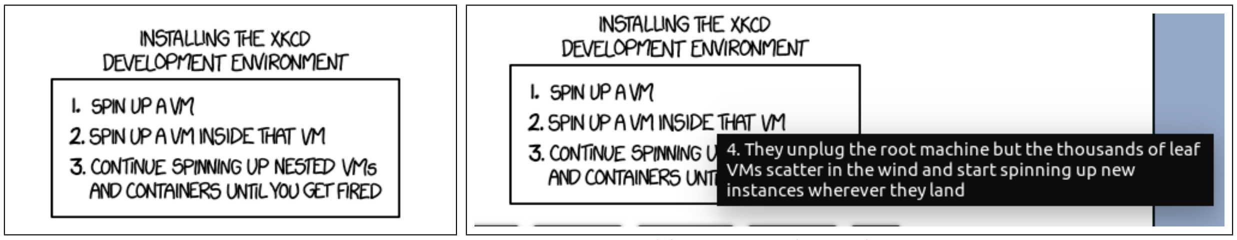
ESS201: Programming-II
Module: C++

Jaya Sreevalsan Nair
jnair@iiitb.ac.in
International Institute of Information Technology, Bangalore

Term I: 2018-19; Lecture-05 on October 29, 2018


INSTALLING THE XKCD DEVELOPMENT ENVIRONMENT
1. SPIN UP A VM
2. SPIN UP A VM INSIDE THAT VM
3. CONTINUE SPINNING UP NESTED VMs AND CONTAINERS UNTIL YOU GET FIRED
4. They unplug the root machine but the thousands of leaf VMs scatter in the wind and start spinning up new instances wherever they land
Image courtesy: https://xkcd.com/1764/

1 Composition – Objects within Objects

Use of objects within classes is referred to as *Composition*, broadly. Composition is not specific/unique to C++, as it is found in all compound data types, e.g. C `struct` or `union`. In C++, one must, however, know the restrictions that come in during initializations.

Composition, similar to other object-oriented concepts, such as *association* and *aggregation*, entail inter-class relationships. These inter-class relationships encapsulate some of the relationships, such as, *part-of*, *has-a*, *uses-a*, *depends-on*, and *member-of*[1]. These are different from the *is-a* relationship, which is covered in the concept of inheritance.

Comparing the three inter-class relationships, namely composition, association, and aggregation, in C++[2]:

- **Composition**: the object and its member have a strong part-whole relationship. While the part does not know about the whole, the whole knows of the part – which is a unidirectional relationship. e.g. x, y, and z components/coordinates in a `vector3d` class. This also means that the member does not exist outside of the object. Thus, the relationship involved here is "part-of" relationship, e.g. a CPU is "part-of" a computer.

- **Aggregation**: It is a weaker form of composition, where the part can belong to multiple objects. Just as composition, aggregation is a "has-a" relationship. However, different from composition, the part can exist without the whole. e.g. a company-employee relationship (with a simplification that the employee belongs to a single company at a time). In a composition, we typically add our parts

[1]. For example: a square "is-a" shape. A car "has-a" steering wheel. A computer programmer "uses-a" keyboard. A flower "depends-on" a bee for pollination. A student is a "member-of" a class. A body organ, e.g. brain or heart, exists as "part-of" a human being.

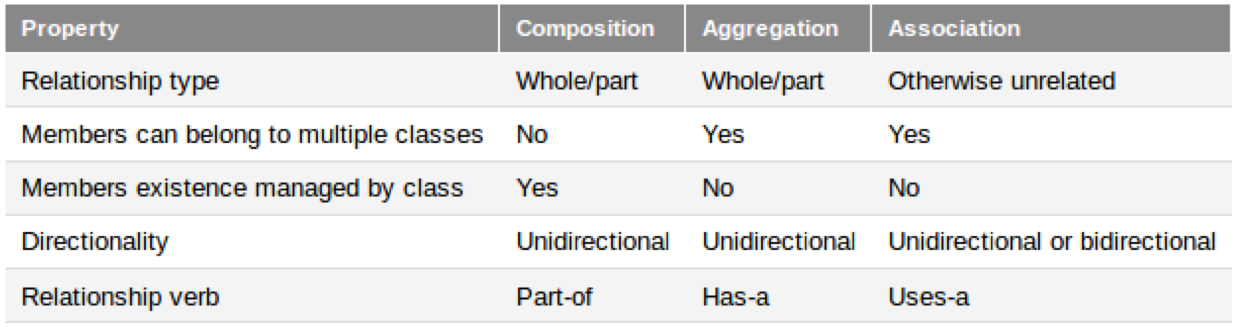
[2]Reference: https://www.learncpp.com/cpp-tutorial/10-1-object-relationships/

to the composition using normal member variables (or pointers where the allocation and deallocation process is handled by the composition class). In an aggregation, we also add parts as member variables. However, these member variables are typically either references or pointers that are used to point at objects that have been created outside the scope of the class. Consequently, an aggregation usually either takes the objects it is going to point to as constructor parameters, or it begins empty and the subobjects are added later via access functions or operators.

