Safer Programming via *const*

Textbook p. 25; 130; A17

Safe Programming Practices

- Goal: protect us from our enemies
  - Protect client from implementer
  - Protect implementor from client
  - Protect us from ourselves!
- Public/private is one safety technique
- Avoiding global variables is another
- Passing pointers and references around can make things less safe
  - but can't always be avoided
- *const*: a safety tool provided in C++

The Mystery of *const*

- You've used it as a replacement for #define
  - const int MAX_NAMELENGTH = 60;
  - rather than
  - #define MAX_NAMELENGTH 60
- In the text, you will notice other usages:

```
class listClass {
 public:
   bool ListIsEmpty ( ) const;
...}
void BinarySearch (const int A[], int First, ...);
```


Basic Meaning: "Can't Change"

- *const* means "if you try to change this thing, I will squawk, read loud"
- Also: "if I suspect somebody might try to change it, I will try to warn about it."
- Enforced by compiler
- Adds a level of fail-safeness
  - but can get complicated in certain cases
- const is a part of the type
  - A non-const converts automatically to const when needed, but not vice-versa

const Variables: True Constants

- Simple and easy to use

```
const double PI = 3.14159;
...
  PI = 3.0; //squawk
  cin >> PI; //squawk
```

- const variables could be global or local
- A const variable *must* be initialized
  - Otherwise, would be no way to give it a value!

```
const double PI;  //a no-no
```

- consts can be class members, too
  - requires special syntax for initialization

const as Argument

- Consider these function calls:
  - *funct (PI);* //example A
  - *funct (&PI);* //example B
- If compiler can determine that the function may try to alter PI: squawk.
- If compiler is assured that function cannot alter PI: no squawk.
- Some prototypes: which ones are squawkers when called with *funct (PI)*? with *funct (&PI)*?

```
void funct (double i);
void funct (double * i);
void funct (double &i);
```

const Argument, Reference Parameter

```
void comp2 (int & N) {
        ...
        N = N+1; //?
        ...
}
```

- Calling *comp2*

```
const int asize = 30;
int bsize = 4;
comp2 (asize); //?
comp2 (bsize); //?
comp2 (4); //?
```


Fill In the Table

- OK; C (const error), or E (other error)

caller:	funct (PI)	funct (&PI)
callee		
void funct (int)		
void funct (int &)		
void funct (int *)		

const on a Pass-by-Value Parameter

```
void recompute (const int N) {
        ...
        N = N+1;  //??
        ...
}
```

const here may protect the implementer of recompute from a programming error.

But -- doesn't add protection to the client (caller) -- the value is passed by copy anyway.

const Parameters Case 2

```
void safe (const team TArray[ ], int N) {
  ...
  N = N*2; //Squawk?
  TArray[N].setGamesWon = 162;  //Squawk?
  ...
}
```

- Calling *safe*

```
const int asize = 30;
team Mymainarray [asize];
...
safe (Mymainarray, 30); //?
safe (Mymainarray, asize); //?
```


const Reference Parameter

```
void comp2 (const Vector & v1, Vector v2) {
        ...
        v1 = v2; //?
        v2 = v1; //?
        v1.insert (0, 414); //????                ...
}
```

- Review: ref. params are frequently used to avoid copying large objects
- New twist: *const* ref. params are frequently used to avoid copying large objects, when the object is not being changed: safe and efficient
- New danger: *const* protects the member variables of the object, but *not* any dynamic memory already pointed to.
  - If the object contains a pointer:

```
You cannot change the pointer
But you can change the contents pointed to!
```


const methods

- Special notation, special meaning

```
class listClass {
  private:
        int listLength;
  public:
    bool ListIsEmpty ( ) const;
...}
```

- Means: "this function can't change *any* member variable of the class."
  - Note: says nothing about parameters
- Very commonly used

const Advice

- For true constants, use *const* variables
  - with whatever scope is appropriate
  - remember that these cannot be passed to non-const reference parameters
- Use *const* on member functions whenever possible
- Use *const* on parameters when appropriate
  - const on a value parameter is a check on the implementer
  - const on a ref. parameter protects the caller, too.
- Adding *const* retroactively sometimes causes cascades of changes, so... put them in from the start!

Fill In the Table

- OK; C (const error), or E (other error)

caller:	f1(i)	f1(&i)	f1(PI)	f1(&PI)
callee				
void f1(int)				
void f1(const int)				
void f1(int &)				
void f1(const int &)				
void f1(int *)				
void f1(const int *)				