ITEM 3.

Date:.....

Use const

I

const – a particular object should not be modified.

→ used for both static & non-static data members.

→ Pointers, can specify whether ptr is itself const, data it points to is const or neither or both.

(See program 2.*)

(1) Non-const pointer & data

```
char greeting[]="Hello";
char * p = greeting;
```

(2) Non-const pointer, const data

```
const char * p = greeting;
```

(3) const pointer, non-const data

```
Char * const p = greeting;
```

(4) const pointer, const data

```
const char * const p = greeting;
```

Date:.....

IV

```
void f1 (const Widget *pw);
void f2 (Widget const *pw);
```

→ Both are same.
f1 & f2 take a pointer to constant Widget object.

⇒ STL analogue of const T* pointer.
const_iterator.

```
std::vector<int> vec;

const std::vector<int>::iterator iter =
    vec.begin();

    *iter = 10;   // Changes what iter points to
    ++iter;       // error, iter is constant

const std::vector<int>::const_iterator citer =
    vec.begin();

    *citer = 10;  // error citer is const
    ++citer       // Change citer, fine.
```

In the above program, this is what we want, it should allow to increment the iterator, but not change its value

Date:.....

this is not possible then const hence vector uses special const for iterator const_iterator.

III Function returning constant value reduces client's error.

Example.

```
Class Rational {...};

const Rational operator * (const Rational & lhs,
    const Rational & Rhs);

Rational a, b, c;

(a*b) = c;
```

-> if the client does like this & return value is not constant, then (a*b) no matter what it returns, will be equal to c (as assignment operator will be called) so if c = 10, then (a*b) result becomes 10, and it will create problem in below statement

```
if ( a*b = c)
```

Date:.....

The above statement will always return true since we are assigning (a * b) value c, Instead of ==, if by mistake we use =, then output will be modified and result will be always true.

If we use const as return type in overloaded operator * then if we do (a * b) = c it will result in compiler error and client will not be able to make the mistake.

IV

Same function can be overloaded if it is a const.

```
class A
{ public:

    const char & operator[](std::size_t pos) const
    { return text[pos]; }

char & operator[](std::size_t pos)
    { return text[pos]; }
```

See program → ~~Non~~ ConstFunctionOverloading.cpp

Date:.....

```
A tb("Abc");                    // calls non-const
std::cout<<tb[0];               // A::operator[]

A cb("Def");                    // calls const
                                // A::operator[]
cout<<cb[0];

tb[0]='X';                      // works fine.

cb[0]='X';                      // Error : cb const
```

=> There are two type of constness
Bitwise & logical.

Bitwise constness passes even though it is not correct.

For eg:-

```
Class A {
    public:

    char & operator[] (size_t pos) const
        { return pText[pos]; }

    char * pText;
};
```

Date:.....

```
const A ab("Abc");
char *pc = &ab[0];
*pc = 'J';  // ab is now Jbc
            not correct.
```

→ In same way, if we want to use some variable but since it belongs to const, we are not able to use it.

For eg:-

```
class A {
    public : size_t length() const;

    private:
        char * ptext;
        size_t textlen;
        bool isValid;
    };

size_t A::length() const
  { if (!isValid) {

        textlen = strlen(pText);
// Error, cannot assign textlen and
   isValid in a const member function
```

Date:.....

To avoid such scenario we can use Mutable keyword.

It allows variables to be modified in a const member function.

⇒ Since we can overload a const and non-const function

This may lead to code duplication.

⇒ Code Duplication can be solved by using Cast.

we can use the non-const function to call the const one.

⇒ const-cast ⇒ which removes constant

⇒ Static-cast ⇒ adds const.

Having const member function call a non-const function is wrong since object which need not to modify would be changed.