CSCI251: Advanced Programming

Some loose ends: `const`, `constexpr`,
type deduction (`auto`, `decltype`), class boilerplate, templates, lambda expression, etc.
based on Bjarne Stroustrup's A Tour of C++;
Lippman, Lajoie & Moo's C++ Primer

Spring 2023

Outline of Topics

Qualifiers - `const`, `constexpr`

A `const` is used to qualify an object whose value we do not intend to change.

Prefer to use `const` instead of `#define` to indicate a constant value

```
const int bufSize = 1024; // ok and preferred
#define bufSize 1024; // not recommended
```

```
const int shoeSiz = get_size();//ok: run-time initialization
const int numBrries = 43; //ok:compile-time initialization
const hValue; //error: hValue is uninitialized const
```

A `const` object can only be used in operations that do not change its value:

- initialization

```
int i = 42;
const int ci = i; //value of i copied into ci
const j = ci; //value of ci copied into j
```

Qualifiers - `const`, `constexpr`

To share a `const` object among multiple files, you must define the variable as `extern`

```
//file_1.hpp
#ifndef FILE_1_HPP
#define FILE_1_HPP
extern const int bufSize; // same bufSize defined in file_1.cpp
...
#endif /* FILE_1_HPP */
```

```
//file_1.cpp
...
//fcn() is some function evaluated at compile time
extern const int bufSize = fcn(); // define and initialize
                                  // bufSize and accessible
                                  // to other other files
...
```

Qualifiers - `const`, `constexpr`

Reference to a `const` cannot be used to change the object to which the reference is bound.

```
const int cInt = 876;
const int &refInt = cInt; // ok: both reference
                           // and underlying
                          // object are const
refInt = 67;              // error: refInt is a
                          // reference to const
int &refInt2 = cInt;      // error: non-const
                          // reference to a const object
```

We cannot assign directly to `cInt`, hence we should not be able to use a reference to change `cInt`. Therefore, initialization of `refInt2` is an error.

Qualifiers - `const`, `constexpr`

Binding a reference to `const` to an object says nothing about whether the underlying object itself is `const`.

```
int iVar = 42;
int &refVar1 = iVar; // refVar1 bound to iVar
const int &refVar2 = iVar; //refVar2 also bound to iVar
                           //but cannot be used to change
                           // iVar
refVar1 = 60; // refVar1 is not const; iVar is updated
refVar2 = 60; // error: refVar2 is a reference to const
```

Qualifiers - `const`, `constexpr`

1. Pointer to `const` may not be used to change the object to which the pointer points.
2. We may store the address of a `const` object only in a pointer to `const`.

```
const double pi = 3.142;// pi is const; its value may
                         // not be changed
double *ptr = &pi; // error: ptr is a plain pointer
const double *cptr = &pi;//ok: cptr may point to a
                             // double that is const
*cptr = 256;             // error: cannot assign to *cptr
double epsilon = 1.0e10; //epsilon is double and value
                              // can change
cptr = &epsilon; //ok: but cannot change the value
                 // of epsilon
```

Qualifiers - `const`, `constexpr`

`constexpr`:

A constant expression is one whose value cannot change and that can be evaluated at **compile time**.

A literal is a constant expression

A `const` object initialized from a constant expression is also a constant expression

```
const int maxFiles = 150;//maxFiles is a constant expression
const int limit = maxFiles + 1; //limit is a constant expression
int staffSz = 27;// staffSz is not a constant expression
const int sz = getSize();// sz is not a constant expression;
```

`staffSz` is initialized from a literal but is it not a constant expression because is a plain `int` variable, not a `const int`.

`sz` is a `const` variable but the initializer is not known until run time.

If `getSize` is declared `constexpr`, `sz` becomes constant expression.

Qualifiers - const, constexpr

C++11 standard allows the declaration of a variable as constexpr and this indicates to the comipler to verify that it is a constant expression.

```
constexpr double licenseFee = 15.75;//15.75; constant expression
constexpr double limitFee = licenseFee + 2.5;// licenseFee + 1 is
                                            // constant expression
constexpr int sz = size();// ok only if size() is
                        // constexpr function
constexpr int newSize(){return 76;}// constexpr function
constexpr int someVar = newSize(); //ok newSize is constexpr
```

constexpr function

A constexpr function must satisfy the following:

1. return type and the type of each parameter must be a literal type.
2. The body must contain exactly one return statement.

```
constexpr int newSize(){return 76;}
```

is an example of constexpr function.

