decreases an integer value by one

7.2 Relational Operators

==	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

7.3 Logical Operators

&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.

7.4 Bitwise Operators

Bitwise operators work on bits and perform bit-by-bit operations. The truth tables for bitwise AND `&`, bitwise OR `|` and bitwise exclusive OR (XOR) `^` are:

p	q	p&q	p|q	p^q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

`&`	Binary AND Operator copies a bit to the result if it exists in both operands.
`|`	Binary OR Operator copies a bit if it exists in either operand.
`^`	Binary XOR Operator copies the bit if it is set in one operand but not both.
`~`	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
`<<`	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
`>>`	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

7.5 Assignment Operators

`=`	Simple assignment operator, Assigns values from right side operands to left side operand.
`+=`	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.
`-=`	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.
`*=`	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.
`/=`	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.
`%=`	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand.

7.6 Assignment Operators

`<<=`	Left shift AND assignment operator.
`>>=`	Right shift AND assignment operator.
`&=`	Bitwise AND assignment operator.
`^=`	Bitwise exclusive OR and assignment operator.
`|=`	Bitwise inclusive OR and assignment operator.

7.7 Misc Operators

`sizeof`	The sizeof operator returns the size of a variable in bytes. For example, sizeof(a), where 'a' is integer, and will return 4.
`?X:Y`	If Condition is true then it returns value of X otherwise returns value of Y.
`,`	The comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
`.`	The member operator `.` accesses a member of the operand on the left as specified by the operator on the right.
`->`	The member operator `->` is used to dereference the operand on the left and access one of its members indicated by the operand on the right.
Casts	Casting operators convert one data type to another. For example `int(2,200)` would return 2.
`&`	The address operator returns the address of a variable.
`*`	The dereference operator returns the value the operand points to.

7.8 Operator Precedence and Associativity

P.	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right
2	`a++ a--`	Suffix/postfix increment and decrement	
	`type() type{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Locigal NOT and bitwise NOT	
	`(type)`	C-style cast	
	`*a`	Dereference	
	`&a`	Adress-of	
	`sizeof`	Size-of	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	
4	`.* ->*`	Pointer-to-member	Left-to-right
5	`a*b a/b a%b`	Multiplication, division, remainder	
6	`a+b a-b`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator	
9	`< <= > >=`	Relational operators	
10	`== !=`	Relational operators	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR	
13	`|`	Bitwise OR	
14	`&&`	Logical AND	
15	`||`	Logical OR	
16	`a?b:c`	Ternary conditional	Right-to-left
	`throw`	throw operator	
	`=`	Direct assignment	
	`+= -=`	Compound assignments	
	`*= /= %=`		
	`<<= >>=`		
	`&= ^= |=`		
17	`,`	Comma	Left-to-right

7.8.1 How to use this table

```
1  cout<<a&&b;     //(cout<<a)&&b;
2
3  *p++            //*(p++);
4
5  a = b = c = d; //a = (b =(c = d)));
6
7  a + b - c;      //(a + b) - c;
8
9  delete ++*p;    //delete(++(*p))
```

1. By its precedence `<<` is evaluated before `&&`.
2. By its precedence `++` is evaluated before `*`.
3. Operators with the same precedence are evaluated based on their associativity. For right-to-left associative operators as `=`, the evaluation proceeds from right to left.

   Thus the assignments made in line 5 are in the order of their execution: `c = d;` which returns a reference to `c`, `b = c;` which returns a reference to `b` and `a = b;`.
4. Operators with the same precedence are evaluated based on their associativity. For left-to-right associative operators as `+` and `-` the evaluation proceeds from left to right.
5. `++()`, `*()` and `delete` have the same precedence, and are thus evaluated based on their associativity, which is right-to-left. Therefore `++()` is evaluated after `*()` and `delete` is evaluated last.

If written badly expression can result in undefined behaviour:

```
1  f(++i, ++i);
2  n = ++i + i;
3  b = ++a - a++;
```

Avoid changing variables which are used again in the same expression.

8 Loop Types

8.1 while
8.2 for
8.3 do...while
8.4 Loop Control Statements

9 Conditional Statements