Programarea C++0x /++1y/2z

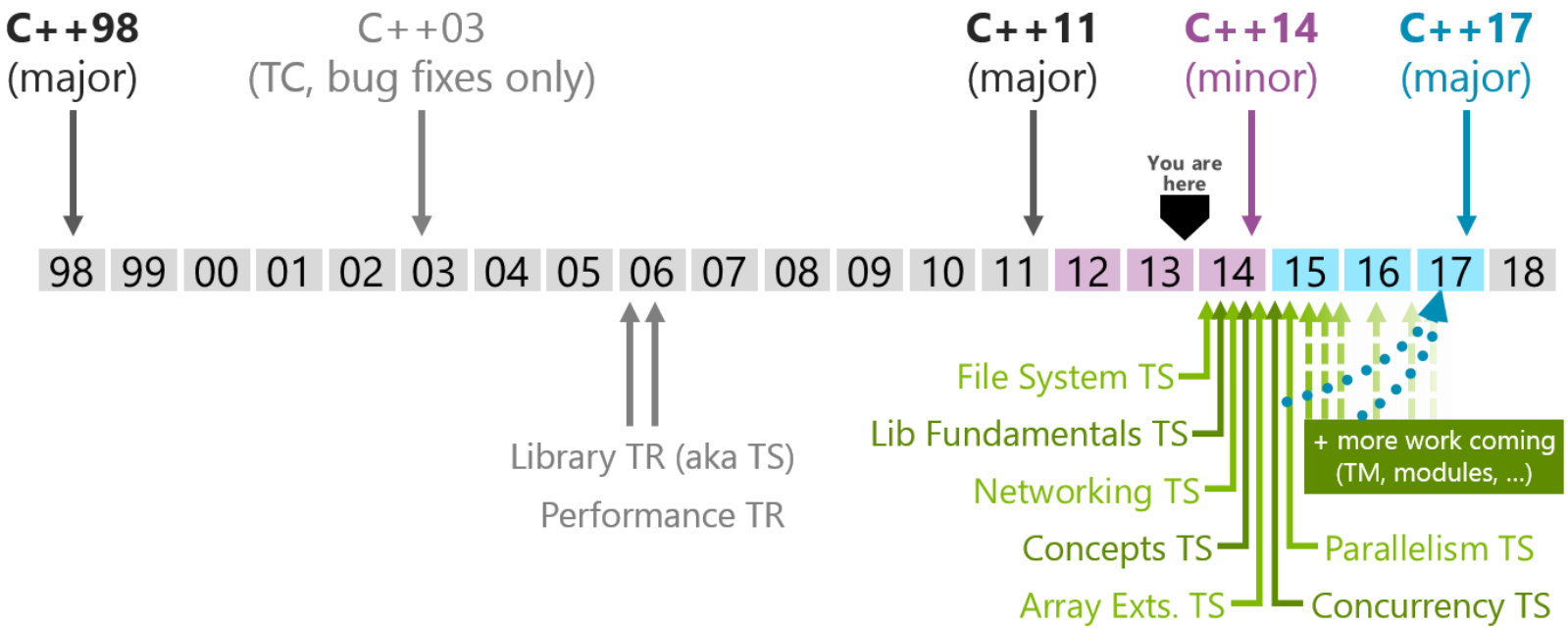
Evolutie, tendinte.

