7. Example of application:

a. Assume a simply dynamic Array Class.

(this is how NOT to do it)

How NOT TO Do it

```
class Array{
  public:
     int size;
     int*data;
     explicit Array (int size) : size (size),
                                 data (new int[size]) { }
     ~Array() {
        delete [] data;
} // Array

int main(){
   Array first Array (20);
   firstArray. data [0] = 25;
(a)  {
      Array copyArray = firstArray;
      std::cout << firstArray. data [0] << " " << copyArray. data [0]
               << std::endl;
     }
(b)  firstArray. data [0] = 10;
   return 0;
} // main
```

i) output: 25 25
segmentation fault

ii) The compiler generated this code: `Array (const Array &copy) : size(copy.size), data(copy.data) { }`

iii) This is a shallow copy! of data pointer.
It only copies the address of the original data member, so 'firstArray' and 'copyArray' object contains a pointer to the same chunk of memory (bad)

$\Rightarrow$

Copy Constructor (continued)

7. (Continued)
a) (continued)
iv) In ⓐ:

```
ⓐ [ {
      Array copyArray = firstArray;
      std::cout << firstArray.data[0] << " " << copyArray.data[0] << std::endl;
    }
```

I) CopyArray's destructor gets called ~~because~~ (because objects on the stack are destroyed automatically when their scope ends)
II) Array's destructor deletes the data array of the original; therefore when it deleted copyArray's data, because they share the same pointer, it also delete firstArray's data!!!

v) In ⓑ

```
b[ firstArray.data[0] = 10
```

I) The line try to access a now invalid data and try to write in it
II) this produce a segmentation fault!!!

b) To eliminate this problem we need to perform a deep copy by writting our own copy constructor

i)
```
Array (const Array & copy) : size (copy.size),
                             data (new int [copy.size]) {
    std::copy(copy.data, copy.data + copy.size, data);
}
```

I) to use std::copy you must include algorithm: #include <algorithm>
II) we are creating a new int & copying the content to it
III) Now, copyArray's destructor only delete its own data and not firstArray's data as well
IV) This eliminate the segmentation fault!
V) std::copy the elements in the range [First, Last) into range at result (It return an iterator to the last element in the destination range)

8) Copy constructor and templates

a) A template constructor is not a user defined copy constructor

b) A user-defined, non-template copy constructor must also be provided

c) this code would not work (at least we define for different uses)

```
template <typename A> Array::Array (const A &copy): size (copy.size()),
                                                    data(new int[copy.size])
    {
      std::copy (copy.begin(), copy.end(), data);
    }
```

9) Explicit Copy Constructor

a) It is declared explicit by using explicit keyword:
b) It is used to prevent copying of objects at function calls or the copy-initialization syntax.
c) example:

$\boxed{\text{explicit X (const X\& copy\_from\_me)}}$;

10) It is common to define an assignment operator when defining a copy constructor

Use of Const with Copy Constructors

1) Only pass by non-const reference if the function will modify the parameter and it is the intention of changing the caller's copy of the data

2) Else pass by const reference

3) There is a C++ clause that says that non-const references cannot bind to temporary objects.
   a) A temporary object is an instance of an object that does not have a variable name: eg `std::string("Hello world!");`

   since we haven't give it a name, it is a temporary

   b) This is a not temporary: `std::string str("Hello world!");`

   The object's name is 'str'

   c) Example:
      i) Improperly declared function: parameter should be const reference:

Bad
```
void print_me_bad (std::string &s){
    std::cout << s << std::endl;
}
```
✗

      ii) Properly declared function: function has no intent on modifying s:

Good
```
void print_me_good (const std::string &s){
    std::cout << s << std::endl;
}
```
✓

      iii)
Bad
```
std::string hello ("Hello");
print_me_bad( hello);  //Compiles ok: hello is not temporary
print_me_bad (std::string("World")); //Compile error: temporary object;
print_me_bad ( "text"); // Compiler error: compiler wants to construct temporary
                        //   std::string from const char*
```
Good
```
print_me_good(hello);  //compile ok
print_me_good(std:: string ("text")); //compile ok
print_me_good ( "text"); //compile ok
```

