Preluare si compilare din diverse surse

- Alexandrescu Andrei - Modern C++ Design: Generic Programming and Design Patterns Applied
- https://www.geeksforgeeks.org/smart-pointers-cpp/
- https://www.geeksforgeeks.org/auto_ptr-unique_ptr-shared_ptr-weak_ptr-2/
- https://beginnersbook.com/2017/08/cpp-functions/
- https://www.softwaretestinghelp.com/lambdas-in-cpp/
- https://riptutorial.com/cplusplus/example/1854/what-is-a-lambda-expression-
- https://en.cppreference.com/w/cpp/language/lambda
- https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=msvc-170

Smart pointers

Problem: no delete

```
class TEST {    …    };

void fun(){   TEST * p = new TEST(); }

int main()
{   // Infinite Loop
    while (1) {
        fun();
    }
}
```

p will be destroyed as it is a local variable. But, the memory it consumed won't be deallocated because we forgot to use *delete p;*

Smart pointers

Problem: no delete

not deallocating a pointer causes a memory leak that may lead to crash of the program

C++11 comes up with its own mechanism to smartly deallocate unused memory: *Smart Pointer.* When the object is destroyed it frees the memory as well.

The idea is to take a class with a pointer, destructor and overloaded operators like * and ->. Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted (or reference count can be decremented).

Smart pointers

- C++ objects that simulate simple pointers by implementing operator-> and the unary operator*.

- perform useful tasks—such as memory management or locking.

- almost all good-quality smart pointers in existence are templated by the pointee type.

- replace pointer definitions with smart pointer definitions without incurring major changes to your application’s code

- Smart pointers have value semantics, whereas some simple pointers do not; An object with value semantics is an object that you can copy and assign to.

Smart pointers

```
class SmartPtr  {
public:
        explicit SmartPtr (int* ptr = NULL) : { p = ptr; }
        ~SmartPtr() { delete p; }
        int& operator*() const  {   return *p; }
private:
int* p;
};
int main()  {
    SmartPtr ptr(new int());
    *ptr = 20;
    cout << *ptr;
}
```

```
// We don't need to call delete ptr: when the object
    // ptr goes out of scope, the destructor for it is automatically
    // called and destructor does delete ptr.
```

Smart pointers

```
template<class T>
class SmartPtr  {
public:
        explicit SmartPtr (T* ptr = NULL) : { p = ptr; }
        ~SmartPtr() { delete p; }
        T& operator*() const  {   return *p; }
        T* operator->() const {return p;}
private:
T* p;
};
int main()  {
    SmartPtr<int> ptr(new int());
    *ptr = 20;
    cout << *ptr;  }
```

Smart pointers

Types of smart pointers

auto_ptr **(is deprecated)**

1. unique_ptr

unique_ptr stores one pointer only. We can assign a different object by removing the current object from the pointer.

2. shared_ptr

By using *shared_ptr* more than one pointer can point to this one object at a time and it'll maintain a **Reference Counter** using ***use_count()*** **method.**

3. weak_ptr

It's much more similar to shared_ptr except it'll not maintain a **Reference Counter**. In this case, a pointer will not have a stronghold on the object.

Smart pointers

auto_ptr

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

class A {
public:
    void show() { cout << "A::show()" << endl; }
};

int main()
{
    auto_ptr<A> p1(new A);
    p1->show();

    cout << p1.get() << endl;    /// returns the memory address of p1

    auto_ptr<A> p2(p1); /// copy constructor called, this makes p1 empty.
    p2->show();
    cout << p1.get() << endl;
    cout << p2.get() << endl;

    return 0;
}
```

Smart pointers

unique_ptr

std::unique_ptr was developed in C++11 as a replacement for std::auto_ptr. unique_ptr is a new facility with similar functionality, but with improved security (no fake copy assignments), added features (deleters) and support for arrays.

any attempt to make a copy of unique_ptr will cause a compile-time error.