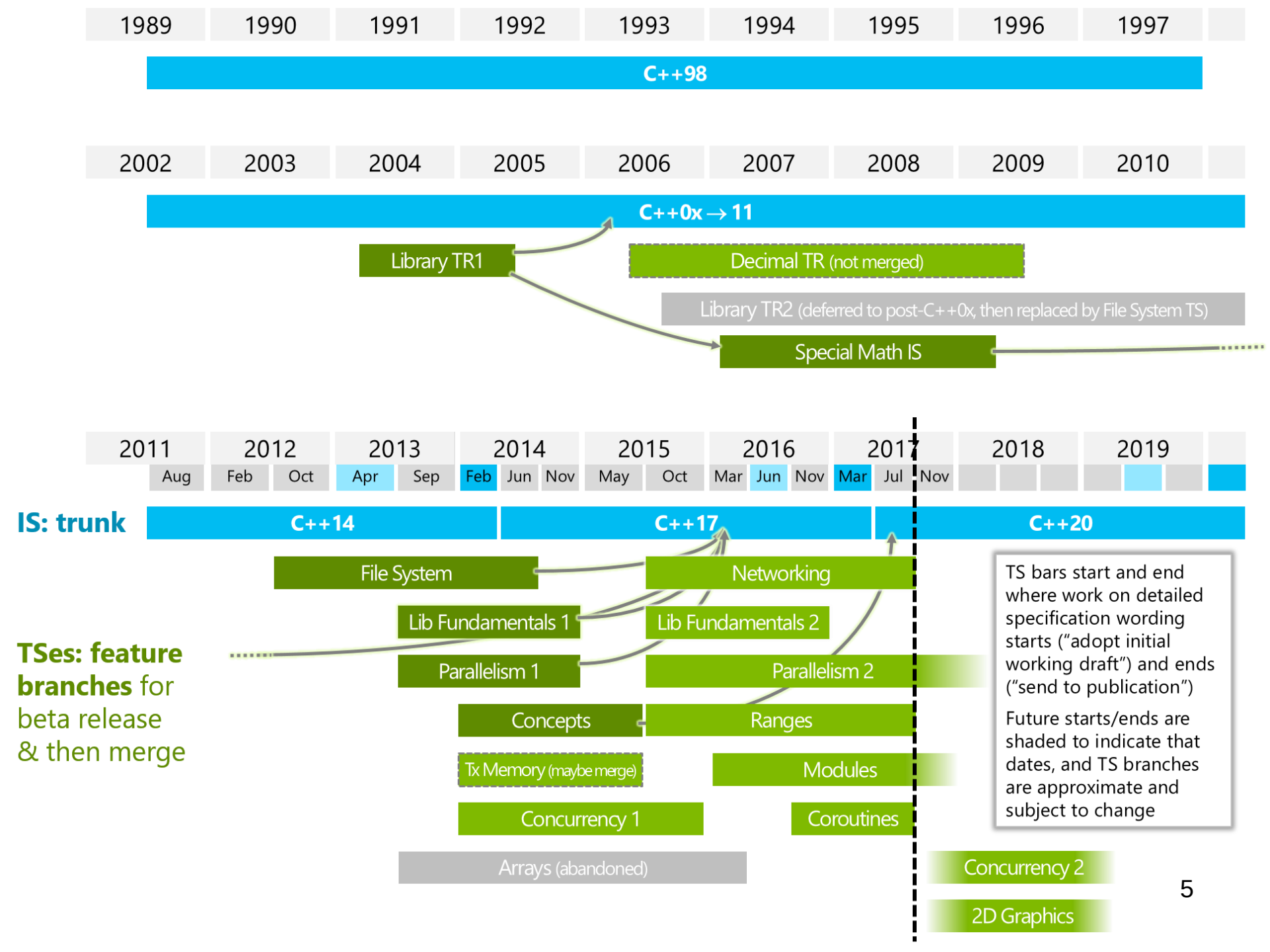
Cuprins

- Privire de ansamblu
- Prezentare generală
- Imbunătățiri aduse noii versiuni
  - Performanțe de utilizare
  - Imbunatatiri ale functionalitatilor
  - Modificările aduse bibliotecilor standard
- Concluzii
- Bibliografie

Privire de ansamblu

- C++0x a aparut in 2003 ca un proiect sustinut de B. Stroustrup pentru a mentine limbajul C++ pe piata limbajelor moderne
- **C++11**, aprobat de ISO în 12 august 2011, reprezintă prima versiune **1y** a limbajului de programare C++, proiect inițiat de Bjarne Stroustrup, versiune integrata in mediile de programare (VC++2012-13, etc.)
- A fost publicat în septembrie 2011, denumirea oficială fiind “ISO/IEC 14882: 2011”
- In 16 ianuarie 2012 sunt doar corectii editoriale ale standardului C++11.
- **C++14**, ISO/IEC 14882:2014(E), 18 august 2014, e urmatoarea versiune, cu modificari minore fata de C++11
- C++17 a aparut, cu detalii la: https://isocpp.org/std/status
- C++20 e in curs de revizie finala
- C++23 se preconizeaza a fi urmatoarea varianta


C++98
(major)
C++03
(TC, bug fixes only)
C++11
(major)
C++14
(minor)
C++17
(major)
You are here
98 99 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18
Library TR (aka TS)
Performance TR
File System TS
Lib Fundamentals TS
Networking TS
Concepts TS
Array Exts. TS
Parallelism TS
Concurrency TS
+ more work coming
(TM, modules, ...)


