C++ syntax-only reference card v.0.28

P. Areias (UEVORA) August 24, 2013

1 Basic considerations

1.1 Notation

≺•≻ indicates that the programmer must provide an appropriate replacement.
For example ≺expr≻ can be replaced by 10.0*sin($pi/6$)
[•] indicates that the programmer can either provide a replacement or opt to omit it.
For example ≺T≻ ≺aT≻[=≺expr≻] can be replaced by either double d or double d=3.0
•|⋆ indicates an option for the programmer.
For example const ≺T≻ ≺aT≻ =≺expr≻|(≺expr≻) can be replaced by either const bool b=true or const bool b(x==3)

Case-sensitivity and semicolon: C++ is a case-sensitive language, there are (usually) no continuation lines and the end of a statement (except a block statement) is marked with a semicolon (;)

1.2 Comments and main segment

```
// comment terminating with a new-line character
/* comment terminating with the symbol */

int main()                      // argument-free version main segment
{[≺statements≻]}                // corresponding statement block
int main(int argc, char **argv) // main segment with arguments, argc is the number of arguments passed to program and, if not zero,
                                // char[0]..char[argc-1] are pointers to character sequences containing command-line arguments
{[≺statements≻]}                // corresponding statement block
```

Relevant utilities:
Premature termination can be forced by using the function exit() from the header #include<cstdlib>.
A system command can be executed by using the function system("≺command≻") available from the header #include<cstdlib>.
Assertions can be established by using the macro assert(≺boolean_expr≻) from the header #include<cassert>.

1.3 File inclusion and namespace use

```
#include <≺system_header≻>          // inclusion of a system header
#include "≺file≻"                   // inclusion of a user header (the extension .hpp is common for ≺file≻)
using namespace ≺namespace_name≻;   // provides access to the scope of a given namespace (in this case ≺namespace_name≻)
```

For example:

```
using namespace std;                // provides access to the Standard namespace
#pragma once                        // ensures that the corresponding header is included once (non-standard but usually accepted)
```

Source code files and header files (combined files sent to linker):

```
≺file≻ with ≺cpp≻ extension         // Source code file: contains stand alone source code to be compiled (other extensions possible!)
≺file≻ with ≺hpp≻ extension         // Header file: contains source code intended for inclusion (other extensions possible!)
extern "C" {≺C functions≻}          // functions compiled by C typically enclosed by the curly braces
```

For example:

```
extern "C" {int mult(int* i,int* j);}   // example of a C function prototype
extern "C" { #include "header.h" }      // example of a C header
```

2 Types and declarations

2.1 Fundamental types

≺C≻ is either a fundamental type or a type introduced by the user[1].
Common fundamental types:
void, char, int, bool, double, float
auto: in the context of variable initialization, leaves to the compiler the decision to determine the appropriate type
For sizes of containers, the header #include<cstdlib> includes the type size_t.

Type use: The keyword decltype can be used to determine the type of an expression:

```
decltype(≺expr≻) ≺avariable≻;
decltype(≺expr≻) ≺areference≻=≺expr≻;
```

For example[2]:

```
typename vector<int>::iterator aiterator=v.begin() // can be replaced by auto aiterator=v.begin()
decltype(adouble) anotherdouble;                   // make use of an existing type
decltype((adouble)) adoublereference=adouble;      // another use of an existing type
```

Examples of declarations:

```
int       ≺aint≻;          // integer variable declaration
size_t    ≺aunsigned≻;     // non-negative integer size_t is a type synonym (requires cstdlib) for a unsigned integer
char      ≺achar≻;         // character variable declaration ('a', 'b', ...)
float     ≺afloat≻;        // real (32 bits in many implementations) variable declaration
double    ≺adouble≻;       // real (64 bits in many implementations) variable declaration
bool      ≺abool≻;         // boolean variable declaration (values are either true or false)
auto      ≺aauto≻=≺expr≻;// the type is inferred by the compiler from the type of ≺expr≻;
```

[1] In the present context, these can be struct and enum
[2] auto can effectively replace most typename specifiers

2.2 Typical declaration on the stack

```
[static] ≺T≻ ≺aT≻ [(≺LE≻) | =≺C≻() | =≺expr≻];  // declares a type where ≺C≻() invokes the default constructor
```

Examples:

```
static int aint(10);                                // global int variable initialized with the value 10
float afloat=30.0;                                  // float variable initialized with the value 30.0
double adouble=double();                            // local double variable initialized with the default value, 0.0
complexclass acomplex(0.0,1.0);                     // hypothetical complex number initialized as a unit imaginary
```

