Page No.
Date

initializer_list<>

C++'11 introduced concept of uniform initialization. It is used for initializing object, dynamic/static array or other containers.

```
int a{10};
float b{2.2f};
int arr[]{11,22,33};
string str{"hello"};
double *ptr = new double[3]{1.1,2.2,3.3};
```

initializer_list<T> is a light weight proxy object that provides access to an array of objects of type const T.

initializer_list object is automatically created

① a braced init list used to list-initialize an object, where ctor accept initializer_list as param.

```
vector<int> v{1,2,3,4};  → initializer_list<int>
auto v1 = {1,2,3};
```

② braced-init-list on right side of assign operator or as fn call argument, which accept initializer_list as arg.

③ braced-init-list is bound to auto, including a ranged for loop.

Page No.
Date

for a initializer_list<T> - braced-init-list underlying array is const T[N]. Lifetime of this temporary array is similar to any temporary object, however initializer_list object from that array extends lifetime of array. Note that underlying array may be allocated in readonly memory (depends on compiler implementation).

initializer_list<> is readonly (not meant to be modified) and typically accessed through iterator or range-based for loop.

```
class Bag {
    int arr[10] = {};
    int size {};
public:
    Bag(const initializer_list<int>& l) {
        int max = sizeof(arr)/sizeof(arr[0]); i=0
        for(auto v: l) {
            if (i >= max)
                break;
            arr[i++] = v;
        }
        size = i;
    }
    int* begin() {          int* end() {
        return arr;             return arr + size;
    }                       }
```

Page No.
Date

```
Bag& operator=(const initializer_list<int>& l){
    // Same as ctor
    return *this;
}
};
```

In main:

Bag bl{1,2,3,4}; → ctor called

bl = {11,22,33,44}; → op=() called

for(auto v: bl) → begin & end() called

cout << v;

refer range based for loop.

Page No.
Date 10 3 21

Inheriting Constructors

In C++, following fns are not inherited

1. Constructor
2. destructor
3. assignment operator

It means that derived class must provide its own implementation. If not implemented compiler will synthesize.

C++11 allow inheriting ctor of base class. This is done by simply redeclaring ctor of base in derived with "using" keyword;

```
class Emp {
    private:
        int id, sal;
    public:
        Emp(): id(0), sal(0) { }
        Emp(int i, int s): id(i), sal(s) { }
        =
};
class Mgr : public Emp {
        int bonus;
    public:
        using Emp::Emp;   // inherit all ctors of base
        Mgr(int i, int s, int b): bonus(b), Emp(i, s) { }
        =
};
```

```
In main():
    Mgr m1;
    Mgr m2(1, 1000);
    Mgr m3(2, 20000, 200);
```

Page No.
Date

Raw string literals

In typical string literal escape sequences may interpret strings in undesired way. Obviously this can be fixed by escaping escape seq.

tab ↙ ↙ newline

string path = "D:\temp\new.txt"; ← undesired.

string path = "D:\\temp\\new.txt"; ← fixed

Such chars can be more complex while dealing with XML, HTML or regex.

C++11 have feature called raw string literals. In this compiler ignores any special character.

string path = R"(D:\temp\new.txt)";

To handles)" in string, we can use custom delimiter (start/end markers).

string msg = R"(Hello user "(Nilesh)" ";
↑ Considered as end of string. ← Error

string msg = R"DELIM(Hello User "(Nilesh)")DELIM";
↑ Custom delimiter - array of chars.

Cannot contain whitespaces & backslash

Lambda Functions

* Function pointer vs Function Object.

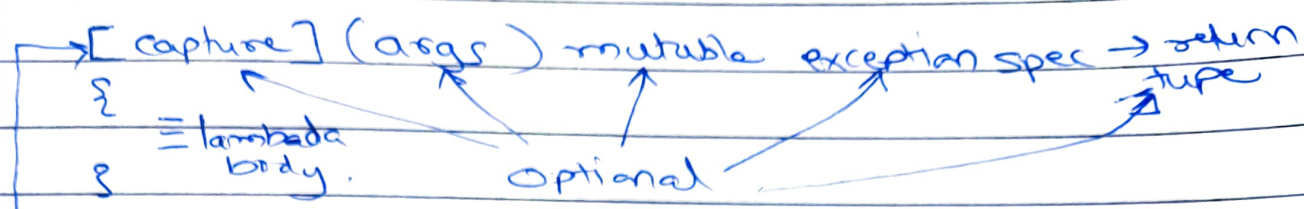
Function pointers are used to implement callback functions.

```
template <typename T, int size, typename Comparator>
void Sort(T (&a)[size], Comparator comp) {
    for(int i=0; i<size-1; i++) {
        for(int j=i+1; j<size; j++) {
            if(comp(a[j], a[i])) {
                T temp = std::move(a[i]);
                a[i] = std::move(a[j]);
                a[j] = std::move(temp);
            }
        }
    }
}
bool comp_asc(int x, int y) {
    return x < y;
}
Struct Comp_desc {
    bool operator()(int x, int y) {
        return x > y;
    }
};
In main():
    int arr[] = {1, 4, 1, 5, 3};
    Sort(arr, comp_asc);
    Comp_desc comp;
    Sort(arr, comp);
}
```

Function pointer	Function object.
① C prog feature - pointer to fn.	C++ feature - overloading operator ()
② Invoked on pointer.	Invoked on an object.
③ Dynamic	Static.
④ Can be given at run time.	Must be given at compile time.
⑤ Difficult to optimize	Easy to optimize
⑥ Slower	Faster
⑦ Cannot store state	Can store state (as object data member).

*Lambda expression

- Lambda expression is an anonymous fn object.
- It is syntactic sugar for fn object.
  - Skip defining class/struct with overloaded () operator.
  - also shorter syntax for few lines of code.
- Lamba expression can be passed as arg to fn.
- Lambda expression take args & return value like fn object.
- Internally implemented as nameless fn object.
- Can use auto keyword to name lambda fn object.




```
[]() {
    std::cout << "hello" << std::endl;
}();
```
↑ lambda call.

```
auto fn = []() { std::cout << "hi" << endl; };
fn();
```
↑ named lambda object
↑ lambda call.

```
cout << typeid(fn).name() << endl;
```
→ prints: class <lambda_unique id> on VS

```
auto sum = [](int x, int y) {
    return x + y;
};
```
← args
↑ return type is auto detected.

```
cout << sum(10,20);
```