Projektowanie złożonych systemów telekomunikacyjnych

Lecture 4: Standard Library

Łukasz Marchewka

21-03-2019

Agenda:

- Memory management
    - Leaks
    - RAII
    - Smart pointers: std::unique_ptr, std::shared_ptr, std::weak_ptr

- STL – Standard Template Library
    - Containers:
        - Sequence containers: **std::array**, **std::vector**, **std::list** (std::forward_list), **std::deque**
        - Associative containers: **std::set**, **std::map** (std::multimap)
        - Containers adapters: **std::queue**, **std::priority_queue**, **std::stack**
        - **std::tuple**

- Iterators

C++ History

- Started in 1979 as '**C with Classes**' by Bjarne Stroustrup
- 1983 – renamed as the **C++** (C incremented)
- 1994 – first appearance of the **STL** (A. Stepanov), HP implementation
- ISO standards (ISO/IEC 14882)
  - 1998 – first ISO standard (**C++98**)
  - 2003 – minor corrections (**C++03**)
  - 2007 – Technical Report 1, additions to std. library (**C++TR1**)
  - 2011 – major revision of language (**C++11**, former *C++0x*)
  - 2014 – minor improvements (**C++14**)
  - **2017 – several improvements** (**C++17**)
  - 2020 – major revision (**C++20**)
  - 2023– next standardization

New features since C++03

- ***performance:*** r-value references, move semantic, constant expressions, data alignment, unions
- ***less errors:*** nullptr, override, final, default, deleted, static asserts, explicit conversion, enumerations, loops, exceptions, integral types
- ***automatic memory management:*** smart pointers, STL allocators
- ***concurrency support:*** memory model, kinds of variables, thread creation and synchronization, tasks
- ***metaprogramming:*** auto, decltype, variadic templates, right angle bracket bug, template aliases, type traits
- ***functional programming:*** callable objects, lambda expressions, functional types, binding
- ***strings and characters:*** unicode, new character types, new string literals
- ***data initialization:*** initializer lists, member initialization, constructors, user defined literals
- ***new STL stuff:*** tuples, containers, regular expressions, random number generation, rational numbers, timers

Memory Management

Memory Leaks

- Memory leak occurs when the memory is allocated by using ”new” k and is not deallocated by using "delete" or delete[].
- A program with memory leaks is increasing memory usage of a system and all systems have limited amount of memory.
- Even if a program is written "correctly", a memory leak can occur caused by an exception

```
#include <iostream>

void memoryLeak()
{
    int* ptr = new int(5);
    return;
}


int main()
{
    memoryLeak();
    return 0;
}
```


Memory leaks

- Use smart pointers as often as possible, instead of managing memory manually (raw pointers)
- Use std::string instead of char *. The std::string class handles all memory management internally, is fast and well-optimized.
- Never use a raw pointer (exception: an interface to an older library)
- Keep as few new/delete calls at the program level as possible – ideally NONE.
- Prefer „new”/„delete” over malloc/free
- Apply RAII pattern

RAII - Resource Acquisition is Initialisation

- The resource is aquired in the constructor
- The resource is released in the destructor (e.g. closing a file, deallocating a memory)
- Instances of the class are stack allocated
- Follow the Rule of 5 in RAII
  - if you need to write nontrivial version of any of:
    1. Desctructor
    2. Copy constructor
    3. Move constructor
    4. Copy assign operator
    5. Move assign operator,
  - then write all of them!

- If an object requires dynamic memory it should allocate a memory in a constructor and release in a destructor -> it is a guarantee that a memory is deallocated when a variable leaves the current scope

- **C++ guarantees that the destructors of objects on the stack will be called, even if an exception is thrown**

RAII - Resource Acquisition is Initialisation

- Find a bug
- Apply RAII pattern

```
#include <iostream>

struct A
{
    int m_name{0};
    A(int p_name) : m_name(p_name) { std::cout << "A(" << m_name << ") constructed successfully\n"; }
    ~A() { std::cout << "A(" << m_name << ") destroyed\n"; }
};

struct B
{
    A* a1 = new A{5};
    B() { std::cout << "B constructed successfully\n"; }
    ~B() { std::cout << "B destroyed\n"; }
};

int main()
{
    B b{};
    return 0;
}
```


Smart pointers: `unique_ptr and shared_ptr, weak_ptr`

- The C++11 standard introduces new type of pointers for avoiding memory leaks. Pointers known from previous standards (with asterisk: i.e. int* ptr) of C++ are called raw pointers.