≺T≻ is given in general as

```
[const] ≺C≻ [♣ [const]]
```

The above keywords are:

```
static                                              // maintains the variable value after leaving the scope of declaration
                                                    // it automatically initializes to the default value
const                                               // qualifier that, as prefix, forces the variable to hold a constant value
                                                    // (attempts to change the value by de-referencing are flagged as errors at compile time)
const                                               // qualifier that, as suffix, declares the address stored in the pointer to be constant
```

Warning: Temporary variables are forced to be constant (const) by the compiler

The symbol ♣ is one of the following:[3]

```
*      // pointer, when an address (and type) of a variable is stored, only use for heap allocation
       // with 'new' and when a 0 can possibly be returned
&      // reference, when an alias of a given variable is stored, requires initialization
&&     // rvalue reference, similar to & but allowing the assignment of temporaries
*&     // reference of a pointer, when an address stored at a pointer is allowed to change,
       // useful to pass pointers to functions with new and delete operations
```

The list of sent expressions ≺LE≻ is an ordered comma-separated set of N expressions[4].
If N==0 parenthesis are omitted around ≺LE≻.
For N expressions, with N≠0,
≺LE≻≡ ≺expr_0≻,≺expr_1≻,...,≺expr_N-1≻

Examples:

```
const double adouble(3.0);           // declares a constant double with the value 3
const int* apint=&aint;              // declares a pointer to a constant integer
double* const apdouble=new double(); // declares a constant pointer to a double
bool& abool2=abool1;                 // declares and initializes a boolean reference
const bool& abool3=false;            // declares and initializes a constant boolean reference
```

2.3 Array in the stack

```
≺T≻ ≺array≻[≺n≻];  // declares an array containing, sequentially, ≺n≻ elements of type ≺T≻.  For large arrays, heap allocation
                  // is recommended.  Multidimensional arrays are also possible, but seldom needed.
```

2.4 Heap space management

Space in the heap will have to be accessed by variables which are declared in a given local scope.
≺apT≻ and ≺arrayT≻ are pointers of type ≺T≻ and their declarations are:

```
≺T≻* ≺apT≻=0;              // declares a pointer to ≺T≻ and sets it to 0
≺T≻* ≺arrayT≻=0;           // also declares a pointer to ≺T≻ but additional space will be reserved, the heap space is managed as:
[≺apT≻]=new ≺T≻[(≺LE≻)];// reserves space for a variable of type ≺T≻
...                        // uses ≺apT≻
delete ≺apT≻;              // frees the space which was previously reserved
≺apT≻=0                    // do not free twice, set to zero after deletion
≺arrayT≻=new ≺T≻[≺n≻];     // reserves space for an array of type ≺T≻ with ≺n≻ elements
...                        // uses ≺arrayT≻
delete[] ≺arrayT≻;         // frees the space which was previously reserved
≺arrayT≻=0                 // do not free twice, set to zero after deletion
```

Smart pointers (the header #include<memory> is required, see also RAII for a generalization):

```
unique_ptr<≺T≻> ≺apT≻=new ≺T≻[(≺LE≻)];           // declares a single-owned smart pointer, container-friendly
scoped_ptr<≺T≻> ≺apT≻=new ≺T≻[(≺LE≻)];           // declares a single-owned smart pointer with transference on copy
shared_ptr<≺T≻> ≺apT≻=new ≺T≻[(≺LE≻)];           // declares a reference-counted smart pointer, container-friendly
```

Destroy utility function for standard pointers:

```
template<typename ≺T≻> inline void destroy(≺T≻*& ≺apT≻){if(≺apT≻)delete ≺apT≻; ≺apT≻=0;}
```

2.5 Type synonyms or 'typedefs'

```
typedef ≺T≻ ≺Tsynonym≻       // this specifier declares the identifier ≺Tsynonym≻ as a synonym of ≺T≻
                             // as it is often easier to write ≺Tsynonym≻.  Can also be used in structs.  An example of use of synonyms:
typedef int* pint;           // pointer to int also identified as pint
pint apint=new int;          // typical use of pint
```

2.6 Enumerations

Enumerations are objects which can have a finite number of values, for which synonyms are defined.

Definition of a enumeration:

```
enum ≺E≻
{
≺name_0≻[=≺integer_expr_0≻],
≺name_1≻[=≺integer_expr_1≻],
...
};
```

Declaration and use of a enum object:

```
≺E≻ ≺aE≻;
≺aE≻ =≺integer_expr_I≻;
```

---

[3] Double, triple, etc pointers, **, ***, also exist but are not necessary with C++ since Standard Library containers can be nested and are preferable.

