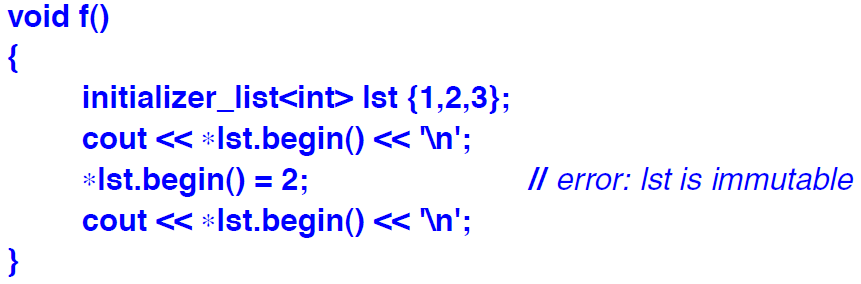
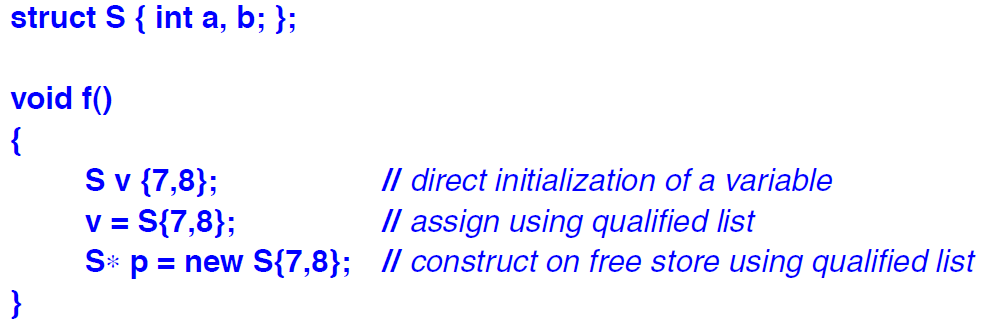
* **Logical Operators –**
* &&(and), ||(or), and !(not) take operands of arithmetic and pointer types.
* && and || operators evaluate their operands only if necessary.
* **Bitwise Logical Operators –**
* &(and), |(or), ^ (exclusive-or / xor), ~ (complement), >> (right shift), << (left shift) are applied to integral types.
* Integral types are char, short, int, long, long long and their unsigned counterparts, and bool, wchar\_t, char16\_t, char32\_t.
* A plain enum can be implicitly converted to an integer type and used as an operand to bitwise logical operations.
* **Conditional Expressions –**
* Conditional expressions are important in that they can be used in constant expressions.
* **Free Store –**
* A C++ implementation does not guarantee the presence of a garbage collector that looks out for unreferenced objects.
* Consequently, objects created by *new* are manually freed using *delete*.
* The operation *delete* can be applied only to a pointer returned by *new* or to a *nullptr*.
* Applying *delete* to *nullptr* has no effect.
* If the deleted object is of a class with a destructor, that destructor is called by *delete* before the object’s memory is released for reuse.
* **Memory Management –**
* The main problems with free store are –
* ***Leaked Objects:*** People use new and then forget to delete the allocated object. This can cause a program to run out of space.
* ***Premature Deletion:*** People delete an object that they have some other pointer to and then later use that other pointer. The pointer to the deleted object no longer points to a valid object. Reading it may give bad results. Writing to it may corrupt an unrelated object.
* ***Double Deletion:*** An object is deleted twice, invoking its destructor (if any) twice. By the second delete, the memory pointed to by another object may have been deleted, causing disastrous results.
* There are two approaches to resource management to avoid the problems of free store –
* Prefer scoped variables to free store objects.
* Use standard-library containers such as string, vector, unique\_ptr, shared\_ptr. This rule is referred to as RAII (Resource Acquisition Is Initialisation).
* **Getting memory space –**
* The free store operators *new*, *delete*, *new[]*, *delete[]* have their definitions in the *<new>* header.