Unique pointers

- The `std::unique_ptr` is a kind of smart pointer which eliminates the risk of resource leaks

- Unique pointers have unique ownership of the internal objects
  - no more than one unique pointer can own the same object
  - destructors of unique pointers automatically destroy owned objects

- Unique pointers replace auto pointers from C++03
  - unique pointers support only move semantic
  - unique pointers properly support arrays and allow replacing the default `delete` and `delete[]` operators used to release owned objects
  - (auto pointers support only copy semantic, but actually perform move (!))

The `std::auto_ptr` is deprecated now, do not use it! Actually it is removed from newer standards

Unique pointers

- The example use cases of unique pointers:
  - ```
    std::unique_ptr<int> ptr1(std::make_unique(13));
    std::unique_ptr<int[]> ptrToArray(std::make_unique<int[]>(5));
    std::unique_ptr<int> ptr2 = ptr1; // compile error,
    std::unique_ptr<int> ptr3 = std::move(ptr1); // OK
    ptr1.reset(); // OK, but no effect
    ptr3.reset(); // Forces to destroy object
    ```
  - ```
    std::unique_ptr<float> func() {…}     // OK, fine with
    std::unique_ptr<float> rslt = func(); // move semantic
    ```
- The function std::make_unique is available since C++14
  - ```
    int* ptr1 = new int(13);
    std::unique_ptr<int> ptr2(ptr1); // compiles OK,
    std::unique_ptr<int> ptr3(ptr1); // but serious error
    ```

function `std::make_unique` assures that this kind of errors is impossible
    ```

Shared pointers

- Shared ownership for dynamically created object
- Keeps internal number of references to the object -> copy of shared_ptr increments that number
- Number of shared_ptr controlling one object is not changed by move operation. Just the pointer is set to nullptr, however it is faster then standard copy
- Shared pointers provide automatic memory management using reference counting
  - attaching a shared pointer increments reference counter
  - destroying a shared pointer decrements the counter, freeing the object if and only if the counter drops to zero
- Performance penalty: heap fragmentation and two actual memory dereferences performed by dereference operators
- Not foolproof – objects in circular references would never be destroyed
  - Use std::weak_ptr to break circular references

Shared pointers

- use_count() -> displays how many the shared pointers point the resource

```
#include <iostream>
#include <memory>

struct A
{
    int m_name{0};
    A(int p_name) : m_name(p_name) { std::cout << "A(" << m_name << ") constructed successfully\n"; }
    ~A() { std::cout << "A(" << m_name << ") destroyed\n"; }
    void getName() { std::cout << __FUNCTION__ << ": A(" << m_name << ")\n"; }
};

void foo(std::shared_ptr<A> p_ptr)
{
    p_ptr->getName();
    std::cout << "Function End\n";
}