24 Assignment Operator

1- The assignment operator allow to use the '=' to assign one instance to another eg:

```
MyClass c1, c2;
...
c1 = c2 // Assign c2 to c1
```

2- Different possible signatures:

```
a) MyClass & operator = (const MyClass &rhs);
b) MyClass & operator = (MyClass rhs);
c) const MyClass & operator = (const MyClass &rhs);
d) const MyClass &operator = (MyClass rhs);
e) MyClass operator = (const MyClass &rhs);
f) MyClass operator = (Myclass rhs
```

3- these signatures (2) permute both the return type and the parameter type.

4- While the return type may not be important, the choice of the parameter is critical

5- The followig signatures pass the right-hand side by non-const reference and they are not recommended:

```
a) MyClass &operator = (MyClass &rhs);
b) const MyClass &operator = (Myclass &rhs);
c) MyClass operator = (Myclass &rhs);
```

d) these signatures would make the following code to not compile:

```
MyClass c1
c1 = MyClass (4, 'b', "Hello,pepe"); // Assume this constructor exist
```

a) Because the right-hand side of this assignment expression is a temporary (un-named) object, the compile ~~prohibit~~ forbit it since c++ standard forbit to pass a temporary object through a non-const reference parameter.

6- If you donot provide the compiler will declare one implicity The implicit assignment operator does only a shadow copy (member-wise assignment of each data member) eg:

```
MyClass & MyClass :: operator = (const MyClass & rhs){ {other
  x = other.x;
  c = other.c
  s = other.c
  return *this; }
```

Assignment Operator (continued)

7- When do we need to write an assignment operator?
   a) When we require a deep copy (pointers, com object ~~[illegible]~~
   b) Everytime you write your own copy constructor.
8- Check note about "Exception Safe Code"

Exception Safe Code

1- A function which modifies some "global" state is said to be exception safe if it leaves the global state well-defined in the event of an exception that is thrown at any point during the function.
   a) Modifying a "global" state means, for example:
      i) Modify a reference parameter
      ii) Modifying the data members of its instance

2- Example: the following class wraps an array of some user-defined type
   It has two data members
   a) A pointer to the array
   b) A number of element in the array
   c) code:

```
template <typename T> class
class MyArray {
    size_t numElements;
    T *pElements;
    public:
        size_t count() const {return numElements}
        MyArray &operator = (const MyArray &rhs;}
};
```

   d) Let now assume we assign one MyArray to another:

3rd Coding

```
template <>
MyArray <T> :: operator = (const MyArray &rhs) {
    if (this != rhs){
        delete [] pElements;
        pElement = new T [rhs.numElements];
        for (size_t i = 0; i < rhs.numElements; ++i){
Problem in this line [ pElements [i] = rhs.pElements [i];]
        }
        numElements = rhs.numElements
    }
    return *this
```

e) The problem is with this line: $\boxed{\text{pElement [i] = rhs. pElements [i];}}$

i) this line could throw an exception

ii) This line invokes operator = for type T, which could be some user-defined type whose assignment operator might throw an exception. Perhaps

iii) I) An out-of-memory (std:: bad_alloc) exception

II) Some other exception that the program created

iii) What would happends if it did throw copying the 3rd element of 10 totals?

I) the stack is unwound until an appropriate handler is found

II) the state of the object what is it?

A) we've reallocated our array to hold 10 T's elements; However, we've copied only 2 elements successfully.

B) The third element failed and the remain seven never been attempted to be copied.

