# Const

The keyword states that the value of a variable or of an argument may not be modified.

```
int main()    {
    int const ival = 3;          // initialized to 3
    ival = 4;              // assignment leads to an error message
    return 0;
}
```

**It is possible to declare a pointer to a constant variable by using the const before the data type.**

```
const char *buf = "hello";
char const *buf = "hello";  is an alternative syntax which does the same
```

**Whatever is pointed to by buf may not be changed**

**The pointer buf itself however may be changed.**

```
int main()    {
    char const *buf = "hello";
       buf++;                // accepted by the compiler
       *buf = 'u';           // rejected by the compiler. Cannot modify via buf.
      return 0;}
```

**Const pointer**

char *const buf;

buf itself is a **const pointer** which may not be changed. Whatever chars are pointed to by buf may be changed at will.

To declare a const pointer, use the *const* keyword between the asterisk and the pointer name.

```
int nValue = 5;

int *const pnPtr = &nValue;
```

A const pointer must be initialized to a value upon declaration, and its value cannot be changed. In the above case, pnPtr will always point to the address of nValue.

However, because the value being pointed to is still non-const, it is possible to change the value being pointed to via dereferencing the pointer:

```
*pnPtr = 6; // allowed, since pnPtr points to a non-const int
```

Finally, it is possible to declare a const pointer to a const value:

```
const int nValue;

const int *const pnPtr = &nValue;
```

A const pointer to a const value can not be redirected to point to another address, nor can the value it is pointing to be changed.

## Const Reference

If the designer is to avoid modification of the passed/returned object, the reference should be made const.

```
class Backpack {
  public:
        const Book& getContents();
         void setContents(const Book& bk);
  private:
        Book myText;};
  const Book& Backpack::getContents(){
         return myText;}
  void Backpack::setContents(const Book& bk){
        myText=bk;
        Book x;
        bk=x;   //This line is now illegal and won't compile.
        }
int main(){
        Book nonfiction;
        Backpack myPack;
        myPack.getContents()=nonfiction;  // This line is now illegal and won't compile
        return 0;}
```

**If a *member* function will not modify its data members, make the *member* function const.**

```
class Backpack {
  public:
        const Book& getContents() const;
        void setContents(const Book& bk);
  private:
        Book myText;
};
```

Note that the setContents() member function is not made const because it modifies the object's data members (it changes myText).

## Const class objects and member functions

Once a const class object has been initialized via constructor, any attempt to modify the member variables of the object is disallowed, as it would violate the constness of the object. This includes both changing member variables directly (if they are public), or calling member functions that sets the value of member variables:

```
class Something   {
public:
    int m_nValue;

    Something() { m_nValue = 0; }

    void ResetValue() { m_nValue = 0; }
    void SetValue(int nValue) { m_nValue = nValue; }
    int GetValue() { return m_nValue; }   };

int main()   {
    const Something cSomething; // calls default constructor
    cSomething.m_nValue = 5; // violates const
    cSomething.ResetValue(); // violates const
    return 0;
}
```

Const member functions declared outside the class definition must specify the const keyword on both the function prototype in the class definition and on the function prototype in the code file:

```
class Something   {
public:
   int m_nValue;

   Something() { m_nValue = 0; }
   void ResetValue() { m_nValue = 0; }
   void SetValue(int nValue) { m_nValue = nValue; }
   int GetValue() const;
};

int Something::GetValue() const
{
   return m_nValue;
}
```

Any const member function that attempts to change a member variable or call

a non-const member function will cause a compiler error to occur.

```
class Something  {
public:
   int m_nValue;

   void ResetValue() const { m_nValue = 0; }
};
```

ResetValue() has been marked as a const member function, but it attempts to change m_nValue. This will cause a compiler error.

Note that constructors should not be marked as const. This is because const objects should initialize their member variables, and a const constructor would not be able to do so.

Although it is not done very often, it is possible to overload a function in such a way to have a const and non-const version of the same function:

```
class Something
{
public:
    int m_nValue;

    const int& GetValue() const { return m_nValue; }
    int& GetValue() { return m_nValue; }
};
```

The const version of the function will be called on any const objects, and the non-const version will be called on any non-const objects:

```
Something cSomething;
cSomething.GetValue(); // calls non-const GetValue();
const Something cSomething2;
cSomething2.GetValue(); // calls const GetValue();
```

## References

A reference to a variable is like an *alias*; the variable and the reference can both be used in statements which affect the variable:

```
int int_value;
int &ref = int_value;
```

reference ref addresses the same memory location which int_value occupies.

& indicates that ref is not itself an integer but a reference to one.

The two statements

```
int_value++;          // alternative 1
ref++;               // alternative 2
```

have the same effect.

☐ In those situations where a called function does not alter its arguments, a copy of the variable can be passed (pass by value):

```
void some_func(int val) {
   cout << val << endl;
}

int main()   {
   int  x;
   some_func(x);       // a copy is passed, so
   return 0;           // x won't be changed
}
```

☐ When a function changes the value of its argument, the address or a reference can be passed, whichever you prefer:

```
void by_pointer(int *valp){
    *valp += 5;}

void by_reference(int &valr) {
    valr += 5;}

int main () {
    int x;
    by_pointer(&x);        // a pointer is passed
    by_reference(x);       // x is altered by reference
    return 0;              // x might be changed
}
```

☐ References have an important role in those cases where the argument will not be changed by the function, but where it is desirable to pass a reference to the variable instead of a copy of the whole variable. Such a situation occurs when a large variable, e.g., a struct, is passed as argument, or is returned from the function.

## Virtual Constructors and Destructors

A constructor cannot be virtual: The simple answer is that when a class is constructed, if it inherits from another class, then the base class must be constructed first and only then the inherited class. If the constructor was virtual, polymorphism would prevent from the base class to be constructed first.

A virtual destructor is one that is declared as virtual in the base class and is used to ensure that destructors are called in the proper order.

Destructors are called in the reverse order of inheritance. If a base class pointer points to a derived class object and we some time later use the delete operator to delete the object, then the derived class destructor is not called.

```
#include <iostream.h>
class base{
   public:
   ~base()  {
    ……
  }
};

class derived : public base{
   public:
  ~derived()   {
   ……
   }
};

void main(){
   base *ptr = new derived();
   // some code
   delete ptr;
}
```

In this case the type of the pointer would be considered. Hence as the pointer is of type base, the base class destructor would be called but the derived class destructor would not be called at all. The result is memory leak. In order to avoid this, we have to make the destructor virtual in the base class.

```
#include <iostream.h>
class base {
public:
virtual ~base(){
}
};
class derived : public base{
  public:
   ~derived()
  {

  }
};

void main(){

  base *ptr = new derived();
  // some code
  delete ptr;
}
```

A class must have a virtual destructor if it meets both of the following criteria:

- You do a delete p.
- It is possible that p actually points to a derived class.