```
unique_ptr<A> ptr1 (new A);

// Error: can't copy unique_ptr
unique_ptr<A> ptr2 = ptr1;
```

unique_ptr can be moved using the new move semantics i.e. using std::move() function to transfer ownership of the contained pointer to another unique_ptr.

Smart pointers

unique_ptr

```cpp
#include <memory>
using namespace std;

class A {
public:
    void show() { cout << "A::show()" << endl; }
};

int main()
{
    unique_ptr<A> p1(new A);
    p1->show();
    cout << p1.get() << endl;     /// returns the memory address of p1

    unique_ptr<A> p2 = move(p1);   /// transfers ownership to p2
    p2->show();
    cout << p1.get() << endl;
    cout << p2.get() << endl;

    unique_ptr<A> p3 = move(p2);    /// transfers ownership to p3
    p3->show();
    cout << p1.get() << endl;
    cout << p2.get() << endl;
    cout << p3.get() << endl;

    return 0;
}
```

Smart pointers

shared_ptr

A shared_ptr is a container for raw pointers. It is a **reference counting ownership model** i.e. it maintains the reference count of its contained pointer in cooperation with all copies of the shared_ptr. So, the counter is incremented each time a new pointer points to the resource and decremented when the destructor of the object is called.

An object referenced by the contained raw pointer will not be destroyed until reference count is greater than zero i.e. until all copies of shared_ptr have been deleted.

Smart pointers

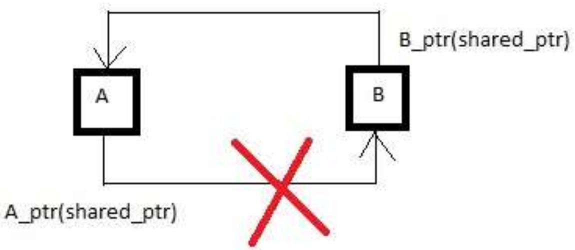
shared_ptr

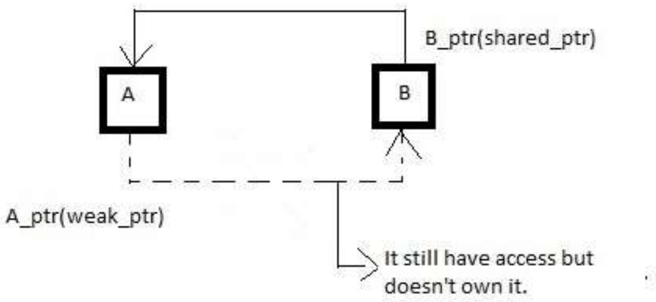
```cpp
#include <iostream>
#include <memory>
using namespace std;

class A {
public:
    void show() { cout << "A::show()" << endl; }
};

int main()
{
    shared_ptr<A> p1(new A);
    cout << p1.get() << endl;
    p1->show();
    shared_ptr<A> p2(p1);
    p2->show();
    cout << p1.get() << endl;
    cout << p2.get() << endl;

    /// Returns the number of shared_ptr objects referring to the same managed object.
    cout << p1.use_count() << endl;
    cout << p2.use_count() << endl;

    /// Relinquishes ownership of p1 on the object and pointer becomes NULL
    p1.reset();
    cout << p1.get() << endl;
    cout << p2.use_count() << endl;
    cout << p2.get() << endl;
```

Smart pointers

weak_ptr

A weak_ptr is created as a copy of shared_ptr. It provides access to an object that is owned by one or more shared_ptr instances but does not participate in reference counting. The existence or destruction of weak_ptr has no effect on the shared_ptr or its other copies. It is required in some cases to break circular references between shared_ptr instances.





Move Constructors in C++

Copy constructors in C++ work with the l-value references and copy semantics(copy semantics means copying the actual data of the object to another object rather than making another object to point the already existing object in the heap).

Move constructors work on the r-value references and move semantics(move semantics involves pointing to the already existing object in the memory).

Work of move constructor looks a bit like default member-wise copy constructor but in this case, it nulls out the pointer of the temporary object preventing more than one object to point to same memory location.

Move Constructors in C++

```cpp
#include <vector>
using namespace std;
class Move {
    int* data;
public:
    Move(int d)  {
        data = new int;
        *data = d;
        cout << "Constructor is called for "<< d << endl;
    };
    Move(const Move& source): Move{ *source.data }   {
        cout << "Copy Constructor is called - "<< "Deep copy for "<< *source.data<< endl;
    }

    ~Move()  {
        if (data != nullptr)
            cout << "Destructor is called for "<< *data << e
        else
            cout << "Destructor is called for nullptr"<< end
        delete data;
    }
};
int main()
{
    vector<Move> vec;
    vec.push_back(Move{ 10 });
    vec.push_back(Move{ 20 });
    return 0;
}
```

