1) Basic aspects

1.1) Comments and main segment

// one-line comment
/*
comment throughout several lines
*/

int main() // argument-free main segment
{<
instructions>
}

int main(int argc, char **argv) // main segment with arguments
{<
instructions>
}

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

<T> // pre-defined type or introduced by the user
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> [([<L>])][<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 ([<L>]) 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

<T> 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

<Expr> // 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>
3.2) Typical operations (a,b,c are expressions)

- Assignment
  - `a+=b;`  // pre-increment and decrement
  - `a-=b;`  // post-increment and decrement

- Comparison
  - `a==b;`  // equal
  - `a!=b;`  // not equal
  - `a>=b;`  // greater than or equal
  - `a>b;`  // greater than
  - `a<=>b;`  // less than, not equal

- Assignment
  - `a*=<b;`  // use of * () in an assignment

- Access to member variables
  - `a->b;`  // access to a member variable
  - `a.b;`  // access to a method

- Access to stored permanent object
  - `*a++;`  // dereferencing a pointer, accessing the contents stored at address `a`

- Access to a member variable
  - `&a++;`  // address of `a`

- Access to a method when `a` is a pointer
  - `*(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
```cpp
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:

- Dynamic cast:
  - `T1* var1=new T1;`  // use of new in a function invocation
  - `T1* var2=new T2;`
  - `var1=dynamic_cast<T1*>(var2);`

- Static cast:
  - `var1=static_cast<T1*>(var2);`

- reinterpret_cast:
  - `var1=reinterpret_cast<T1*>(var2);`

3.1) Prototype and definition

- Prototype:
  - `<T> name=<LP>;`  // prototype

- Definition:
  - `<T> name=<LR>;`  // definition (note the difference between <LP> and <LR>)

The return value can provide the default value of the argument, and must be set at the right end of the function definition.

4.2) Overloading

When two or more instructions are needed, curly braces must be used.

```cpp
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; }
```

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>`
- `<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

```cpp
[<val>=]<name=<LE>];  // invocation of a function with a list of sent expressions
```
The list of sent $N$ expressions, $\langle LE \rangle$ can be written as:
$\langle \text{expr}_0 \rangle, \langle \text{expr}_1 \rangle, \ldots, \langle \text{expr}_{N-1} \rangle$
where each of the expressions must be $\langle \text{Expr} \rangle$ if the corresponding type in $\langle LP \rangle$ is $\langle C \rangle\&$ or $\langle C \rangle\*\&$
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$>$
//note that Typ1 and Typ2 can now be used as types
usage: $[\langle$var$\rangle]=\langle$T1$\rangle,\langle$T2$\rangle,\ldots;\langle\langle LE \rangle\rangle$; where $\langle\langle T1\rangle\rangle,\langle\langle T2\rangle\rangle,\ldots$ are types

### 5) Instructions

#### 5.1) Condition (if)

if($<$boolean_expr$>$)$<$instruction$>$; // the shortest version of the "if" condition
if($<$boolean_expr$>$) $<$instruction_0$>$; // with a else
$<$instruction_0$>$; // executes an instruction if the $<$boolean_expr$>$ is true
else $<$instruction_1$>$; // executes another instruction if not
if($<$boolean_expr$>$) $<$instruction_0$>$; // if else if else version
else if($<$boolean_expr$>$) $<$instruction_1$>$;
else if($<$boolean_expr$>$) $<$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$>$($<$LR$>$){...} // friend function declaration
// typedef can be accessed as $<$name_struct$>::$<$type_name$>$
typedef $<$T$>$ $<$name_struct$>::$<$name_variable$>$; // static variable initialization
}$<$T$>$ $<$name_struct$>::$<$name_variable$>$; // static variable initialization
where:

\[ <\text{LE}> \] is the list of received variables, \( <\text{LP}> \) is the list of received types and
\[ <\text{L}> \] is the initialization list with the form \( \langle 1 \rangle, \langle 2 \rangle, \langle 3 \rangle, \ldots \)
where each item is either \( \langle \text{var}\rangle \langle \text{LE} \rangle \) or \( \langle \text{name\_ancestor\_struct}\rangle \langle \text{LE} \rangle \) where the constructor of the ascending structure is used. In \( \langle \text{var}\rangle \langle \text{LE} \rangle\), \text{var}3 must be an object variable and \text{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

d 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

\[ <\text{name\_ancestor\_struct}\rangle::<\text{name\_of\_function}\rangle(\text{LE}); \] // access to ancestor function

// access to virtual functions
\[ <\text{name\_struct}\rangle \langle \text{var}\rangle; \]
\[ <\text{name\_ancestor\_struct}\rangle \star \langle \text{var}\_address\rangle = \&<\text{var}; \]
\[ <\text{var}\_\langle \text{name\_of\_function}\rangle(\text{LE}); \] // calls name\_struct virtual function or

\[ <\text{name\_ancestor\_struct}\rangle \star \langle \text{var}\_address\rangle = \text{new} <\text{name\_struct}> \ldots \] // the same with heap allocation

6.4) "this" pointer

\[ \text{this} \rightarrow \langle \text{name\_of\_a\_function}\rangle(\text{LP}); \] // calls a method on the given object

\[ \text{this} \rightarrow \langle \text{var}\rangle; \] // accesses a variable of a given object

\[ \text{return} \ast \text{this}; \] // returns the object

6.5) Use of a structure

\[ <\text{name\_struct}> \langle \text{var}\rangle(\text{LE}); \] // object declaration

\[ <\text{name\_struct\_base}\rangle \langle \text{var}\rangle = \text{new} <\text{name\_struct}>(\text{LE}); \] // heap declaration of an address

\[ <\text{name\_struct}>::<\text{var\_static}>..; \] // use of a static variable

\[ <\text{name\_struct}>::<\text{name\_static}>(\text{LE}); \] // invocation of a static method

\[ <\text{name\_ancestor\_struct}>::<\text{name\_of\_method}>(\text{LE}); \] // invocation of a method by an object

\[ <\text{name\_ancestor\_struct}>::<\text{name\_of\_method}>(\text{LE}); \] // invocation of a method by an address

\[ <\text{name\_struct}>::<\text{type\_name}>(\text{var}); \] // 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>);
friend ostream& operator[[](unsigned i){...} // access operator, non-constant
const T& operator[][](unsigned i) const {...} // access operator, constant
friend operator[](const <name\_struct>& &<lr>){...} // call operator overload
<name\_struct>{}[](...) // default constructor
explicit <name\_struct>(<LR>){...} // 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) Templates

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

\[ \text{base}\ast \rightarrow \text{derived}\ast \]; // declares \text{b} as a pointer to a base class and initializes as a derived
\[ \text{derived}\ast \rightarrow \text{dynamic\_cast}<\text{derived}\ast >(\text{db}); \] // casts to a pointer to a derived class
\[ \rightarrow \langle \text{name\_of\_function}\rangle(\text{LE}); \] // 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/