Section 24 : C++ 11

15 April 2022 16:49

© Rajat Kumar
https://www.linkedin.com/in/imRajat/
https://github.com/im-Rajat

Section 24 : C++ 11

Auto :

- When we don't know the data type that we require or it depends on the result, then we can declare it as auto.
- auto x=2*5.7+ 'a'; // automatically it will become double
- Cout<<x;

```
double d=12.3;
int i=9;
auto x=2*d+i;
cout<<x;
```

- We can use auto when we are calling a function and we don't know what's the return type of that function.

```
float fun()
{
    return 2.34f;
}
int main()
{
    auto x=fun();
    cout<<x;
}
```

- One more feature of c++ 11 is when we won't to copy data type of another variable but we don't know it's data type but we have the variable name, then we can use **decltype** function.

```
int x=10;
float y=90.5;

decltype(y) z=12.3;
```

  - Data type of z will be same as data type of y.
  - Variable z will be of float type because y is of float type.

Final Keyword :

- One usage of **final** is it restrict inheritance.
- If we declare a class as final then it can't be inherit.

```
class Parent final
{

};

class Child:Parent
{
        Base 'Parent' is marked 'final'

};
```

- Only virtual function can be marked as final.
- The final function of parent class cannot be override in the child class.

```
class Parent
{
    virtual void show() final
    {

    }

};
```

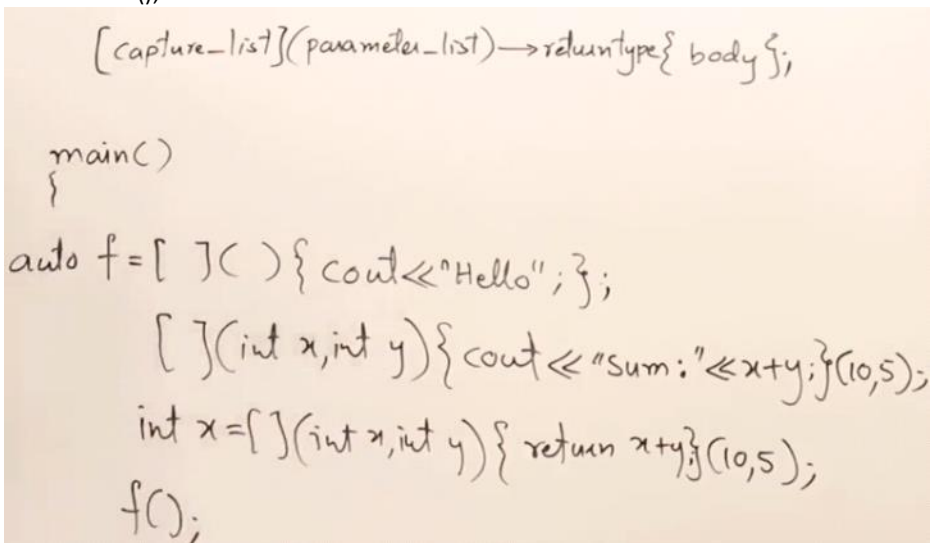
Declaration of 'show' overrides a 'final' function

```
class Child:Parent
{
    void show()
    {

    }
};
```

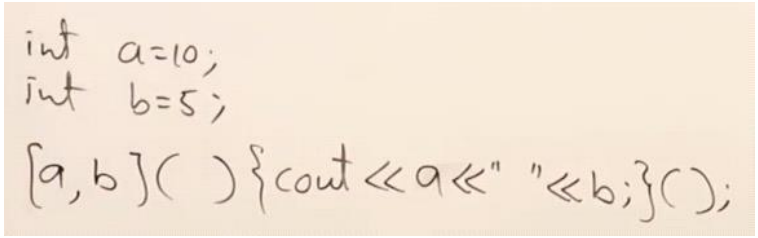
- **Final keyword** is used in C++ for **restricting inheritance** as well as **restricting overriding all functions**.
- Same feature is available in Java also, only difference is the final key word is used before the return type of a function and here it is written after the name of a function or after the parameter list of a function.