```
Constructor is called for 10
Constructor is called for 10
Copy Constructor is called - Deep copy for 10
Destructor is called for 10
Constructor is called for 20
Constructor is called for 20
Copy Constructor is called - Deep copy for 20
Constructor is called for 10
Copy Constructor is called - Deep copy for 10
Destructor is called for 10
Destructor is called for 20
Destructor is called for 10
Destructor is called for 20
```

Move Constructors in C++

Syntax of the Move Constructor:

```
Object_name(Object_name&& obj)
   : data{ obj.data }
{
   // Nulling out the pointer to the temporary data
   obj.data = nullptr;
}
```

This unnecessary use of the memory can be avoided by using move constructor.

Move Constructors in C++

```cpp
#include <vector>
using namespace std;
class Move {
    int* data;
public:
    Move(int d) {
        data = new int;
        *data = d;
        cout << "Constructor is called for "<< d << endl;
    };
    Move(const Move& source): Move{ *source.data }  {
        cout << "Copy Constructor is called - "<< "Deep copy for "<< *source.data<< endl;
    }
/// Move Constructor
    Move(Move&& source): data{ source.data } {
        cout << "Move Constructor for "<< *source.data << endl;
        source.data = nullptr;
    }

    ~Move() {
};
int main()
{
    vector<Move> vec;
    vec.push_back(Move{ 10 });
    vec.push_back(Move{ 20 });
    return 0;
}
```

```
Constructor is called for 10
Move Constructor for 10
Destructor is called for nullptr
Constructor is called for 20
Move Constructor for 20
Constructor is called for 10
Copy Constructor is called - Deep copy for 10
Destructor is called for 10
Destructor is called for nullptr
Destructor is called for 10
Destructor is called for 20
```

Lambda expressions

Lambdas, as they are commonly called, are basically small inline snippets of code that can be used inside functions or even function call statements. They are not named or reused.

General syntax

```
[Capture clause] (parameter_list) mutable exception ->return_type
 {
 Method definition;
    }
```

Lambda expressions

Capture closure: Lambda introducer as per C++ specification.

Parameter list: Also called as lambda declarations. Is optional and is similar to the parameter list of a method.

Mutable: Optional. Enables variables captured by a call by value to be modified.

exception: Exception specification. Optional. Use “noexcept” to indicate that lambda does not throw an exception.

Return_type: Optional. The compiler deduces the return type of the expression on its own. But as lambdas get more complex, it is better to include return type as the compiler may not be able to deduce the return type.

Method definition: Lambda body.

Lambda expressions

Capture clause

A lambda can introduce new variables in its body (in C++14), and it can also access, or *capture*, variables from the surrounding scope.

Variables that have the ampersand (&) prefix are accessed by reference and variables that don't have it are accessed by value.

An empty capture clause, [ ], indicates that the body of the lambda expression accesses no variables in the enclosing scope.

```
int a = 0;                             // Define an integer variable
auto f = []()   { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]()  { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
                                       //      Note: It is the responsibility of the programmer
                                       //      to ensure that a is not destroyed before the
                                       //      lambda is called.
auto b = f();                          // Call the lambda function. a is taken from the capture
```

Lambda expressions

If we have a capture-default & a capture clause, then we cannot have an identifier in the capture of that particular capture can have the & identifier. Similarly, if the capture clause contains capture-default =, then the capture clause cannot have the form = identifier.

```
[&sum, sum_var]          //OK, explicitly specified capture by value
[sum_var, &sum]         //ok, explicitly specified capture by reference
[&, &sum_var]          // error, & is the default still sum_var preceded by &
[i, i]                    //error, i is used more than once
```

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};      // OK
    [&, &i]{};     // ERROR: i preceded by & when & is the default
    [=, this]{};   // ERROR: this when = is the default
    [=, *this]{ }; // OK: captures this by value. See below.
    [i, i]{};      // ERROR: i repeated
}
```

Lambda expressions

Given below is a basic Example of a Lambda Expression in C+·

```
#include <iostream>
#include <string>
using namespace std;
int main()
   {
      auto sum = [](int a, int b) {
      return a + b;
      };

   cout <<"Sum of two integers:"<< sum(5, 6) << endl;

   return 0;
   }
```

