Extensions of the C language in the C++ language

There are two types of ***extensions*** of the C language:

- adding some facilities that *are not related* to object-oriented programming paradigm (reference type, in-line substitution of the functions, etc.)
- adding elements in order *to provide support* for object-oriented programming paradigm (class, inheritance, polymorphism, etc.)

A. A short history of C++

- The history of the C++ language can be divided in 3 periods:
  - ***Early C++***, starting to 1979, when ***Bjarne Stroustrup*** worked for his Ph.D. thesis on the ***Simula*** language (which was too slow for practical use)
    - The first language developed by Stroustrup was called "***C with classes***" – a ***superset of the C language***, which:
      - includes some ***object-oriented concepts*** (***classes***, ***inheritance***, …)
      - can produce ***high speed programs***
    - In 1983 the name of the language was changed in ***C++***, and new features were added (***virtual functions*** and ***polymorphism***, ***function overloading***, ***lvalue references***, ***new*** and ***delete*** operators, …)
    - In 1989 other new features were added (***multiple inheritance***, ***abstract classes***, …)
    - In 1990, ***The Annotated C++ Reference Manual*** was released. This book described the language, including some features (***namespaces***, ***exception handling***, ***nested classes***, ***templates***)

- ***Classical C++***:
  - In 1991: ISO C++ Committee was founded
  - In 1992: ***Standard Template Library*** (STL) was implemented
  - In 1998, ***the first ISO standard*** for C++ was published (***C++98***)
    - New features were added (***RTTI***, ***covariant return types***, ***cast operators***, ***mutable***, ***bool***)
    - It includes the ***Standard Template Library*** (***containers***, ***algorithms***, ***iterators***, ***function objects***)
  - The ***second standard*** was ***C++03***
    - This was a minor revision of C++98

- ***Modern C++***:
  - In 2011, the ***third standard*** was published: ***C++11***:
    - A large number of changes were introduced (***auto*** and ***decltype***, ***defaulted*** and ***deleted*** functions, ***final*** and ***override***, ***trailing return type***, ***rvalue references***, ***move semantics***, ***constexpr***, ***nullptr***, ***long long***, ***variadic templates***, ***lambda expressions***, ***range for***, …)

- In 2014, the ***fourth standard*** was published: **C++14**:
  - A minor revision of the C++11 standard
  - Some new features were added (***variable templates***, ***polymorphic lambdas***, ***return type deduction for functions***, ***aggregate initialization***)
- In 2017, the ***fifth standard*** was published: ***C++17***:
  - Some new features were introduced (***fold-expressions***, ***class template argument deduction***, ***auto non-type template parameters***, ***compile-time if constexpr***, ***inline variables***, ***structured bindings***, ***initializers for if and switch***, …)
- The next major revision of the C++ standard: ***C++20*** …

B. Classical C++

B1. New data types

- ❑ C++ has additional ***built-in data types***

a) The `bool` datatype

- represents *logical values* (boolean),
- uses two predefined *constants*: **`true`** and **`false`**.

- ❑ There is a *similarity* with the **Pascal** language (**`Boolean`** datatype), and with the **Java** language (**`boolean`** datatype).

- ❑ There is compatibility between the data type **`bool`** and arithmetic data types.
  - ❑ The **`bool`** variables can be assigned with integer values because any C++ compiler *automatically converts* integer values to the **`bool`** value.

Example. For the following sequence :

```
bool boolVar;
int intVar;
// ...
boolVar = intVar;
```

the C++ compiler generates an equivalent expression :

```
boolVar = intVar ? true : false;
```

- Similarly, there is also an *automatic conversion* from the bool values to the integer values. For example:

```
intVal = boolVal ? 1 : 0;
```

- Using the **`bool`** data type allows writing code with a simpler an intuitive meaning. For example:

```
bool BelongsTo(double x, double a, double b);
```

b) The `wchar_t` datatype (`wide character`)

- It is an extension of the dataype **`char`**;
- It allows to using characters represented internally on *two bytes* (for example the **`Unicode`** set of characters).

- Usually, for Windows, **`sizeof(wchar_t)=2`**, allowing to use sets of characters having more than 64000 characters, while for Linux the size is 4 bytes.

- To assign a character to **`wchar_t`** type a letter “L” is added in front of the character:

```
wchar_t wc = L’c’;
```

B2. Variable declaration and namespaces

- In the C++ language the *local declarations* can be appear anywhere within a block (unlike the C language).

- The *scope* of such local declared variables starts to the line of the declaration and it ends at the end of the current block.

- All the variables used in different modules of a C program are related to the whole program.
  - So, the variables with the *same name* declared in *different modules* of a program access the same memory zone and *represent the same variables*.

- The C++ language attaches the variables to a ***namespace***, which allows the variables with the *same name* but in *different modules* to represent *distinct variables*.