[4] The (≺LE≻) option may fail in certain odd cases, in that case an extra set of parenthesis must be added.

2.7 Unions

Unions are amalgamations of data. Memory sharing is allowed for certain data types (C++11).

Definition of a union:

```
union ≺U≻
{
≺C_0≻ ≺aC_0≻;
≺C_1≻ ≺aC_1≻;
...
};
```

Declaration and use of a union object:

```
≺U≻ ≺aunion≻;
≺aunion≻=≺compatible_expr≻;
```

2.8 Type identification during run time - RTTI

The header <typeinfo> provides the functions typeid(≺T≻) and typeid(≺expr≻). This should only be used for comparisons.

Examples of application:

```
if(typeid(complex)!=typeid(notacomplex))cout<<"Not a complex";
cout<<typeid(complex).name()<<endl; // compiler-dependent name
```

3 Operators and expressions

3.1 Nomenclature of Rexpr and Lexpr

```
≺Lexpr≻          // an expression that, const apart, can be used either in the left or right-hand-side of an attribution
≺Rexpr≻          // an expression which cannot be used, even if non-constant, in the left-hand-side of an attribution
≺expr≻           // a general expression, either ≺Rexpr≻ or ≺Lexpr≻
≺boolean_expr≻   // an expression which has a boolean value or one that can be implicitly converted to a bool
≺integer_expr≻   // an expression which has an integer value
```

3.2 Typical operations (≺a≻, ≺b≻ and ≺c≻ are expressions, ≺i≻ is an integer expression and ≺l1≻ and ≺l2≻ are boolean expressions)

```
≺a≻=≺b≻;                                  // assignment, the value stored in ≺a≻ will be lost and
                                          // ≺a≻ will now contain the value of ≺b≻
++≺i≻, --≺i≻, ≺i≻++, ≺i≻--;               // pre-increment, pre-decrement, post-increment and post-decrement
!≺l1≻, ≺l1≻||≺l2≻, ≺l1≻&&≺l2≻;            // negation, "or" and "and". The result is a boolean
≺a≻==≺b≻, ≺a≻!=≺b≻;                       // equal, not equal
≺a≻>=≺b≻, ≺a≻<=≺b≻, ≺a≻<≺b≻, ≺a≻>≺b≻;     // greater than or equal, less than or equal, less than, greater than
≺a≻+≺b≻, ≺a≻-≺b≻, -≺a≻, +≺a≻;             // addition, subtraction, unary minus, unary plus
≺a≻*≺b≻, ≺a≻/≺b≻, ≺a≻%≺b≻;                // multiplication, division, remainder
≺a≻@=≺b≻;                                 // composition of attribution and any binary operator @ this is ≺a≻=≺a≻@≺b≻;
≺a≻(≺b≻);                                 // use of () in a function invocation
≺a≻*(≺b≻+≺c≻);                            // use of () in expression grouping
≺a≻.≺b≻;                                  // access to a member variable
≺a≻.≺b≻(≺c≻);                             // access to a member function
≺a≻->≺b≻(≺c≻);                            // access to a member function when ≺a≻ is a pointer
*≺a≻;                                     // value of ≺a≻: contents stored at address ≺a≻
&≺a≻;                                     // address of ≺a≻
≺a≻[≺i≻] or *(≺a≻+≺i≻) or ≺i≻[≺a≻];       // accesses the contents stored at address ≺a≻+≺i≻
```

3.3 Block statement

Syntactically, when one statement is expected and the programmer needs to insert more than one,
curly braces are used {} to create a block statement. For N statements, the block statement has the form:

```
{≺statement_0≻;≺statement_1≻;...;≺statement_N-1≻;}
```

Example:

```
if(a!=b)a=b;              // one statement expected from the if condition
if(a!=b)
{ a=b;
c=b; }                    // when two or more statements are needed, a block statement must be used
```

3.4 Old-fashioned type cast operator

For conciseness reasons, I prefer the old-fashioned C-style type cast operator.
The most common application of type cast occurs in pointers of structures (or classes) when inheritance is involved:
struct ≺C2≻:≺C1≻ and access member functions in ≺C2≻ not present in ≺C1≻:

```
≺C1≻* ≺aP1C1≻=new ≺C1≻;            // base allocated as base
≺C1≻* ≺aP1C2≻=new ≺C2≻;            // base allocated as derived
≺C2≻* ≺aP2C2≻=(≺C2≻*)≺aP1C2≻;      // cast to a ≺C2≻* pointer
```

The cast operator can invoke the overloaded cast operator if it is defined for a given struct or class.
Type conversion operations are performed at compile-time.

