C++11

Introduction

C++11 is a major revision of the C++ programming language standard.

It represents a substantial enhancement of the language’s capabilities compared to the C++98/C++03 standards.

Introduction

Among the changes introduced we can highlight:

- Move semantics, rvalue references and universal references.
- Brace initialization
- `auto` keyword
- `constexpr` keyword
- Lambda functions
- Smart pointers
- Delegating constructor
- Variadic templates

Move semantics

Move instead of copy.

Reuse instead of recreate.

Move semantics

The move semantics allows to move a variable content to another, without copying that.

It allows binding non-named values (*rvalues*) or expiring variables to reuse their content.

Move semantics

The move semantics works by handling references to values, instead of copying the value.

The *move to* variable gets a value and the *moved from* variable loses it.

A moved from variable will/must have a valid value, bit it is unknown.

Move semantics

```cpp
#include <iostream>
#include <string>
#include <utility>

int main()
{
    std::string foo{"foo"};
    // Moves the foo's value to bar.
    std::string bar = std::move(foo);

    // Foo may be empty now.
    std::cout << foo << " >> " << bar;

    return 0;
}
```

Minimal move semantics demo

Move semantics

WARNING !

One can not make assumptions about moved from variables.

Moved from variable must not be reused after the move unless it is reinitialized.

Move semantics

User classes can have their objects moved at the same way of basic types.

It sometimes requires implementing custom *move constructors* and override the move assignment operator.

Moving semantics

with non trivial classes

Creating a non trivial class that supports move semantics requires attention to constructors and assignment operations.

Implicit conversions are also source of undesired behaviors.

Move semantics

with non trivial classes

A class is trivial if:

It is composed only of trivial types or objects.

It has a trivial default constructor, trivial copy constructor, trivial assignment operator and trivial destructors.

It does not have virtual functions, constructors, destructors and operators.

Move semantics

with non trivial classes

A trivial class object occupies contiguous memory space and thus it is direct to use move semantics on it

Move semantics

with non trivial classes

```cpp
#include <iostream>
#include <cstring>
using std::ostream;

class String
{
public:
    String() = default;
    String(const char *value);
    ~String();
    String(const String &other);
    String &operator=(const String &other);
    String(String &&other) noexcept;
    String &operator=(String &other) noexcept;
    friend ostream &operator<<(ostream &os, const String &value);

private:
    char *str{};
};
```

Custom string header file.

Move semantics

with non trivial classes

```cpp
String::String(const char *value)
{
    auto length = strlen(value);
    str = new char[length + 1];
    std::strcpy(str, value);
}

String::String(String &&other) noexcept
{
    str = other.str;
    other.str = nullptr;
}

String &String::operator=(String &other) noexcept
{
    str = other.str;
    other.str = nullptr;
    return *this;
}
```

Custom string source file (partial).

Move semantics
with non trivial classes

WARNING !

If a class copy constructor, destructor or assignment operator are declared the compiler does not create a default move constructor and assignment operator.

This behavior occurs even when use the keyword default on the header.

Move semantics

with non trivial classes

```
// … Regular includes and std usings…
#include "string.h"
#include "utils.h"
int main()
{
    String first{"first"};
    String second{"second"};

    // Original variable values.
    cout << first << " | " << second << endl;

    String temp = move(first);
    first = move(second);
    second = move(temp);

    // Variables values after swap.
    cout << first << " | " << second << endl;
    return 0;
}
```

Example 1: Swap using Move semantics.

Move semantics

with non trivial classes

CONSTRUCTOR/ASSIGNMENT CALLS OUTPUT

Const char * constructor with value "first"
Const char * constructor with value "second"

Move constructor with value "first"
Move assignment with value "second"
Move assignment with value "first"

```cpp
// … Regular includes and std usings…
#include "string.h"
#include "utils.h"
int main()
{
    String first{"first"};
    String second{"second"};

    // Original variable values.
    cout << first << " | " << second << endl;

    String temp = move(first);
    first = move(second);
    second = move(temp);

    // Variables values after swap.
    cout << first << " | " << second << endl;
    return 0;
}
```

Example 1: Swap using Move semantics.

Move semantics

with non trivial classes

```
// … Regular includes and std usings…
#include "string.h"
#include "utils.h"

int main()
{
    String *stringPointer = new String{"first"};
    String destinationString = move(*stringPointer);
    // The pointer is not null here.
    cout << *stringPointer << " | " << destinationString << endl;

    String stringVariable{"second"};
    destinationString = move(stringVariable);
    // The variable value is unknown (it can remain the same).
    cout << stringVariable << " | " << destinationString << endl;
    return 0;
}
```

Example 2: Move semantics with custom string.

Move semantics
with non trivial classes

CONSTRUCTOR/ASSIGNMENT CALLS OUTPUT

Const char * constructor with value “first”
Move constructor with value "first"

Const char * constructor with value “second”
Move assignment with value “second”

```
// … Regular includes and std usings…
#include "string.h"
#include "utils.h"

int main()
{
    String *stringPointer = new String{"first"};
    String destinationString = move(*stringPointer);
    // The pointer is not null here.
    cout << *stringPointer << " | " << destinationString << endl;

    String stringVariable{"second"};
    destinationString = move(stringVariable);
    // The variable value is unknown (it can remain the same).
    cout << stringVariable << " | " << destinationString << endl;
    return 0;
}
```

Example 2: Move semantics with custom string.

Moving semantics

on initialization and function override

Initialize objects and references using move semantics is straightforward. Understanding what is happening might be not.

Move semantics

on initialization and function override

```
int main(){
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String constStringToReference{"Non const string to ref"};
    String &refToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    String &&oldStyleInitializedRef = "Old style initialization";
    String &&braceInitializedRef{"Brace initialization"};
    String &&moveInitializedRef = move(nonConstantString);
    String &&referenceMoveInitializedRef = move(refToNonConstString);
    String &&functionInitializedRef = toString(15);
    String &&userDefinedFunctionInitializedRef = getString();
    String &&charArrayMoveInitizedRef = move(charArray);
    ...

    return 0;
}
```

Example 3: Rvalues reference initialization (Part I).

Move semantics

on initialization and function override

```
// ... Regular includes and usings ...
#include "utils.h"
#include "string.h"

String toString(int value)
{
    String returnedValue{to_string(value).c_str()};
    return returnedValue;
}

String getString()
{
    String nonConstantString{"Returned string"};
    return nonConstantString;
}
```

Example 3: Rvalues reference initialization (Part II).

Move semantics

on initialization and function override

```
int main(){
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String constStringToReference{"Non const string to ref"};
    String &refToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    String &&oldStyleInitializedRef = "Old style initialization";
    String &&braceInitializedRef{"Brace initialization"};
    String &&moveInitializedRef = move(nonConstantString);
    String &&referenceMoveInitializedRef = move(refToNonConstString);
    String &&functionInitializedRef = toString(15);
    String &&userDefinedFunctionInitializedRef = getString();
    String &&charArrayMoveInitizedRef = move(charArray);
    ...

    return 0;
}
```

String literals are lvalues, but they create a dev::String rvalue.
Only const char* constructor is called

Example 3: Rvalues reference initialization (Part III).