Lambda expressions

lambdas generalized for all data types. This is done from C++14 onwards.

```
#include <iostream>
#include <string>
using namespace std;
int main()
   {

      // generalized lambda
      auto sum = [](auto a, auto b) {
         return a + b;
         };

      cout <<"Sum(5,6) = "<< sum(5, 6) << endl; // sum of two integers

      cout <<"Sum(2.0,6.5) = "<<sum(2.0, 6.5) << endl; // sum of two floa

      cout <<"Sum((string(\"SoftwareTesting\"), string(\"help.com\")) =
      "<<sum(string("SoftwareTesting"), string("help.com")) << endl; // s

      return 0;
   }
```

Lambda expressions

Parameter list

`()` is the **parameter list**, which is almost the same as in regular functions. If the lambda takes no arguments, these parentheses can be omitted (except if you need to declare the lambda `mutable`). These two lambdas are equivalent:

```
auto call_foo  = [x](){ x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

C++14

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs,
std::vector<T>::const_reference rhs) { return lhs < rhs; };

auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

Lambda expressions

```
#include <iostream>

int main()
{
    using namespace std;
    int n = [] (int x, int y) { return x + y; }(5, 4);
    cout << n << endl;
}
```

Function body

`{}` is the **body**, which is the same as in regular functions.

Calling a lambda

A lambda expression's result object is a closure, which can be called using the `operator()` (as with other function objects):

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // Prints 10

multiplier = 15;
std::out << timesFive(2); // Still prints 2*5 == 10
```

Lambda expressions

Return Type

By default, the return type of a lambda expression is deduced.

```
[](){ return true; };
```

In this case the return type is `bool`.

You can also manually specify the return type using the following syntax:

```
[]() -> bool { return true; };
```

Lambda expressions

Lambda body

The lambda body of a lambda expression is a compound statement. It can contain anything that's allowed in the body of an ordinary function or member function. The body of both an ordinary function and a lambda expression can access these kinds of variables:

- Captured variables from the enclosing scope, as described previously.
- Parameters.
- Locally declared variables.
- Class data members, when declared inside a class and `this` is captured.
- Any variable that has static storage duration—for example, global variables.

Lambda expressions

Before C++11

```
class compare
{
    public:
    compare(int a) {x = a;}

    bool operator()(int a) const
    {
        return a < x;
    }
    int x;
};

int main()
{
const int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> vec(arr, arr+5);

int x = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), compare(x));
return 0;
}
```

ter C++11

```
vector<int> vec{ 1, 2, 3, 4, 5 };

int x = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value < x; });
```

Mutable Lambda

Objects captured by value in the lambda are by default immutable. This is because the `operator()` of the generated closure object is `const` by default.

```
auto func = [c = 0](){++c; std::cout << c;};  // fails to compile because ++c
                                              // tries to mutate the state of
                                              // the lambda.
```

Modifying can be allowed by using the keyword `mutable`, which make the closer object's `operator()` non-`const`:

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

If used together with the return type, `mutable` comes before it.

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

constexpr lambda expressions

(available in /std:c++17 mode and later): You may declare a lambda expression as constexpr (or use it in a constant expression) when the initialization of each captured or introduced data member is allowed within a constant expression.

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

Lambda expressions

Examples

```cpp
// Assign the lambda expression that adds two numbers to an auto variable.
auto f1 = [](int x, int y) { return x + y; };

cout << f1(2, 3) << endl;

// Assign the same lambda expression to a function object.
function<int(int, int)> f2 = [](int x, int y) { return x + y; };

cout << f2(3, 4) << endl;
```

```cpp
int i = 3;
int j = 5;

// The following lambda expression captures i by value and
// j by reference.
function<int (void)> f = [i, &j] { return i + j; };

// Change the values of i and j.
i = 22;
j = 44;

// Call f and print its result.
cout << f() << endl;
```

Lambda expressions

Examples

```
int n = [] (int x, int y) { return x + y; }(5, 4);
cout << n << endl;
```

```
// Create a list of integers with a few initial elements.
list<int> numbers;
numbers.push_back(13);
numbers.push_back(17);
numbers.push_back(42);
numbers.push_back(46);
numbers.push_back(99);

// Use the find_if function and a lambda expression to find the
// first even number in the list.
const list<int>::const_iterator result =
    find_if(numbers.begin(), numbers.end(),[](int n) { return (n % 2) == 0; });

// Print the result.
if (result != numbers.end()) {
    cout << "The first even number in the list is " << *result << "." << endl;
} else {
    cout << "The list contains no even numbers." << endl;
}
```