C) we haven't change numElement so it still holds whatever was holding before; therefore, it will not reflex the current state

D) this could affect other members that relied on numElements

E) This instance became a time-bomb

iv) This implementation of the assignment operator '=' is not exception safe;

I) There is no telling what the object state is

II) we can only assume to be unusable

III) Its violate some of its own invariants

IV) If the object is in bad state, it may not be possible to destroy the object without crashing the program. or causing MyArray to throw another exception

V) When the compiler runs destructor while unwinding the stack to search for the handler.
If an exception is throw while unwinding the stack, the program stop and terminate

28 std::string x = "hello"

x.c_str(); - Returns $\overset{\text{const}}{\text{char}}$ * w/ a concluded NULL ['h','e','l','l','o','\0']

x.data(); - Returns $\overset{\text{const}}{\text{char}}$ * w/out promise to conclude w/ null

&x[0]: ~~even do data yield const char~~

same as data() but yields a non-constant char

Const Correctness

1. The main purpose of the keyword 'const' is to provide documentation and prevent programming mistake.

2. The keyword 'const' allow to specify whether, for example, a variable is modifiable or not.

3. It provide guarantees to you users that your program would not modify your uses' data.

4. It is possible to locate the keyword 'const' before or after the type of variable, when declaring a const variable e.g.

```
int const x = 2;
```
or
```
const int x = 2;
```

   a. x is a constant integer
   b. The value of the variable in this instance is specified in the declaration, There is no way to set it later
   c.

5. Const References: usefull to declare reference parameters to functions as const reference.

```
bool verifyObjectCorrectness (const myObj &obj);
```

   a. Lets assume verifyObjectCorrectness is suppose to make sure that the object is in a valid state.

   b. myObj object is passed by reference into verifyObjectCorrectness

   c. 'const' is used to ensure that verifyObjectCorrectness cannot change the object

   d. By declaring the argument 'const', users of the function can be assured that their objects will not be changed and not need to worry about the possible side effect of making the function call

   e. When you declare const reference, you are only making the data referenced to 'const'

Exception safe Code (continued)

3- How to write an exception safe assignment operator?

a) write an exception safe assignment operator via the copy-swap idiom

b) the copy-swap idiom is a two-step algorithm:

I) First make a copy

II) then swap w/ the copy

III) example code:

```
template <>
MyArray <T> :: operator = (const MyArray &rhs) {
    // copy the right-hand side
    MyArray tmp(rhs);
    // swap data members w/ the temporary
    std:: swap (numElement, tmp. numElements);
    std:: swap ( pElement , tmp. pElements);
    return *this;
}
```

IV) your swap always must succeed so be careful if you overload it.

c) By changing the signature, we can let the compiler make a copy for us:

```
template <>
MyArray <T> :: operator = (MyArray tmp) {
    std::swap (numElements , tmp. numElements);
    std:: swap ( pElements, tmp. pElements);
    return *this;
}
```

6. Const Pointers:

a. When using reference, by their nature, it cannot be changed what they '~~reference~~ refer to'

b. Pointers, on the other hand, have two ways that can be used:
   i) you can change the data pointed to, or
   ii) Change the pointer to

c. therefore, there are two ways of declaring a const pointer:
   i) One that prevents you from changing what is pointed to
   ii) One that prevents you from changing the data pointed to

d. Declaring a pointer to constant data:

before*

```
const int *p_int;
```

i) In this case, the way is that the keyword const appears before the '*'
ii) this is read as *p_int is a const int.
iii) the pointer may be changeable, but you cannot touch what p_int points to

```
 p_int++;   //ok : Change where is pointing to
*p_int = 3; //NO! : cannot change value of what is pointing to
```

iv) If you wish the address stored in the pointer itself to be const, then you have to put the keyword 'const' after the '*'

```
int x;
int * const p_int = &x;
```

```
 p_int++;    //No! it cannot be changed where is points to
*p_int = 5;  //OK! you can change the data to which is pointing to
```

Const Correctness (Continued)

7. The effects of declaring a variable to be const propagate throughout the program

8. Once an object is declared 'const', this object cannot be assigned to a non-const reference.

9. Once an object is declared 'const', we cannot use functions that are known to be capable of changing the state of the object.