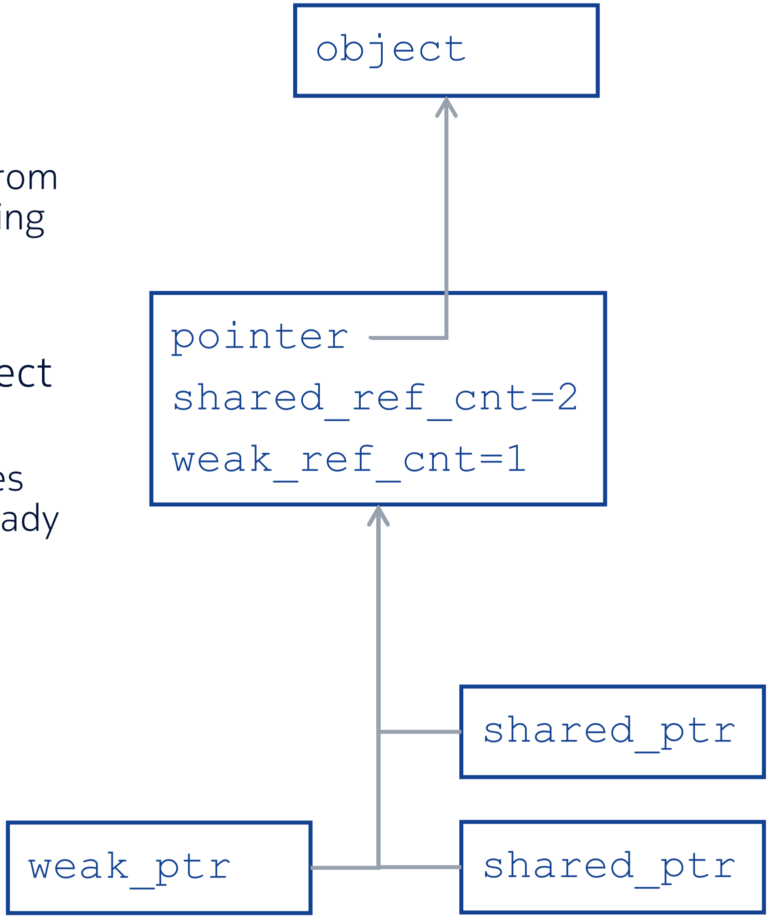
int main()
{
    std::shared_ptr<A> ptr1(std::make_shared<A>(13));
    foo(ptr1);

    std::cout << "PROGRAM END\n";
}
```


Shared and weak pointers

- Weak pointers can be used to break circular references
  - any shared pointer pointing to an object prevents from deleting it weak pointers do not prevent from deleting object

- Weak pointers can be queried if the pointed object still exists
  - careful usage of shared and weak pointers eliminates the possibilities of double-delete and access to already deleted objects



Weak pointers

- Weak pointers have two functions for querying object existence
  - `shared_ptr<T> lock() const;`
    returns either a shared pointer to the object if it still exists or null pointer otherwise
  - `bool expired() const;`
    verifies if weak pointer still points to an object
- The example use cases of shared and weak pointers:

```
std::shared_ptr<int> ptr1(std::make_shared(13));
std::shared_ptr<int> ptr2(ptr1); // refcnt=2
std::weak_ptr<int> wptr(ptr1);   // still refcnt=2
ptr1.reset();  // refcnt=1, no delete
{ wptr.lock(); } // returns a non-null shared pointer
ptr2.reset();  // refcnt=0, deletes object
{ wptr.lock(); } // now returns a null shared pointer
```


STL – Standard Template Library

- The Standard Template Library defines template-based, reusable components that implements common data structures and algorithms

- STL extensively uses generic programming based on templates

- Divided into three components:
    - Containers: data structures that store objects of any type
    - Iterators: used to manipulate container elements
    - Algorithms: searching, sorting and many others

Containers

- Three types of containers
  - Sequence containers:
    - linear data structures such as vectors and linked lists
  - Associative containers:
    - non-linear containers such as hash tables
  - Container adapters:
    - constrained sequence containers such as stacks and queues

- Sequence and associative containers are also called first-class containers

Sequence Containers

- STL provides sequence containers as follows:
  - array
  - deque (double-ended queue): based on arrays
  - list: based on linked lists
  - vector: based on arrays

Array - `template<class T, size_t N> class array;`

- Array – fixed size container similar to vector
  - zero overhead over classic C array `[]`
  - just member functions that make C array compatible with STL containers and tuples
  - Element can be reached by operator []