Move semantics

on initialization and function override

```
int main(){
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String constStringToReference{"Non const string to ref"};
    String &refToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    String &&oldStyleInitializedRef = "Old style initialization";
    String &&braceInitializedRef{"Brace initialization"};
    String &&moveInitializedRef = move(nonConstantString) ;
    String &&referenceMoveInitializedRef = move(refToNonConstString);
    String &&functionInitializedRef = toString(15);
    String &&userDefinedFunctionInitializedRef = getString();
    String &&charArrayMoveInitizedRef = move(charArray);
    ...

    return 0;
}
```

Reference assignment.
Objects created on top do not change.
Constructor and operator= are not called.

Example 3: Rvalues reference initialization (Part III).

Move semantics

on initialization and function override

```
int main(){
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String constStringToReference{"Non const string to ref"};
    String &refToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    String &&oldStyleInitializedRef = "Old style initialization";
    String &&braceInitializedRef{"Brace initialization"};
    String &&moveInitializedRef = move(nonConstantString);
    String &&referenceMoveInitializedRef = move(refToNonConstString);
    String &&functionInitializedRef = toString(15);
    String &&userDefinedFunctionInitializedRef = getString();
    String &&charArrayMoveInitizedRef = charArray;
    ...

    return 0;
}
```

Rvalue references extends returned function local variable lifespan. Only constructors are called

Example 3: Rvalues reference initialization (Part III).

Move semantics

on initialization and function override

```
int main(){
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String constStringToReference{"Non const string to ref"};
    String &refToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    String &&oldStyleInitializedRef = "Old style initialization";
    String &&braceInitializedRef{"Brace initialization"};
    String &&moveInitializedRef = move(nonConstantString);
    String &&referenceMoveInitializedRef = move(refToNonConstString);
    String &&functionInitializedRef = toString(15);
    String &&userDefinedFunctionInitializedRef = getString();
    String &&charArrayMoveInitizedRef = charArray;
    ...

    return 0;
}
```

Rvalue references works as a regular lvalue reference (points to char array address).

Example 3: Rvalues reference initialization (Part III).

Move semantics

on initialization and function override

```
int main(){
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String constStringToReference{"Non const string to ref"};
    String &refToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    ...
    String &&moveInitializedRef = move(nonConstantString);
    String &&referenceMoveInitializedRef = move(refToNonConstString);
    ...

    return 0;
}
```

Removing the && on variable declarations above will call a move constructor (except on char array case).

Example 3: Rvalues reference initialization (Part III).

Move semantics

on initialization and function override

WARNING !

It is not allowed a function that returns a reference to a local variable.

There is not advantage in returning a local variable using move semantics (i.e. writing something like `return std::move(local);`).

Move semantics

on initialization and function override

```
// All functions print the string passed, but each one specifies if it is understood as a rvalue reference or a lvalue (const or non const) reference.
void printReference(String &&value)
{
    cout << "The string \"" << value << "\" is received as a rvalue. \n";
}

void printReference(const String &value)
{
    cout << "The string \"" << value << "\" is received as a const lvalue reference. \n";
}

void printReference(String &value)
{
    cout << "The string \"" << value << "\" is received as a non-const lvalue reference. \n";
}
```

Example 4: Function overrides (Part I).

Move semantics

on initialization and function override

```
String &String::operator+=(String &&right)
{
    char *newStr = new char(1 + strlen(str)
                                + strlen(right.str));
    strcpy(newStr, str);
    strcat(newStr, right.str);
    delete[] str;
    str = newStr;
    return *this;
}

String String::operator+(String &&right) const
{
    String result{str};
    result += right;
    return result;
}
```

Example 4: Function overrides (Part II).

Move semantics

on initialization and function override

```cpp
int main()
{
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String *stringPointer = new String{"Non const string ptr"};
    String constStringToReference{"Non const string to ref"};
    String &referenceToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    printReference("Literal string");
    printReference({"Literal string inside braces"});
    printReference(String{"Concat string "} + String{" init"});
    printReference(move(nonConstantString));
    printReference(move(referenceToNonConstString));
    printReference(move(*stringPointer));
    printReference(move(charArray));
    ...
    return 0;
}
```

Example 4: Function overrides (Part III).

Move semantics

on initialization and function override

```
int main()
{
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String *stringPointer = new String{"Non const string ptr"};
    String constStringToReference{"Non const string to ref"};
    String &referenceToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    printReference("Literal string");
    printReference({"Literal string inside braces"});
    printReference(String{"Concat string "} + String{" init"});
    printReference(move(nonConstantString));
    printReference(move(referenceToNonConstString));
    printReference(move(*stringPointer));
    printReference(move(charArray));
    ...
    return 0;
}
```

As before, a lvalue (literal string) generates a rvalue. Calls only the move constructor.

Example 4: Function overrides (Part III).

Move semantics

on initialization and function override

```
int main()
{
    String nonConstantString{"Non constant string"};
    const String constString{"Constant string"};
    String *stringPointer = new String{"Non const string ptr"};
    String constStringToReference{"Non const string to ref"};
    String &referenceToNonConstString = constStringToReference;
    char charArray[] = "This is a char array";

    printReference("Literal string");
    printReference({"Literal string inside braces"});
    printReference(String{"Concat string "} + String{" init"});
    printReference(move(nonConstantString));
    printReference(move(referenceToNonConstString));
    printReference(move(*stringPointer));
    printReference(move(charArray));
    ...
    return 0;
}
```

The operator + returns an expiring lvalue that is handled as a rvalue.

Example 4: Function overrides (Part III).

CONSTRUCTOR/ASSIGNMENT CALLS OUTPUT

Const char * ctor with value "Concateneted string "
Const char * ctor with value " init"
Move concatenation
Const char * ctor with value "Concateneted string init"
printReference(String &&value)

Move semantics

on initialization and function override

```
int main()
{
     ...
    cout << "\n\n Tracking lvalue strings\n\n";
    printReference(nonConstantString);
    printReference(constString);
    printReference(referenceToNonConstString);

    return 0;
}
```

Example 4: Function overrides (Part III).

Move semantics

on initialization and function override

WARNING !

It is counterintuitive creating functions with a const rvalue reference as parameter, because move semantics changes the argument value.

Moving semantics performance

Not always move semantics is useful.

Avoid a simple copy with a simple copy is “changing from two to one pair”

Move semantics

performance

```
// Bubble sort using only move
template <class T>
void sortByMove(vector<T> &vectorToBeSorted)
{
    if (vectorToBeSorted.size() == 0)
        return;

    T &&temporaryRvalue{T{}};
    for (int i = 0; i < vectorToBeSorted.size() - 1; i++)
        for (int j = i + 1; j < vectorToBeSorted.size(); j++)
        {
            if (vectorToBeSorted[j] < vectorToBeSorted[i])
            {
                temporaryRvalue = move(vectorToBeSorted[j]);
                vectorToBeSorted[j] = move(vectorToBeSorted[i]);
                vectorToBeSorted[i] = move(temporaryRvalue);
            }
        }
}
```

Example 5: Performance on sorting (Part I)

Move semantics

performance

```cpp
// Bubble sort using only copy
template <class T>
void sortByCopy(vector<T> &vectorToBeSorted)
{
    if (vectorToBeSorted.size() == 0)
        return;

    T temporaryLvalue;
    for (int i = 0; i < vectorToBeSorted.size() - 1; i++)
        for (int j = i + 1; j < vectorToBeSorted.size(); j++)
        {
            if (vectorToBeSorted[j] < vectorToBeSorted[i])
            {
                temporaryLvalue = vectorToBeSorted[j];
                vectorToBeSorted[j] = vectorToBeSorted[i];
                vectorToBeSorted[i] = temporaryLvalue;
            }
        }
}
```