3.5 Sizeof operator

```
sizeof(≺C≻)     // returns the number of bytes of type ≺C≻
sizeof(≺expr≻)  // returns the number of bytes resulting from the evaluation of ≺expr≻
```

Examples:

```
sizeof(double)  // returns 8 in most cases
float f=3.0;
sizeof(f-33.0)  // returns 4 in most cases
```

3.6 Conditional (ternary) operator

```
≺boolean_expr≻?≺expr_1≻:≺expr_2≻;   // returns ≺expr_1≻ if ≺boolean_expr≻ evaluates to true and ≺expr_2≻ if not
                                     // note that ≺expr_1≻ and ≺expr_2≻ evaluations may be, in general, of different types
```

For example:
```
apint==0?int():*apint // returns the default value of int if apint is 0 or returns the value stored at apint's address.
```

4 Functions

4.1 Prototype and definition

```
≺T≻ ≺name≻(≺LP≻);                // function prototype or declaration
[inline] ≺T≻ ≺name≻(≺LR≻)        // definition (note the difference between ≺LR≻ and ≺LP≻ and the lack of semicolon)
{// ⟵ beginning of the function block
≺statements≻
[return ≺expr≻;]                 // return of ≺expr≻ only if ≺T≻ is not void
}// ⟵ end of the function block
```

≺LR≻ is the list of N received variables
If N==0 parenthesis are not omitted.
For N arguments, ≺LR≻ is given as:
≺T_0≻ ≺var_0≻[=≺expr_0≻],≺T_1≻ ≺var_1≻[=≺expr_1≻], ,...,≺T_N-1≻ ≺var_N-1≻[=≺expr_N-1≻]

≺LP≻ is the list of N received types
If N==0 parenthesis are not omitted.
For N arguments, ≺LP≻ is given as:
≺T_0≻,≺T_1≻,...,≺T_N-1≻

In ≺LR≻, the expressions ≺expr_i≻ provide the default value of the corresponding argument, and must satisfy:
If ≺expr_i≻ is present then ≺expr_i+1≻ must also be present. Strict reference arguments (&) cannot have default values.

Examples[5]:
```
void test(const int&,char*,const double&);            // declaration of a function named test
void test(const int& i,char* c,const double& d=100.0);// definition of test
{...}
double* newvector(size_t dim=10)
{...
 return new double[dim] ;                             // with return value
}
```

4.2 Function overloading

We can declare and define functions with the same name but with different lists of received types.
For example:

```
void print(const int& val)    // will be invoked when an int is used in ≺LE≻
void print(const double& val) // will be invoked when a double is used in ≺LE≻
```

4.3 Argument passing

Each ≺T_I≻ in ≺LP≻ and ≺LR≻ should have one of the following forms:

```
≺C≻         // passing by value, can receive any ≺expr≻ and will
            // not modify the argument (avoid except with reference-counting smart pointers)
const ≺C≻&  // passing by constant reference, can receive any ≺expr≻ and the expression will not be modified (inspector)
≺C≻&        // passing by reference, can only receive ≺Lexpr≻ and the expression can be modified (mutator)
≺C≻&&       // passing by rvalue reference, can receive any ≺expr≻, and the expression can be modified if it is a ≺Lexpr≻
≺C≻*&       // a reference to pointer, for new and delete operations inside the function, also ≺Lexpr≻
≺C≻*        // a pointer, also ≺Lexpr≻ but allowing a default value (useful for returning 0)
≺C≻* const  // a pointer const, inhibiting reseat (including new and delete).
```

4.4 Return value

The return value (≺T≻) can be:

```
≺C≻          // a copy of the calculated object is returned to the caller
[const] ≺C≻& // a reference to an argument or to a static or heap object is returned
[const] ≺C≻* // a pointer to an argument or to a heap variable defined inside the function is returned
≺C≻* const   // a pointer const, inhibiting new and delete of the result
```

4.5 Invocation

```
[≺Lexpr≻=]≺name≻(≺LE≻); // invocation of a function with a list of expressions
```
The list of sent N expressions, ≺LE≻, can be written as:
≺expr_0≻, ≺expr_1≻,..., ≺expr_N-1≻

When N==0 then parenthesis cannot be omitted in the invocation.
Each of the expressions must be ≺Lexpr≻ if the corresponding type in ≺LP≻ is either ≺C≻& or ≺C≻*& (see previous sub-section).
When default values are defined, then the corresponding argument can be omitted in the invocation.