1989
1990
1991
1992
1993
1994
1995
1996
1997
C++98
2002
2003
2004
2005
2006
2007
2008
2009
2010
C++0x → 11
Library TR1
Decimal TR (not merged)
Library TR2 (deferred to post-C++0x, then replaced by File System TS)
Special Math IS
2011
2012
2013
2014
2015
2016
2017
2018
2019
Aug
Feb
Oct
Apr
Sep
Feb
Jun
Nov
May
Oct
Mar
Jun
Nov
Mar
Jul
Nov
IS: trunk
C++14
C++17
C++20
File System
Networking
Lib Fundamentals 1
Lib Fundamentals 2
Parallelism 1
Parallelism 2
Concepts
Ranges
Tx Memory (maybe merge)
Modules
Concurrency 1
Coroutines
Arrays (abandoned)
Concurrency 2
2D Graphics
TSes: feature branches for beta release & then merge
TS bars start and end where work on detailed specification wording starts ("adopt initial working draft") and ends ("send to publication")
Future starts/ends are shaded to indicate that dates, and TS branches are approximate and subject to change

Privire de ansamblu

- **Istoric (standardizări):**
  - 1998, ISO/IEC 14882:1998 (C++98)
  - 2003, ISO/IEC 14882:2003 (C++03)
  - 2007, ISO/IEC TR 19768:2007 (C++TR1)
  - 2011, ISO/IEC 14882:2011 (C++11)
  - 2014, ISO/IEC 14882:2014 (C++14)
  - 2017, ISO/IEC 14882:2017 (C++17)
  - 2020, ISO/IEC 14882:2020 (C++20)

Prezentare generală

- **Principalele îmbunătățiri includ suportul pentru multithreading, noi facilități de programare generică, inițializare uniformă și performanțe sporite.**
- **Secțiuni generale urmărite:**
  - Performanțele timpilor de rulare (run-time performance)
  - Performanțele build-time (timp de generare) (build-time performance)
  - Performanțe de intrebuintare (usability enhancements)
  - Noi functionalități

Performanțe de intrebuintare

- **Caracteristicile îmbunătățite aduse de C++0x/1y/2z, fac acest limbaj mai ușor de utilizat**

- **Ele minimizează necesitatea repetării codului, reduc rata de aparitie a erorilor, etc.**

Performanțe de intrebuintare

- **Performanțele sunt aduse de:**
  - **Initializer lists --**sunt extinse și pot fi folosite pentru orice clase, inclusiv pentru containere
  - **Initializare uniforma**
  - **Bucla for -- pentru colectii cu** sintaxă foarte simpla (for – range)
  - **Functii Lambda --**functii anonime, nu au identificator, iar tipul returnat e implicit
  - **Sintaxa alternativă a funcțiilor, etc.**

Performanțe de intrebuintare

- **Imbunătățirea constructorilor** --constructorii pot apela alți constructori din aceeași clasă
- **Constanta: pointer *null*** **--**se schimbă denumirea pentru pointerul null: din “0” in “***nullptr***” pentru a nu exista ambiguități, cu pastrarea si a lui *null* = 0
- **Enumerari in condiții de siguranță** **--**valorile dintr-o enumerare nu sunt convertite implicit în integer
- **Template-uri alias** **--**se crează template-uri typedef, etc.

Liste de inițializare (initializer lists)

- C++0x/1y/2z a “moștenit” initializer lists de la C. Pentru orice structură sau tablou de structuri se dă între acolade o listă de argumente. Aceste liste sunt recursive, deci un tablou de structuri, sau o structură conținând alte structuri, poate sa le folosească.
- Acest lucru este foarte util pentru liste statice sau pentru a inițializa o structură cu o valoare anume.

```
struct Object
{
float first;
int second;
};
```

- *Object scalar = {0.43f, 10}; //One Object, with first=0.43f and second=10*
- *Object anArray[] = {{13.4f, 3}, {43.28f, 29}, {5.934f, 17}}; //An array of three Objects*