Example 5: Performance on sorting (Part II)

Move semantics

performance

```
// Sort a vector<int>
   vector<int> initialVector{13, 10, 24, 2, 8, 9, 4, 7, 15, 1, 18, 6};
   vector<int> vectorIntSortedByCopy{initialVector};
   vector<int> vectorIntSortedByMove{initialVector};

  // Call sort and calculate elapsed time
   auto startSortByMove{steady_clock::now()};
   sortByMove(vectorIntSortedByMove);
   auto endSortByMove{steady_clock::now()};
  //… Do the same with the sort by copy

   // Sort vector<string>
   vector<String> initialStringVector{"qwert", "wert", "erty", "rtyu",
"tyui", "yuio", "uiop", "asdf", "sdfg", "dfgh", "fghj", "gjkl"};
   vector<String> vectorStringSortedByCopy(initialStringVector);
   vector<String> vectorStringSortedByMove(initialStringVector);
  // Call sort and calculate elapsed time…
```

Example 5: Performance on sorting (Part III)

Move semantics
performance

WARNING !

Moving basic types can cost more than copying them.

Moving semantics and universal reference

Bind everything (not exactly) to a single function call and call the correct function from a single forward.

Keep variable full specification, without losing reference and constantness information.

Move semantics

and universal reference

WARNING !

Universal and forwarding references are two terms that mostly refers to the same thing.

Universal reference was used first (2016) to explain the concept.
Forwarding reference was chosen by C++ standards committee later (with C++17).

Move semantics

and universal reference

Universal references are not universal in usage sense (see discussion [here](#)).

Sometimes a universal reference is not forwarded.

```
vector<string> coll;
auto&& range = coll; // initialize a universal reference
auto pos = range.begin();
auto end = range.end();

for ( ; pos != end; ++pos ) {
      const auto& s = *pos;
}
```

Move semantics

and universal reference

```
void printReference(String &&value)
{
    cout << "\"" << value << "\" is a rvalue. \n";
}

void printReference(const String &value)
{
    cout << "\"" << value << "\" is a const lvalue reference. \n";
}

template <class T>
void fowardFunction(T &&param)
{
    printReference(std::forward<T>(param));
}
```

Universal reference used to forward value

Move semantics
and universal reference

```
void printReference(String &&value)
{
    cout << "\"" << value << "\" is a rvalue. \n";
}

voi
{
    cout << "\"" << value << "\" is a const lvalue reference. \n";
}

template <class T>
void fowardFunction(T &&param)
{
    printReference(std::forward<T>(param));
}
```

This is a rvalue reference, but it is not a universal reference.

Universal reference used to forward value

Move semantics

and universal reference

WARNING !

The val variable in function foo(T&& val); inside a template<class T>class Bar is not a universal reference.

```
template<typename T>
class Bar {
    T&& member;         // member is not a universal reference
    void foo(T&& val); // val is not a universal reference
};
```

Move semantics
and universal reference

Passing a rvalue reference as an argument to a function with universal reference make the function parameter type to be deducted as a rvalue reference.

In fowardFunction(std::move(x)); param is a rvalue reference.

```
template <class T>
void fowardFunction(T &&param)
{
    printReference(std::forward<T>(param));
}
```

Move semantics
and universal reference

WARNING !

Passing a rvalue reference as an argument to a function with lvalue reference parameter makes the function parameter to be deducted as a lvalue reference.

Brace initialization

Create a list, initialize a vector, set a pointer to null, using the same syntax.

Brace initialization

Brace initialization uniformize variables and constant initializations on C++11.

It connects values inside a pair of braces (i.e. { }) to constructors, assignment operators and performs member by member initializations on structs.

Brace initialization

```
#include <iostream>
#include <cassert>

int main()
{
    int integerValue{10};
    double doubleValue{5.05};
    string stringValue{"brace yourself"};
    bool booleanValue{true};
    int integerArray[]{5, 4, 3, 2, 1};
    int integerMatrix[][3]{{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};

    assert(integerValue == 10);
    assert(doubleValue == 5.05);
    assert(stringValue == "brace yourself");
    assert(booleanValue == true);

    return 0;
}
```

Example 1: Basic types brace initializations.

Brace initialization

```cpp
#include <iostream>
using std::cout;
using std::endl;
using std::string;

int main()
{
    int *nullPointer{};
    string emptyString{};
    bool undefinedBooleanValue{}; // expected to be false.
    bool undefinedIntValue{};     // expected to be zero.

    assert(nullPointer == nullptr);
    assert(emptyString.size() == 0);

    cout << "Bool value " << undefinedBooleanValue << endl;
    cout << "Integer value : " << undefinedIntValue << endl;
    return 0;
}
```

Example 1: Basic types brace initializations.

Brace initialization

```
//… includes, usings and printStdContainer generic function …

int main()
{
    string stringByCharList{'c', 'h', 'a', 'r'};
    vector<string> stringVector{"this", "is", "a", "string",
"list"};
    vector<int> vectorInitSpecialCase(5, 10);
    vector<int> twoValuesVectorInit{5, 10};

    cout << stringByCharList << endl;
    printStdContainer(stringVector);
    printStdContainer(vectorInitSpecialCase);
    printStdContainer(twoValuesVectorInit);

    return 0;
}
```

Example 3: List initialization with braces.

Brace initialization

```
//… includes and usings …
template <class T1, class T2>
void printPair(pair<T1, T2> inputPair){ //… only a cout … }

int main()
{
    pair<int, string> braceInitPair{5, "init"};
    tuple<string, int, bool> simpleTuple{"tuple", 2, true};
    map<string, int> simpleMap{{"Italy", 39}, {"Brazil", 55},
{"Austria", 43}};

    printPair<int, string>({8, "automatic"});
    printPair<int, string>(braceInitPair);

    cout << get<0>(simpleTuple) << " " << get<1>(simpleTuple)
         << " " << get<2>(simpleTuple) << endl;
    cout << simpleMap["Brazil"] << " " << simpleMap["Austria"];

    return 0;
}
```

Example 3: List initialization with braces.

Brace initialization

```
struct SimpleStruct
{ int integer; double real; string label; };

int main()
{
    char someCharArray[] = "Char array";
    SimpleStruct simpleStruct{5, 3.9, "test"};

    // Cast char[] to string is valid. Cast double to int is not.
    SimpleStruct implicitCastStruct{5, 2, someCharArray};
    // Builds without problems.
    SimpleStruct explicitCastStruct{int(5.2), 2, someCharArray};

    assert(implicitCastStruct.integer == explicitCastStruct.integer);
    assert(simpleStruct.label == "test");

    return 0;
}
```

Example 4: Struct initialization with brace initialization.

Auto **keyword**

Make your code short and let the compiler decide the right type to each data.

Make generic programming less painful by using a shortcut.

Auto **keyword**

```cpp
#include <iostream>

int main()
{
    auto integer = 5;
    auto real = 0.5;
    auto boolean = false;
    auto charPointerArray = "Test";

    return 0;
}
```

Minimal auto keyword usage demo

Auto **keyword**

WARNING !

If a variable is declared with the *auto* keyword it must be initialized.

If compiler type deduction is not the one desired, it is necessary to force it with a cast.