```
#include <iostream>
#include <array>

int main()
{
    std::array<int, 3> arr1 = {1, 2, 3};
    std::array arr2 = {1, 2, 3}; // introduced in C++17

    arr1[0] = arr2[2] = 8;

    std::cout << "Count: " << arr1.size() << ", " << arr2.size() << std::endl;

    for(int it : arr1)
    {
        std::cout << it << " " ;
    }

    std::cout << std::endl;
    auto [v1, v2, v3] = arr2; // introduced in C++17
    std::cout << v1 << " " << v2 << " " << v3;
}
```

Vector - std::vector<T, Alloc = std::allocator<T>>

- The implementation of a vector is based on arrays, it encapsulates dynamic size array
- Vectors allow direct access to any element via indexes O(1)
- Insertion at the end is normally efficient, the vector simply grows
- Insertion and deletion in the middle is expensive, an entire portion of the vector needs to be moved
- Uses more memory to handle future growth
- When the vector capacity is reached then
  - A larger vector is allocated
  - The elements of the previous vector are copied and
  - The old vector is deallocated

- Some functions: size, capacity, insert, push_back, erase
  - data() – returns pointer to array containing all vector elements, similar to &vec.first(), but safe on empty vectors
  - shrink_to_fit() – reduces pre-allocated memory to be not much larger than necessary to contain all elements, it is a hint only!

Vector - std::vector<T, Alloc = std::allocator<T>>

- First choice for data structure in C++

```
#include <iostream>
#include <vector>

int main()
{
std::vector<int> vec;
std::vector<int> vec1 {1, 2, 3};
std::cout << "Count: " << vec.size() << std::endl;
std::cout << "Capacity: " << vec.capacity() << std::endl;

vec.push_back(3);
vec.push_back(1);
vec.push_back(2);

std::cout << "Count: " << vec.size() << std::endl;
std::cout << "Capacity: " << vec.capacity() << std::endl;

for (int it : vec)
{
    std::cout << it << " " ;
}
std::cout << std::endl;

for (int it : vec1)
{
    std::cout << it << " " ;
}

}
```


List - std::list <T, Alloc = std::allocator<T>>

- List is implemented using a doubly-linked list
- Insertions and deletions are efficient (constant time) at any point of the list
    - But you have to have access to an element in the middle of the list first

- Bidirectional iterators are used to traverse the container in both directions
- Some functions: push_front, pop_front, remove, unique, merge, reverse and sort

List - std::list <T, Alloc = std::allocator<T>>

```
#include <iostream>
#include <list>
#include <algorithm>

int main()
{
    std::list<int> l_list;
    l_list.push_back(3);
    l_list.push_back(2);
    l_list.push_back(1);
    std::cout << "Count: " << l_list.size() <<  std::endl;

    for(int it : l_list)
    {
        std::cout << it << " ";
    }


    auto it = std::find(l_list.begin(), l_list.end(), 3);
    if (it != l_list.end())
    {
        l_list.insert(it, 4);
    }

    std::cout << std::endl;
    for(int it : l_list)
    {
        std::cout << it << " ";
    }
}
```


Forward lists

Forward lists – singly linked lists

- zero space or time overhead relative to a hand-written C-style singly linked list
- support only forward iterators, no bidirectional and reverse ones
- no `size()` member function
- pointer to the last element is not stored – no `back()`, `push_back()` and `pop_back()` member functions
- `insert_after()`, `erase_after()` and `splice_after()` instead of respective methods of `std::list`
- additional iterator position – `before_begin()`

Deque - std::deque<T, Alloc = std::allocator<T>>

- Deque stands for double-ended queue
- Deque combines the benefits of vector and list
- It provides indexed access using indexes (which is not possible using lists)
- It also provides efficient insertion and deletion in the front (which is not efficient using vectors) and the end
- Elements are not stored contigously
- Additional storage is allocated using blocks of memory, that are maintaned as an array of pointers to those blocks
- Same functions as for vector

Deque - std::deque<T, Alloc = std::allocator<T>>

```
#include <iostream>
#include <deque>

int main()
{
    std::deque<int> deq{1,2,3};
    deq.push_back(4);
    deq.push_front(0);
    std::cout << "Count: " << deq.size() << std::endl;

    for(int it : deq)
    {
        std::cout << it << " ";
    }
}
```


Associative Containers

- Associative containers use keys to store and retrieve elements
- There are: **std::set, std::multiset, std::map, std::multimap**
    - all associative containers maintain keys in sorted order
    - all associative containers support bidirectional iterators
    - set does not allow duplicate keys
    - multiset and multimap allow duplicate keys
    - multimap and map allow keys and values to be mapped

Set - std::set<Key, Comp=std::less<Key>, Alloc=std::allocator<Key>>

- Contains sorted unique objects (does not allow duplicates)
- Set is implemented using a red-black binary search tree for fast storage and retrieval of keys O(logN)
- The ordering of the keys is determined by the STL comparator function object less<T>
- Keys sorted with less<T> must support comparison using the < operator