- **Association**: It is a weaker form of aggregation. Unlike a composition or aggregation, where the part is a part of the whole object, in an association, the associated object is otherwise unrelated to the object. The relationship can be uni- or bi-directional (i.e. the member and the class can be unaware about each other). e.g. Doctor-patient relationship. A doctor can see many patients, and a patient can consult as many doctors as needed. Association, thus, is "uses-a" relationship. *Reflexive* association is where both entities are of the same type. e.g. an academic course has a set of other courses as pre-requisites.

  In C++, associations are a relationship between two classes at the class level. That is, one class keeps a direct or indirect "link" to the associated class as a member. For example, a `Doctor` class has an array of pointers to its instantiations of the `Patient` class, as a member. Dependencies are *different* from associations. Dependencies typically are not represented at the class level – that is, the object being depended on is not linked as a member. Rather, the object being depended on is typically instantiated as needed (like opening a file to write data to), or passed into a function as a parameter (like `std::ostream` in the overloaded operator ≪).

Figure 1 gives the relationship between inter-class relationships.



Property	Composition	Aggregation	Association
Relationship type	Whole/part	Whole/part	Otherwise unrelated
Members can belong to multiple classes	No	Yes	Yes
Members existence managed by class	Yes	No	No
Directionality	Unidirectional	Unidirectional	Unidirectional or bidirectional
Relationship verb	Part-of	Has-a	Uses-a

Figure 1: Summary of composition, aggregation, and association. Image courtesy: https://www.learncpp.com/cpp-tutorial/10-4-association/

Member Initialization During Composition: Member initialization always occurs when objects are composed in classes: if no constructors are mentioned in the member initializer list the default constructors of the objects are called. However, this only holds true for *objects*. The implication is that the data members of *primitive data types* (e.g. `int` or `double`, etc.) are not initialized automatically.

The order in which class type data members are initialized is defined by the order in which those members are defined in the composing class interface. If the order of the initialization in the constructor differs from the order in the class interface, the compiler reorders the initialization so as to match the order of the class interface.

2 Example Codes and Practicals

`composition.cpp` and `composition_mod.cpp` to show different relationships, namely, association, aggregation and composition. `composition_modtest.cpp` to test the order of member initializations in classes.

3 The keyword `const`

The `const` keyword is a modifier stating that the value of a variable or of an argument may not be modified.

Not new in C++, `const` is more significantly used than in C. An example of usage:

```
int main() {
  int const ival = 3 ; // a const int
  ival = 100 ;
  // this modifies gives an error message
}
```

Applications in C++ include applying `const` to data types, or to pointers, where the latter are called `const`-pointers. Scenarios where `const` is used in C++:

- Specifying size of an array:

  ```
  int const size = 20 ;
  char buffer[size] ;
  ```

- In the declaration of pointers, use of `const` means that the object that is being *pointed to* cannot be changed; however the pointer itself can change.

  ```
  char const *buffer ; // pointer to const char

  *buffer = "A" ; // gives error
  ++buffer ;      // no error
  ```

- `const`-pointer, on the other hand, is different and cannot be changed; but the object being pointed to can be changed.

  ```
  char *const buffer ; // const pointer of type char

  *buffer = "A" ; // no error
  ++buffer ;      // gives error
  ```

  Note: we had discussed about how "references" are like `const` pointers. To drive home the point – pointers can be reassigned to point to different variables, if they are not `const` pointers. However, references need to refer to a variable, and cannot be re-aliased to another variable. Thus, they are like `const`-pointers.

- A `const`-pointer to `const` data type:

  ```
  char const *const buffer ;

  *buffer = "A" ; // gives error
  ++buffer ;      // gives error
  ```

Placement of the keywork `const`: The rule of thumb is to perform *right-to-left* parsing for declarations of variables.

- Whatever occurs to the *left* to the keyword `const` is the entity which cannot be changed. e.g. `char const buffer` implies `const char` type variable; `char *const buffer` implies `const` *, i.e. pointer of char type.

- In cases where there is nothing in the *left* to parse, `const` is applied to the variable in the *right*. e.g. `const int size` would mean `const int` type variable. This rule however does not work for pointers, as we will see soon.
- The definition or declaration (either or not containing `const`) should always be read from the variable or function identifier back to the type indentifier:

  ```
  char const *const buffer ;
  ```

  is to be read as: "`buffer` is a `const` pointer to `const char` type."

