CSCI251: Advanced Programming

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

Outline of Topics

1. **Qualifiers :** `const, constexpr`
2. **Type deduction -** `auto, decltype`

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 in some cases
```

```
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
```

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
...
```

Practice 1

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.

Practice 2

```
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**.

1. A literal is a constant expression
2. 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+2.5 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;}
```

Why Compile-time evaluation?

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);


int main(){
int num1 = 9, num2 = 10;
int sum = addNum(num1, num2);
}
```

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 `const`s.
- Expectedly in initializations, low-level `const`s 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
```

Practice 3

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
```

Practice 4