CSE 143

# 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++



## 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, ...);
```



## 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



## *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
```



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

## Fill In the Table

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

| caller: / called | funct (PI) | funct (&PI) |
|---|---|---|
| void funct (double) | | |
| void funct (double &) | | |
| void funct (double *) | | |



## *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;                              // complain?
  TArray[N].setGamesWon = 162;   // complain?
  ...
}
```

- Calling *safe*

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



## Reference Parameters

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

- Calling *comp2*

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



## *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);       //?
```



## *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??

## *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)

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