

Controlling Change

Constant Objects

const and Pointers: Scenarios

Location Class

const and Pointers: Syntax

const and Pointers: Syntax

Pointer Parameter to a const Target

const Pointer Parameter

const Pointer to const Parameter

Static Variable

Static Variable Example

Static Function Example

Goal:

Protect the integrity of an object by limiting the ways in which it may be modified.

Scenarios:

- passed as a parameter to a function
- declared as a local variable within a function
- accessed via a pointer

C++ keyword: `const`

Source of confusion: location of `const` keyword varies, and the location (syntax) determines the effect (semantics).

Both primitive types and user-defined objects can be const:

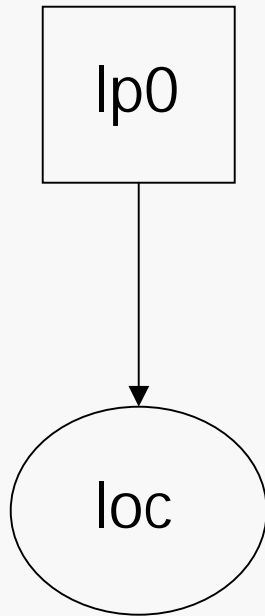
```
const double Pi = 3.141593;           // mathematical constant
const int MAX_ARRAY_LENGTH = 100;    // system limit
const int YesAnswer = 0;             // program convention
const int NoAnswer = 1;              // program convention
const int VersionNumber = 1;         // program information
const int ReleaseNumber = 5;         // program information

const Location dialogLocation (200,200);
                                     // application information
```

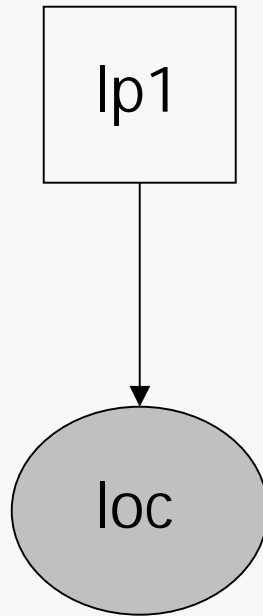
An attempt to modify the value of a const object after its initializing declaration will result in a compile-time error.

When we have a pointer to an object, there are four logical combinations of const-ness:

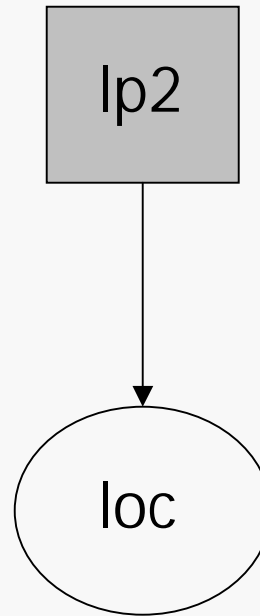
Neither
pointer nor
target is
const



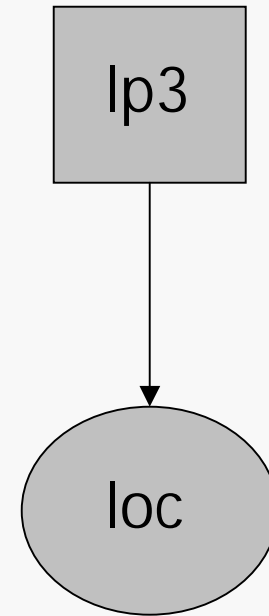
Pointer is
not
const;
target is
const



Pointer is
const;
target is
not const



Both
pointer and
target are
const



Shaded parts are const

Assume a class `Location` that encapsulates a pair of integer coordinates:

```
class Location {
private:
    int X;
    int Y;
public:
    Location();
    Location(int initX, int initY);
    int  getX() const;
    int  getY() const;
    void setX(int initX);
    void setY(int initY);
    void Shift(int deltaX, int deltaY);
    bool operator==(const Location& Other);
};
```

Assume a class `Location` that encapsulates a pair of integer coordinates:

```
Location Center(100, 100);

Location* lp0 = &Center;    // neither pointer nor target is const
*lp0 = Location(25, 15);    // OK
lp0 = NULL;                // OK

const Location* lp1 = &Center; // constant target
*lp1 = Location(25, 15);     // not allowed - compiler error*
lp1->Shift(5, 100);         // not allowed - compiler error
lp1 = NULL;                 // OK
```

```
* main.cpp(17) : error C2678: binary '=' : no operator defined
  which takes a left-hand operand of type 'const class Location'
  (or there is no acceptable conversion)
```

... continued:

```
Location* const lp2 = &Center; // constant pointer
*lp2 = Location(25, 15);      // OK
lp2 = NULL;                   // not allowed - compiler error*

const Location* const lp3 = &Center; // both are constant
*lp3 = Location(25, 15);          // not allowed - compiler error
lp3 = NULL;                       // not allowed - compiler error
```

```
* main.cpp(22) : error C2166: l-value specifies const object
```

Hint: read the declaration from right to left:

```
Location* const lp2 = &Center;
```

lp2 is a const pointer to a Location

Consider adding a member function to Location:

```
void Location::Add(const Location* Other) {  
  
    X += Other->getX();  
    Y += Other->getY();  
  
}
```

The target of this pointer will not be changed by calling this function

What would happen if the member functions `getX()` and `getY()` were not specified as `const` functions?

