1) Basic aspects

1.1) Comments and main segment

```
// one-line comment
/*
  comment throughout several lines
*/
int main()                          // argument-free main segment
{<instructions>}                    // corresponding instruction block
int main(int argc, char **argv)     // main segment with arguments
{<instructions>}                    // instruction block
extern “C” {...}                    // functions compiled by C can be enclosed in the curly braces, e.g.
extern “C” {int mult(int* i,int* j);} // example of a prototype of a function compiled in C
```

1.2) File inclusion and namespace use

```
#include < <system_header> > // inclusion of a system header
#include "<file>.h"          // inclusion of a user header
using namespace std;         // using the std namespace
#pragma once                 // indicates that the header is included only once
Source code files of a class:
<file>.cpp                   // source code with the implementations
<file>.h                     // header file with the interfaces
```

2) Types and declarations

2.1) Pre-defined types and classes

```
<C>                          // pre-defined type or introduced by the user
Typical pre-defined types: void, char, int, unsigned int, bool, double, float
auto: used when the type is automatically determined by the compiler
Examples of declarations:
int    <var_int>;            // integer variable declaration
char   <var_char>;           // character variable declaration
float  <var_float>;          // real (32 bits) variable declaration
double <var_double>;         // real (64 bits) variable declaration
bool   <var_bool>;           // boolean variable declaration (values true or false)
auto   <var_auto>=<expr>;    // the type is determined by the compiler from the type of <expr>
```

2.2) General declaration in the stack

```
[static] <T> <var> [(<LE>)|=<C>()]; // <C>() calls the default constructor
<T> is given as [const] <C> [§ [const]]
the keywords are:
static // maintains the variable beyond the scope of declaration
const  // as prefix declares the value of the variable to be constant
const  // as suffix declares the address to be constant
The symbol § is one of:
*      // address, when a variable address is stored
&      // reference, when an alias of a given variable is stored, requires the initiation
*&     // reference of an address, when an address is allowed to change
The list of sent expressions (<LE>) is a ordered comma-separated set of N expressions (if N=0 parenthesis are omitted):
(<expr_0>,<expr_1>,...,<expr_N-1>)
```

2.3) Array in the stack

```
<T> <array>[<n>]; // declares an array of type <T> with n elements
```

2.4) Heap space management

```
<var> and <array> are pointers of type <T> and their declarations are:
<T>* <var>;   // declares a pointer to <T>
<T>* <array>; // also declares a pointer to <T> but will be differently allocated
the heap space is managed as:
<var>=new <T>;         // reserves space for a variable of type <T>
delete <var>;          // frees the space previously reserved
<array>=new <T>[<n>]; // reserves space for an array of type <T> with <n> elements
delete[] <array>;      // deletes the reserved space of the array
```

2.5) Type synonyms

```
typedef <T> <name>
Example of use of synonyms
typedef int* pint; // pointer to int declared as pint
pint Pint=new int; // allocation in the heap
```

3) Operators and expressions

3.1) Nomenclature of Rexpr and Lexpr

```
<Lexpr>        // an expression which can be used either in the left or right hand-side of an attribution
<Rexpr>        // an expression which cannot be used 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
<*integer_expr*> // an expression which has an integer value

3.2) Typical operations (a,b,c are expressions)

<*a*>=<*b*>; // assignment
++<*a*>, --<*a*>,<*a*>++,<*a*>--; // pre-increment and decrement and post-increment and decrement
!<*a*>, <*a*>||<*b*>, <*a*>&&<*b*>; // negation, "or" and "and" the result is a boolean
<*a*>==<*b*>, <*a*>>=<*b*>, <*a*><=<*b*>, <*a*><<*b*>, <*a*>><*b*>, <*a*> != <*b*> ; // equal, greater than or equal, less than or equal, less than, greater than, not equal
<*a*>+<*b*>,<*a*>-<*b*>,-<*a*>,<*a*>*<*b*>, <*a*>/<*b*>, <*a*>%<*b*>; // addition, subtraction, minus, times, division, remaining
<*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 grouping an expression
<*a*>.<*b*>; // access to a member variable
<*a*>.<*b*>(<*c*>); // access to a method
<*a*>-><*b*>(<*c*>); // access to a method when <*a*> is a pointer
*<*a*>; // dereferencing a pointer, accesses the contents stored at address <*a*>
&<*a*>; // address of <*a*>
<*a*>[<*i*>] or *(<*a*>+<*i*>); // accesses the contents stored at address <*a*>+<*i*>

3.3) Use of curly braces

When an instruction is expected and there is the need of inserting more than one, we can use curly braces {}.
For *N* instructions: {<*inst_0*>;<*inst_1*>;...;<*inst_N-1*>}
As an example