4.6 Function pointers and references

Using pointers:
```
≺T≻ (*≺name≻)(≺LP≻);                    // declares ≺name≻ as a pointer to a function with ≺T≻ as return type
                                       // and ≺LP≻ as list of received arguments
≺name≻=&≺name_of_function≻;             // assigns an existing function (the & is not required)
[≺Lexpr≻=](*≺name≻)(≺LE≻);              // invokes the function
```
Using references:
```
≺T≻ (&≺name≻)(≺LP≻)=≺name_of_function≻;  // function reference
[≺Lexpr≻=](≺name≻)(≺LE≻);               // invokes the function
```

[5] One can declare a function inline to improve performance, but this is better left to the compiler

```
With type synonyms:
typedef ≺T≻ (*≺name≻)(≺LP≻);               // this allows the direct use of name as a function name:
≺name≻ ≺var≻;                               // declares ≺var≻ as a pointer to function
```

4.7 Template function

```
Templatization provides type parametrization (i.e.  in practice types will also be arguments)
Declaration and definition formats:
template <typename Typ1[=≺T1≻],typename Typ2[=≺T2≻],...> // note that Typ1 and Typ2 are type parameters
≺T≻ ≺name≻(≺LR≻)
{...// Typ1 and Typ2 can now be used as types }
invocation:  ≺name≻< ≺T1≻,≺T2≻,...>(≺LE≻); where ≺T1≻, ≺T2≻, ...  are types.
The compiler is often able to determine the types, so that invocation may also be:
≺name≻(≺LE≻);
```

5 Statements

5.1 Condition (if)

```
if(≺boolean_expr≻)≺statement≻;// the shortest version of the "if" condition
if(≺boolean_expr≻)             // with a else branch
≺statement_0≻;                 // executes an statement if the ≺boolean_expr≻ is true
else                           // executes another statement if not
≺statement_1≻;

if(≺boolean_expr_0≻)           // if..else if..else version
≺statement_0≻;
else if(≺boolean_expr_1≻)
≺statement_1≻;
else if(≺boolean_expr_2≻)
≺statement_2≻;
...
else
≺statement_N≻;
```

5.2 Selection (switch)

```
switch(≺integer_expression≻)
{
case ≺const_integer_0≻:
≺statement_0≻;
break;
case ≺const_integer_1≻:
≺statement_1≻;
break;
default:  // this branch will be executed when none of the above are satisfied
≺default_statement≻;
}
```

5.3 Loop (while)

```
while(≺boolean_expression≻)
≺statement≻;
```

5.4 Loop (do while)

```
do
≺statement≻;
while(≺boolean_expression≻)
```

5.5 Loop (for)

```
for([≺initialization≻];[≺continuation_condition≻];[≺incrementation≻])
≺statement≻
Note that ≺initialization≻ and ≺incrementation≻ can have multiple statements separated by commas.

Examples:

for(int i=0;i!=n;++i)       // classical loop (note the prefix increment)
{...}

int i=1;                    // with separated initialization
for(;(i<n)&&(j>=0);++i,--j)
{...}

for(;;)                     // infinite loop with break condition
{...
if(i<n)break;}
```

5.6 Loop exit and skipping

```
break;          // breaks out of the loop
continue;       // skips the remaining part of the loop
goto ≺label≻;   // goes to a line marked with a label
≺label≻:        // this marks a line with a label
```

6 Structs and classes

6.1 General considerations

Struct (an extension of C-language structs) are user-defined types (user-defined ≺C≻).
Each struct definition should be valid for both const and non-const objects.
Struct may contain type synonyms, enumerations, variables and member functions.
Struct and class are similar: however, they have distinct default access for members and inheritance (public in struct and private in class).
Often, it is necessary to use pointers or references to a struct before it is defined. A forward declaration can then be used:

```
struct ≺name_of_used_struct≻; // forward declaration of a struct
struct ≺name_struct≻
{... // use of pointers or references[6] of objects of type ≺name_of_used_struct≻ }
```