- All the variables declared in the standard libraries of the C++ language have a *predefined namespace*, denoted by **`std`**.

- For using a namespace different to the current compilation unit, the *directive* **`using`** is used:

```
using namespace std;
```

- For example, for working with the input/output operations the following sequence should be used:

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

Remark.

- *Header files* related to the standard library of the C++ language *do not contain* the suffix "*.h*" as in the C language.
- All header files related to the standard library of the C language are *rewritten* in the C++ language, and their names have the character '*c*' as prefix. For example:

```
#include <alloc.h>
```

is equivalent with:

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

- However, in order to keep the *compatibility* with the C programs, the *syntax for including* the standard header files of the C language *can be also used* in the C++ programs.

B3. Lvalue references

- The C language allows *only one way of passing the parameters* when calling a function, ***call by value***, which requires using pointers in the case when a function modifies the value of a certain parameter.

- The C++ language adds the notion of ***lvalue reference***. A reference is an alternative name (*alias*) for a variable.

- The ***reference type*** is a *compound* type, which is realized by using the operator `&`. For example:

  `T&`

  represents the reference type derived from the base type *T*.

- The values of ***reference types*** are similar to ***pointers***, in the sense that a reference has as value the memory address of a variable belonging to a base type.

- However, there are some important ***differences between pointers and references***:
  a) A reference *must be always initialized* at the declaration. For example:

```
int k;
int &r = k;
```

  b) References are *automatically dereferred* when using them in a program. For example :

```
int k = 5, &r = k, *p;
p = &k;
r = r + 1; //that means k = k + 1
*p = *p + 1;
```

- The main way to use the reference mechanism is related to ***passing parameters*** in functions.

Example. Swapping two values:

```
 void Swap1(int *a, int *b) {
   int c = *a;
   *a = *b;
   *b = *c;
 }
 void Swap2(int &a, int &b) {
   int c = a;
   a = b;
   b = c;
 }
 void Process() {
   int x = 7, y = 5;
   Swap1(&x, &y);
   printf(″%d%d″, x, y);
   x = 7; y = 5;
   Swap2(x, y);
}
```

B4. Inline functions

- Initially In the case of *small functions* (with small number of statements):
  - the *calling mechanism can be significant* in respect with the execution time of the function,
  - the *execution time* of the program *can increase* and its efficiency decreases.

- The C++ language offers the possibility to expand ***inline*** theses small functions.

- When the ***inline*** function is called whole code of the inline function *gets inserted or substituted* at the point of inline *function call*.

- Declaring an **`inline`** function can be made either:
  - a) *for non-members functions* of classes: by *using* the keyword **`inline`** before its definition;
  - b) *for a member function of a class*: by *including* the implementation of the function block in the class declaration.

Example:

```
inline int minim(int a, int b) {
  return ((a < b) ? a : b);
}
```

- ❑ In the case of the ***inline functions***, the compiler tries to place an instance of the calling function in the same code segment as the called function, but this fact *is generally not guaranteed*.

- ❑ For *complex* functions (recursive functions, or functions having repetitive statements) the **`inline`** mechanism is *not performed*.

- ❑ In general, the using of ***inline functions*** *is more efficient than usual functions*, but it is *less efficient than the using of macros*.

- ❑ **Remark**. An ***inline*** function can be defined inside of a header file.

- In this case, each *translation unit*, which include this header will contain the same function that will be inlined.
- In this way, the compiler allows the definition of a function to be visible in multiple translation units (that include the header file)

Example.

```
// head.h
inline int f(int n) {
    return 2 * n;
}
```

```
// pr1.cpp
#include “head.h”
static int a = 10;
int g1(int k) {
    return a * f(k);
}
```

```
// pr1.cpp
#include “head.h”
static int a = 20;
int g2(int k) {
    return a * f(k);
}
```

```
// main.cpp
extern int g1(int);
extern int g2(int);
int main() {
    cout << “g1 = “ << g1(4);
    cout << “g2 = “ << g2(4);
    return 0;
}
```

- In the C++ language it is better to use inline function than macros:

- Inline functions are managed by the compiler, while macros are managed by the pre-processor
- C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. The preprocessor is not able of doing this for macros
- Macro cannot access private members of class

B5. Default arguments for function parameters

- Usually, an important rule for many programming languages imposes the *same number of parameters* both for the function *definition* and for the function *call*.

- The C language allows the definition (quite difficult) of some functions with *variable number of parameters*, with the help of the operator '**…**'.

- In addition to the C language, the C++ language provides a *simpler* and *more efficient* method for functions with a variable number of parameters: ***functions with default values for parameters***.

- A parameter with *a default value* is declared as usually through a name and a data type, but *in addition* it is *initialized* with an appropriate *value*.