10. Const Functions:

   a) When an object is enforced the const-ness, it means that you may need to state that a function should not make changes to an object. (easy in non-object-oriented code)

   b) In classes w/ methods, if you have a const object, you should not want to call methods that can change the object; therefore, you need to let know the compiler which methods can be safely called

   c) These methods are called "const functions" and are the only functions that can be called on a const object.

   d) Only member methods make sense as const methods

   e) Recall that every method receives an implicit 'this' pointer to the object

   f) Const methods can receive a const 'this' pointer

   g) To declare that a function is safe for const objects mark it as const.

```
<Return-value><class>::<member-function>(args) const {
   ...
}
```

ex:

```
int Loan::calcInterest() const {
   return loan_value * interest_rate;
}
```

i) Just because a function is declared const that doesn't prohibit non-const functions from use it while follow the following rules:
   I) const functions can always be called
   II) Non-const functions can only be called by non-const objects

j) When a const function return references or pointers to member of the class, they must also be const;

11. Const Overloading

a) Since const functions cannot return non-const reference to an object's data many times is appropriate to have both const and non-const version of the function that returns a non-const reference:

```
int &  myClass::getData(){
    return data;
}
```

I) On the other hand, you ~~may~~ do not want prevent some one using a const version of your object ~~from~~ getting the data. eg

```
myClass constDataHolder;
```

b) C++ allows you to overload based on the const-ness of a method (so you can have both const and non-const methods)
   I) The correct version will be chosen.

c) If you wish to return a non-const ~~non-reference~~ reference, you must also declare a second, const version of the method that returns a const method:

```
//Called for const object only since a non-const version also exist
const int &  myData::getData() const {
    return data;
}
```

Const Correctness (continued)

12. Const iterator

a) In order to enforce const-ness, C++ requires that const functions return only const pointers and references.

b) Iterators can be used to modify the underlying collection.

c) When an STL collection is declared const, then any iterators used over the collection must be const iterators

d) Const iterators are like normal iterators with the exception that cannot be used to modify the underlying data

e) To turn an iterator to const iterator in STL just append "const_" to the type of iterator you desire.
Example: an iterator over a vector:

```
std::vector<int> vec;
vec.push_back(3);
vec.push_back(4);
vec.push_back(5);
for( std::vector<int>::const_iterator itr = vec.begin(), end=vec.end
     itr!=end;
     ++itr){
     // Print out values
     std::cout << *itr <<std::endl;
  }
```

I) Note the use of a const iterator to iterate over a non-const collection
II) It prevents the possibility of silly programming mistakes
III) Also, it documents that we never intend to use the iterators to change the collection

13. Const cast:

1. Lets assume you have a const variable that you wish to pass into a function
   a) However, you are not sure that function doesn't declare its argument as const.
   b) Now lets assume a C library such as strlen were declare w/out using const
   c) ~~Finally lets assume that~~ If you know that you are safe in passing a const variable into a function that doesn't explicitly indicate that it will not change the data then you can use a const_cast in order to temporarily strip away the const-ness of an object
   d)

```
//Bad version of strlen which doesn't
// declare its argument const
int bad_strlen (char *x){
   return strlen (x);
}

const char *x = "abc"; //notice that the extra const
                       // is actually implicit in this
                       // declaration since string literals
                       // are constant

//cast away const-ness for our bad strlen function
bad_strlen (const_cast <char *> (x));
```

2. Const casts look like regular typecasts with the exception that they only be used for casting away constness (or volatile-ness) but not converting between type or casting down a class
   const → [const cast – remove const ness] → non-const

[non-const] → [const-cast] → [const]

Const Correctness (Continued)

14. Conceptual vs. Bitwise Constness

a) There is a misconception of efficiency related w/ optimization

b) The reason of misconception is due to the fact that:

i) it is possible to cast away the constness using const_cast

ii) There is an issue w/ classes, even const classes can be changed because of the mutable keyword.