6.2 Definition of a structure

```
struct ≺name_struct≻ [:[virtual] ≺name_ancestor_struct_1≻],...,[virtual] ≺name_ancestor_struct_m≻]
// struct name and ancestor inheritance (virtual keyword indicates merging of the ancestors)
{// ⟵ beginning of the struct member declaration and definition
/* one ancestor type */
typedef ≺name_ancestor_struct_1≻ super;
/* friend structs and functions will have full access to the contents of the structure (they are not inherited) */
friend struct ≺name_of_another_struct≻;                      // friend struct declaration
friend ≺T≻ ≺name_of_a_function≻(≺LP≻);                       // friend function declaration
/* typedef is a new type synonym defined by the class and can be accessed as ≺name_struct≻::≺type_name≻ */
typedef ≺T≻ ≺type_name≻;
/* enumerations can also be defined in the class, such as ≺E≻, and can be accessed as ≺name_struct≻::≺E≻ */
enum ≺E≻
{...};
/* now the member variable and functions subsequent to access specifiers */
[public:|protected:|private:]                                // access specifier
[mutable] ≺T≻ ≺var≻;                                         // object variable
[virtual] ≺T≻ ≺name_of_function≻(≺LR≻) [const] [=0][{...}|;] // member function
[virtual] ≺T≻ operator @ (≺LR≻) [const] [=0][{...}|;]        // member operator
using ≺name_ancestor_struct≻::≺name_of_function≻;            // access to ancestor member
/* classical constructor and destructor */
[explicit] ≺name_struct≻(≺LR≻)[:≺LI≻]{...}                   // constructor, ≺LI≻ is the list of direct initialization
[virtual] ~≺name_struct≻[=0]{...}                            // destructor - should be virtual with polymorphic structs
/* static variables and functions */
static ≺T≻ ≺var_static≻;                                     // static variable (prefer a
                                                             // static function returning a reference)
static ≺T≻ ≺name_static≻(≺LR≻){...}                          // static function
};// ⟵ end of the struct member declaration and definition
≺T≻ ≺name_struct::var_static≻=≺expr≻;                        // compulsory static variable initialization
```

Note that the Lazyc++ tool simplifies the process of struct creation and allows the initiation of static variables in the struct definition.

In the struct, we have:
≺LR≻ as the list of received variables
≺LP≻ as the list of received types and
≺LI≻ as the direct list of initialization with the general form:
≺I1≻,≺I2≻,≺I3≻,...

Where each item ≺IJ≻ in the list of initialization is either
≺varJ≻(≺LE_J≻) or ≺name_ancestor_struct≻(≺LE≻) where the constructor of the ascending structure is used.
Note that the compiler interprets the initialization list from the last to the first item.
In ≺varJ≻(≺LE_J≻), ≺varJ≻ must be an object variable and ≺LE_J≻ is the list of arguments of a corresponding constructor.

Access specifiers:

public:	Access is granted to member functions and variables under this scope
protected:	Access is only granted to functions and variables from structs publicly derived [7]from the present one
private:	Access is only granted to functions and variables of objects of the same struct

In addition:
The virtual specifier indicates that the function is dynamically determined in a struct hierarchy.
In addition, when the =0 suffix is used (it requires the virtual keyword), the function must be redefined by the descendants when needed.
The virtual specifier only needs to be used in the base struct.

The post const keyword:
Post const means that the function cannot alter non-mutable variables. Objects declared with the const keyword will only be able to invoke the const version of member functions. The mutable keyword indicates that the member variable is allowed to be changed by a const function.
Mutator functions can alter the values of a given object and inspector functions cannot. The latter should have the const keyword.

Explicit keyword in the constructor:
The explicit keyword removes automatic type conversion effected by constructors that accept a single-argument,
therefore eliminating possible cast ambiguities

Not inherited:
Constructors, destructors and the copy operator overload are not inherited by derived structs. Friends are also not inherited.

Automatically invoked by inherited objects:
Default constructor and destructor

Virtual inheritance:
When the virtual keyword is used before the ancestor name, the following structs inheriting from more than one of
the 'virtual-inheritance' structs will have a common base struct. The common base struct is called base node, the two or
more 'virtual-inheritance' structs are called derived and the most derived class is the join struct.

[6]Variables are not allowed

6.3 Access to functions

```
≺name_ancestor_struct≻::≺name_of_function≻(≺LE≻);             // access to ancestor function
Access to virtual member functions:
≺name_struct≻ ≺aname_struct≻;                                // declaration of a object of a struct
                                                             // (does not allow use of polymorphism)
≺name_ancestor_struct≻ * ≺pname_struct≻=&≺aname_struct≻;
≺pname_struct≻->≺name_of_function≻(≺LE≻);                    // invokes name_struct virtual function or
≺name_ancestor_struct≻ * ≺pname_struct≻=new ≺name_struct≻;   // the same effect with heap allocation
```

6.4 "this" const pointer

```
this->≺name_of_a_function≻(≺LE≻);     // invokes a function on the given object
this->≺var≻;                          // accesses a variable of a given object
return *this;                         // returns the object
```

