Miscellaneous
(Inlines, const, mutable)

What are inline functions?

- Inline functions are those for which the compiler expands the code in-line, i.e., there is no function call made to execute the code in that function.

- Example:
  - *inline* double Complex::GetReal () const

```
{
        return  m_Real;
}
int main ()
{
   Complex a( 10, 5);
   cout  <<  a.GetReal()  <<  endl;      ← No function call is made for GetReal()
}
```

How to make a function inline?

- To declare a member function inline, put the keyword *inline* in front of the definition of the function, and place the definition in the .h file (usually)

- A member function can also be made inline by providing its definition inside the class declaration:
  - class Complex {

    …

    double GetReal () const { return m_Value; }

    …

    }

- However, this tends to clutter the class declaration, so it is best to put the implementation outside. (See example of class Complex)

Non-member inline functions

- Inlining is not restricted to member functions. Non-member functions can be inlined too.

- Example:
  - ```
    inline void PrintComplex ( const Complex & c )
    {
       cout << "Complex passed in is  (" <<  c.GetReal() << ", " << c.GetImaginary()
            << ")"  << endl;
    }
    ```
  - As long as this definition appears before its use, the compiler can inline it.
    ```

Inlining...

- Inlining is a suggestion or a hint to the compiler.
- This means that it is not guaranteed that a function that is declared as inline will be inlined by the compiler.
- For a function to be inlined, its definition should be visible to the compiler at compile time.
- So, a function defined in another library cannot be inlined unless its definition is available in the .h file (since in that case, the compiler can see it at compile time).

Inlining...

- So, when is a function not inlined:
  - A recursive function is not inlined.
  - A function that has a lot of code in it may not be inlined.

- Inlines can be compared to macros *(#define)* in C.

- Inlines are better than C macros because :

  - Inlines provide type checking of arguments (since they are really functions).
  - Arguments passed in are evaluated only once, whereas in the case of C macros, there is a possibility (depending on the macro) that the argument(s) are evaluated more than once, resulting in bugs.
  - Easier to debug than macros since you cannot step into macros, but can step into inline functions (as long as you tell compiler not to inline during debug code generation).

Pros & Cons

- Performance benefit since a function call is avoided, so savings in not having to build activation record, etc.
- Careless inlining can increase the size of the object code, since the function is expanded inline at every point of call in the source code.
- So, typically one or two statement functions are inlined. Most common examples are Get and Set methods. (See class Complex example).

Can virtual functions be inlined ?

Const members

- Const data members:
  - This just means that these data members do not change their values once initialized.
  - Const data members need to be initialized in the initialization list of the constructor.

- Const method members:
  - This just means that these methods do not modify the object.
    - Example:
      - GetReal() or GetImaginary()

    - Set methods are not const (or any method that modifies some data member of the object, in other words, modifies the state of the object).

Mutable

- When a data member needs to be modified, but doesn't really affect the state of the object, it can be declared as *mutable*.

- What this means is that a method can modify a mutable data member, and this method can still be declared as a *const*.

- Example:
  - Lets say we have a class that holds some integer values.
  - Let there be a Find (int value) member method that searches for a given value and returns *true* if it finds it, and *false* otherwise. Since it simply does a search and does not modify the object, it can be declared as a *const.*

  - Now, Find() may try to be efficient, and may try to store the index of the value it just found, so as to use that index as the starting point of the search in the next call to Find(). Now, to store this index, it needs to modify a data member, and this data member needs to be declared as mutable in order for Find() to be declared as a *const.*