

## Safer Programming via *const*

Textbook p. 25; 130; A17

10/9/00 K-1

## Safe Programming Practices

- Goal: protect us from our enemies
  - Protect client from implementer
  - Protect implementer 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++

10/9/00 K-2

## Many Uses for *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, ...);
```

10/9/00 K-3

## Basic Meaning: "Can't Change"

- *const* means "if you try to change this thing, I will complain, real 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

10/9/00 K-4

## *const* Variables: True Constants

- Simple and easy to use
 

```
const double PI = 3.14159;
```
- ...
 

```
PI = 3.0;           //complain
cin >> PI;          //complain
```
- *const* variables could be global, could be local, or could be member variables of a class, as appropriate
- A *const* variable *must* be initialized when defined
  - Otherwise, would be no way to give it a value!
 

```
const double PI; // not allowed
```

10/9/00 K-5

## *const* as Argument

- Consider these function calls:
  - `func(PI);` //example A
  - `func(&PI);` //example B
- If compiler can determine that the function may try to alter PI: complain.
- If compiler is assured that function cannot alter PI: no complaint.
- Some prototypes: which ones generate complaints if *const* variables are used as arguments?
 

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

10/9/00 K-6

## Fill In the Table

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

caller:	func (PI)	func (&PI)
called		
void func (double)		
void func (double &)		
void func (double *)		

10/9/00 K-7

## 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.

10/9/00 K-8

## const Parameters Case 2

```
void safe (const team TArray[ ], int N) {
    ...
    N = N*2; // complain?
    TArray[N].setGamesWon = 162; // complain?
    ...
}
• Calling safe
const int asize = 30;
team theArray [asize];
...
safe (theArray , 30); //?
safe (theArray , asize); //?
```

10/9/00 K-9

## Reference Parameters

```
void comp2 (int &N) {
    ...
    N = N+1;
    ...
}
• Calling comp2
const int asize = 30;
int bsize = 4;
comp2 (asize); //?
comp2 (bsize); //?
comp2 (4); //?
```

10/9/00 K-10

## const Reference Parameters

```
void comp3 (const int &N) {
    ...
    N = N+1;
    ...
}
• Calling comp3
const int asize = 30;
int bsize = 4;
comp3 (asize); //?
comp3 (bsize); //?
comp3 (4); //?
```

10/9/00 K-11

## const methods

- Special notation, special meaning
- ```
class listClass {
    private:
        int listLength;
    public:
        bool ListIsEmpty ( ) const;
    ...}

```
- Means: "this function won't change any member variable of the class."
    - Note: says nothing about parameters
    - Means this function can be called on a const instance of this class
  - Puzzler: would const ever make sense on a constructor??

10/9/00 K-12

## 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.

10/9/00 K-13

## Fill In the Table

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

| caller:               | f1 (i) | f1 (&i) | f1 (PI) | f1 (&PI) |
|-----------------------|--------|---------|---------|----------|
| called                |        |         |         |          |
| void f1 (int)         |        |         |         |          |
| void f1 (const int)   |        |         |         |          |
| void f1 (int &)       |        |         |         |          |
| void f1 (const int &) |        |         |         |          |
| void f1 (int *)       |        |         |         |          |
| void f1 (const int *) |        |         |         |          |

10/9/00 K-14