6.5 Use of a structure

```
≺name_struct≻ ≺aname_struct≻(≺LE≻);                           // object declaration
≺name_struct_base≻* ≺pname_struct≻=new ≺name_struct≻(≺LE≻);   // heap declaration of a pointer to base
≺name_struct≻::≺var_static≻=...;                              // use of a static variable
≺name_struct≻::≺name_static≻(≺LE≻);                           // invocation of a static function
≺name_struct≻::≺type_name≻ ≺aTinname_struct≻;                 // use of a struct type synonym in a declaration of a variable;
≺name_struct≻::≺E≻ ≺aEinname_struct≻;                         // use of a struct enumeration in a declaration of a variable;
≺aname_struct≻.≺name_of_function≻(≺LE≻);                      // invocation of a function by an object
≺aname_struct≻.≺name_static≻(≺LE≻);                           // an alternative invocation of a static function
≺aname_struct≻.operator@(≺LE≻);                               // a form of invocation of a operator by an object
≺pname_struct≻->≺name_of_function≻(≺LE≻);                     // invocation of a function by a pointer, same as
                                                              // (* ≺pname_struct≻).≺name_of_function≻(≺LE≻);
typename ≺name_struct≻::≺type_name≻ ≺atypeinname_struct≻;     // use of a structure typedef (with disambiguation)
```

6.6 Other typical functions and operators - friends defined in struct scope

```
#include≺iostream≻                                                            // includes the input/output library
using namespace std;                                                          // access the standard namespace
struct... (see above) {
friend ostream& operator<<(ostream&≺out≻,const ≺name_struct≻ *|& ≺rhs≻);    // insertion operator (defined only in the base
                                                                              // should invoke a virtual function
                                                                              // for each derived struct, similar for binary)
friend istream& operator>>(istream&≺in≻,≺name_struct≻ *|& ≺rhs≻);           // extraction operator (defined only in the base
                                                                              // should invoke a virtual function
                                                                              // for each derived struct, similar for binary)
bool operator==(const ≺name_struct≻&≺other≻) const {...};                   // equality comparison
bool operator!=(const ≺name_struct≻&≺other≻) const {...};                   // inequality comparison
bool operator<(const ≺name_struct≻&≺other≻) const {...};                    // less-than comparison
[virtual] ≺T≻& operator[](size_t index){...}                                 // access operator, non-constant
[virtual] const ≺T≻& operator[](size_t index) const {...}                    // access operator, constant
[virtual] ≺T≻& operator()(≺LR≻) {...}                                        // invocation operator
[virtual] const ≺T≻& operator()(≺LR≻) const {...}                            // invocation operator (constant version)
[virtual] ≺name_struct≻* to≺name_struct≻{return(this);}                      // avoid dynamic_cast, requires the same
                                                                              // function in the base class
≺name_struct≻(){...}                                                         // default constructor
explicit ≺name_struct≻(≺LR≻):≺LI≻ {...}                                      // constructor with initialization list
operator ≺T≻() const{}                                                       // type cast (conversion) operator
                                                                              // will explicitly be invoked when a type cast is used
≺name_struct≻(const ≺name_struct≻& ≺other≻){...}                             // copy constructor
≺name_struct≻(≺name_struct≻&& ≺other≻){...}                                  // move constructor, will subtract the resources of other
[virtual] void swap(≺name_struct≻&){...}                                     // swap function to be used in copy constructor
[virtual] ≺name_struct≻* clone() const[=0]{...}                              // virtual clone (uses copy constructor)
[virtual] ≺name_struct≻* create() const[=0]{...}                             // virtual create (uses default constructor)
≺name_struct≻& operator=(const ≺name_struct≻& ≺other≻)                       // assignment operator (don't forget self-assignment
{if(this!=&≺other≻){...≺name_ancestor_struct≻::operator= (≺other≻);...}     // and ancestor assignment)
return *this;}                                                                // the assignment operator is not inherited
≺name_struct≻& operator++(){...}                                             // prefix increment (usually faster)
≺name_struct≻ operator++(int trash){...}                                     // suffix increment (a dummy int argument is used)
≺name_object≻& operator*(){...}                                              // dereference (*) operator
const ≺name_object≻& operator*() const {...}                                 // dereference (*) operator, constant
≺name_object≻* operator->(){...}                                             // dereference (->) operator
const ≺name_object≻* operator->() const {...}                                // dereference (->) operator, constant
friend const ≺name_struct≻ operator+                                         // addition operator as friend function
(const ≺name_struct≻& first,const ≺name_struct≻& second){...};               // other binary operators can also be friend functions
};
```