- C++ oferă de asemenea constructori pentru inițializarea unui obiect, dar de multe ori nu sunt la fel de utili ca și initializer lists. Aceste liste sunt extinse, astfel încât să poată fi folosite pentru orice clase, inclusiv pentru containere, ca de exemplu *std::vector*.
- Acest concept este legat și de un template: *std::initializer_list*. Acest lucru permite constructorilor și altor funcții să ia ca parametru initializer lists.
- *class SequenceClass {*
- *public:*
- *SequenceClass(std::initializer_list<int> list); };*
- În exemplul de mai sus, un obiect din *SequenceClass* poate fi construit (inițializat) astfel:
- *SequenceClass some_var = {1, 4, 5, 6};*
- Acest constructor este unul special, numit initializer list constructor.

Inițializare uniformă

- C++0x/1y/2z oferă posibilitatea de inițializare uniformă, ce se scrie între acolade.
- În exemplul urmator se poate vedea cum se inițializează un tablou unidimensional:

```
int i[ ]={1,4,9};
```

- Diferența față de versiunile anterioare de C++ este că se poate folosi aceeași sintaxă pentru a inițializa orice obiect.
- Mecanismul poate fi folosit si la initializari simple de variabile. Astfel se simplifică structura codurilor scrise.
- O particularitate o reprezintă inițializarea containerelor

- Până acum era destul de derutantă inițializarea ca în exemplul urmator:

```
string a[ ]= {"hello", "bye"};
vector<string> v(a,a+1);
```

- Acest exemplu se poate scrie în C++0x/1y/2z astfel:

```
vector<string> v= {"hello", "bye"};
```

- **Alt exemplu:**

```
struct IdString { std::string name; int identifier;};

IdString get_string(){
return {"foo", 42}; //se poate observa lipsa mentionarii tipului de data folosita
 }
```

Bucla for range (Range based for)

- În C++0x/1y/2z s-a creat o sintaxa mult mai simplă pentru instrucțiunea “for”, asa cum în alte limbaje de programare precum Java și C# există o sintaxa simplă de tip for_each, sau pentru functia STL “for_each”, care parcurge un șir, un vector, etc., de la început la sfârșit.

```
int my_array[5] = {1, 2, 3, 4, 5};

for (int x : my_array) {

x *= 3;

}
```

- Sintaxa instrucțiunii *for( )* funcționează pentru parcurgerea vectorilor, initializer lists, și a oricărui tip care are funcții *begin()* și *end()* definite pentru el, care returnează iteratori. Toate containerele care au perechea *begin*/*end* vor funcționa folosind sintaxa instrucțiunii *for( )*.

- Sintaxa ***for-range***:

attr(optional) for (range_declaration : range_expression)
loop_statement

- Execută o buclă pentru pentru un interval, fara control pe fiecare element.
- Folosit ca un echivalent mai lizibil cu bucla tradițională pentru funcționarea pe o gamă de valori, cum ar fi toate elementele dintr-un container.

Example range-based loop

```
#include <iostream>
#include <vector>
int main( ) {
int a[ ] = { 0, 1, 2, 3, 4, 5 };
for (int i = 0; i < sizeof(a) / sizeof(int); i++)// standard for
std::cout << a[i] << ' ';
std::cout << '\n';
for (int i : a) // the initializer may be an array
std::cout << i << ' ';
std::cout << '\n';
for (int i : {0, 1, 2, 3, 4, 5}) // the initializer may be a braced-init-list
std::cout << i << ' ';
std::cout << '\n';
for (int i : a)
std::cout << 7 << ' '; // the loop variable need not be used
std::cout << '\n';
```

```
std::vector<int> v = { 0, 1, 2, 3, 4, 5 };
for (const int& i : v) // access by reference
std::cout << i << ' ';
std::cout << '\n';
for (auto i : v) // access by value, the type of i is int
std::cout << i << ' ';
std::cout << '\n';
for (auto&& i : v) // access by forwarding reference, the type of i is int&
std::cout << i << ' ';
std::cout << '\n';
const auto& cv = v;
for (auto&& i : cv) //access by forward reference, the type of i is const int&
std::cout << i << ' ';
std::cout << '\n';
}
```