Lambda Expressions :

- It is useful for defining unnamed functions.
- Syntax :
  - [capture_list](parameter_list) -> return_type { body };
- Define as well as call function to print "Hello" in main.
  - []() { cout<<"Hello"; } ( ); // () this is for calling the function.
- With parameters :
  - [](int x, int y) { cout << "sum: "<<x+y;};
  - For calling :
    - [](int x, int y) { cout << "sum: "<<x+y;} (10, 5) ; // we are passing 10 and 5 as x and y respectively.
- With return type :
  - Int x = [](int x, int y) {return x + y;} (10, 5);
  - I don't have to mention the return type, because C++ support auto type of declaration. So automatically, whatever the data type is, that data type is return.
- Instead of calling the function, even we can assign it with some variable auto.
  - Auto f = []() { cout<<"Hello"; };
  - The function is still unnamed, it's just reference to that function.
  - For calling this function :
    - f();

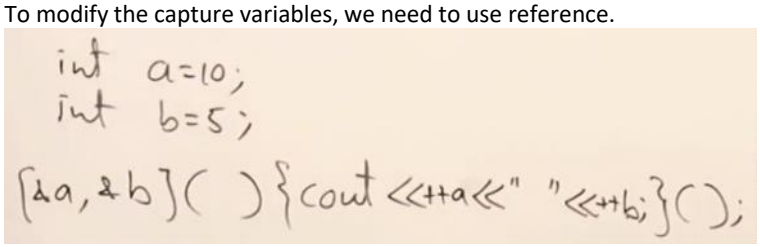


- If we want to mention the return type (it doesn't require but we still can)
  - int s=[ ] (int n, int y) -> int { return x+y; } (10,5);
- We can access the local variables of all function inside unnamed function, but we need to capture them using capture_list.

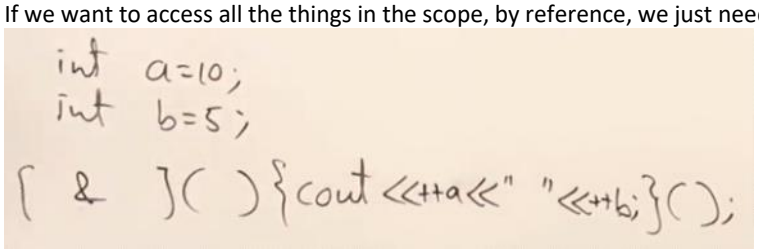


- But we cannot modify these captured variables like by using a++, ++b directly.

- To modify the capture variables, we need to use reference.



- If we want to access all the things in the scope, by reference, we just need to write reference symbol & in capture list.



Lambda Expression Examples :

```
int main()
{
    [] () { cout<<"Hello"<<endl; } ();       // Hello
    [] (int x, int y) { cout<<"Sum is: "<<x+y<<endl; } (10, 30); // Sum is: 40
    cout << ( [] (int x, int y) { return x + y; } (20, 80) ) <<endl;;   // 100

    int a = [] (int x, int y) -> int { return x + y; } (20, 80);
    cout<<a<<endl;       // 100;

    int b = 10;
    [b] () { cout<<b<<endl; } ();    // 10
    auto f = [b] () { cout<<b<<endl; };
    f();         // 10

    int c = 55;
    auto g = [&c] () { cout<<c<<endl; };
    g();         // 55
    c++;
    g();         // 56

    int d = 20;
    auto h = [&d] () { cout<<++d<<endl; };
    h();         // 21

    return 0;
}
```