Both the following declarations are valid and are semantically the same:

```
int const val = 2 ; // Declaration - 1
const int val = 2 ; // Declaration - 2
```

Stroustrup has said that `Declaration-2` is less confusing since it reads *left-to-right*, just like the English language. However, it leads to ambiguous results in certain cases. e.g.:

```
const* int ptr = new int(2) ; // Left-to-right parsing would be:
                              // "const pointer to int"
                              // But this gives compilation error
```

List of declarations corresponding to *right-to-left* parsing:

```
int *ptr ;                // ptr is pointer to int
int const *ptr ;          // ptr is pointer to const int
int *const ptr ;          // ptr is const pointer to int
int const * const ptr ;   // ptr is const pointer to const int
int ** const ptr ;        // ptr is const pointer to a pointer to an int
int * const *ptr ;        // ptr is pointer to a const pointer to an int
int const **ptr ;         // ptr is a pointer to a pointer to a const int
int * const * const ptr ; // ptr is const pointer to const pointer to an int
```

and also:

```
const int *ptr;          // ptr is pointer to const int
const *int ptr ;         // Gives error
```

Then how does one interpret a fairly complex declaration as the following?

```
char const *(*const (*(*ip)()) []))[]
```

Identify *nesting* and interpret innermost-to-outermost.
"Recipe" for reading complex declarations:

1. We start reading at the name of the variable;
2. We read (left-to-right) as far as possible until we reach the end of the declaration or an (as yet unmatched) closing parathesis;
3. We return to the point where we started reading, and read backwards until you reach the beginning of the declaration or a matching opening paranthesis;
4. Upon reaching an opening parathesis, we continue at step 2 beyond the paranthesis where we stopped previously.

Step-by step reading of our example:

```
char const *(*const (*(*ip)()) []))[]
```

```
                          ip                      Start at the variable's name:
                                                      'ip' is
                          ip)                     Hitting a closing paren: revert
                           →
                        (*ip)                     Find the matching open paren:
                           ←                          'a pointer to'
                        (*ip)()                   The next unmatched closing paren:
                           →                           'a function (not expecting
                                                        arguments)'
                      (*(*ip)())                  Find the matching open paren:
                           ←                           'returning a pointer to'
                      (*(*ip)()[])                The next closing par:
                           →                           'an array of'
             (* const (*(*ip)()[])                Find the matching open paren:
                           ←                           'const pointers to'
             (* const (*(*ip)()[])[]              Read until the end:
                           →                           'an array of'
char const *(* const (*(*ip)()[])[]              Read backwards of what's left:
                           ←                           'pointers to const chars'
```

Thus, `char const *(* const (*(*ip)())[])[] ;` means:
"ip is a pointer to a function (not expecting arguments) returning a pointer to an array of const pointers to an array of const pointers to an array of pointers to const chars".

4 Typecasting

C-style: `(typename) expression` are deprecated, hence, C++ styles may be used instead. Note: typecasting in C++ is different from oft-used constructor notation: `typename(expression)`.

- `static_cast<type>(expression)` operator: converts "conceptually comparable or related types" to each other. Following cases exist in C++:
  - **Case 1**: `int` to `double`, and vice-versa.

```
int x = 19 ;
int y = 4 ;
int z1 = sqrt(x/y) ;
double z2 = sqrt(static_cast<double> (x)/y) ;
```

  - **Case 2**: when converting related pointers to each other, in the case of inheritance (typecasting from derived to base class), generic function pointers such as `(void *).`

```
void const *p1 ;
static_cast<int const *>(p1) ;
};
cout << static_cast<int>(VALUE); // show the numeric value
```

  - **Case 3**: When undoing or introducing the signed-modifier of an int-typed variable (e.g. `char, int`)

```
tolower(static_cast<unsigned char>(c)) ;
```

- `const_cast` operator: to undo the `const` attribute of a (pointer) type. Needed for some of the standard C library functions which may not be `const`-aware.

```
char const hello[] = "hello";
strfun(hello) ;                    // gives warning:
                                   // passing 'const char *' as argument 1
                                   // of 'fun(char *)' discards const
strfun(const_cast<char *>(hello)) ; // Works
```

5 `const` member functions and `const` objects

First encounter with const member functions – Access to the (private) data members in a class is controlled by encapsulating its accessor members. Accessors ensure that data members cannot suffer from uncontrolled modifications. Since accessors conceptually do not modify the data of the object (but only retrieve the data) these member functions are given the predicate `const`. They are called `const` member functions, as they are guaranteed not to modify the data of their object, and are available to both modifiable and constant objects.

```
class Person {
    std :: string name, address ;
public:
    std::string const &name() const;
    std::string const &address() const;
};
string const &Person::name() const {
    return _name;
}
```

Except for constructors and the destructor only `const` member functions can be used with (plain, references or pointers to) `const` objects.
`const` objects are frequently seen in the form of `const &` parameters of functions. Inside such functions only the `const` members of the object may be used.

```
void displayAddress(std::ostream &out, Person const &person) {
   out << person.name() << " resides at " << person.address() << std::endl;
}
```

`const` member function attributes can be overloaded.

```
class Members {
public:
  Members();
  void member();
  void member() const;
};

Members::Members() {}

void Members::member() {
  std::cout << "non const member" << std :: endl ;
}

void Members::member() const {
  std::cout << "const member" << std :: endl ;
}
```

When functions are overloaded by their `const` attribute, the compiler uses the member function matching most closely the `const`-qualification of the object:

- When the object is a `const` object, only `const` member functions can be used.
- When the object is not a `const` object, non-`const` member functions are used, unless only a `const` member function is available. In that case, the `const` member function is used.

General rule of thumb: member functions should always be given the const attribute, unless they are intended to modify the data of the object.