Switch range *gcc* new compilers, no VC++19

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

int main( ){
    int score;
cout << "Score values 0-100:";
   cin >> score;
    //Switch
    switch(score){
       case 0:
            cout << "aa"; break;
       case 1 ... 10:
            cout << "bb"; break;
       case 11 ... 24:
            cout << "cc"; break;
       case 25 ... 49:
            cout << "dd"; break;
       case 50 ... 100:
            cout << "ee"; break;
        default:      cout << "BAD VALUE";
                }
    cout << endl;
}//end_main
```

Multiple *switch-case*

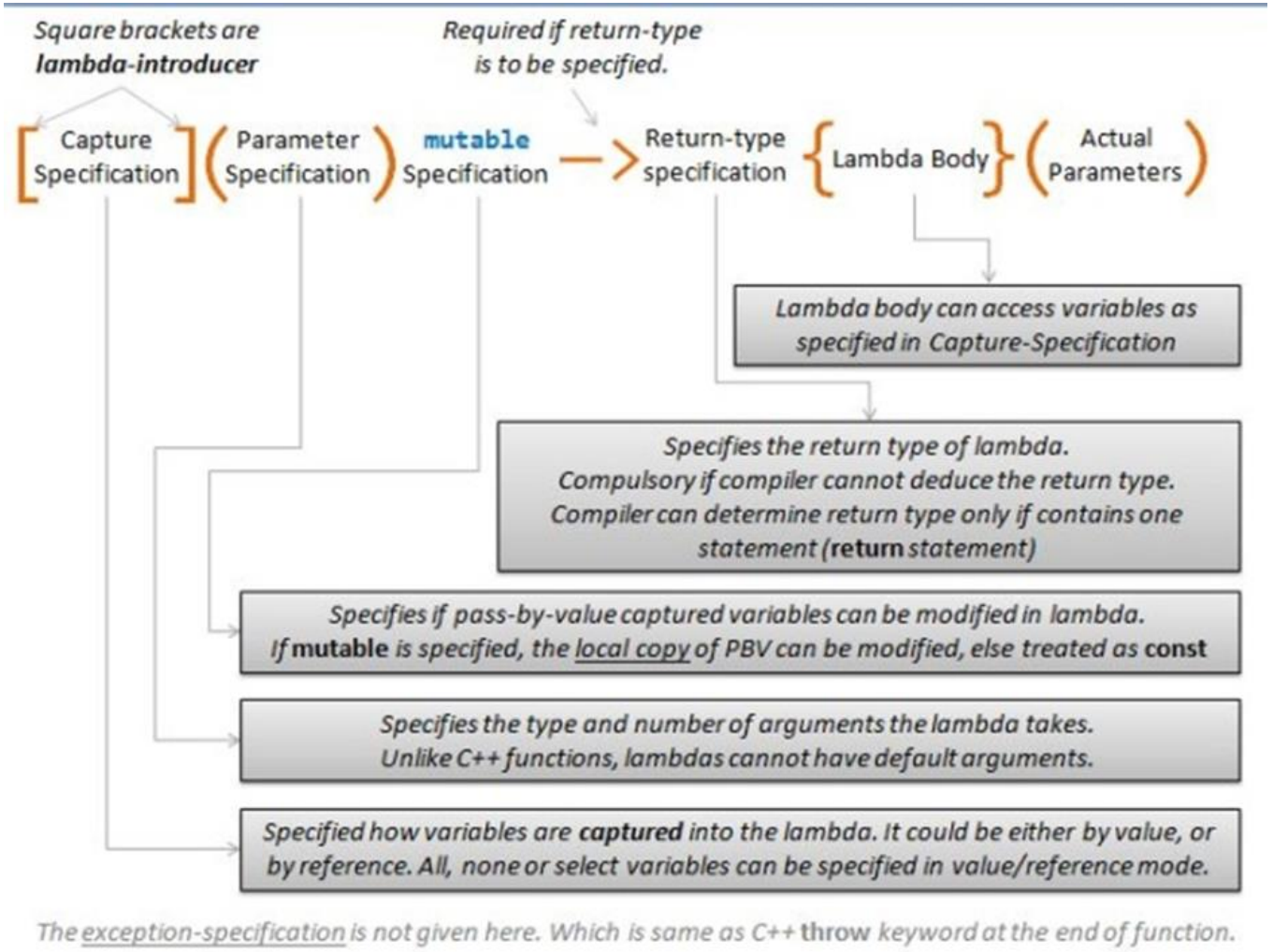
```
//Multiple switch-case
#define _CRT_SECURE_NO_WARNINGS
#include <cwchar> // wide functions and types