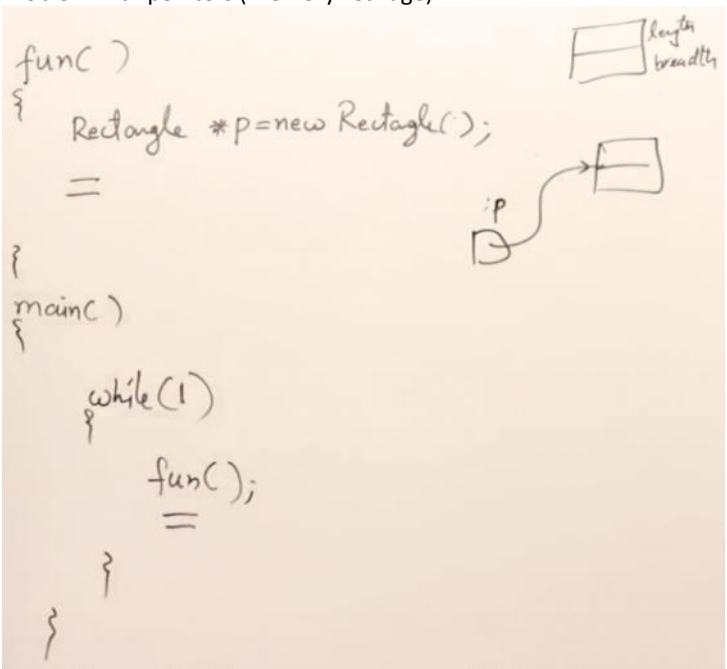
- We can also send a lambda expression to a function as a parameter.

```
template<typename T>
void fun(T p)
{
    p();
}

int main()
{
    int a=10;
    auto f=[&a](){cout<<a++<<endl;};

    fun(f);
    fun(f);
}
```

- Lambda expressions are very flexible for writing or defining a function or the scope of a function within a line or within a block of statement. It is helpful for writing nested functions also.
- And this is a feature of functional programming which is used in artificial intelligence.

Smart Pointers :

- Pointers are used for accessing the resources which are external to the program like Heap Memory.
- The problem with the heap memory is that when we don't need it, we must deallocate it.
- Languages like java, C# or dot net framework that provides garbage collection mechanism to be look at objects which are not in use.
- C++ provides smart pointers which automatically manage memory and they will deallocate the object when they are not in use. When pointer is going out of scope, automatically it will deallocate the memory.
- There are 3 types of smart pointer available :
  - **Unique_ptr**
  - **Shared_ptr**
  - **Weak_ptr**
- Problem with pointers (Memory Leakage) :



  - Good practice : at the end of function delete memory, for example **delete p;** in function fun() last line.

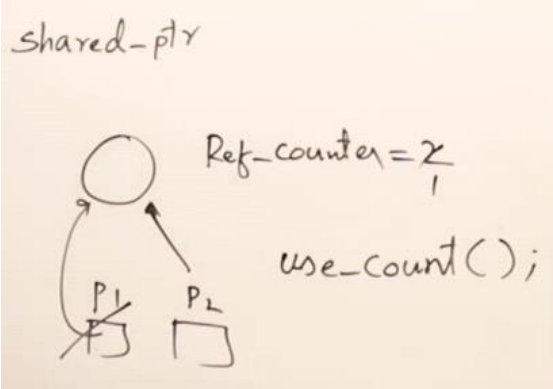
Unique_ptr :

- Instead of writing Rectangle *p = new Rectangle(10, 5);, we write, syntax :
  - **Unique_ptr<Rectangle> p1(new Reactangle(10, 5));**
  - Cout<<p1->area();
  - Cout<<p1->perimeter();
- Unique pointer will take care of deletion of object/memory when program goes out of scope.
- Only one pointer can to one object. We cannot share the object with another pointer.

- But we transfer the control to another pointer by removing the first pointer first.
- It means upon ab object at a time only one pointer can point.
- unique_ptr' is defined in header - **#include <memory>**

Shared_ptr :

- **shared_ptr<Rectangle> p1(new Rectangle(10, 5));**
- More then 1 pointer can point to this one and this will maintain a reference counter (ref_counter) that how many pointers are pointing to the object.