6.7 Synthesized member functions - present if not defined by the programmer

- Default constructor (empty)
- Copy constructor (copies all member variables)
- Destructor (by default non-virtual if the base class does not possess a virtual destructor)
- Assignment operator (copies all member variables)
- Dereference operators

6.8 Template structs and specialization

Basic declaration and usage:

```
template <typename Typ1,typename Typ2,...> // note that Typ1 and Typ2 are type parameters
struct ≺name_struct≻{                     // Typ1 and Typ2 can now be used as types}
```
usage: ≺name_struct≻<≺T1≻,≺T2≻,...> ≺var≻; where ≺T1≻, ≺T2≻, ... are types

Specialization. When a specific instantiation of a template for type ≺T≻ is needed:

```
template <typename Typ>
struct ≺name_struct≻{...};              // general template
template <>
struct ≺name_struct≻ <≺T≻> {...};        // specialized template
template <typename Typ>
struct ≺name_struct≻ <Typ*> {...};      // specialized template for pointer (partial specialization)
```

6.9 Pointer to struct members

We can use a variable to select which member function to invoke. The syntax is:

```
typedef ≺T≻ ≺name_struct≻::*≺synonym_pointer≻;                // defines ≺synonym_pointer≻ as a pointer to a member variable
typedef ≺T≻ (≺name_struct≻::*≺synonym_pointer≻)(≺LP≻)[const]; // defines ≺synonym_pointer≻ as a pointer to a member variable or function
≺synonym_pointer≻ ≺apointer≻;                                  // declares ≺apointer≻ as a object pointer to a member function
≺apointer≻=&≺name_struct≻::≺name_member_function≻;             // assigns a specific member functions to ≺apointer≻
[≺Lexpr≻=] (≺name≻.*≺apointer≻)(≺LE≻);                         // with ≺name≻ being an
                                                               // object of type ≺name_struct≻, invokes a member function
[≺Lexpr≻=] (≺name≻->*≺apointer≻)(≺LE≻);                        // with ≺name≻ being a pointer to an
                                                               // object of type ≺name_struct≻, invokes a member function
```

Access macro (to avoid the previous syntax): #define ACCESS_MEMBER(name,apointer) ((name).*(apointer))
Usage: [≺Lexpr≻=] ACCESS_MEMBER(≺name≻,≺apointer≻)(≺LE≻)

7 User-defined namespaces

To define a scope spanning one file or more, the user can create a named namespace:
namespace ≺namespace_name≻{ ≺contents≻ } // ≺contents≻ is composed of typedefs, variables, structs, etc
The scope operator :: is used to access the contents of a namespace. For example, if a struct point was declared in namespace custom, a variable can be declared as:
custom::point var;
In alternative, we can use the keyword using:
using namespace custom;
point var;
Specific ≺contents≻ use can also be prescribed:
using custom::point;
point var;
Global variables and functions can be used with the global scope operator ::
::var; // in alternative, they can be inserted in a unnamed namespace. Unnamed namespaces force translation unit scope.
Namespace aliases can be defined as follows:
namespace ≺namespace_alias≻=≺existing_namespace≻;

8 Exceptions

```
try
{ // code that may throw exceptions or invocation of such code
throw ≺expr≻ }
catch(≺T≻ ≺var≻) // catches the type returned by a previous ≺expr≻
{ // code that will invoked according to the value of ≺var≻}
catch(...)  // catches any throw
{ // code that will invoked to handle any exception }
```

9 Uniform initialization and initializer lists

The C++11 norm allows the use of a homogeneous container for inserting a braced-delimited list of values.
The following header inclusion is required: #include <initializer_list>

To use a initializer_list in a declaration (≺name_struct≻ must declare a initializer_list constructor):
≺name_struct≻ ≺aname_struct≻ {≺LE≻};

Common use of initializer_list is:

- Constructors
- Functions with an arbitrary number of arguments
- Conversion between containers

Example:
vector<string> vecstr {"First name","Second name","Third name","Fourth name"};

10 Tools

Tool to generate header and source from a .lzz file (Lazyc++): http://www.lazycplusplus.com/
Tool to improve the source-code readability: http://astyle.sourceforge.net/
Tool to create makefiles: http://www.cmake.org/

11 More information, acronyms and patterns

C++FAQ: http://www.parashift.com/c++-faq/index.html
Curiously Recurring Template Pattern (CRTP)
Resource Acquisition Is Initialization (RAII)