```
if(a!=b)a=b; // one instruction expected from if
if(a!=b) // when two or more instructions are needed, curly braces must be used
{ a=b;
  c=b; }
```

3.4) Type-cast

Type conversion can be useful. Three type-cast functions are available:
In pointers of structures with inheritance (struct <*T2*>:<*T1*>) and with virtual methods
we use dynamic casts:
<*T1*>* <*var1*>=new <*T1*>;
<*T1*>* <*var2*>=new <*T2*>;
<*var1*>=dynamic_cast< <*T1*>*>(<*var2*>);
In addresses of other structures, for example from void* to another pointer
var1=static_cast<T1*>(var2);
In any other cases, the reinterpret_cast is used:
var1=reinterpret_cast<T1*>(var2);

4) Functions

4.1) Prototype and definition

<*T*> <*name*>(<*LP*>); // prototype
<*T*> <*name*>(<*LR*>); // definition (note the difference between <*LR*> and <*LP*>)
{<*instructions*>
[return(<*value*>);] // return of value only if <*T*> is not void}
<*LR*> is the list of received variables and is given, for N arguments, 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 received types and is given, for N arguments, as:
<*T_0*>,<T_1>,...,<T_N-1>
In <*LR*>, the expressions <*expr_i*> provide the default value of the argument, and must be set at the right end of the argument list

4.2) Overloading

We can declare and define functions with the same name but with different list of received types. For *example*,

```
void print(const int& val)     // will be called when an int is used
void print(const double& val)  // will be called when a double is used
```

4.3) Argument passing

Each <*T*> in <*LP*> and <*LR*> must be of one of the following types:

```
<C>         // by value, can receive any <expr>
const <C>&  // by constant reference, can receive any <expr> and will not be modified
<C>&        // by reference, can only receive <Lexpr> and can be modified
<C>*&       // a reference to pointer, for new and delete operations inside the function
```

4.4) Return value

The return value can be:

```
<C>         // a copy of the calculated object is returned to the caller
<C>&        // a reference to an argument or to a stored permanent object is returned
<C>*        // a pointer to an argument or to a heap variable defined inside the function
```

4.5) Invocation

[<*val*>=]<*name*>(<*LE*>); // invocation of a function with a list of sent expressions

The list of sent *N* expressions, <*LE*> can be written as:
<*expr_0*>,<*expr_1*>,...,<*expr_N-1*>
where each of the expressions must be <*Lexpr*> if the corresponding type in <*LP*> is <*C*>& or <*C*>*&
when default values are defined, then the corresponding argument can be omitted

4.6 Template function

template **<**typename Typ1,typename Typ2,...**>** //note that Typ1 and Typ2 are type parameters
<*T*> <*name*>{// Typ1 *and* Typ2 *can now be used as types*}
usage: [<*var*>=]<*name*>**<** <*T1*>,<*T2*>,...**>**(<*LE*>); where <*T1*>, <*T2*>, ... are types

5) Instructions

5.1) Condition (if)

if(<*boolean_expr*>)<*instruction*>; // *the shortest version of the "if" condition*
if(<*boolean_expr*>) // with a else
<*instruction_0*>; // executes an instruction if the <*boolean_expr*> is true
else // executes another instruction if not
<*instruction_1*>;
if(<*boolean_expr_0*>) // if else if else version
<*instruction_0*>;
else if(<*boolean_expr_1*>)
<*instruction_1*>;
else if(<*boolean_expr_2*>)
<*instruction_2*>;
else
<*instruction_N*>;

5.2) Selection (switch)

switch(<*integer_expression*>){
case <*const_integer_0*>:
<*instruction_0*>;
break;
case <*const_integer_1*>:
<*instruction_1*>;
break;}

5.3) Loop (while)

while(<*boolean_expression*>)
<*instruction*>;

5.4) Loop (do while)

do
<*instruction*>
while(<*boolean_expression*>)

5.5) Loop (for)

for([<*initialization*>];[<*continuation_condition*>];[<*incrementation*>])
<*instruction*>

5.6) Loop alterations

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 label*

6) Structs and classes

6.1) Differences between struct and class

Distinct default access for members and inheritance (public in struct and private in class) is the only difference. Struct is used here because it results more concise. Encapsulation of variables and functions and work savings are the main motivations for the use of structs.

6.2) Definition of a structure

struct <*name_struct*> [: <*name_ancestor_struct*>]{
// friend structs and functions will have full access to the contents of the structure
friend struct <*name_of_another_struct*>; *// friend struct declaration*
friend <*T*> <*name_of_function*>(<*LP*>); *// friend struct declaration*
// typedef can be accessed as <name_struct>::<type_name>
typedef <*T*> <*type_name*>;
[public:|protected:|private:] *// access specifier*
[mutable] <*T*> <*var*>; *// object variable*
[virtual] <*T*> <*name_of_function*>(<*LR*>) [const] [=0;|{...}] *// object function*
[explicit] <*name_struct*>(<*LR*>)[:<*LI*>]{...} *// constructor, <LI> is the initialization list*
[virtual] ~<*name_struct*>{...} *// destructor*
static <*T*> <*var_static*>; *// static variable*
static <*T*> <*name_static*>(<*LR*>){...} *// static method*};
<*T*> <*name_struct::var_static*>=<*expr*>; *// static variable initialization*

where:
<*LR*> is the list of received variables, <*LP*> is the list of received types and
<*LI*> is the initialization list with the form <*I1*>,<*I2*>,<*I3*>,...
where each item is either <*varJ*>(<*LE_J*>) or <*name_ancestor_struct*>(<*LE*>) where the constructor of the ascending structure is used. In <*varJ*>(<*LE_J*>), *varJ* must be an object variable and *LE_J* is the list of arguments of the corresponding constructor
public: Access is granted to all functions and variables under this scope
protected: Access is granted to functions and variables from structs derived from the present one
private: Access is denied
virtual specifier means that the function is dynamically determined in a struct hierarchy, when the =0 suffix is used, the function must be redefined by descendents
post const means that the function will not and cannot alter non-mutable variables
explicit removes automatic type conversion using the constructor

6.3) Access to functions

<*name_ancestor_struct*>::<*name_of_function*>(<*LE*>); *// access to ancestor function*
// access to virtual functions
<*name_struct*> <*var*>;
<*name_ancestor_struct*> * <*var_address*>=&<*var*>;
<*var*>.<*name_of_function*>(<*LE*>); *// calls name_struct virtual function or*
<*name_ancestor_struct*> * <*var_address*>=new <*name_struct*>... *// the same with heap allocation*

6.4) “this” pointer

this-><*name_of_a_function*>(<*LP*>); *// calls a method 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*> <*var*>(<*LE*>); *// object declaration*
<*name_struct_base*>* <*var*>=new <*name_struct*>(<*LE*>); *// heap declaration of an address*
<*name_struct*>::<*var_static*>=...; *// use of a static variable*
<*name_struct*>::<*name_static*>(<*LE*>); *// invocation of a static method*
<*var*>.<*name_of_method*>(<*LE*>); *// invocation of a method by an object*
<*var*>-><*name_of_method*>(<*LE*>); *// invocation of a method by an address*
<*name_struct*>::<*type_name*> <*var*>; *// use of a structure typedef*

6.6) Recommended functions and operators

#include<iostream> *// includes the input/output library*
using namespace std; *// and uses the standard namespace*
struct <*name_struct*> [:<*name_ancestor_struct*>]{
friend ostream& operator<<(ostream&<*out*>,const <*name_struct*>& <*rhs*>);
friend istream& operator>>(istream&<*in*>,<*name_struct*>&<*rhs*>);
friend bool operator==(const <*name_struct*>&<*lhs*>,const <*name_struct*>&<*rhs*>);
friend bool operator!=(const <*name_struct*>&<*lhs*>,const <*name_struct*>&<*rhs*>);
friend bool operator<(const <*name_struct*>&<*lhs*>,const <*name_struct*>&<*rhs*>);
friend bool operator>(const <*name_struct*>&<*lhs*>,const <*name_struct*>&<*rhs*>);
<*T*>& operator[](unsigned i){...} *// access operator, non-constant*
const <*T*>& operator[](unsigned i) const {...} // access operator, constant
<*T*>& operator() (<*LR*>) {...} *// call operator overload*
<*name_struct*>(){...} *// default constructor*
explicit <*name_struct*>(<*LR*>):<*LI*> {...} *// constructor with initialization list*
operator <*T*>() const{} *// type cast operator*
<*name_struct*>(const <*name_struct*>& <*rhs*>){...} *// copy constructor*
<*name_struct*>& operator=(const <*name_struct*>& <*rhs*>) *// assignment operator*
{if(this!=&<*rhs*>){...}
return *this;}

6.7) Template structs

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

6.8) Downcast to access descendent functions

<*base*>* <*b*>=new <*derived*>; // declares b as a pointer to a base class and initializes as a derived
<*derived*>* <*d*>=dynamic_cast< <*derived*>* >(<*b*>); // casts to a pointer to a derived class
<*d*>-><*name_of_function*>(<*LE*>); *// accesses a function defined in derived but not in base*

7) Tools

Automatically generate header and source from a .lzz file: http://www.lazycplusplus.com/
Reformat tool to improve the source-code readability: http://astyle.sourceforge.net/
Commercial tools for Visual Studio: http://www.wholetomato.com/, http://www.devexpress.com/