Today’s lecture: Miscellaneous C++ Topics
- abstract classes
- C preprocessor
- constants

Things that didn’t fit in previous lectures
- abstract classes
- C preprocessor
- constants

Recall that compile time binding is the default for class members in C++
To define a member that is bound at runtime, declare it to be virtual

```cpp
class RAM {
  public:
    virtual int load(...) {...
  ...
};
class Cache : public RAM {
  public:
    int load(...) {...
  ...
};
```
Pure Virtual Function Declaration

- A method that is not implemented in the base class is a pure virtual function.
- To declare a pure virtual function, "initialize it" with =0:
  ```cpp
class Memory { // in header (Memory.h)  
  public:  
    virtual int load(int addr) = 0;  
  };
```
- This declaration says load() is a "place-holder" for actual methods that will be declared in derived classes.
- It is not necessary to define Memory::load() in Memory.C because load is already defined with =0.

Pure Virtual Function in Derived Classes

- When you define each type of Memory, you will declare a method named load for that type:
  ```cpp
class RAM : public Memory { // in RAM.h  
  public:
    int load(int addr); // prototype  
  };
```
- And include a definition in the implementation file:
  ```cpp
  int RAM::load(int addr) { // in RAM.C  
  }
  ```

Abstract Classes

- If a class contains one or more pure virtual functions, it is called an abstract class.
- Abstract classes cannot be instantiated.
- This is a syntax error:
  ```cpp
  Memory M; // syntax error: can't make abstract object
  ```
- This is OK:
  ```cpp
  Memory *M[10];  
  M[0] = new RAM;  
  M[1] = new Disk;  
  ```

Derived Classes May Be Abstract

- A class derived from an abstract base class does not have to declare all the pure virtual functions.
- It may itself be an abstract class and pass the virtual functions down to its own subclasses:
  ```cpp
  class RandomAccessMemory : public Memory { // no declaration for load()  
  };
  ```

Comparison to Java

- In Java, all member functions are virtual.
- A C++ abstract class that contains only pure virtual functions is the equivalent of a Java interface.
- A Java class that implements an interface promises it will provide a definition for each method named in the interface.
- Similarly, a C++ concrete class implements all the pure virtual functions promised in its abstract base class.

II: Preprocessor Directives

- In C and C++, lines that begin with a # are compiler directives.
- These lines are instructions to the compiler on how to compile the program.
- They are not part of the language.
- Examples:
  ```cpp
  #include <iostream>  
  #include "Feature.h"
  ```
Uses of Directives

✓ All compilers support a common set of directives
   - e.g. includes, conditional compilation
✓ Most compilers also have an additional set of directives
✓ Example: SGI uses directives to specify parallel regions in large applications
  #pragma omp parallel
  for (int i = 0; i < N; i++)
  surface.updateRegion(i);
  (lets a parallel processor update different regions simultaneously)

Defining Constants

✓ A very common use of directives is to define a constant
  #define N 10
  means “find every occurrence of the symbol N and replace it by the string ‘10’”
✓ Can also use it to define “macros”
  #define inc(N) N + 1
  int i = 3;
  inc(i);
* But read Chapter 1 of The Practice of Programming...

Don't Define Constants

✓ Constants defined with #define are
   - not type-checked
   - do not have scopes (they are defined over the entire file)
✓ It is much better in C++ to use the const keyword to define constants

Conditional Compilation

✓ Two directives you will use often can control whether the compiler should process part of a file
  #ifdef SYM
  // this code is compiled if SYM has been defined
  #endif
  // What symbols are defined?
  - some host or OS specific constants (e.g. Solaris, Sparc)
  - symbols defined in a Makefile (e.g. DEBUG)
  - symbols you define with #define

Very Large Comments

✓ One use of ifdef is for “commenting out” large sections of your program
  - C++ does not allow nested comments
✓ To comment out a big section of a program:
  #ifdef NEVER
  many lines of code here, including lines that contain comments
  #endif
  // resume compilation here

Directives in Header Files

✓ Conditional compilation is required in class header files
✓ In large programs, a header may be included several times
✓ Example: in your application:
  #include <string> // because your app
  // uses strings
  #include "Feature.h" // and Feature objects
  // in Feature.h:
  #include <string> // because features
  // have strings
✓ Classes defined in string will be multiply defined, which is a syntax error
Directives in Header Files (cont’d)

- A common organization for header files is
  - #ifndef _Feature_h_
  - #define _Feature_h_
  - // declarations
  - #endif
- #ifndef SYM means "compile if SYM not defined"
- Choose a distinctive symbol name for each header
- The first time the file is included, the symbol is defined
  along with the classes
- The second time the file is included the declarations are
  skipped

III: Constants

- The keyword const is used in several contexts in C++
  - variable declarations
  - pointer declarations
  - class data members
  - parameters to functions
  - return types of functions
  - class methods
- Declaring an item to be a constant has several benefits
  - helps the compiler generate efficient code
  - helps document the program
  - will help you catch bugs (e.g. unintended side effects)

Local and Global Variables

- If you declare a variable to be a constant, the compiler will
  catch any attempts to modify it:
  ```cpp
  const int i = 10;
  i = 3;  // syntax error
  ```
- The compiler won’t let you make an alias and change a
  constant through trickery:
  ```cpp
  int *p = &i;  // syntax error
  *p = 3;  // nice try...
  ```
- When you define a variable to be a constant, you must
  initialize it (because you can’t supply the value later)
  ```cpp
  const int n;  // syntax error
  ```