Qualifiers - `const`, `constexpr`

Passing parameter by reference ...

- We pass parameter by reference to functions to avoid copying variables. But, the variables can be modified inside the function. How to prevent this situation?
- We make the function argument `const` reference.

```
#include <iostream>
#include <string>
#include<vector>
int addNum(const int &v1, const int &v2);
std::string putInAString(const std::vector<string> &strings);

int main(){
int num1 = 9, num2 = 10;
std::vector<string> niceWords{"Computer",
"Science", "rocks"};
int sum = addNum(num1, num2);
std::string slogan = putInAString(niceWords);

std::cout << "Say " << slogan <<  " "
            << sum " times" << std::endl;
}
```

auto, decltype

auto

If a variable is initialised when it is declared, in such a way that its type is unambiguous, the keyword auto can be used to declare its type.

The type of item is deduced from the type of the result of adding `val1` and `val2`

```
// item initialized to the result of val1 + val2
auto item = val1 + val2;

auto i = 0, *p = &i;//ok: i is int and p is a pointer to int
auto sz = 0, pi = 3.14;//error: inconsistent types for sz and pi
```

Note:

Type inferred by compiler for `auto` is not always exactly the same as the initializer's type. Rather, compiler adjusts the type to conform to normal initialization rules. Example below: reference is used as initializer; hence initializer is the corresponding object (here an `int`)

```
int i = 0, &r = i;
auto a = r; // a is an int (r is an alias for i, which has type int)
```

auto

Note:

- auto will ordinarily ignore top-level consts.
- Expectedly in initializations, low-level consts are kept for example when an initializer is a pointer to const, :

```
const int ci = i, &cr = ci;
auto b = ci; //b is an int (top-level const in ci is dropped)
auto c = cr; //c is an int (cr is alias for ci whose const is top-level)
auto d = &i; //d is an int* (& of an int object is int*)
auto e = &ci; //e is const int* (& of a const object is low-level const)
```

If we want the deduced type to have a top-level const, we must say so explicitly (e.g.):

```
const auto f = ci; // deduced type of ci is int ; f has type const int
```

decltype

decltype

- The keyword `decltype` can be used to say “same type as that one”
- `decltype` returns the type of its operand. The compiler analyzes the expression to determine its type but does not evaluate the expression, e.g.

```
// sum has whatever type f returns
     decltype(f()) sum = x;
```

Note:

- `decltype` handles top-level `const` and references somewhat differently from the way `auto` does.
- When applied to an expression that is variable, `decltype` returns the type of the variable, including top-level-const and references

```
const int ci = 0, &cj = ci;
decltype(ci) x = 0; //x has type const int
decltype(cj) y = x; //y has type const int& and is bound to x
decltype(cj)  z; // error: z is a reference and must be initialized
```

decltype

Note

- When we applied to an expression that is not a variable, we get the type yielded by the expression.

```
// decltype of an expression can be a reference type
int i = 42, *p = &i, &r = i;
decltype(r + 0) b; // ok: addition yields an int ; b is int
decltype(*p) c; // error: c is int& and must be initialized
```

Note:

- `decltype((variable))` (note, double parentheses) is always a reference type, but `decltype(variable)` is a reference type only if variable is a reference.

```
// decltype of a parenthesized variable is always a reference
decltype((i)) d; // error: d is int& and must be initialized
decltype(i) e; // ok: e is an (uninitialized) int
```

Classes - key ideas

Key ideas of a `class`

1. Data abstraction: a programming and design technique of separating interface and implementation.
2. Interface of the `class` are the operations made available to users of the `class` to execute.
3. Implementation includes data member and bodies of functions (those of the interface and other helper functions not exposed to the user).
4. Encapsulation ensures the separation of interface and implementation; users see the interface but not the implementation.
5. If a `class` uses data abstraction and encapsulation, it defines an Abstract Data Type (ADT).

Classes - key ideas

Some questions for class designers

1. What behaviours do I need?
2. Do I need to overload operators?
3. Are the compiler defaults correct?
4. What user-defined conversions do I need?