```cpp
#include <iostream>
#include <set>

int main()
{
    std::set<int> l_set;
    l_set.insert(3);
    l_set.insert(122);
    l_set.insert(2);
    std::cout << "Count: " << l_set.size() << std::endl;

    for(int it : l_set)
    {
        std::cout << it << " " ;
    }

    return 0;
}
```

Map - std::map<Key, T, Comp=std::less<T>, Alloc=std::allocator<std::pair<const Key, T>>>

- Implemented using red-black binary search trees
- Allows storage and retrieval of unique key/value pairs
- Does not allow duplicates of keys
- The class map overloads the [ ] operator to access values in a flexible way

```
#include <iostream>
#include <map>

int main()
{
    std::map<int, int> l_map;
    l_map[1] = 5;
    l_map[2] = 7;
    l_map[8] = 4;
    l_map[1] = 2;
    std::cout << "Count: " << l_map.size() <<   std::endl;
    for(const auto& it : l_map)
    {
        std::cout << it.first << "->" << it.second << "\n";
    }
    std::cout << std::endl;

    for(const auto& [key, value] : l_map) // C++17
    {
        std::cout << key<< "->" << value << "\n";
    }

    return 0;
}
```

Container Adapters

- STL supports three container adapters
  - **std::stack, std:: queue** and **std:: priority_queue**
- Implemented using the containers seen before
- Do not provide new data structure
- Container adapters do not support iterators
- The functions push and pop are common to all container adapters

Stack - std::stack<T, Container = std::deque<T>>

- Last-in-first-out data structure
- Implemented with vector, list, and deque (by default)
- Example of creating stacks
  - A stack of int using a vector: stack < int, vector < int > > s1;
  - A stack of int using a list: stack < int, list < int > > s2;
  - A stack of int using a deque: stack < int > s3;

```cpp
#include <iostream>
#include <stack>

int main()
{
    std::stack<int> l_stack;
    l_stack.push(1);
    l_stack.push(2);
    l_stack.push(3);
    std::cout << "Count: " << l_stack.size() << std::endl;
    while(!l_stack.empty())
    {
        std::cout << l_stack.top() << " ";
        l_stack.pop();
    }

    return 0;
}
```


Queue - std::queue<T, Container = std::deque<T>>

- First-in-first-out data structure
- Implemented with list and deque (by default)
- Example:
  - A queue of int using a list: queue <int, list<int>> q1;
  - A queue of int using a deque: queue <int> q2;

```cpp
#include <iostream>
#include <queue>

int main()
{
    std::queue<int> l_queue;
    l_queue.push(1);
    l_queue.push(2);
    l_queue.push(3);
    std::cout << "Count: " << l_queue.size() << std::endl;
    while(!l_queue.empty())
    {
        std::cout << l_queue.front() << " ";
        l_queue.pop();
    }

    return 0;
}
```


Priority Queue - std::priority_queue<T, Container = std::vector<T>>

- Insertions are done in a sorted order
- Deletions from front similar to a queue
- They are implemented with vector (by default) or deque
- The elements with the highest priority are removed first
    - less<T> is used by default for comparing elements (largest at front)

```cpp
#include <iostream>
#include <queue>

int main()
{
    std::priority_queue<int> pqueue;
    pqueue.push(122);
    pqueue.push(2);
    pqueue.push(33);
    std::cout << "Count: " << pqueue.size() << std::endl;
    while(!pqueue.empty())
    {
        std::cout << pqueue.top() << " ";
        pqueue.pop();
    }

    return 0;
}
```

Tuple - template <class... Types> class tuple;

- Structures with any number of elements of arbitrary types
- Tuples provide comparison operators, similarly as pairs
- Implemented with variadic templates
- Supports logical operators. The logical conditions are performed for every element of tuple