- Shared pointer, It can be used by more than one pointers, more than one pointer can point on the same object.
- We can know the reference counter by calling function use_count() on a shared pointer.

Weak_ptr :

- Same as shared pointer, but it will not maintain reference counter.
- **Weak_ptr<Rectangle> p2(new Rectangle(10, 5));**
- Pointing will be there, but it's weak point. The pointer will not have string hold on the object.
- The reason is if suppose - The pointers are holding the objects and requesting for other objects, then they may form a deadlock between the pointers. So to avoid a deadlock, weak pointers are useful.
- So it doesn't have a reference counter. So it's more like unique, but it allows the pointers to share it. It's between unique and shared.

Unique Pointer example :

```
class Rectangle {
private:
    int length;
    int breadth;
public:
    Rectangle(int l, int b) {
        length = l;
        breadth = b;
    }
    int area() {
        return length * breadth;
    }
};

int main()
{
    unique_ptr<Rectangle> ptr(new Rectangle(10, 5));
    cout<<ptr->area()<<endl;       // 50

    // unique_ptr<Rectangle> ptr2 = ptr;     // error
    unique_ptr<Rectangle> ptr2;
    ptr2 = move(ptr);

    cout<<ptr2->area()<<endl;       // 50
    // cout<<ptr->area()<<endl;     // program crash
    // because ptr is not pointing on any valid object.


    return 0;
```

Shared Pointer example :

```
shared_ptr<Rectangle> ptr3(new Rectangle(100, 2));
cout<<ptr3->area()<<endl;       // 200

shared_ptr<Rectangle> ptr4;
ptr4 = ptr3;

cout<<ptr4->area()<<endl;       // 200
cout<<ptr3->area()<<endl;     // 200
cout<<ptr3.use_count()<<endl;     // 2 is reference count
```

- Both ptr3 and ptr4 are pointing on the same object.

- It's suggested that we use shared pointer or a unique pointer instead of using normal pointers.
- The benefit is that this will not cause memory leak. It is just like garbage collection.

InClass Initializer and Delegation of Constructors :

- In C++ 11, in-class initialization is allowed.

```
class Test
{
    int x=10;
    int y=13;

};
```

- Direct initialization of variable in class is allowed now, previously it's not allowed in c++.

Delegation of Constructors :

- We can pass or allow non parametrized constructor to call parametrized constructor by this :

```
Test(int a,int b)
{
    x=a;
    y=b;
}
Test():Test(1,1)
{}
```

  - This is delegation of constructor.
  - This is also a new feature of C++, so it means one constructor can call another constructor within the same class.

Ellipses : (Need to include **#include <stdarg.h>** or **#include <cstdarg>**)

- Ellipses is used for taking variable number of arguments in a function.
- For example, I want to write a function for finding the sum of elements for some of integers. But I can pass any number of elements in function, 2, 3, 4, 5, etc.
- For ellipses to work we must tell the function how many arguments we are calling.
- Syntax :
  - Int sum(int n, …) { … };
  - Calling :
    - Sum(3, 1, 3, 3) where n = 3 elements;
    - Sum(5, 1, 4, 3, 10) where n = 5 elements;
  - We use va_list, va_start, va_arg, va_end functions to work with ellipses.
- This was the feature of C Language, but now also available in C++.
- In C++ we have function overloading, which also help in different number of arguments.
- Example to use ellipses :

```cpp
#include <iostream>
#include <stdarg.h>
//#include <cstdarg>
using namespace std;

int sum(int n, ... ) {
    va_list list;
    va_start(list, n);

    int x;
    int s = 0;
    for (int i = 0; i < n; i++) {
        x = va_arg(list, int);
        s = s + x;
    }

    return s;
}

int main()
{
    int s = sum(4, 1, 2, 3, 4);
    cout<<s<<endl;

    s = sum(7, 10, 10, 20, 20, 30, 40, 50);
    cout<<s<<endl;

    return 0;
}
```

- This is very useful feature like printf and scanf functions of C language uses this ellipses for variable number of arguments.