The rule of 3/5 applies to constructors.

Resouce Allocation Is Initialization (RAII).

Classes define constructors to control what happens when objects of the class type are initialized.

Classes control what happens when objects are copied, assigned, moved, and destroyed.

Boilerplate approach

A boilerplate approach may make class design consistent and robust.

Class design boilerplate

```
#ifndef Class_NameH
#define Class_NameH
Class Class_Name{
private:
// private data and functions here ...
    void copy(const Class_Name &); //copy object
    void free();                //free object
public:
//Constructors and destructor
    Class_Name();    //default constructor
    Class_Name(const Class_Name & obj){// copy constructor
        copy(); // copy object
    }
// other constructors here
~Class_Name(){free();} // destructor
//Modifiers
Class_Name & operator= (const Class_Name & obj){//asssignment
    if(this != &obj){free(); copy(obj);} //free resources
    return *this; //copy object
}
//other member functions ... selectors, etc, here ...
// const member functions here ...
};
#endif /* Class_NameH */
```

Class design boilerplate

Class Design Checklist

- Initialization
- Copy initialization
- Assignment
- Destruction
- Conversions
- Comparison operators
- Arithmetic operators
- Input and output
- Subscritping, iteration
- Nonmember or member function impelementation

Class design boilerplate

Constructor

A constructor that supplies default arguments for all its parameters also defines the default constructor.

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
class SalesData {
 std::string bookNo;          // uninitialized
 unsigned unitsSold = 0;      // initialized
 double unitPrice = 0.0;      // initialized
 double revenue = 0.0;        // initialized
 public:
 // defines the default constructor as
//well as one that takes a string argument
SalesData(std::string s = ""): bookNo(s) { }
// other constructors
SalesData(std::string s, unsigned cnt,
      double price):  bookNo(s), unitsSold(cnt),
            unitPrice(price), revenue(price*cnt) { }
};
#endif
```

Class design boilerplate

Constructor - with `const` & `ref` data members

- By the time the body of the constructor begins executing, initialization is complete.
- Our only chance to initialize `const` or reference data members is in the constructor initializer

```
class ConstRef {
public:
    ConstRef(int ii);
private:
    int i;
    const int ci;
    int &ri;
};
```

```
// error: ci and ri must be initialized
ConstRef::ConstRef(int ii){
 // assignments:
i = ii; // ok
ci = ii; // error: cannot assign to a const
ri = i; // error: ri was never initialized
}
```

```
// ok: explicitly init ref & const members
ConstRef::ConstRef(int ii): i(ii),
                  ci(ii), ri(i) {}
```

Class design boilerplate

Copy Constructor

A constructor is the copy constructor if its first parameter is a reference to the class type and any additional parameters have default values. Example ⇒

```
#ifndef SALES_DATA_H
#define SALES_DATA_H
#include <string>
class SalesData {
 std::string bookNo;           // uninitialized
 unsigned unitsSold = 0;       // initialized
 double unitPrice = 0.0;       // initialized
 double revenue = 0.0;         // initialized
 public:
 SalesData(std::string s = ""): bookNo(s) { }
// copy constructor
SalesData(const SalesData &item):
 bookNo(item.bookNo),unitsSold(item.unitsSold),
 unitPrice(item.unitPrice),
 revenue(unitPrice*unitsSold) { }
};
#endif
```

Class design boilerplate

Assignment : `operator=()`

- Assignment is NOT initialization and never calls constructors.
- Initialization happens when a variable is given a value when it is created.
- Assignment obliterates an object's current value and replaces that value with a new one.

```
class HasPtr{
public:
HasPtr(const std::string &s = std::string()):
ps(new std::string(s)), i(0) { }
// each HasPtr has its own copy of the string to which ps points
HasPtr(const HasPtr &p): ps(new std::string(*p.ps)), i(p.i) { }
HasPtr& operator=(const HasPtr &);
~HasPtr() { delete ps; }
private:
    std::string *ps;
    int     i;
};
HasPtr& HasPtr::operator=(const HasPtr &rhs){
auto newp = new string(*rhs.ps); // copy the underlying string
delete ps;// free the old memory
ps = newp;// copy data from rhs into this object
i = rhs.i;
return *this;}// return this object
```

Class design boilerplate

Move constructor : `ClassName(ClassName && )noexcept`

If a class has a move constructor, then copy initialization sometimes uses the move constructor instead of the copy constructor.