Auto **keyword**

```
#include <iostream>
#include <functional>
using std::cout;
using std::endl;
using std::function;

int main()
{
    auto helloWord = []()
    { cout << "Hello world from lambda expression!" << endl; };
    function<void()> helloWorldFromFunction = []()
    { cout << "Hello world from function!" << endl; };

    helloWord();
    helloWorldFromFunction();

    return 0;
}
```

Example 1: Use auto keyword to declare to hold lambda expressions.

Auto **keyword**

WARNING !

Even if an auto variable and a function receives a same lambda expression, function is less efficient if the lambda uses significant amount of memory.

Auto **keyword**

```cpp
// ... Includes and usings ...

int main()
{
    // It does not compile without a '=' operator.
    auto integerList = {2, 4, 3, 6, 9, 10};
    // This is initialized as a list initialized, not a pair.
    auto twoElementsIntegerList = {2, 4};
    // Initialize vector with list initializer.
    auto integerVector{static_cast<vector<int>>(integerList)};
    // Initialize with make_pair as list initializer is not
    // possible.
    auto mixedPair = make_pair("value", 4);
    // Without the cast the compiler deduces a const char*
    auto autoString = string("test string");

    return 0;
}
```

Example 2: Use auto keyword to declare to hold lambda expressions.

Auto **keyword**

WARNING !

It is impossible to initialize an auto variable with a set of integers containing more than one value through brace initialization.

A single element in a brace initialization will make the compiler deduce the element type, not a list.

Auto **keyword**

```
// ... Includes and usings ...
int main()
{
    // This is not a vector, but a std::initializer_list<int>.
    auto integerList = {2, 4, 3, 6, 9, 10};
    for (auto value : integerList)
        cout << "Current value: " << value << endl;

    map<string, int> autoMap{make_pair("1st", 1), make_pair("2nd", 4),
                             make_pair("3rd", 3)};

    // With C++17 is possible to use a structured bind,
    for (auto &item : autoMap)
        cout << "\nKey: " << item.first << ", value: " << item.second;
    for (auto it = begin(autoMap); it != end(autoMap); it++)
        cout << "\nKey: " << it->first << ", value: " << it->second;

    return 0;
}
```

Example 3: For loops using auto keyword.

Auto **keyword**

```cpp
// ... Includes and usings ...
int main()
{
    auto intList = {12, 4, 31, 16, 19, 1};
    vector<int> intVector(intList);

    // This function does not build with a std::initializer_list<int>
    // because the container cannot be changed.
    sort(begin(intVector), end(intVector), [](int &first, int &second)
         { return first < second; });
    for (auto value : intList)
        cout << "Current value: " << value << endl;

    // This is ok because std::find_if does not change the list.
    find_if(begin(intList), end(intList), [](int value)
            { return value == 19; });

    return 0;
}
```

Example 4: Auto deducing initializer_list<T>.

Auto **keyword**

```
// ... Includes and usings ...
int main()
{
    int nonConstInt{5};
    int &intReference{nonConstInt};
    const int constInt{6};
    volatile int volatileInt{7};
    vector<bool> booleanVector{false, false, true, false};

    auto missedReference = intReference;
    auto missedConstantness = constInt;
    auto missedVolatileness = volatileInt;
    // proxy types requires cast.
    auto isReferenceButNotBool = booleanVector[2];

    // ...
    return 0;
  }
```

Example 5: Qualifiers decays and proxy type issues (Part I)

Auto **keyword**

```
// ... Includes and usings ...
int main()
{
    // Requires #include <type_traits>
    cout << "missedReference is a reference? "
         << std::is_reference<decltype(missedReference)>::value
          << endl;
    cout << "missedVolatileness is volatile? "
         << std::is_volatile<decltype(missedVolatileness)>::value
         << endl;

    // The types are not the same.
    assert(strcmp(typeid(isReferenceButNotBool).name(),
                  typeid(false).name()) != 0);

    return 0;
  }
```

Example 5: Qualifiers decays and proxy type issues (Part II)

Auto **keyword**

WARNING !

The operator [] in a vector<bool> is a proxy to a bool. Internally it vector elements are single bits.

Proxy class objects will cause similar problems when used to initialize an auto variabile.

To keep qualifiers insert them on definition (i.e.).

Auto **keyword**

WARNING !

Both decltype(auto) and typedef(auto).name() returns the argument type. The first at compile time and cannot be used as variable. The second executes at run time and returns a char*.

typedef(auto).name() discard qualifiers and it is compiler dependent.

Auto **keyword**
advanced bonus example

```
template <class T, class U>
auto exoticMaxFunction(T first, U second) -> decltype(first)
{
    return first > second ? first : second;
}

int main()
{
    int intValue = 6;
    double doubleValue = 3.2;

    auto firstResult = exoticMaxFunction(intValue, doubleValue);
    auto secondResult = exoticMaxFunction('a', doubleValue);

    cout << typeid(firstResult).name() << " "
         << typeid(secondResult).name() << endl;

    return 0;
}
```

Example 6: Using decltype to define return type

Auto **keyword**

advanced bonus example

```
template <class T, class U>
auto exoticMaxFunction(T first, U second) -> decltype(first)
```

From C++14 it can be written as
decltype(auto) exoticMaxFunction(T first, U second);

```
}

int main()
{
    int intValue = 6;
    double doubleValue = 3.2;

    auto firstResult = exoticMaxFunction(intValue, doubleValue);
    auto secondResult = exoticMaxFunction('a', doubleValue);

    cout << typeid(firstResult).name() << " "
         << typeid(secondResult).name() << endl;

    return 0;
}
```

Example 6: Using decltype to define return type

Auto **keyword**

advanced bonus example

WARNING !

You can use conditionals inside decltype to define the return type, but this condition will use only information available on compile type.

Passing a derived class object to a decltype will make it return the base class.

Lambda expressions

Make your code short and let the compiler decide the right type to each data.

Make generic programming less painful by using a shortcut.

Lambda expressions

```
class HelloWorldPrint
{
public:
    void operator()()
    {
        cout << "Hello world from class function!" << endl;
    }
};
int main()
{
    auto helloWord = []()
    { cout << "Hello world from lambda expression!" << endl; };

    helloWord();
    HelloWorldPrint helloWorldObject;
    helloWorldObject();

    return 0;
}
```

Example 1: Lambda expression equivalences

Lambda expressions

```
class HelloWorldPrint
{
public:
    void operator()()
    {
        cout << "Hello world from class function!" << endl;
    }
};
int main()
{
    auto helloWord = []()
    { cout << "Hello world from lambda expression!" << endl; };

    helloWord();
    HelloWorldPrint helloWorldObject;
    helloWorldObject();

    return 0;
}
```

Captures on lambda expressions are equivalent to class data members.

Example 1: Lambda expression equivalences

Lambda expressions

```
int main()
{
    auto helloWord = []()
    { cout << "Hello world from lambda expression!" << endl; };

    function<void()> helloWorldFromFunction = []()
    { cout << "Hello world from function!" << endl; };

    helloWord();
    helloWorldFromFunction();

    ...

    return 0;
}
```

Example 2: Lambda expressions and std::function

Lambda expressions

```
int main()
{
    ...
    auto printCustomPhrase = [](const char *phrase)
    {
        cout << phrase << endl;
    };

    function<void(const char *)> printCustomPhraseFromFunction =
    [](const char *phrase)
    { cout << phrase << endl; };

    printCustomPhrase("This is a custom phrase!");
    printCustomPhraseFromFunction("A phrase from function!");

    return 0;
}
```

Example 2: Lambda expressions and std::function

Lambda expressions

REMEMBER !