- If the function call contains an actual parameter, this value is used as initialization; if the actual parameter is missing, the actual value is considered as the initialization value.

Example.

```
double Distance(double x, double y,
    double x0 = 0, double y0 = 0)
{
  return sqrt((x-x0)*(x-x0)+(y-y0)*(y-y0)) ;
}
void Processing() {
  double x1 = 3, y1 = 5, x2 = 4, y2 = 6, d1, d2 ;
  //distance between(x1,y1) and origin
  d1 = Distance(x1, y1);
  //distance between (x1, y1) and (x2, y2)
  d2 = Distance(x1, y1, x2, y2);
  // ...
}
```

Remarks :

a) A parameter with a default value can be initialized only with *a constant expression*, which *can be evaluated during compilation*;

b) A function *can have more parameters with default values*, but in this case, they must take the *last positions* (because otherwise the current values of the parameters cannot be determined when calling the function)

B6. Function overloading

- ***Overloading of the functions name*** means the existence of *two or more functions* with the *same name* which perform *different tasks*.

- The C++ language allows the definition of overloaded functions. For example, the definitions of two functions with the same name *add* :

```
double add(double a, double b) {
  return a + b;
}
char* add(char *a, char *b) {
   strcat(a, b); return a;
}
void Processing() {
    double s = add(1.5, 8.4);
    char *s1 = "abc", *s2 = "xyz";
    char *s3 = add(s1, s2) ;
    // ...
}
```

Remarks.

a) For defining two different overloaded functions they must have *different number of parameters* or at least the *data type of one of its parameters*.

b) Two overloaded functions *can not differ only by the type of the returned value*, because the type of the returned value is not verified by the compiler.

c) The compiler *determines* the *effective function* which will be called depending of *types of the actual parameters and their number*.

B7. Operators for memory handling

- The C++ language has in addition to the C language two operators represented by the keywords ***new*** and ***delete***. The used syntax is:

```
<pointer> = new [‘(‘] <type> [‘)’] [(<expression>)];
delete <pointer>;
```

Example :

```
int *p = new int(4);
double *p = new(double);
struct point { double x, y; };
struct point *p = new struct point;
```

- These operators can be used also for memory allocation/deallocation for *compound elements*. In the case of arrays, the length of the array must be explicitly specified.

- In the case of the **`delete`** operator, if *the number of the components is not specified*, this number is *automatically determined* by the compiler. The used syntax is:

```
<pointer> = new <type> '[' <dimension> ']';

delete '[' [<dimension>] ']' <pointer>;
```

Example:

```
int *p, *q, *r;
*p = new int[10];
*q = new int[10];
*r = new int[10];
// Not O.K. Only the first element is deallocated
delete p;
// O.K. 10 elements are deallocated
delete[10] p;
// O.K. All elements are deallocated
delete[] p;
```

- The operator **`new`** can be used in addition for the *creation of multi-dimensional arrays*. In this case all the dimensions of the array must be specified. For example, the following expression:

  ```
  new int[2][3][4]
  ```

  allocates the memory for two arrays of the type:

  ```
  int [3][4]
  ```

  and it returns a pointer to the first array, that is a pointer of the following type:

  ```
  int (*)[3][4]
  ```

- Regardless the *number of the dimensions* of an array that is allocated by the operator **`new`**, the *syntax for deallocation* of this array by using the operator **`delete`** *is the same* (only one pair of brackets).

Example:

```
int a[2][4] = {1, 2, 3, 4}, (*p)[4];
p = new int[2][4];
for (int i=0; i<2; i++)
  for (int j=0; j<2; j++)
     p[i][j] = a[i][j];
// ...
delete[] p;
```

Remarks:

a) The operator ***new*** *calls by default a constructor* the class if the data type is an instance of certain class.

b) The operator ***delete*** *calls by default the class destructor*, if the pointer indicates an instance of a certain class.

B8. Template functions

- ❑ The C++ language offers support for ***data abstraction*** and parameterization:
  - ***template functions***
  - ***template classes***.

- ❑ A ***template function*** contains at least a *generic* (*unspecified*) data type.

- ❑ The syntax for defining a template function impose the presence of the following construction before the header of the function:

  `template '<' class ⟨name⟩ '>'`

  where ⟨**`name`**⟩ represents the name of the data, which is a parameter for the template function, and it can be used inside the block of the function.

- A ***template function*** describes a set of functions having similar code but different data types. It can be ***instantiated***, each *instance* of a template function being a *usual function*.

- The syntax to instantiate a template function is similar to a function call. In addition, the *actual name* of the used data type must be specified in *angle brackets*.

Example.

