COMSC-200 Lecture Topic 4
The Principle Of Least Privilege

□ Reference

Deitel Ch.10.1-10.2
const pointer syntax

□ Why Restrict Ourselves?

self-documentation
avoid programming errors
  e.g., `strcpy(char*, const char*)`
compiler optimizations are possible

□ Implementing The Principle

the `private` keyword (or by default)
data abstraction (hiding the details)
the `const` keyword

□ `const` Global And Local Variables

e.g., `const int N = 100;`
must be assigned upon initialization
globals are shared among functions *(for constants only)*
for non-pointers: cannot *reassign*
for references: cannot change data members

□ `const` Parameter Variables

e.g., `void fun(const int N){...}` // prototype
for non-pointers: cannot *reassign*
for references: cannot change aliased value
`const` pointers can be assigned!
default parameter values are allowed

□ `const` Pointers

constant pointers and read-only pointers
leading `const` protects pointed-to values
  ...and is part of the data type!
trailing `const` protects pointer value itself
leading *and* trailing `const`s
"casting" away read-only constness

□ `const` Return Values

`const int` is meaningless
  assigned as a copy anyway
used to return references and pointers
e.g., `const int& getHour();` // prototype
  can store in non-`const` variable
    `int x = t.getHour();` // statement
    ...x is just a copy
  cannot store in non-`const` ref variable
    `int& x = t.getHour();`
    ...COMPILE ERROR!
    `const int& x = t.getHour();` // ok

□ `const` Data Members

requires constructor to initialize
  but cannot set in constructor body

```
class Time
{
  const int h;
  ...
  Time() // inline function definition
  {
    h = 0; // ERROR
```

requires special syntax (e.g., inline function defs)
  "initializer lists" (in topic 5)

```
  Time():h(0),m(0),s(0){}
```

or

```
  Time(int h, int m, int s):h(h),m(m),s(s){}
```

may use variables or function calls instead of literals
evaluation order matches class declaration order!

□ `const` Member Functions

e.g., `int getHour() const;` // prototype
function cannot change data members of host object
  either directly, through assignment
  or indirectly through function calls
keyword is part of function *signature*
e.g., okay to have in same class:
  `int getHour();` // prototype
  `int getHour() const;` // prototype
compiler choice depends on `const`-ness of host object
getters and dynamic memory

□ Using `const` in OOP

getters: `const` member functions
  do not change host object
setters: `const` parameters
  typically `void` return type, but...
  may return *self-reference* (later...)
constructors: `const` parameters, special syntax
data members: `const` if logically immutable
pointers: leading `const` if not used to change pointed-to data
  trailing `const` if not to be reassigned

□ Not Exactly Java's `final`

`final` data members can be assigned in the constructor
`final` "return types" means function cannot be overridden!
`final` object references do not protect object data

How To Create Arrays Of Objects Without A Default Constructor

```
class Time
{
  int h, m, s;

  public:
  Time(int);
};

int main()
{
  Time a[] = {Time(1), Time(12), Time(18)};
```

How To Create Arrays Of Objects With A Default Constructor

```
class Time
{
  int h, m, s;

  public:
  Time();
};

int main()
{
  Time a[3]; // ...or...
  Time b[] = {Time(), Time(), Time()};
```