I) mutable: . This keyword can only be applied to non-static and and non-const data of a class.

. If a data member is declared mutable, then it is legal to assign a value to this datamember from a const member function.

. mutable member-variable-declaration

. This code example will compile w/out error because m_accessCount has been declared to be mutable; and therefore can be modified by GetFlag even though GetFlag is a const member function.

```
//mutable.cpp
class X {
   public:
      bool GetFlag() const {
         m_accessCount ++;
         return m_flag;
      }
   private:
      bool int flag;
      mutable int m_accessCount;
};
```

mutable allow const function to change the field regardless of whether or not the object itself was declared as const

~~15. What is the different Between?~~

15. What's the difference between:

a) Fred const * p: This means "p points to a constant field Fred"
The Fred object cannot be changed via p.

b) Fred * const p: this means "p is a const pointer to a Fred"
You cannot change the pointer p itself,
~~and~~ but you can change the Fred object via p.

c) Fred const * const p: This means "p is a constant pointer to a constant Fred"
• You cannot change the pointer p itself,
~~and~~ you cannot change the Fred object via p

16. What does: "Fred const &x" means?

Fred const &x: this means x aliases a Fred object, but cannot be used to change that Fred object
• Example: If class Fred has a const member function called inspect(), saying x.inspect is OK
However if class Fred has a non-const member function called mutate(), saying x.mutate() would produce an error (caught by the compiler)

17. What does "Fred X &x" means?
a) const X &x } and they are both equivalent (but which one should be use?)
X const &x }
b) X const &x is old-fashioned while X const & is new-and rising

18. What does "Fred & const x" means? Does make sense?
~~bad to use~~
a) You have to read it right-to-left: "Fred & const x" means "x is a const reference to a Fred"
b) this is redundant because references are always const
c) "Fred & const x" is functionally equivalent to "Fred &x"
this only help to confuse people
~~d) the const will~~
d) therefore it is non sense to use it

Const correctness (continued)

18. **What does "const X * x" mean?**
    a) "const X * x" } same
       "X const * x"
    b) real question: is it should be used?
       i) const X * x is old fashioned
       ii) X const * x is new and rising

19. **What is the relationship between a return-by-reference member function and a const member function?**
    a) If you wish to return a member of your 'this' object by reference from an inspector method (A member function that inspect, rather than mutates, the object), you should return it using reference-to-const (that is, X const &)
       i) example code:

```
class Person {
  public:
     std::string const & name_good() const;
     // right ✓: the caller can't change the name
     std::string & name_evil() const;
     // wrong: ✗ the caller can change the name
     ....
};
void myCode()
```

20. **What's the deal with "const-overloading"?**
    1- you will have an inspector method and a mutator method w/ the same name and the same type of parameters.
       i) The only difference is that one uses 'const' and the other is not-const
       ~~class Fred {...};~~
       ~~class MyFredList {~~
       public:

⇒

20. (continued)

ii)
```
class Fred {
  ...
};
...

class MyFredList {
  public:
(Inspector method) Fred const & operator [] (unsigned index) const;
(mutable method) Fred &         operator [] (unsigned index);
mutator          ...
};
```

I) Inspector method and ~~mutable~~ mutator method always come in pairs;

II) they are commonly called subscript operators

III) when we apply the subscript operator to a MyFredList object that is non-const, the compiler will call the non-const subscript operator.
Since this one will return a normal Fred&, you can both inspect and mutate the corresponding Fred object

IV) Example: Let assume class Fred has an inspector called Fred::inspect() const and a mutator method called Fred::mutate():

```
void f(MyFredList & a) { // the MyFriendList is non-const
  // Call method that Don't change Fred at a[3] : OK
  Fred x = a[3];
  a[3]. inspect ();

  //Call methods that Do change Fred at a[3]
  Fred y;
  a [3] = y;
  a [3]. mutate ();
}
```

$\Longrightarrow$