Declaring a lambda expression using auto often requires less memory than using std::function

Use std::function to specify parameter type on functions which take a lambda expressions as argument.

Lambda expressions

```
void runLambda(function<void()> lambda)
{ lambda(); }

void runParamLambda(string param, function<void(string &)> lambda){
    lambda(param);
}

int main()
{
    auto helloWord = []()
    { cout << "Hello world from lambda expression!" << endl; };

    auto customPhrase = [](const string &customPhrase)
    { cout << customPhrase << endl; };

    runLambda(helloWord);
    runParamLambda("A custom phrase", customPhrase);
    return 0;
}
```

Example 3: Lambda expressions as functions arguments

Lambda expressions

```
int main()
{
    auto cost = 50;
    auto evaluateSum = [=](int quantity)
    {
        return cost * quantity;
    };

    auto evaluateRemains = [=](int quantity, int total = 2000)
    {
        return total - cost * quantity;
    };
    ...

   return 0;
}
```

Example 4: Lambda expression captures

Lambda expressions

```
int main()
{
    ...

    auto quantities = {10, 20, 3, 18, 9};
    for (auto quantity : quantities)
        cout << "Cost to buy " << quantity << " units: "
             << evaluateSum(quantity) << endl;

    for (auto quantity : quantities)
        cout << "Remaining after buying " << quantity << " units: "
             << evaluateRemains(quantity) << endl;

    return 0;
}
```

Example 4: Lambda expression captures

Lambda expressions

```
int main()
{
    auto cost = 50;
    auto evaluateSum = [&](int quantity)
    {
        return cost * quantity;
    };

    auto evaluateRemains = [&](int quantity, int total = 2000)
    {
        return total - cost * quantity;
    };

    ...
}
```

Capturing by reference can lead to undesirable results.

Example 4: Lambda expression captures

Lambda expressions

```
int main()
{
    ...

    auto quantities = {10, 20, 3, 18, 9};
    for (auto quantity : quantities)
        cout << "Cost to buy " << quantity << " units: "
             << evaluateSum(quantity) << endl;

    cost = 50;

    for (auto quantity : quantities)
        cout << "Remaining after buying " << quantity << " units: "
              << evaluateRemains(quantity) << endl;

    return 0;
}
```

Changes capture value and make output inconsistent

Example 4: Lambda expression captures

Lambda expressions

WARNING !

Lambda expression captures does not change variable scopes.

Only capture by value will hold the captured values during the entire lambda expression lifespan.

Lambda expressions

```
function<int(int)> getEvaluationFunction()
{
    auto cost = 50;
    // Using a capture by reference is an error. The value
    // must be captured by value.
    auto evaluateSum = [&](int quantity)
    {
        return cost * quantity;
    };

    return evaluateSum;
}

void runLambda(function<void(const string &)> lambda,
               const string &value)
{
    lambda(value);
}
```

Example 5: Lambda capture scope

Lambda expressions

```
int main()
{
    auto evaluationFunction = getEvaluationFunction();
    auto quantities = {10, 20, 3, 18, 9};
    for (auto quantity : quantities)
        cout << "Cost to buy " << quantity << " units: "
             << evaluationFunction(quantity) << endl;

    string customString = "This is a custom string.";
    auto printCustomString = [&](const string &complement)
    {
        cout << customString << " " << complement << endl;
    };

    runLambda(printCustomString, " And this is the complement");

    return 0;
}
```

Example 5: Lambda capture scope

Lambda expressions

```
int main()
{
    auto evaluationFunction = getEvaluationFunction();
    auto quantities = {10, 20, 3, 18, 9};
    for (auto quantity : quantities)
        cout << "Cost to buy " << quantity << " units: "
             << evaluationFunction(quantity) << endl;
```

The variable captured by reference does not exist anymore. Thus the output is unpredictable.

```
        cout << customString << " " << complement << endl;
    };

    runLambda(printCustomString, " And this is the complement");

    return 0;
}
```

Example 5: Lambda capture scope

Lambda functions

```
int main()
{
    auto evaluationFunction = getEvaluationFunction();
    auto quantities = {10, 20, 3, 18, 9};
    for (auto quantity : quantities)
        cout << "Cost to buy " << quantity << " units: "
             << evaluationFunction(quantity) << endl;

    string customString = "This is a custom string.";
    auto printCustomString = [&](const string &complement)
    {
        cout << customString << " " << complement << endl;
    };

    runLambda(printCustomString, " And this is the complement");

    return 0;
}
```

No problems with capture by reference here. The referenced variable still exists on lambda call.

Example 5: Lambda capture scope

Lambda expressions

WARNING !

Lambda expressions are functions created inside other functions. They can reduce code readability.

Use common functions when the lambda expression becomes large and/or does not need to be used as an argument.

Smart pointers

Use pointers in a safer way

Forget delete without leaking memory

Share heap allocated objects without asking to keep them alive

Smart pointers

Smart pointers are special pointers that deallocate the memory they point when they are destructed.

A block of memory pointed by a smart pointer is deallocated only if all smart pointers pointing to it are destructed.

Smart pointers

```cpp
#include <iostream>
#include <memory>
using std::cout;
using std::endl;

int main()
{
    int *holder{};
    {
        std::unique_ptr<int> smartPointer(new int(6));
        holder = smartPointer.get(); // Get handled memory address
        cout << "Value with smart pointer alive: " << *holder
            << endl;
    };

    cout << "Value with smart pointer dead: " << *holder << endl;
    return 0;
}
```

Example 1: Live cycle of memory pointed by smart pointer.

Smart pointers

WARNING !

A raw memory pointer will become dangled if the smart(s) pointer(s) that owned the memory it points goes out of scope.

Do not manually deallocate memory pointed by a smart pointer.

Smart pointers

```
int *getSmartPointerPointedElement()
{
    unique_ptr<int> smartPointer(new int(6));
    int *holder = smartPointer.get(); // Get handled memory address
    cout << "Memory with smart pointer alive: " << holder << endl;

    return holder;
}

int main()
{
    int *holder = getSmartPointerPointedElement();
    cout << "Memory with smart pointer dead: " << holder << endl;

    return 0;
}
```

Example 2: Live cycle of memory pointed by a smart pointer created in another function.

Smart pointers

```
int *getSmartPointerPointedElement()
{
    unique_ptr<int> smartPointer(new int(6));
    int *holder = smartPointer.get(); // Get handled memory address
    cout << "Memory with smart pointer alive: " << holder << endl;

    return holder;
}

int main()
{
    int *holder = getSmartPointerPointedElement();
    cout << "Memory with smart pointer dead: " << holder << endl;

    return 0;
}
```

“holder” points to “smartPointer” deallocated memory. It does not become a nullptr.

Example 2: Live cycle of memory pointed by a smart pointer created in another function.

Smart pointers

Smart pointers can be divided in unique (unique_ptr), shared (shared_ptr) and weak (weak_ptr).

They differ by their replication capability and memory deallocation responsibility.

Unique Smart pointers

Allocate a block of memory on heap without worrying about deallocating them.

Making sure that only one variable will handle that object.

Make heap memory to have an owner, not only a pointer to it.

Unique

Smart pointers

Unique smart pointers (from now only unique_ptr) owns the memory they point to. When they are destructed, the memory is freed.

A unique_ptr is a move only type.

A unique_ptr can handle both a pointer to a single object as well as to a array.