```
#include <iostream>
#include <tuple>

int main()
{
    auto l_tuple1 = std::make_tuple(1, 2.3, "Lukasz");
    std::tuple<int, double, std::string> l_tuple2(1, 3.14, "PI");

    std::size_t s = tuple_size<decltype(l_tuple1)>::value;
    std::cout << "Size: " << s << std::endl;

    std::cout << "Tuple1: " <<
              std::get<0>(l_tuple1) << ", " <<
              std::get<1>(l_tuple1) << ", " <<
              std::get<2>(l_tuple1) << std::endl;

    std::tuple<int, float, char*> l_tuple3(1, 2.0f, nullptr);
    l_tuple3.get<2>() = new char[13];

    // C++17
    std::tuple l_tuple4{ 2, 6.28, "2*PI" };
    auto [index, value, name] = l_tuple4;
    std::cout << "Tuple4: " << index << ", " << value << ", " << name << std::endl;
}
```

Iterators

- Iterators are pointers to elements of first-class containers
  - Type const_iterator defines an iterator to a container element that cannot be modified
  - Type iterator defines an iterator to a container element that can be modified
  - `cbegin()`, `cend()`, `crbegin()`, `crend()` – return always const iterators, even on non-const objects

- All first-class containers provide the members functions begin() and end()
  - return iterators pointing to the first and one-past-the-last element of the container

- **it++** (or **++it**) points to the next element
- ***it** refers to the value of the element

NOKIA

Copyright and confidentiality

Revision history and metadata

Please delete this slide if document is uncontrolled

Document ID: DXXXXXXXXX
Document Location:
Organization:

Version	Description of changes	Date	Author	Owner	Status	Reviewed by	Reviewed date	Approver	Approval date
		DD-MM-YYYY					DD-MM-YYYY		DD-MM-YYYY

R-value references

- Avoid unnecessary copying of temporary values
- Efficient objects like `std::vector<std::string>`
- Move constructors and move assignment operators

  defined similarly as typical constructors and assignment operators, but using `&&` (r-value reference)

```
class C {
  C(const C& rhv);
  C(C&& rhv);
  C& operator=(const C& rhv);
  C& operator=(C&& rhv);
  …
};
```

  typically both move and copy semantic should be defined

R-value reference can be used in normal functions as well

  they can be overloaded, `T&` and `T&&` are distinct types

Move semantic

- Move constructors and move assignment operators can *steal* data from its arguments
- Formally, after move the arguments may remain in unspecified state, but the state still has to be valid
- No further functions can be called on these objects
  - such functions would cause unspecified results
- However, destructors are always executed

Typical implementation of move constructor

```
C(C&& rhv)
{
  ptr_data = rhv.ptr_data; // data stealing
  rhv.ptr_data = nullptr;  // the state of rhv
}                          // must remain valid

~C() { delete[] ptr_data; }    // safe even after move
```

Using move

Compiler decides when move semantic is used

```
C obj1;       // always copy semantic,
C obj2(obj1); // obj1 still can be used

C func(int x);// move semantic (if defined)
C o3(func(0));// result of func cannot be used anymore
```

The move semantic can be forced using `std::move` from `<utility>` header

```
C obj2(std::move(obj1)); // force move semantic
```

now it is programmer responsibility not to use obj1, since it is in unspecified state

however, the new value can be safely assigned to obj1

the destructor for obj1 will be called automatically

The `std::move` utility function is overload to ranges of objects (using iterators)

Constant expressions

- May be used to mark constants and functions
- Keyword `constexpr` guarantees that a constant initializer is evaluated to a compile-time constant, or causes a compile error if this is not possible
- Functions marked as `constexpr`
  - can have only a single `return` statement
  - can depend only on its arguments and globals marked as `constexpr`
  - can call only `constexpr` functions

In C++14 these requirements were relaxed, allowing:

- branchs and loops (without `goto`)
- automatic (non-static) variables
- calling non-const functions on objects with lifetime limited to the `constexpr` function

Constant expressions

Example:

```
constexpr int const1 = 1;    // OK
int const2 = 2;

constexpr int func(int x) {
  return x*x + const1; }     // OK
constexpr int func2(int x) {
  return x*x + const2; }     // compile error

template<int N, int M> class Matrix {…};
Matrix<func(1), func(2)> m; // OK

int i = 3;
int j = func(i);             // OK, j is non-constexpr
constexpr int k = func(i);   // compile error
```

- Numeric limits are redefined to be `constexpr`

```
Matrix<std::numeric_limits<short>::max(),
  std::numeric_limits<short>::max()> m; // OK in C++11
```


Less errors

Less error prone code by detecting more errors at compile time

Nullptr (1/2)

The numeric constant 0 is used as both an integer and a pointer

```
int i = 0;
void* ptr = 0;
```

Not really a problem in C

The following macros are used for better code readability

```
#define NULL (void*)0 // in C
#define NULL 0        // in C++
```

Does not work with function overloading

```
void f(int i);
void f(char* ptr);
f(NULL);    // probably an error, calls int version!
```

Does not provide type control

```
ptr = NULL; // OK
i = NULL;   // also OK
```


Nullptr (2/2)

But from C++11 onwards:

```
f(nullptr);     // calls char* version
ptr = nullptr; // OK
i = nullptr;    // compile error
```

The `nullptr` expression evaluates to a distinct type value that can be implicitly casted to any *pointer* (but not to an integer)

The type of `nullptr` is *NOT* `void*`

`std::nullptr_t`, defined as

```
typedef decltype(nullptr) nullptr_t;
```

Can be explicitly overloaded

```
f(void* ptr);
f(char* ptr);
f(nullptr); // compile error
            // ambiguity
```

```
f(std::nullptr_t ptr);
f(char* ptr);
f(nullptr); // OK
```

Override specifier

Virtual functions in C++03 are prone to errors

```
class Base {
  virtual void func(double x);
};
class Derived: public Base {
  void func(float x); // probably an error
};
Base* obj = new Derived();
obj->func(1); // compiles OK, but calls Base::func
```

But from C++11 onwards compiler can detect such errors

```
class Derived1: public Base {
  void func(float x) override;  // compile error
};
class Derived2: public Base {
  void func(double x) override; // OK
};
```


Final

Classes and member functions can be marked as `final`

```
class Cf final {…};
```

```
class Base {
  virtual void func();
};
```

```
class Derived: public Base {
  void func() final;
};
```

If `final` is used, some constructions cause compile errors

```
class Derived1: public Cf {…};       // compile error
```

```
class Derived2: public Derived {…}; // OK
```

```
class Derived3: public Derived {
  void func(); // compile error
};
```

Default / Deleted

Constructors and assignment operators can now be explicitly specified as `default` or `deleted`

```
class C1 {
  C1(const C1& rhv) = default;
};
class C2 {
  C2(const C2& rhv) = deleted;
  C2& operator=(const C2& rhv) = deleted;
};
```

Objects of type `C1` are explicitly specified to be copied with default copy constructor

Objects of type `C2` explicitly cannot be copied

- no more need for undefined c'tors in private sections

Any function, as well as function template, can be marked as `deleted`

For each

The C++11 provides new alternative syntax for the `for` loop:

```
int data[4] = {1, 2, 3, 4};
for (int& x: data)
  std::cout << x << ” ”;
```

- the loop operates on the entire range of data
- counter handling and termination condition is managed automatically

Global `std::begin()` and `std::end()` have to be defined for any data range that is used in such a loop

- standard library defines these functions for arrays in `<iterator>` header
- similar template functions are also defined for conainers which define `begin()` and `end()` as member functions

Static asserts

Asserts which are tested during compile time

Useful for quick detection of errors, especially within templates and constant expressions (`constexpr`)

Example:

```
template<class T>
class Flags {
  static_assert(sizeof(T) >= sizeof(int),
    ”Provided type is to small”);
  T data;
};
```

```
Flags<char> f1; // compile error
```

```
Flags<long> f2; // OK
```

If an assertion fails, a `static_assert` cause a compile error, using given string in an error message