int main( ) {
	wchar_t ch;
	wprintf(L"\nEnter a wide character:");
	//wscanf_s(L" %lc", &ch, sizeof(ch));
	wscanf(L" %lc", &ch);
	switch (ch) {
	case 'a':
	case 'e':
	case 'i':
	case 'o':
	case 'u': wprintf(L"\nIt is a vowel! %lc", ch); break;
	default: wprintf(L"\nIt is a consonant! %lc", ch);
	}
}//end_main
```

Funcții lambda

- se mai numesc și funcții anonime;
- se declară inline;
- tipul de date returnat este unul implicit;
- tipul de date returnat de aceste funcții este unul de forma ***decltype(expresia_funcției)***;

```
    auto lambda_media =[](int n1,int n2,int n3,int n4,int n5)
{
    return (n1 + n2 + n3 + n4 + n5)/5.0;
};
```



The exception-specification is not given here. Which is same as C++ **throw** keyword at the end of function.

Funcții lambda

O funcție lambda este de fapt o funcție anonimă (adica o funcție fără nume). Este deosebit de utilă când un programator are nevoie de o funcție mică, pentru care nu este necesară scrierea unei funcții obișnuite.

Exemplu:

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

class Apple{
public:
double m_weight; // Greutatea marului in kg
int m_age; // Varsta marului in zile
Apple(double weight = 1., int age = 1) { m_weight=weight;
m_age=age;}
friend ostream& operator<<(ostream &, Apple ob);
}//Apple_class;

ostream& operator<< (ostream& stream, Apple ob) {
stream << ob.m_weight<< " " << ob.m_age << '\n';
return stream;
}
```

```
int main( ) {
vector<Apple>myApples;
myApples.push_back(Apple(0.50, 35));
myApples.push_back(Apple(0.75, 40));
myApples.push_back(Apple(0.35, 37));

// Sort the apples by their weight
sort(myApples.begin( ), myApples.end( ),[ ](const Apple a, const Apple b) -> bool {
return (a.m_weight < b.m_weight); });
for(int i = 0; i < myApples.size( ); ++i)
cout << myApples[i] << ' ';
cout << endl;
}
```

- Se poate observa în funcția de sortare prezența funcțiilor lambda. Sintaxa simplificata a unei astfel de funcții este urmatoarea:

```
[ ](Type1 parameter1, Type2 parameter2, ...) -> ReturnType {
// Function contents
}
```

- Funcțiile lambda pot face referire și la variabile care sunt în afara funcției. Acestea intra în asa numitul “closure” al funcției.
- Exemplu:

```
int appleCount = 0; int orangeCount = 2;
 [appleCount, orangeCount]( )
{
++appleCount;
++orangeCount;
}( );
// appleCount == 1
// orangeCount == 3
```

Variabilele locale din sfera exterioară functiilor Lambda pot fi capturate (captures) in 2 moduri adică:

- prin valoare

- prin referință.

- **Exemplu:**

```
#include <iostream>
#include <string>
int main(int argc, char **argv){
    std::string msg = "Hello";
    int counter = 10;
    // Defining Lambda function and  capturing Local variables by Value
    auto func = [msg, counter] ( ) mutable { //auto func = [&msg, &counter]  -by reference
                std::cout<<"Inside Lambda :: msg = "<<msg<<std::endl;
                std::cout<<"Inside Lambda :: counter = "<<counter<<std::endl;
    // Change the counter msg, and counter captured by value in Lambda function
                msg = "Temp";
                counter = 20;
                std::cout<<"Inside Lambda :: After changing :: msg = "<<msg<<std::endl;
                std::cout<<"Inside Lambda :: After changing :: counter = "<<counter<<std::endl;
                };
    //Call the Lambda function
    func( );
    //Values of local variables are not changed by value, changed by reference
    std::cout<<"msg = "<<msg<<std::endl;
    std::cout<<"counter = "<<counter<<std::endl;
}//main
```