Unique

Smart pointers

```cpp
int main()
{
    std::unique_ptr<int> firstOwnerPointer(new int(6));

    // It does not compile.
    // std::unique_ptr<int> newOwnerPointer = firstOwnerPointer;

    // It compiles, but causes a double free error. Do not do it.
    // std::unique_ptr<int> newOwnerPointer(firstOwnerPointer.get());

    // The right way to transfer ownership.
    std::unique_ptr<int> newOwnerPointer = move(firstOwnerPointer);

    assert(firstOwnerPointer.get() == nullptr);

    return 0;
}
```

Example 3: Unique_ptr creation and ownership transfer.

Unique

Smart pointers

REMEMBER !

By transferring a memory block ownership a unique_ptr will become “non-initialized”.

It can be reinitialized later using the function *reset.*

A reset call from an initialized unique_ptr reset will delete the current memory block to receive the new one.

Unique

Smart pointers

```cpp
int main()
{
    const auto arraySize = 10;
    auto integerDeleter = [](int *pointer)
    {
        cout << "The pointer about to be deleted holds the value: "
             << *pointer << endl;
        delete pointer;
    };

    auto arrayDeleter = [arraySize](int *array)
    {
        cout << "The array about to be deleted holds the values : {";
        for (auto i{0}; i < arraySize; i++)
            cout << array[i] << (i < arraySize - 1 ? ", " : "}\n");

        delete[] array;
    };
    …
```

Example 4: Unique_ptr custom deleters (Part I)

Unique

Smart pointers

```
int main()
{
    ...
    unique_ptr<int, decltype(integerDeleter)>
                    pointerOwner(new int(6), integerDeleter);

    unique_ptr<int[], decltype(arrayDeleter)>
                    arrayOwner(
                    new int[]{1, 2, 3, 4, 5, 10, 9, 8, 7, 6},
                    arrayDeleter);

    return 0;
}
```

Example 4: Unique_ptr custom deleters (Part II)

Unique

Smart pointers

```
int main()
{
    ...
    unique_ptr<int, decltype(integerDeleter)>
                    pointerOwner(new int(6), integerDeleter);

    unique_ptr<int[], decltype(arrayDeleter)>
                    arrayOwner(
                    new int[]{1, 2, 3, 4, 5, 10, 9, 8, 7, 6},
                    arrayDeleter);

    return 0;
}
```

Deleter type must be explicit on unique_ptr template.

Example 4: Unique_ptr custom deleters (Part II)

Unique

Smart pointers

```cpp
int main()
{
    const auto arraySize = 10;
    auto integerDeleter = [](int *pointer)
    {
        cout << "The pointer about to be deleted holds the value: "

    };

    auto arrayDeleter = [arraySize](int *array)
    {
        cout << "The array about to be deleted holds the values : {";
        for (auto i{0}; i < arraySize; i++)
            cout << array[i] << (i < arraySize - 1 ? ", " : "}\n");

        delete[] array;
    };
    …
```

Deleter parameter must coincide with the type allocated on unique_ptr.

Example 4: Unique_ptr custom deleters (Part I)

Shared
Smart pointers

Allocate memory blocks and share it with another variables without worrying if it will be deleted.

Track how many pointers are pointing the same memory block.

Shared
Smart pointers

Shared smart pointers (from now only shared_ptr) are smart allows the same memory block to be pointed by more than one shared_ptr.

Each shared_ptr that points to the memory has a counter indicating the number of shared_ptr’s pointing to the memory block.

Shared

Smart pointers

```
int main()
{
    shared_ptr<int> init(new int(5), [](int *p)
                                {   cout << "Delete value: " << *p << endl;
                                    delete p; });
    cout << "Pointers sharing memory: " << init.use_count() << endl;

    shared_ptr<int> firstCoPointer(init);
    cout << "Pointers sharing memory: " << init.use_count() << endl;
    shared_ptr<int> secondCoPointer{init};
    cout << "Pointers sharing memory: " << init.use_count() << endl;

    {
    shared_ptr<int> thirdCoPointer = firstCoPointer;
    cout << "Pointers sharing memory: " << init.use_count() << endl;
    }
    cout << "Pointers sharing memory: " << init.use_count() << endl;
    return 0;
}
```

Example 5: Shared_ptr minimal usage.

Shared

Smart pointers

```cpp
int main()
{
    shared_ptr<int> init(new int(5), [](int *p)
                                              << *p << endl;
    cout << "Pointers sharing memory: " << init.use_count() << endl;

    shared_ptr<int> firstCoPointer(init);
    cout << "Pointers sharing memory: " << init.use_count() << endl;
    shared_ptr<int> secondCoPointer{init};
    cout << "Pointers sharing memory: " << init.use_count() << endl;

    {
    shared_ptr<int> thirdCoPointer = firstCoPointer;
    cout << "Pointers sharing memory: " << init.use_count() << endl;
    }
    cout << "Pointers sharing memory: " << init.use_count() << endl;
    return 0;
}
```

Deleter type is not inserted on template

Example 5: Shared_ptr minimal usage.

Shared

Smart pointers

```
int main()
{
    shared_ptr<int> init(new int(5), [](int *p)
                                  {    cout << "Delete value: " << *p << endl;
                                       delete p; });
    cout << "Pointers sharing memory: " << init.use_count() << endl;

    shared_ptr<int> firstCoPointer(init);
                                                                nt() << endl;

                                                                nt() << endl;


    {
    shared_ptr<int> thirdCoPointer = firstCoPointer;
    cout << "Pointers sharing memory: " << init.use_count() << endl;
    }
    cout << "Pointers sharing memory: " << init.use_count() << endl;
    return 0;
}
```

use_count() returns 4 inside the block, but after the block it will return 3.

Example 5: Shared_ptr minimal usage.

Shared

Smart pointers

A shared_ptr cannot hold a pointer to an array (only from C++17).

A shared_ptr can be initialized using make_shared<T> function. There C++ is not a equivalent function for unique_ptr’s on C++11 (on C++14 is introduced the function unique_ptr<T>)

Shared

Smart pointers

```
int main()
{
    auto integerSharedPointer = make_shared<int>(10);
    auto stringSharedPointer = make_shared<string>("test");
    auto pairSharedPointer =
         make_shared<pair<int, string>>(10, "test");
    auto vectorSharedPointer =
         make_shared<vector<int>>(vector<int>({10, 15, 20, 30}));

    assert(*integerSharedPointer == 10);
    assert(*stringSharedPointer == "test");
    assert((*pairSharedPointer).second == *stringSharedPointer);
    assert((*vectorSharedPointer).size() == 4);

    return 0;

}
```

Example 6: make_shared<T> used to initialize shared_ptr<T>

Shared

Smart pointers

```
int main()
{
    auto integerSharedPointer = make_shared<int>(10);
    auto stringSharedPointer = make_shared<string>("test");

        make_shared<vector<int>>(vector<int>({10, 15, 20, 30}));

    assert(*integerSharedPointer == 10);
    assert(*stringSharedPointer == "test");
    assert((*pairSharedPointer).second == *stringSharedPointer);
    assert((*vectorSharedPointer).size() == 4);

    return 0;

}
```

make_shared<T> function can bind to correct constructor and forward variadic arguments.