Lambda expressions

Examples

```
    // The following lambda expression contains a nested
// lambda expression.

    int timestwoplusthree = [](int x) { return [](int y) {
return y * 2; }(x) + 3; }(5);

    // Print the result.
    cout << timestwoplusthree << endl;
}
```

Lambda expressions

Examples

```
// capture "this" by reference
void ApplyScale(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [this](int n) { cout << n * _scale << endl; });
}

// capture "this" by value (Visual Studio 2017 version 15.3 and later)
void ApplyScale2(const vector<int>& v) const
{
   for_each(v.begin(), v.end(),
      [*this](int n) { cout << n * _scale << endl; });
}
```

Lambda expressions

Examples

```
// template_lambda_expression.cpp
// compile with: /EHsc
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

// Negates each element in the vector object. Assumes s
template <typename T>
void negate_all(vector<T>& v)
{
    for_each(v.begin(), v.end(), [](T& n) { n = -n; });
}

// Prints to the console each element in the vector object.
template <typename T>
void print_all(const vector<T>& v)
{
    for_each(v.begin(), v.end(), [](const T& n) { cout << n << endl; });
}
```

```
int main()
{
    // Create a vector of signed integers with a few elements.
    vector<int> v;
    v.push_back(34);
    v.push_back(-43);
    v.push_back(56);

    print_all(v);
    negate_all(v);
    cout << "After negate_all():" << endl;
    print_all(v);
}
```

Generic Lambda

c++14

Lambda functions can take arguments of arbitrary types.

```
auto twice = [](auto x){ return x+x; };

int i = twice(2); // i == 4
std::string s = twice("hello"); // s == "hellohello"
```

This is implemented in C++ by making the closure type's `operator()` overload a template function. The following type has equivalent behavior to the above lambda closure:

```
struct _unique_lambda_type
{
  template<typename T>
  auto operator() (T x) const {return x + x;}
};
```

https://riptutorial.com/cplusplus/example/1969/generic-lambdas

Generic Lambda

Not all parameters in a generic lambda need be generic:

```
[](auto x, int y) {return x + y;}
```

Here, `x` is deduced based on the first function argument, while `y` will always be `int`.

Generic lambdas can take arguments by reference as well, using the usual rules for `auto` and `&`. If a generic parameter is taken as `auto&&`, this is a *forwarding* reference to the passed in argument and not an *rvalue* reference:

```
auto lamb1 = [](int &&x) {return x + 5;};
auto lamb2 = [](auto &&x) {return x + 5;};
int x = 10;
lamb1(x); // Illegal; must use `std::move(x)` for `int&&` parameters.
lamb2(x); // Legal; the type of `x` is deduced as `int&`.
```

https://riptutorial.com/cplusplus/example/1969/generic-lambdas

Lambda expressions

Future

Syntax

[*captures*] (*params*) *lambda-specifiers* *requires*(optional) { *body* }	(1)	
[*captures*] { *body* }	(2)	(until C++23)
[*captures*] *lambda-specifiers* { *body* }	(2)	(since C++23)
[*captures*] *<tparams>* *requires*(optional) (*params*) *lambda-specifiers* *requires*(optional) { *body* }	(3)	(since C++20)
[*captures*] *<tparams>* *requires*(optional) { *body* }	(4)	(since C++20) (until C++23)
[*captures*] *<tparams>* *requires*(optional) *lambda-specifiers* { *body* }	(4)	(since C++23)

1) Full declaration.

2) Omitted parameter list: function takes no arguments, as if the parameter list were ().

3) Same as 1), but specifies a generic lambda and explicitly provides a list of template parameters.

4) Same as 2), but specifies a generic lambda and explicitly provides a list of template parameters.

https://en.cppreference.com/w/cpp/language/lambda

Lambda expressions

To do:

```cpp
int i = 8;
auto f =
    [i]() mutable
{
    int j = 2;
    auto m = [&i,j]()mutable{ i /= j; };
    m();
    cout << "inner: " << i;
};
f();
cout << " outer: " << i;
}
```

```cpp
int i=1, j=2, k=3;
auto f =  [i,&j,&k]() mutable    {
    auto m =  [&i,j,&k]() mutable
    {
        i=4;
        j=5;
        k=6;
    };
    m();
    cout << i << j << k;
};
f();
cout << " : " << i << j << k;
```