Announcement

- Lab #5
  - Programs and Mini project due 3/19 (Thursday) at 11:59PM
- Quiz #2 on 3/19 (Thursday)
- Mid-term exam on 3/31 (Tuesday) during lecture
  - Review on 3/19
- Final project out this Friday (in UCMCROPS)
  - Due date: 5/11 (Monday) at 11:59PM
  - Presentation date: 5/12 (Tuesday) at 3:00PM
- Reading assignment:
  - Ch. 12
A **const** may replace a define in some cases:

```plaintext
#define BUFSIZE 100
const int bufsize = 100;
```

A **const** can be used everywhere:

```plaintext
int main() {
    cout << "type a character & CR:";
    const char c = cin.get(); // Can't change c later on
    const char c2 = c + 'a';
    cout << c2;
    // ...
}
```
The **const** modifier binds to the thing it is **closest** to; important examples:

- To prevent any changes to the element you are pointing to:
  ```
  const int* u;
  ```
- To make the pointer itself unchangeable, that is, to prevent any change to the address contained inside the pointer:
  ```
  int* const u;
  ```
- Other variations possible:
  ```
  const int* const u;
  ```
  (see the book for many interesting variations...)

**Constants in Pointers**
void f2 ( int ic ) {
    const int& i = ic;
    i++;  // Illegal -- compile-time error
}

void u ( const int* cip ) {
    //! *cip = 2; // Illegal -- modifies value
    int i = *cip; // OK -- copies value
    //! int* ip2 = cip; // Illegal: non-const type ip2
}

void g2 ( const X& ) // Pass X object by const reference
{...}
enum can be used to generate consts that are evaluated at compile time.

- You can also explicitly establish the values of the enumerators:

```cpp
def enum { one = 1, two = 2, three }
```

- With integral enum types, the compiler will continue counting from the last value, so the enumerator three will get value 3.
A `const` can also be used to specify that functions cannot modify the variable members of the class:

```cpp
//: C08:ConstMember.cpp
class X {
  int i;
public:
  X(int ii);
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; } // ok to return but
  // not ok to modify
```
Mutables and Constants

- Allows an exception to const members:
  - A const function *can* modify a variable member if the member is mutable.

```cpp
//: C08:Mutable.cpp The "mutable" keyword
class Z {
    int i;
    mutable int j;
public:
    Z();
    void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
    //! i++; // Error -- const member function
    j++; // OK: j is mutable
}
```
A class can have members defined in two ways

- **inline**
  - Implementation of a function is given inside the class declaration.
  - Compiler will not create a function to be linked but will **replace the code** at the location where the function is called. Perfect for short get/set functions, “accessors” and “mutators”.

- **normal**
  - Function is implemented in a .cpp, outside of the class definition
  - Compiler will create compiled code for a function to be linked.
//: C09:Rectangle.cpp - Accessors & mutators

class Rectangle {
    int _w, _h;
public:
    Rectangle( int w = 0, int h = 0 ) : _w(w), _h(h) {}  
    int width() const { return _w; } // Read
    void width(int w) { _w = w; } // Set
    int height() const { return _h; } // Read
    void height(int h) { _h = h; } // Set
};

int main() {
    Rectangle r(19, 47);
    // Change width & height:
    r.height ( 2 * r.width() );
    r.width ( 2 * r.height() );
}
//: C09:Rectangle2.cpp - Accessors & mutators with "get" and "set"
class Rectangle {
    int width, height;
public:
    Rectangle(int w = 0, int h = 0) : width(w), height(h) {}
    int getWidth() const { return width; }
    void setWidth(int w) { width = w; }
    int getHeight() const { return height; }
    void setHeight(int h) { height = h; }
};

int main() {
    Rectangle r(19, 47);
    // Change width & height:
    r.setHeight(2 * r.getWidth());
    r.setWidth(2 * r.getHeight());
}
Inline Functions

- It is also possible to write a normal (not a member function of a class) function that is supposed to be expanded inline:

```c
inline int plusOne(int x) { return ++x; }
```
 Inline Functions (a cleaner interface)

//: C09:Noinsitu.cpp
// In the header file, only the declarations should appear:
class Rectangle {
   int width, height;
public:
   Rectangle(int w = 0, int h = 0);
   int getWidth() const;
   void setWidth(int w);
   int getHeight() const;
   void setHeight(int h);
};

// In the source file, the definitions can still be inline:
inline Rectangle::Rectangle(int w, int h) : width(w), height(h) {}

inline int Rectangle::getWidth() const { return width; }

inline void Rectangle::setWidth(int w) { width = w; }

inline int Rectangle::getHeight() const { return height; }

inline void Rectangle::setHeight(int h) { height = h; }