```
class HasPtr{
public:
HasPtr(const std::string &s = std::string()):
  ...
ps(new std::string(s)), i(0) { }
// move constructor
HasPtr(const HasPtr &&p)noexecpt
: ps(p.ps), i(p.i) {
p = nullptr;
i = 0;}
private:
    std::string *ps;
    int     i;
};
```

Class design boilerplate

Move Assignment : `ClassName &operator=(ClassName &&rhs)noexcept`

```
class HasPtr{
public:
HasPtr(const std::string &s = std::string()):
ps(new std::string(s)), i(0) { }
// each HasPtr has its own copy of the string to which ps points
HasPtr(const HasPtr &p): ps(new std::string(*p.ps)), i(p.i) { }
HasPtr& operator=(const HasPtr &&) noexcept;
~HasPtr() { delete ps; }
private:
    std::string *ps;
    int     i;
};
HasPtr& HasPtr::operator=(const HasPtr &&rhs) noexcept{ //move assignment
  if(this!= &rhs){
    delete ps;
    ps = rhs.ps;
    i = rhs.i
    rhs.ps = nullptr;
    rhs.i = 0;
  }
  return *this;}// return this object
```

Practice 1

Templates

Templates

- A template is a class or a function that is parameterized with a set of types or values.
- It is a way of generalizing an idea/concept and from which we can generate specific types or functions through argument specification.
- For example a `vector<T>` is a class template that can be specialized by specifying the type of `T`, e.g. `double, int, string`, etc.

Templates

Consider the template for a class `Data`:

```
template<typename T>
class Data {
private:
    T* elem; // elem points to an array of sz elements of type T
    int sz;
public:
    explicit Data(int s); // constructor: establish invariant
                          //, acquire resources
    ~Data() { delete[] elem; } // destructor: release resources

// ... copy and move operations ...

    T& operator[](int i);// subscripting for non-cost Data
    const T& operator[](int i) const;// for const Data
    int size() const { return sz; }
};
```

When a type is supplied for `T`, we specialize `Data`

Templates

We can define our various `Data` as follows:

```
Data<char> vc(200); // data of 200 characters
Data<string> vs(17);  // data of 17 strings
Data<list<int>> vli(45);  // data of 45 lists of integers
```

Templates - value template arguments

A template can take value arguments in addition to type arguments (value argument must be `constexpr`):

```
template<typename T, int N>
struct Buffer {
using value_type = T;
constexpr int size() { return N; }
T[N];
// ...
};
```

The alias (value_type) and the constexpr function are provided to allow users (read-only) access to the template arguments.

One usefulness of value arguments; `Buffer` is used to create arbitrarily sized buffers without use of dynamic memory:

```
Buffer<char,1024> glob; // global buffer
                       //of characters (statically allocated)
void fct(){
Buffer<int,10> buf; // local buffer of integers (on the stack)
// ...
}
```

Templates - parameterized operations

Three ways of expressing an operation parameterized by types or values:

- A function template
- A function object: an object that can carry data and be called like a function
- A lambda expression: a shorthand notation for a function object

Function template that calculates the sum of the element values of any sequence that a range-for can traverse:

```
template<typename Sequence, typename Value>
Value sum(const Sequence& s, Value v){
    for (auto x : s)
    v+=x;
    return v;
}
```

`Value` template argument and the function argument v allows the caller to specify the type and initial value of the accumulator →

Templates - parameterized operations

Example user call of template function `sum()`:

```
void user(vector<int>& vi, list<double>& ld,
         vector<complex<double>>& vc){
int x = sum(vi,0); // the sum of a vector of ints (add ints)
double d = sum(vi,0.0); // the sum of a vector
                        //of ints (add doubles)
double dd = sum(ld,0.0); // the sum of a list of doubles
auto z = sum(vc,complex{0.0,0.0}); // the sum of a vector of
                                   //complex<double>s
}
```

Templates: function object (also called a functor)

Example functor - `Less_than`

```
template<typename T>
class Less_than {
const T val; // value to compare against
public:
    Less_than(const T& v) :val{v} { }
    bool operator()(const T& x) const{ // call operator
        return x<val;
    }
};
```

Templates: function object (also called a functor)

Examples of named variable of type `Less_than for some argument type`