Example 6: make_shared<T> used to initialize shared_ptr<T>

Shared

Smart pointers

```
int main()
{
    auto integerSharedPointer = make_shared<int>(10);
    auto stringSharedPointer = make_shared<string>("test");
    auto pairSharedPointer =
         make_shared<pair<int, string>>(10, "test");
    auto vectorSharedPointer =
         make_shared<vector<int>>(vector<int>({10, 15, 20, 30}));
```

Some cases may require a explicit call to constructor, as in auto keyword usage.

```
    assert(*stringSharedPointer == "test");
    assert((*pairSharedPointer).second == *stringSharedPointer);
    assert((*vectorSharedPointer).size() == 4);

    return 0;

}
```

Example 6: make_shared<T> used to initialize shared_ptr<T>

Unique

Smart pointers

REMEMBER !

make_shared<T> function offers performance advantages and may be used as alternative to shared_ptr<T>(new T(args...))

By using make_share<T> function it is not possible to specify a custom deleter.

Unique

Smart pointers

WARNING !

A shared_ptr creation allocate two blocks of memory, one to the memory block to be shared and other to a control block that handles the sharing.

shared_ptr’s must be used only when unique_ptr’s are not enough to implement the desired functionality.

Weak
Smart pointers

Share smart pointers without being responsible for the memory they point.

Point a memory handled by a smart pointer without locking its lifetime.

Weak

Smart pointers

Weak smart pointers (from now only weak_ptr) are smart pointers that points a memory block handled by a shared_ptr, but without owning it.

A weak_ptr can not destruct the memory it points, it only keeps a reference to memory block.

Shared

Smart pointers

```
int main()
{
    auto sharedPointer = make_shared<int>(10);
    auto coOwnerPointer = sharedPointer;
    cout << "Sharing pointers: " << sharedPointer.use_count() << endl;

    {
        weak_ptr<int> weakPointer{sharedPointer};
        {
            weak_ptr<int> coWeakPointer{sharedPointer};
            weak_ptr<int> secondCoWeakPointers{sharedPointer};
            assert(sharedPointer.use_count() == 2);
        }
        assert(weakPointer.use_count() == 2);
    }

    assert(sharedPointer.use_count() == 2);
    return 0;
}
```

Example 7: Weak pointer initialization

Shared

Smart pointers

```
int main()
{
    auto sharedPointer = make_shared<int>(10);
    auto coOwnerPointer = sharedPointer;
    cout << "Sharing pointers: " << sharedPointer.use_count() << endl;

    {
        weak_ptr<int> weakPointer{sharedPointer};
        {
            weak_ptr<int> coWeakPointer{sharedPointer};
            weak_ptr<int> secondCoWeakPointers{sharedPointer};
            assert(sharedPointer.use_count() == 2);
        }
        assert(weakPointer.use_count() == 2);
    }

    asse
    return 0;
}
```

use_count() on both “sharedPointer” and “weakPointer” refers to the same value.

Example 7: Weak pointer initialization

Weak

Smart pointers

A weak_ptr points both to the pointed memory and control block of its shared_ptr.

When a weak_ptr begin to point to a shared_ptr, it increments the shared_ptr weak_count. If the weak_ptr dies, weak_count is decremented.

Shared_ptr control block is deleted only if its weak_count is zero

Weak

Smart pointers

```
void releaseSharedPointer(shared_ptr<int> &pointer)
{
    cout << "This shared pointer points to: " << pointer.get();
    cout << " and is shared with other " << pointer.use_count() - 1
         << " shared_ptr's\n";

    // Release ownership but does not delete the memory block if it
    // is shared with other smart pointer.
    pointer.reset();

    cout << "This shared pointer points to: " << pointer.get();
    cout << " and " << pointer.use_count()
         << " shared_ptr's shares this memory block\n";
}
```

Example 8: Weak pointer expiration (Part I)

Weak

Smart pointers

```
int main()
{
    auto sharedPointer = make_shared<int>(10);
    auto coOwner = sharedPointer;

    weak_ptr<int> weakPointer(sharedPointer);
    releaseSharedPointer(sharedPointer);

    // Check if the memory block pointed by the shared pointer
    // still exists.
    assert(!weakPointer.expired());
    assert(coOwner.use_count() == 1);

    releaseSharedPointer(coOwner);
    assert(weakPointer.expired());
    assert(coOwner.use_count() == 0);

    return 0;
}
```

Example 8: Weak pointer expiration (Part II)

Weak

Smart pointers

```
void releaseSharedPointer(shared_ptr<int> &pointer)
{
    cout <<
    cout <<                                                    - 1
          << " shared_ptr's\n";

    // Release ownership but does not delete the memory block if it
    // is shared with other smart pointer.
    pointer.reset();

    cout << "This shared pointer points to: " << pointer.get();
    cout << " and " << pointer.use_count()
         << " shared_ptr's shares this memory block\n";
}
```

This function will not change extern shared_ptr states if "pointer" is passed by value

Example 8: Weak pointer expiration (Part I)

Weak

Smart pointers

WARNING !

expired() function is equivalent to use_count() == 0

shared_ptr::get() == nullptr is not the proper way to check if a memory pointed by a shared_ptr has been deleted , because only the control block is thread safe.

Weak

Smart pointers

It is not possible to access directly or dereference the memory block pointed by a weak_ptr.

However is possible to create a shared_ptr owning the memory block pointed by a weak_ptr using the function lock().

lock() returns a shared_ptr that owns the memory block or null if weak_ptr is expired

Weak

Smart pointers

WARNING !

If it is needed to check if the memory block is still allocated before an access use lock() function, because it is atomic and marked as noexcept.

Using expired() in this context can cause problems on multi threaded code.

Variadic templates

Create functions that can receive any number of arguments.

Pass several values, even with different types, to functions.

Variadic templates

Variadic templates are templates with at least one parameter pack, i.e. with a template parameter that receives zero or more template arguments.

Template parameter packs are declared using the elision operator . . . (three dots).

Variadic templates

```
void printValues()
{
    cout << endl;
}

template <class T, class... Ts>
void printValues(T first, Ts... slices)
{
    cout << first;
    printValues(slices...);
}

int main()
{
    printValues("Hello ", "world ", " using ",
                "packed ", "params.");
    return 0;
}
```

Example 1: Print string slices using variadic template

Variadic templates

```
void printValues()
{
    cout << endl;
}

template <class T, class... Ts>
void printValues(T first, Ts... slices)
{
    cout << first;
    printValues(slices...);
}

int main()
{
    printValues("Hello ", "world ", " using ",
                "packed ", "params.");
    return 0;
}
```

Variadic functions are commonly recursive or only forward params in a call to another function

Example 1: Print string slices using variadic template

Variadic templates

```
void printValues()
{
    cout << endl;
}

template <class T, class... Ts>
void printValues(T first, Ts... slices)
{
    cout << first;
    printValues(slices...);
}

int main()
{
    printValues("Hello ", "world ", " using ",
                "packed ", "params.");
    return 0;
}
```

A recursive variadic function must have a base case

Example 1: Print string slices using variadic template

Variadic templates

```
void printValues()
{
    cout << endl;
}

template <class T, class... Ts>
void printValues(T first, Ts... slices)
{
    cout << first;
    printValu
}

int main()
{
    printValues("Hello ", "world ", " using ",
                "packed ", "params.");
    return 0;
}
```

In each function call, the first element on pack is used as “first” parameter.

Example 1: Print string slices using variadic template

Variadic templates

```
void printValues()
{
    cout << endl;
}

template <class T, class... Ts>
void printValues(T first, Ts... slices)
{
    cout << first;
    printValues(slices...);
}

int main()
{
    printValues("Hello ", "world ", " using ",
                "packed ", "params.");
    return 0;
}
```

In each call, “slices” reduces its size in one, until become empty and call the base case.