* **List Implementation Model –**
* Having a {}-list be immutable implies that the container taking elements from it must use a copy operation rather than a move operation.



* The lifetime of a {}-list is determined by the scope in which it is used.
* **Qualified Lists –**



* A {}-list is the simplest way of dealing with homogenous lists of varying lengths.
* **Lambda Expressions:** Simplified notation for defining a using an anonymous function object.
* The lambda expressions consist of a sequence of parts –
* A possibly empty *capture list*, delimited by *[]*. Specifying what names from the definition environment can be used in the lambda expression’s body, and whether those are copied or accessed by reference.
* An optional *parameter list*, delimited by *()*. Specifies what arguments the lambda expression requires.
* An optional *mutable* specifier. Indicates that the lambda expression’s body may modify the copies of variables captured by value.
* An optional *noexcept* specifier.
* An optional *return type* declaration of the form *->* type.
* A *body*, delimited by *{}*. Specifies the code to be executed.
* A lambda introducer can take various forms –
* [] is an empty capture list. This implies that no local names from the surrounding context can be used in the lambda body. For such lambda expressions, data is obtained from arguments or from non-local variables.
* [&]: implicitly capture by reference. All local variables are accessed by reference.
* [=]: implicitly capture by value. All names refer to the copies of the local variables taken at the point of call of the lambda expression.
* [capture list]: List of names of local variables to be captured (i.e. stored in the object) by reference or by value. A capture list can also contain this and names followed by … as elements.
* [&, capture list]: Implicitly capture by reference all local variables with names not mentioned in the list. List name can contain this. List names cannot be preceded by &.
* [=, capture list]: Implicitly capture by value all local variables with names not mentioned in the list. List cannot contain this. List names must be preceded by &.
* The choice between capturing by value and by reference is the same as the choice for function arguments.
* When passing a lambda to another thread, capturing by value is typically best. This will avoid accessing another thread’s stack through a reference or a pointer.
* Here are a few things to note about ***Lambda expressions*** –
* If the lambda expression does not take any arguments, the arguments list can be omitted. Thus, the minimal lambda expression is []{}.
* A lambda expression’s return type can be deduced from its body.
* If a lambda body does not contain any return statement, the lambda’s return type is void.
* If the lambda body consists of just a single return statement, the lambda’s return type is the type of return’s expression.
* If neither is the case, we have to explicitly supply a return type.
* **Explicit Type Conversions –**
* C++ offers explicit type conversion operations of varying convenience and safety –
* Construction, using the {} notation, providing type-safe conversion of new values.
* Named conversions, providing conversions of various degrees of nastiness –
* ***const\_cast*** for getting write access to something declared ***const***.
* ***static\_cast*** for reversing a well-defined implicit conversion.
* ***reinterpret\_cast*** for changing the meaning of bit patterns.
* ***dynamic\_cast*** for dynamically checked class hierarchy navigation.
* C-style casts, providing any of the named conversions and some combination of those.
* Functional notation, providing a different notation for C-style casts.
* **Avice –**
* Prefer prefix ***++*** over suffix ***++***.
* Use resource handles to avoid leaks, premature deletion and double deletion.
* Don’t put objects on the free store if you don’t have to; prefer scoped variables.
* Avoid naked ‘*new*’ and naked ‘*delete*’.
* Use RAII (Resource Acquisition Is Initialisation).
* Prefer a named function object to a lambda if the operation requires comments.
* Prefer a named function object to a lambda if the operation is generally useful.
* Keep lambdas short.
* For maintainability and correctness, be careful about capture by reference.
* Let the compiler deduce the return type of a lambda.
* Use the *T{e}* notation for construction.
* Avoid explicit type conversion (casts).
* When explicit type conversion is necessary, prefer a named cast.
* Consider using a run-time checked cast, such as *narrow\_cast<>()*, for conversion between numeric types.