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

template <class T>
void Swap(T & a, T & b) {
  T temp;
  temp = b;
  b = a;
  a = temp;
}
```

```
void main() {
  int a=3, b=5;
  double x=33, y=55;
  Swap<int>(a, b);
  cout << a << " " << b << endl;
  Swap<double>(x, y);
  cout << x << " " << y << endl;
}
```

Remark. In the above example the two calls of *Swap* can be replaced also by the following sequence:

```
Swap(a,b);

Swap(x,y);
```

because the compiler can detect automatically the data types `int` and `double` to which *T* will be instantiated

C. Modern C++

C1. New datatypes and syntax

a) The `long long int` datatype

- It is an integer type whose values are stored at least 64 bits;
    - The exact dimension depends on the compiler;
- The limits values are defined in the header file **`climits`**:
    - For **`long long int`**:
        - **`LLONG_MIN`**: $(-2^{63}+1)$ or less
        - **`LLONG_MAX`**: $(2^{63}-1)$ or greater
    - For **`unsigned long long int`**:
        - **`ULLONG_MAX`**: $(2^{64}-1)$ or greater

b) The `auto` keyword

- In C++11, the meaning of the **`auto`** keyword has changed
- When initializing a variable, **`auto`** is used to tell the compiler to infer the type of them variable from the type of the initializer.
- This is called ***type inference***

Examples:

- For a variable:

```
auto x = 7.5;  // double
auto n = 7;    // int
```

- For the return values from functions:

```
int triple (int a) {
   return 3 * a;
}
int processing() {
   auto n = triple(4);
   return n;
}
```

- When **`auto`** sets the type of a declared variable from its initializing expression, it proceeds as follows:
  - If the initializing expression is a reference, the reference is ignored.
  - If, after the above step 1 has been performed, there is a top-level **`const`** and/or **`volatile`** qualifier, it is ignored
- **Example**:

```
const int c = 0;
auto rc = c;   // type of rc is int
rc = 44;       // OK
```

- **Remark**. The reference **`auto&`** related to a **`const`** value does not remove the **`const`** qualifier

```
const int c = 0;
auto& rc = c;  // type of rc is const int&
rc = 44; // error: const qualifier was not removed
```

- ❑ Starting to C++14, the auto keyword was extended to infer the return type of a function:

```
auto triple (int n) {       // int
    return 3 * n;
}
```

c) Trailing return type syntax

- ❑ C++11 also added a trailing return syntax, where the return type is specified after the rest of the function prototype
- ❑ The following function declaration:

```
int triple (int a);
```

could be equivalently written as:

```
auto triple (int a) -> int;
```

- ❑ In this case, auto does not perform type inference, it is just part of the syntax to use a trailing return type;
- ❑ This rare C++ feature was added to aid writing of generic code and to provide consistency (will be later discussed)

d) The null pointer

- Before C++11, for the null pointer was used the **`NULL`** macro:
    - It was typically defined as **`(void *)0`**
    - Conversion of **`NULL`** to integral types is allowed (and is implicit)
- For this reason, the using of **`NULL`** can be ambiguous.

- For **example**, for two overloaded functions:

```
void f(int n) {
   cout << "int";
}
void f(char* s) {
   cout << "char *";
}
int main() {
   f(NULL);    // error: call of f(NULL) is ambiguous
   return 0;
}
```

- For solving this problem, the literal **`nullptr`** was introduced:
    - It has the type **`nullptr_t`**
    - Like **`NULL`**, **`nullptr`** is implicitly convertible to any pointer type
    - Unlike **`NULL`**, it is not implicitly convertible to integral types
- For the above example:

```
void f(int n) {
   cout << "int";
}
void f(char* s) {
cout << "char*";
}
int main() {
   f(nullptr);    // is called f(char*)
   return 0;
}
```

e) Type alias

- In ***C++11*** another variant to ***rename*** a data type was added
- An ***alias declaration*** is used to declare a ***name*** to use as a ***synonym*** for a ***previously declared type***, similar to **`typedef`** from the ***C language***:

```
using <identifier> = <type>;
```

Examples:

```
using counter = long;
typedef long counter; // is similar
```

- Aliases also work with function pointers, but are much more readable than the equivalent typedef:

```
using func = void(*)(int);
typedef void (*func)(int);
```

- A limitation of **`typedef`** is that it doesn't work with ***templates***. However, the ***type alias*** syntax in C++11 enables the creation of ***alias templates***:

```
template<typename T> using Ptr = T*;
// Ptr<T>' is an alias for a pointer to T
Ptr<int> ptrInt;
```

f) Uniform initialization

- ***Uniform initialization*** is a feature in ***C++11*** that allows the usage of a ***consistent*** syntax to initialize ***variables*** and ***objects*** ranging from ***primitive type*** to ***aggregates***
- It introduces ***brace-initialization*** that uses braces **{ }** to enclose ***initializer values***
- **Syntax**:

```
<type> <variable> {<argument list>};
```

Examples.

I. Classical syntax:

```
int i;      // uninitialized built-in type
int j=5;    // initialized built-in type
int k(5);   // initialized built-in type
int a[]={1, 2, 3, 4}; // array initialization
```

II. New syntax

```
int i{};       // uninitialized built-in type
int j{5};      // initialized built-in type
int a[]{1, 2, 3, 4}; // array initialization
```

- ❑ ***Aggregate initialization*** initializes an ***aggregate*** from a ***braced-init-list***
- ❑ An ***aggregate*** is one of the following types:
  - ○ ***array*** type
  - ○ ***class*** type:
    - ▪ ***struct*** or ***union*** that has no ***private*** or ***protected*** data members

Examples (for arrays):

```
int a[]{1, 2, 3, 4};  // array initialization
char a[] = "abc";     // classic character array
// char a[4] = {'a', 'b', 'c', '\0'};
char b[]{"abc"};      // aggregate initialization
// char b[4] = {'a', 'b', 'c', '\0'};
char c[5]{"abc"};     // aggregate initialization
// char b[5] = {'a', 'b', 'c', '\0', '\0'};
```

Examples (for structures):

```
struct S { char c; double x; int n; };
// aggregate initialization with initializer list
S a{‘t’, 2.5, 2};
S b{‘u’, 1.5};  // OK - incomplete initializer list
// S b{‘u’, 1.5, 0};

struct A {
    int n;
    struct B { int i; int j; int a[3]; } b;
};
A a1 = {1, {2, 3, {4, 5, 6}}};   // classical
A a2 = {1, 2, 3, 4, 5, 6};    // same, brace elision
// same, direct-list-initialization syntax
A a3{1, {2, 3, {4, 5, 6}}};
// until C++14, error:
// brace-elision only allowed with equals sign
A a4{1, 2, 3, 4, 5, 6};
```

g) Structured bindings

- Starting to ***C++17***, ***structured bindings*** allows a way define ***several objects*** instead of ***one***, in a ***more natural way*** than in the previous versions of C++
- ***Structured bindings*** gives the ability to declare ***multiple variables initialized*** from a ***composite object*** (an ***array***, a ***struct***, or a ***tuple***)
  - Like a ***reference***, a ***structured binding*** is an ***alias*** to an ***existing object***
  - Unlike a reference, the ***type*** of a ***structured binding*** does not have to be a ***reference type***
- ***Syntax*** can have 3 forms:

```
auto [<identifier-list>] = <expression>;
auto [<identifier-list>] {<expression>};
auto [<identifier-list>] (<expression>);
```

where:

- **`<identifier-list>`** : a list of comma separated ***variable names***
- **`<expression>`** : an expression that ***does not have the comma operator at the top level***, and has ***either array*** or ***non-union class type***

- The `auto` keyword can optional be followed by a ***reference operator*** (`&` for ***lvalue references***, or `&&` for ***rvalue references***)
- ***Inference type deduction***. Let `E` denote the ***type of the initializer expression***. Then:
  - if `E` is an ***array*** type,
    - then the ***names*** are ***bound*** to ***the array elements***
  - if `E` supports `tuple_size<E>` and provides `get<N>()` function (***tuples*** from ***STL*** library or other ***containers*** similar to tuple: ***pair***, …),
    - then the "***tuple-like***" ***binding protocol*** is used
  - if `E` contains only ***non static***, ***public members***,
    - then the ***names*** are ***bound*** to the ***accessible data members*** of `E`

Examples:

- Case 1: ***binding an array***:

```
int a[3] = {1, 2, 3};
// x is a copy of a[0], y is a copy of a[1], …
auto [x, y, z] = a;   // x==1, y==2, y==3
auto& [u, v, t] = a;  // u==a[0], v==a[1], t==a[2]
```

- Case 2: ***binding a tuple-like type***:

```
#include <tuple>
#include <string>
tuple<int, double, string> tp(5, 1.2, “abc”);
auto [n, x, s] = tp;  // n==5, x==1.2, s==”abc”
```

- Case 3: ***binding to data members***:

```
struct Point {
    int x;
    int y;
};
Point p{3, 5};
auto [xp, yp] = p;    // xp==3, yp==5
```

Example. A more practical example for using the ***structured bindings***: ***iterating*** over a ***compound collection***.