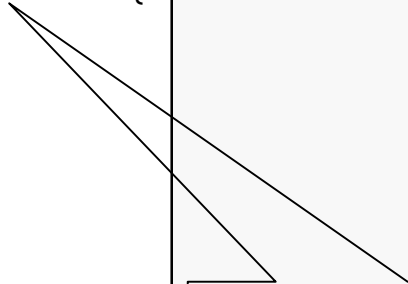
What might a call to `Add()` look like?

```
Location Translate(25, 25);  
Center.Add(&Translate);
```

It's simpler to just implement `Add()` using pass by constant reference...

Of course, this doesn't prevent the pointer itself from being modified:

```
void Location::Add(const Location* Other) {  
  
    Other = this;  
    X += Other->getX();  
    Y += Other->getY();  
}
```



The value of this pointer may be changed within this function.

Not that this makes much sense logically...

But remember that `Other` is a local variable since the pointer isn't passed by reference.

So, this doesn't change the value of the actual parameter passed in the call.

Consider the function:

```
void ReflectHorizontally(Location* const toFlip) {  
  
    int oldX = toFlip->getX();  
    toFlip->setX(-oldX);  
}
```

The target of this pointer will be changed by calling this function, but not the pointer.

Using a const pointer just adds a little safety to the implementation.

Consider the function:

```
void Print(ostream& Out, const Location* const toPrint) {  
  
    Out << '('  
        << setw(3) << toPrint->getX()  
        << ','  
        << setw(3) << toPrint->getY()  
        << ')';  
  
}
```

Neither the target of this pointer nor the pointer itself will be changed by calling this function.

Again, using a const pointer to a const target just adds a little safety to the implementation.

Class-wide data member (aka “class variable”).

One copy per class: shares data among all instances.

Sidesteps the desire for global variables.

Restrictions:

Must be initialized somewhere, but should not be initialized in a constructor.

Why?

```
// Pumpkin.h
enum RipeNess {Green, Ripe, Rotten};

class Pumpkin {
private:
    int      Weight;
    RipeNess Condition;
    static int TotalWeight;
    static int TotalNumber;
    string toString(RipeNess Cond);
public:
    Pumpkin(int Wt = 0);
    void Display(ostream& Out);
    ~Pumpkin();

    static void DisplayTotals(ostream& Out);
};
```

Static variable
declarations

Static function
declaration

```
// Pumpkin.cpp
#include "Pumpkin.h"

int Pumpkin::TotalWeight = 0;
int Pumpkin::TotalNumber = 0;

Pumpkin::Pumpkin(int Wt) {

    Weight      = Wt;
    Condition   = Green;
    TotalWeight += Weight;
    TotalNumber++;
}

void Pumpkin::Display(ostream& Out) {

    Out << "Weight:   " << setw(5)
        << Weight << endl
        << "Condition:"
        << toString(Condition) << endl;
}
```

Class constructor must maintain values as object is created.

Static variable initializations are typically placed in the class implementation.

Lifetime of static variables matches that of program execution; initializations here are set at compile time.

```
// Pumpkin.cpp continued. . .

string Pumpkin::toString(RipeNess Cond) {

    string Response = "Unknown";

    switch (Cond) {
    case Green:  Response = "Green";
                 break;
    case Ripe:   Response = "Ripe";
                 break;
    case Rotten: Response = "Rotten";
    }
    return Response;
}

Pumpkin::~~Pumpkin() {

    TotalWeight -= Weight;
    TotalNumber--;
}
```

Class destructor must also maintain values as object is destroyed.

```
// Pumpkin.cpp continued. . .  
  
void Pumpkin::DisplayTotals(ostream& Out) {  
  
    Out << "Total weight:" << setw(5) << TotalWeight << endl  
        << "Total number:" << setw(5) << TotalNumber << endl;  
}
```

Static member function is NOT associated with any particular object of the class.
Invocation will normally take the form:

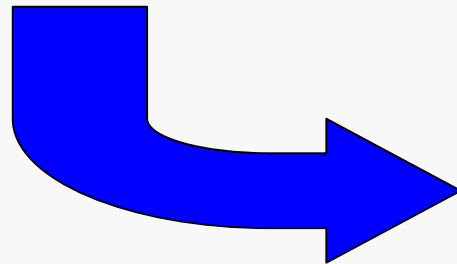
```
Pumpkin::DisplayTotals(cout);
```

```
#include "Pumpkin.h"

void main() {

    Pumpkin P1(3);           // create Pumpkin object
    P1.DisplayTotals(cout);  // invocation via an object

    Pumpkin P2(5);
    Pumpkin::DisplayTotals(cout); // invocation via class name
}
```



```
Total weight: 3
Total number: 1
Total weight: 8
Total number: 2
```