```
Less_than lti {42};// lti(i) compares i to 42 using < (i<42)
Less_than lts {"Backus"}; // lts(s) compare
                          // s to "Backus" using < (s<"Backus")
```

Example useage in calls

```
void fct(int n, const string & s){
    bool b1 = lti(n); // true if n<42
    bool b2 = lts(s); // true if s<"Backus"
    // ...
}
```

Templates: function object (also called a functor)

Consider a function template to count elements in a sequence

```
template<typename C, typename P>
int count(const C& c, P pred){
    int cnt = 0;
    for (const auto& x : c)
        if (pred(x))
            ++cnt;
    return cnt;
}
```

The template function `count()` can be used with a predicate in conjunction with a container

Predicate is something that we can invoke to return true or false.

Templates: function object (also called a functor)

Example usage in a call

```
void multiCount(const vector<int>& vec, const list<string>& lst,
                int x, const string& s){

    cout << "number of values less than " << x << ": "
    << count(vec, Less_than{x}) << '\n';
    cout << "number of values less than "
    << s << ": " << count(lst, Less_than{s}) << '\n';
}
```

Function `multiCount()` takes a `vector<int>`, `list<string>` and comparison variables `int` and `string`

Passes these parameters to `count()` along with the predicate `Less_than`

Output will be the number of items in `vector<int>`, `list<string>` satisfying the predicate `Less_than` for the given parameter (`int` or `string`)

Function object - Lambda expressions

Notions:

- An object or expression is callable if we can apply the call `operator()` to it. That is, if `e` is a callable we can write `e(args)`.
- We can pass any kind of callable object to an algorithm.
- We already saw functors as callable objects.
- Lambda expression is a callable unit of code.

The form of a lambda expression is:

```
[capture list](parameter list) mutable -> return type { function body }
```

Optional keyword mutable allows variables captured by value to be changed inside the lambda function.

Return type is optional if there is one return statement; unlike ordinary functions, a lambda must use a trailing return to specify its return type.

Function parameters list is optional if empty

We must always include the capture list and function body:

Function object - Lambda expressions

Examples of lambda expressions

```
[ ](int a, int b)->bool{return a>b;}
[=](int a)->bool{return a>somevar;}
[&](int a){somevar += a;}
[=,&somevar](int a){somevar+=max(a,othervar);}
[a,&b]{f(a,b);}
```

```
[]          // Capture  nothing
[=]         // Capture  all by value (copy)
[=,&x]      //Capture all by value, except x by reference
[&]         // Capture all by reference
[&,x]       //Capture  all by reference, except x by value
```

Function object - Lambda expressions; more examples

Example (Using lambda expression as predicate to `stable_sort`)

- We want to sort words by size, but maintain alphabetical order for words of the same size
- We use `stable_sort` along with a lambda as follows:

```
    stable_sort(words.begin(), words.end(),
    [](const string &a, const string &b)
    { return a.size() < b.size();});
```

- When `stable_sort` needs to compare two elements, it will call the given lambda expression.

Function object - Lambda expressions; more examples

Example (Using the capture list)

- How do we pass the value of a variable `sz` in a function to a lambda expression.
- A lambda expression may use a variable local to its surrounding function only if the lambda captures that variable in its capture list.
- Say we need to find the first element whose size is at least as big as `sz` using the `find_if()` algorithm:

```
    ....
    std::size_t sz = 7;
    std::string words;
    ...
    auto wc = find_if(words.begin(), words.end(),
                [sz](const string &a){ return a.size() >= sz; });
```

- `find_if` returns an iterator to the first element that is at least as long as the given `sz`, or a copy of `words.end()` if no such element exists.

Function object - Lambda expressions; more examples

Example (Can we use variables or function not captured or part of parameter list?)

- We illustrate with `for_each` algorithm:

```
// print words of the given size or longer,
// each one followed by a space
// wc is the iterator returned from find_if()
// in previous example

for_each(wc, words.end(),
[](const string &s){cout << s << " ";});
cout << endl;
```

- The capture list is used for local nonstatic variables only; lambdas can use local `static`s and variables declared outside the function directly; hence `cout` from `#include <iostream>`.