```
#include <iostream>
#include <utility>     // for pair container
#include <map>         // map container
using namespace std;
typedef pair<double, double> Coord;     // geog. coord.
int main() {
    map<string, Coord> cities;    // map of cities
    Coord c1(44.339241, 23.796380);
    Coord c2(44.434053, 26.120410);
    Coord c3(45.670482, 25.575787);
    // insert in the map
    cities[“Craiova”] = c1;
    cities[“Bucharest”] = c2;
    cities[“Brasov”] = c3;
    // iterating over the map
    for (auto& [name, coord] : cities) {
        cout << “City: “ << name << endl;
        cout << “lat. = “ << coord.first << endl;
        cout << “long. = “ << coord.second << endl;
    }
    return 0;
}
```

h) **Binary literals** (since C++14)

- ❑ A ***binary literal*** is compound by the character sequence **`0b`** or **`0B`**, followed by one or more binary digits (0, 1)
- ❑ The ***data type*** of a binary literal is ***integer***

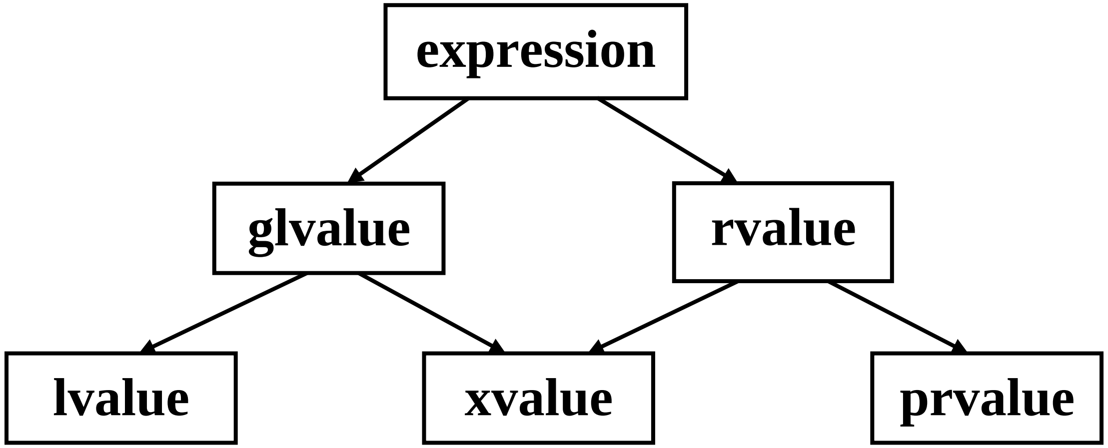
Example.

```
int b1 = 0b101011;        // 43
long int b2 = 0B101010;  // 42
```

C2. Expressions

a) Type of expressions

- Before C++11, the expressions were of two types: ***lvalue*** and ***rvalue***
- Starting to C++11 there are several types for expressions:
  - ***glvalue***, ***rvalue***
  - ***lvalue***, ***xvalue***, ***prvalue***



expression

glvalue

rvalue

lvalue

xvalue

prvalue

- The reason is the introduction of new concepts such as ***move semantics***, ***move constructor***, ***move assignment operator*** and ***rvalue reference***

- The main types are:
    - ***lvalue*** (*Left value*, as before): designates an object, a location in memory
    - ***xvalue*** (*eXpiring value*): an object towards the end of its’ lifetime (typically used in move semantics)
    - ***prvalue*** (*Pure rvalue*): represents an actual value (which is temporary)
- ***glvalue*** means *Generalized lvalue*, which is a *lvalue* or a *xvalue*
- The meaning of ***rvalue*** (*Right value*) has evolved with the introduction of *move semantics*, and it represents a *xvalue* or a *prvalue*

b) `decltype` specifier

- Yields the type of its operand, which is not evaluated
- For a construct **`decltype(expr)`**:
    - If the operand **`expr`** is a class member access without any additional parentheses, then **`decltype(expr)`** is the declared type of the member accessed

Example:

```
struct S {
    int x = 42;
};
const S s;
decltype(s.x) y;
// Equivalent: int y,
// even though s.x is const
```

- In all other cases, **`decltype(expr)`** yields both the *type* and the *value category* of the expression **`e`**, as follows:
  - If **`expr`** is a *lvalue* of type **`T`**, then **`decltype(expr)`** is **`T&`**
  - If **`expr`** is a *xvalue* of type **`T`**, then **`decltype(expr)`** is **`T&&`**
  - If **`expr`** is a *prvalue* of type **`T`**, then **`decltype(expr)`** is **`T`**
- If the name of an object is parenthesized, it is treated as an ordinary *lvalue* expression

- **Remark**. **`decltype`** does not drop the *reference* and the const qualifier. **Example**:

```
const int cx = 42;
const int& crx = x;
auto a = cx;    // a is int
auto b = crx;   // b is int
typedef decltype(cx) cx_type;     // cx_type is const int
typedef decltype(crx) crx_type;   // crx_type is const int&
```

- **Some examples**:

```
int x = 0;
int y = 0;
const int c1 = 42;
const int c2 = 43;
double d1 = 3.14;
double d2 = 2.72;