Sintaxa alternativă a funcțiilor

- Sintaxa de declarare a funcțiilor pentru C s-a modificat în anumite situații o dată cu trecerea la C++. S-au impus anumite limitări, în special pentru modul în care se declară template-urile.
- În C++03, pentru a declara un template se folosește urmatoarea sintaxă:

```
template<typename Lhs, typename Rhs>
ret_v adding_func(const Lhs lhs, const Rhs rhs)
        {return lhs + rhs;},
```

unde *ret_v* va fi de tipul *lhs+rhs*

În C++1y/2z sintaxa devine:

```
template< typename Lhs, typename Rhs>
auto adding_func(const Lhs lhs, const Rhs rhs)
 -> decltype(lhs+rhs)
        {return lhs + rhs;}
```

Sintaxa alternativă a funcțiilor

- este un concept pentru noul mod folosit pentru declararea unei funcții;

- acest mod de declarare fost introdus odată cu lansarea noilor versiuni ale limbajului C++;

```
auto student::retnumar_matricol() -> int
{
    return numar_matricol;
}
```

```
auto student::return_media() -> double
{
    return media;
}
```

Noi tendințe aduse de limbajul “C++1Y”

- Deducerea automată a tipului cu ajutorul tipului *auto;*
- Expresii regulare;
- Tipul decltype;
- Compilarea în timp a expresiilor constante;
- Referințe R-value;
- Pointeri smart;
- Tabele hash;

Deducerea automată a tipului cu ajutorul *auto*

- se face remarcat prin utilizarea tipului de dată auto;

- deduce automat tipul unei date dintr-o expresie;

- declară o variabilă fară a specifica tipul ei;

- compilatorul va deduce tipul unei variabile auto din expresia de inițializare a acesteia (valoare de return);

```
void student::media_semestiala()
{
    auto lambda_media =[](int n1,int n2,int n3,int n4,int n5)
{
    retur
```

double lambda_media(int n1, int n2, int n3, int n4, int n5)

Expresii regulare

- se găsesc în noua bibliotecă <regex>;

- sunt folosite pentru validare;

- conțin noi clase (regex_search, regex_replace, match_result);

- prin utilizarea acestor clase, performanțele legate de timpul de execuție și spațiul de stocare sunt mult mai ridicate;

```
regex pattern(" ");
while(!regex_search(nume,pattern))
{
    cout<<"\nNu ati introdus corect, Reintroduceti numele de forma: Nume Prenume !!";
    cin.getline(nume,30);
}
```

Tipul decltype

- este un operator folosit pentru a furniza informații legate de tipul de dată al unei funcții;

- utilizarea acestui operator este folosită în *programarea generică* pentru a detemina tipul de dată returnat de un anumit apel al unei funcții template;

- acest tip de dată a fost introdus odată cu noua versiune *C++1Y*;

```
decltype(st.retnumar_matricol());  //int

decltype(st.return_media());      //double
```

Compilarea în timp a expresiilor constante

- permite ca anumite calcule / funcții să fie realizate în timpul compilării (acestea au loc in timpul compilării nu în timpul execuției);

- avantaj de performanță (dacă o anumită expresie se execută în timpul compilării, aceasta nu se va mai executa la fiecare rulare a programului);

Exemplu:
```
constexpr int Inmultire(int x, int y){
        return x*y;
    }
const int val = Inmultire(10,10); // apel functie
```

Dezavantaj: Returneaza doar o singura valoare.

Referințe R-value

- din punct de vedere tehnic, referința *Rvalue*, este o valoare de tip anonim care există doar în momentul evaluării unei expresii;

Exemplu: ***x+(y*z);*** //expresia C++ produce o valoare temporară;

- rezultatul *(y*z)* este stocat într-o variabilă temporară *Rvalue* după care este adunat la variabila *x*;

- conceptual această valoare dispare până la sfârșitul evaluării liniei de cod;

- referințele *Rvalue* pot apărea oriunde, chiar dacă nu se folosesc direct în cod;

Pointeri SMART

- aceștia dețin memoria și o eliberează atunci când ies din sfera de aplicare *(garbage collector);*

- în acest fel programatorul este liber să folosească cum vrea memoria deoarece este alocată în mod dinamic;

Tabele Hash

- Tabela Hash este mai putin eficienta decat un arbore binar datorita prezentei unui numar mare de coliziuni, insa au o performanta mai buna in aplicatii reale.
- Aceasta noua biblioteca este recunoscuta prin prefixul *unordered*.