Example 1: Print string slices using variadic template

Variadic templates

```
void printValues()
{
    cout << endl;
}

template <class T, class... Ts>
void printValues(T first, Ts... slices)
{
    cout << first;
    printValues(slices...);
}

int main()
{
    printValues("Hello ", "world ", " using ",
                "packed ", "params.");
    return 0;
}
```

Function call does not need to specify a template type.

Example 1: Print string slices using variadic template

Variadic templates

WARNING !

Parameter packs can have variables of different types. A variadic function must be prepared to handle all the types passed in the pack.

It is possible to define parameter pack size using sizeof...(T) function. However, it is not possible to access an element in the pack using an index.

Variadic templates

```
template <class T>
int sum(T value) { return value; }

template <class T, class... Ts>
int sum(int initialValue, T first, Ts... values)
{
    return sum(initialValue + first, values...);
}

int main()
{
    cout << "The total is: " << sum(1, 2, 3, 4, 5, 6) << "\n";

    // Does not build because the first parameter is a string.
    //cout << sum("fail", 3, 4, 6, 2, 8);

    return 0;
}
```

Example 2: Variadic function with fixed type parameter

Variadic templates

```
template <class T>
int sum(T value) { return value; }
```

The function base class can receive parameters if their types match the template parameter types.

```
{
    return sum(initialValue + first, values...);
}

int main()
{
    cout << "The total is: " << sum(1, 2, 3, 4, 5, 6) << "\n";

    // Does not build because the first parameter is a string.
    //cout << sum("fail", 3, 4, 6, 2, 8);

    return 0;
}
```

Example 2: Variadic function with fixed type parameter

Variadic templates

```
template <class T>
int sum(T value) { return value; }
```

A call like sum(1, 2, 3, 4, 5, “fail”) would fail because it is not possible to convert const char* to int.

```
{
    return sum(initialValue + first, values...);
}

int main()
{
    cout << "The total is: " << sum(1, 2, 3, 4, 5, 6) << "\n";

    // Does not build because the first parameter is a string.
    //cout << sum("fail", 3, 4, 6, 2, 8);

    return 0;
}
```

Example 2: Variadic function with fixed type parameter

Variadic templates

WARNING !

Even if type mismatches indicated on compile time, it is necessary to take care with undesired casts.

Variadic templates

```
template <class T, class... Ts>
void printOrderedDistinct(T first, Ts... values)
{
    set<T> setOfValues = {first, values...};
    cout << "Ordered and distinct values: \n";
    for (auto item : setOfValues)
        cout << item << " ";
    cout << endl;
}

template <class... Ts>
void printFreeDays(Ts... values)
{
    vector<int> freeDays{1, values..., 7};
    cout << "The free days are: \n";
    for (auto item : freeDays)
        cout << item << " ";
    cout << endl;
}
```

Example 3: Packed parameters on container initialization (Part I).

Variadic templates

```
...

int main()
{
    printOrderedDistinct(9, 4, 1, 2, 2, 9, 10, 3, 9, 1, 8);
    printFreeDays(3, 5);
    printFreeDays(2, 4, 6);

    return 0;
}
```

Example 3: Packed parameters on container initialization (Part II).

Variadic templates

```
template <class T, class... Ts>
void printOrderedDistinct(T first, Ts... values)
{
    set<T> setOfValues = {first, values...};
```

The parameter “first” is used to define set template class. It is not necessary if the template class is explicit.

```
    cout << endl;
}

template <class... Ts>
void printFreeDays(Ts... values)
{
    vector<int> freeDays{1, values..., 7};
    cout << "The free days are: \n";
    for (auto item : freeDays)
        cout << item << " ";
    cout << endl;
}
```

Example 3: Packed parameters on container initialization (Part I).

Variadic templates

```
template <class T, class... Ts>
void printOrderedDistinct(T first, Ts... values)
{
    set<T> setOfValues = {first, values...};
    cout << "Ordered and distinct values: \n";
    for (auto item : setOfValues)
        cout << item << " ";
    cout << endl;
}

te
void printFreeDays(Ts... values)
{
    vector<int> freeDays{1, values..., 7};
    cout << "The free days are: \n";
    for (auto item : freeDays)
        cout << item << " ";
    cout << endl;
}
```

This function will work with any type, since all packed parameters have all the same type.

Example 3: Packed parameters on container initialization (Part I).

Variadic templates

```
template <class T, class... Ts>
void printOrderedDistinct(T first, Ts... values)
{
    set<T> setOfValues = {first, values...};
    cout << "Ordered and distinct values: \n";
    for (auto item : setOfValues)

}
```

This function will work only if all packed values are convertible to int.

```
template <class... Ts>
void printFreeDays(Ts... values)
{
    vector<int> freeDays{1, values..., 7};
    cout << "The free days are: \n";
    for (auto item : freeDays)
        cout << item << " ";
    cout << endl;
}
```

Example 3: Packed parameters on container initialization (Part I).

Variadic templates

```
bool isPrime(int first)
{
    if (first == 2) return true;
    if (first % 2 == 0) return false;

    const auto squaredRoot = sqrt(first);
    for (int i = 3; i < sqrt(first); i += 2)
        if (first % i == 0)
            return false;
    return true;
}

template <class... Ts>
vector<bool> checkPrimes(Ts... sequence)
{
    return vector<bool>({isPrime(sequence)...});
}
```

Example 4: Function expansion with packed parameters (Part I).

Variadic templates

REMEMBER !

Packed parameters can be used to represent any sequence of values separated by comma (at compile time).

By initializing a container by packed parameters it is possible to iterate through it.

Variadic templates

```
int main()
{
    vector<bool> results = checkPrimes(3, 12, 5, 29, 2, 4, 15);

    cout << "{";
    for (auto i = 0; i < results.size(); i++)
        cout << results[i] << (i < results.size() - 1 ? " - " : "");
    cout << "}\n";

    return 0;
}
```

Example 4: Function expansion with packed parameters (Part II).

Variadic templates

```
bool isPrime(int first)
{
    if (first == 2) return true;
    if (first % 2 == 0) return false;

    const auto squaredRoot = sqrt(first);
    for (int i = 3; i < sqrt(first); i += 2)
        if (first % i == 0)
            return false;

}
```

checkPrimes(3, 12, 5, 29, 2, 4, 15) generates 7 isPrime calls with results separated by comma(,).

```
template <class... Ts>
vector<bool> checkPrimes(Ts... sequence)
{
    return vector<bool>({isPrime(sequence)...});
}
```

Example 4: Function expansion with packed parameters (Part I).

Variadic templates

```
bool isPrime(int first)
{
    if (first == 2) return true;
    if (first % 2 == 0) return false;

    const auto squaredRoot = sqrt(first);
    for (int i = 3; i < sqrt(first); i += 2)
        if (first % i == 0)
            re
    return tru
}

template <clas
vector<bool> c
{
    return vector<bool>({isPrime(sequence)...});
}
```

It is compiled as:
vector<bool>({
isPrime(3), isPrime(12), isPrime(5),
isPrime(29), isPrime(2), isPrime(4),
isPrime(15)
}).

Example 4: Function expansion with packed parameters (Part I).

Variadic templates

REMEMBER !

Packed parameters are always defined and expanded at compile time. It is not possible to create packed parameters at runtime.