// the type of the product is int,
// the product is a prvalue => type of xy_type is int
typedef decltype(x * y) xy_type;
// the type of the product is int (not const int),
// the product is a prvalue => type of c1c2_type is int
typedef decltype(c1 * c2) c1c2_type;
```

```
// the type of expression is double,
// expression is a lvalue => type of cond_type is double&
typedef decltype(d1 < d2 ? d1 : d2) cond_type;
// the type of expression is double,
// the expression is a prvalue,
// because for translating x to a double,
// a temporary object has to be created
// => type of cond_type1 is double
typedef decltype(x < d2 ? x : d2) cond_type1;
auto c = 0;         // c has type int
auto d = c;         // d has type int
decltype(c) e;      // e has type int, the type of c
// f has type int&, because (c) is a lvalue
decltype((c)) f = c;
// g has type int, because 0 is a rvalue
decltype(0) g;
int f() { return 42; }
int& g() { static int x = 42; return x; }
int x = 42;
decltype(f()) a = f(); // a has type int
decltype(g()) b = g(); // b has type int&
```

- Since C++14, the special form **`decltype(auto)`** :
    - deduces the type of a variable from its initializer, or the return type of a function from the **`return`** statements in its definition,
    - using the type deduction rules of **`decltype`** rather than those of **`auto`**
- **Example**:

```
const int x = 123;
auto y = x;        // y has type int
// z has type const int, the declared type of x
decltype(auto) z = x;
```

c) `constexpr` specifier

- Initially **`constexpr`** was a feature added in C++11 for performance improvement of programs:
    - Performing computations at *compile time* rather than *run time*
        - It is better to spend time in compilation and save time at run time

- Mainly, **`constexpr`** specifies that the value of a *variable* (C++11) or a *function* (C++14) ***can*** be evaluated at compile time and the expression can be used in other *constant expressions*

- A **`constexpr`** *variable* must satisfy the following requirements:
  - It must be immediately *initialized* (as in the **`const`** case)
  - The *initialization* expression must be a *constant expression*
- A **`constexpr`** function must satisfy the following requirements:
  - It must consist of single **`return`** statement
  - It can call only other **`constexpr`** functions
  - It can reference only **`constexpr`** global variables

Example. Consider the following program:

```
#include <iostream>
using namespace std;
constexpr long long int fib(int n) {
    return (n <= 1)? n : fib(n-1) + fib(n-2);
}
```

```
int main () {
    const long long int v = fib(50);
    cout << v;
    return 0;
}
```

- Running on some ***mingw*** compiler the above program takes **0.187 seconds**
- Replacing

```
const long long int v = fib(50);
```

by

```
long long int v = fib(50);
```

on the same compiler the program takes **123.864 seconds**

- The compiling time is reverse: 12 seconds / 1 second
- Because a **`constexpr`** function must have only one return statement, in the case of recursive functions, the *conditional* operator has to be used
- The keywords **`constexpr`** and **`const`** serve different purposes:
  - **`constexpr`** is mainly for *optimization*
  - while **`const`** is for defining *constant* objects

- The principal difference between **`const`** and **`constexpr`** is the time when their initialization values are evaluated:
  - while the values of **`const`** variables can be evaluated at both compile time and runtime,
  - **`constexpr`** are always evaluated at compile time.
- For **example**:

```
int t = rand();           // t is generated at runtime
const int x1 = 10;        // OK - known at compile time
const int x2 = t;         // OK - known only at runtime
constexpr int x3 = 10;    // OK - known at compile time
constexpr int x4 = t;     // ERROR - known only at runtime
```

- There is some similarity between **`constexpr`** functions and ***template metaprogramming*** (*compile-time programming, static metaprogramming*)
- Example of a **`constexpr`** function for factorial:

```
constexpr int factorial (unsigned int n) {
    return (n <= 1 ? n : n * factorial(n-1));
}
```

```
int main () {
    const int f = factorial(10);
    cout << f; }
```

- And the same action by using the ***template metaprogramming***:

```
template <int N>
struct Factorial {
    static const int res = N * Factorial<N-1>::res;
};
template <>
struct Factorial<0> {
    static const int res = 1;
};
int main () { cout << Factorial<10>::res; }
```

C3. Inline variables (C++17)

- Global variables, and static variable can be declared as **`inline`**
- The same rules applied to **`inline`** *functions* are applied to **`inline`** *variables*:
  - There can be *more than one* definition of an **`inline`** variable
  - The definition of an **`inline`** variable must be present in the *translation unit*, in which it is used
  - A *global* **`inline`** variable must be declared inline in every translation unit and *has the same address in every compilation unit*
- As a general benefit, an **`inline`** variable can be defined into a header file and included them more than once in other translation units
- If there is a need to declare *global variables* that are *shared* between several *compilation units*, declaring them as **`inline`** variables in a *header file* is simple and avoids some problems with pre-C++17 workarounds
- For example, one workaround is to use the Scott Meyer *singleton* with an **`inline`** function, which has some drawbacks in terms of performance:

```
// head.h
inline int& instance() {
  static int globalVar;
  return globalVar;
}
// pr1.cpp
#include “head.h”
int a = instance();
// pr2.cpp
#include “head.h”
int a = instance();   // the same global variable
```

- With inline variables, this variable can be directly declared, without getting a multiple definition linker error:

```
// head.h
inline int a;
// pr1.cpp
#include “head.h”
int b = a;
// pr2.cpp
#include “head.h”
int c = a;   // the same global variable
```

C4. Statements

a) Range-based for loop

- Executes a for ***loop*** over a ***range***
- Used as a *more readable* equivalent to the traditional for loop operating over a *range of values*, such as all elements in a *container*
- Syntax:

  `for (` *range_declaration* **`:`** *range_expression* **`)`** *loop_statement*

  - ***range_declaration***: a declaration of a *named variable*, whose type is the type of the element of the sequence represented by *range_expression*, or a *reference* to that type; often uses the **`auto`** specifier for automatic type deduction
  - ***range_expression***: any expression that represents a suitable *sequence*, or a *braced-init-list* (a list of elements between braces)

- **Examples**:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main() {
   // Iterating over array
   int a[] = {1, 2, 3, 4, 5};
   for (auto n : a)
      cout << n << ' ';
   // Iterationg over string characters
   string str = "Language";
   for (auto c : str)
      cout << c << ' ';
   // Iterating over an array
   vector<int> v = {10, 11, 12, 13, 14};
   for (auto i : v)
      cout << i << ' ';
}
```

b) `if` statement with `constexpr` and `init statement`

- Since *C++17* the syntax of the **`if`** statement was modified:

```
if [constexpr] ( [<init-statement>;] <condition> )
   <statement-true>   //Discarded if condition is false
[else
   <statement-false>  //Discarded if condition is true
]
```

- The keyword **`constexpr`** is *optional*. If it is used:
  - The *condition* is evaluated at *compile time*
  - *Determines* which of the two sub-statements *to compile*, *discarding* the other
    - This means that one *branch* can be *rejected* at *compile time*, and thus *will never get compiled*

Example. A function **`get`** that works in a similar way as in the case of STL **`tuple`** container.

```
#include <iostream>
#include <string>
using namespace std;
struct triple {
   int n;
   double x;
   string s;
};
template <size_t I>
auto& get(triple& t) {
   if constexpr (I == 0)
      return t.n;
   else if constexpr (I == 1)
      return t.x;
   else if constexpr (I == 2)
      return t.s;
}
int main() {
   triple t{5, 5.5, "string"};
   cout << get<0>(t) << ", " << get<1>(t) << endl;
}
```

- The **`<init-statement>`** is optional. It is similar to the ***init*** expression from the **`for`** statement.
- The following code:

```
    <init-statement>;
    if (<condition>)
        <statement-true>;
    else
       <statement-false>;
```

is similar to:

```
    if (<init-statement>; <condition>)
        <statement-true>;
    else
       <statement-false>;
```

- The ***scope*** of the ***conditioned variable*** is ***limited*** to the ***current if-else block***
  - This also allows us to ***reuse*** the same named identifier in ***another conditional block***.
    - Which in turn ***avoids variable leaking*** outside the scope

Example.

```
#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;
int main() {
   srand((unsigned)time(NULL));
   if (int rn = rand(); rn % 2 == 0) {
      cout << rn << " is an even number\n";
   } else {
      cout << rn << " is an odd number\n";
   }
   return 0;
}
```

c) `switch` statement with `init statement`

- Similar to the **`if`** statement (**`<init-statement>`** is optional):

```
switch ( [<init-statement>;] <condition> )
   <statement>
```

Example.

```
int integerType(const string &s) {
   // determine the type of an integer literal
   // returns:
   // 1: decimal type (ex. 183)
   // 2: octal type (ex. 017)
   // 3: hexadecimal type (ex. 0x1a3, 0X27c)
   // 4: binary type (ex. 0b101, 0B11)
   // 5: unknown type
   // Implement this function
}
```

```
void printIntegerType(const string &s) {
   switch(auto t = integerType(s); t) {
   case 1:
      cout << “decimal type\n”;
      break;
   case 2:
      cout << “octal type\n”;
      break;
   case 3:
      cout << “hexadecimal type\n”;
      break;
   case 4:
      cout << “binary type\n”;
      break;
   default:
      cout << “unknown type\n”l
   }
}
```

C5. Lambda functions

- Will be discussed later