Type of hash table	Associated values	Equivalent keys
std::unordered_set	No	No
std::unordered_multiset	No	Yes
std::unordered_map	Yes	No
std::unordered_multimap	Yes	Yes

```
template <class T>
void show(const T& x)
{
    for (const auto& i : x)
        cout << i.first << "; " << i.second << endl;
}

template <template <class...> class T>
void test()
{
    T<string, int> x;
    x.insert(make_pair("one", 1));
    x.insert(make_pair("two", 2));
    x.insert(make_pair("three", 3));
    x.insert(make_pair("four", 4));

    show(x);
}
```

```
int main()
{
  test<map>();
  cout << "********************" << endl;
  test<unordered_map>();
}
```

Îmbunătățiri ale funcționalităților

- **Ce sunt thread-urile?**

  Proces - > fire de executie (thread)

- **Multi-threading**

  Fire logice separate.

  Date comune, fără copiere sau mutare.

  Comunicare între thread-uri.

  Dezavantaj: complicat.

- **Multi-threading avantaje:** arhitecturile multi-core sunt programate mai eficient, in paralel.

Îmbunătățiri ale funcționalităților

Noutăți:

-sintaxa de inițializare a thread-urilor{ };

-mai mulți parametrii constructorului *thread*;

-invocarea membrilor funcției într-un *thread*;

-invocarea unei referințe la un membru al funcției într-un *thread*;

-beneficii aduse locking-ului și unlocking-ului manual prin comanda *lock-guard();*

Îmbunătățiri ale funcționalităților

Alte noutăți:

- blocarea mai multor thread-uri fără utilizarea deadlock-ului;
- alocarea memoriei de către operații din biblioteca *Atomic*;
- utilizarea operatorului *sizeof* pe tipuri de date și obiecte;
- un nou tip de date: long long int;

Modificări aduse bibliotecilor standard

- Pentru a completa bibliotecile precedente limbajului s-a implementat un domeniu nou de aplicare (spatiu de nume), pe baza modelului de alocare care a fost inclus în C++11/14/17/20

- Bibliotecile standard existente în limbajul C++0x/1y/2z oferă o serie de caracteristici noi, fiind actualizate unele componente existente până în prezent

Concluzii modificări aduse bibliotecilor standard

- Aceste componente ale bibliotecilor standard includ:
  - Referințele Rvalue
  - Operatorii de conversie explicită
  - Template-uri Variadic
  - Compilarea în timp a expresiilor constante
  - Erasure functions (Funcțiile șterse)
  - decltype

Concluzii modificări aduse bibliotecilor standard

- Modificările aduse bibliotecilor standard:
  - Actualizări pentru componentele bibliotecii standard
  - Facilitățile Threading
  - Expresiile regulare
  - Tuple types
  - Tabele Hash
  - Pointerii Smart
  - Facilitatea extensibilă de numere aleatoare

Concluzii generale

- C++0x/1y/2z reprezintă versiunea noua a limbajului de programare C++
- Aduce imbunătățiri ale performanțelor timpilor de rulare, performanțe de utlizare, performanțe build-time precum și noi funcționalități.
- De asemenea aduce îmbunătățiri ale bibliotecilor.

Referinte

- http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-8-futures-and-promises.html
- http://www.artima.com/cppsource/cpp0x.html

[1] “C++ 1Y Explication”

http://en.wikipedia.org/wiki/C%2B%2B11

[2] “C++11 Tutorial: Rvalue and move constructor”

http://blog.smartbear.com/c-plus-plus/c11-tutorial-introducing-the-move-constructor-and-the-move-assignment-operator/

[3] “What is C++ 0X and 1Y”

http://www.cprogramming.com/c++11/what-is-c++0x.html

[4] “Explicating the new C standard 0X and 1Y”

http://www.codeproject.com/Articles/71540/Explicating-the-new-C-standard-C-0x-and-its-implem

[5] “C++11 Core Language Features”

http://msdn.microsoft.com/en-us/library/hh567368.aspx

[6] “C++11, C++ Standard Library”, Nicolai M. Josuttis, editura Addison_Wesley