Constant Object Instances

- Variables that are instances of classes can be constants
  ```cpp
  const string s = "hello";
  s.append(", world");  // syntax error
  s += ", world";  // syntax error
  ```
- But you can call methods that do not modify the object:
  ```cpp
  cout << "a has " << s.length() << " chars" << endl;
  ```
- We’ll see later how the compiler knows which methods do
  not modify objects

Aside: Enumerations

- A data type inherited from C is enum
- Use it to give symbolic names to a set of values, e.g.
  ```cpp
  enum instr_t {MISC, MEM, REG, BRANCH};
  ```
- Defines instr_t to be a new type and defines MISC, etc
  to be values of the type
  ```cpp
  instr_t x = BRANCH;
  ```
- Coding conventions:
  - type names end in "_t"
  - since the values are constants, use all upper case for names

Enumerations (cont’d)

- Implementations: enumerations are integers, and the
  defined values are constants starting with 0 (unless you
  specify otherwise)
- The compiler guards against extraneous values
  ```cpp
  enum instr_t {MISC, MEM, REG, BRANCH =7};
  instr_t x = REG;
  ```
- x = 2;  // syntax error
  ```cpp
  cout << "x = " << x << endl;
  ```
- prints the integer equivalent of REG (2 in this example)
Const Pointers

✓ It is possible to make pointer variables constants
✓ If you want the value of a pointer to be constant (i.e. the pointer always refers to the same memory location):

```c++
int const *q = new int[10];
q[0] = 4;
// q = new int[10];   // syntax error
```

Note the placement of the keyword `const` between the `*` and the variable name.

Pointer Consts

✓ It is also possible to tell the compiler the location pointed to by a pointer is constant:

```c++
const int *p = new int[10];
// p[0] = 4;           // syntax error
p = new int[10];
```

✓ This is different than saying the pointer variable itself is a constant
✓ Compare the placement of `const` with the previous example

Mnemonic for Const Pointers

✓ How can you remember which is which?
```c++
const int *p = makeBlock(...);
int *const q = makeBlock(...);
```

✓ Here is a trick:
  - read the declaration in reverse order
  - “p is a pointer to an integer constant”
  - “q is a constant pointer to an integer”
  (where “an integer” really means “a block of integers”)

Const Pointer Const

✓ It is possible to define a pointer that is constant in both senses:

```c++
const int *const r = makeBlock(10);
// r[0] = 4;
// r = new int[10];
```

Here `r` is a constant (can’t be updated to point to a new region) and the region it points to is constant.

Constants Inside Classes

✓ It is possible to declare class members to be constants
✓ Problem:
  - data members of a class can only be initialized in a constructor (not in the class declaration)

```c++
class BadFoo {
public:
    int j = 10;     // syntax error
};
```

✓ Constants inside classes allow data members to be initialized when they are declared
```c++
class GoodFoo {
public:
    int j = 10;     // syntax error
    const int k = 20; // syntax error
};
```
Const Initialization Syntax

- Since we can’t assign a value to a const variable, we need a new syntax.
- Supply a list of const variables and their values right after the constructor name.

```cpp
class Foo {
    public:
        Foo(const int n) : i(n) {}  // exception: const members not initialized
    private:
        const int i;
};
```

General Initialization Syntax

- The method of initializing members by listing them after the constructor names applies in other situations.
- Use it to initialize any data members, not just constants.

```cpp
class Foo {
    public:
        Foo(int n) : i(n), j(n+1) {}  // ...}
};
```

Constant Parameters

- Recall value parameters are local variables.
- They can be declared as constants:

```cpp
void foo(const int i) {
    i = 3;  // syntax error
}
```

Constant Reference Parameters

- Declaring a reference parameter to be a const makes it easier to use references for efficiency.
- Use it to call superclass constructors.

```cpp
class RAM {
    public:
        RAM(int size) : Memory(size) {}  // ...}
};
```

Constants Returned by Functions

- Functions can be declared as returning constants.
- For returning values, this doesn’t make much sense.

```cpp
const int foo(int n) {}  // ...}
```

- But if a function returns a reference, particularly to a large structure, it might make sense to declare it a constant:

```cpp
const list<int>& zeroTerminated() {
    list<int>* L = new list<int>;
    L->push_back(2);  // make a big constant list and return
    return *L;  // a reference to it
}
```

Returned Constant Example

- Here is an example of a function that creates and returns a large structure:

```cpp
const list<int>& zeroTerminated() {
    list<int>* L = new list<int>;
    L->push_back(2);
    cout << "list at " << L << endl;
    return *L;
}
```

- The list is allocated on the stack, and the return value is the list itself, not a pointer to the list.
Using a Returned Constant

Here are some examples of how to use a returned constant:

- `list<int> x = zeroTerminated();` // legal -- why?
- `list<int> y = zeroTerminated();` // syntax error
- `const list<int> &z = zeroTerminated();`

The first declaration is OK because `x` is a copy of the constant list. `y` and `z` are references to the list created by the function, but only `z` is legal because it is a `const`.

Verifying a Returned Constant

Here is the complete code that uses a returned constant, verifying it has the same address as the list created in the function:

```cpp
const list<int> &z = zeroTerminated();
cout << "z at " << &z << endl;
// z.push_back(13); // syntax error
list<int>::const_iterator it;
for (it = z.begin(); it != z.end(); it++)
    cout << " " << *it;
cout << endl;
```

Note the iterator is a `const_iterator`.

Aside: Compiler Error Messages

For what it’s worth, here are the compiler error messages for the declaration of `y`:

```cpp
% CC -c constdemo.C
"constdemo.C", line 158: Error: Initializing std::list<int, std::allocator<int>> requires an lvalue.
```

```cpp
% g++ -c constdemo.C
constdemo.C:158: conversion from "const list<int, allocator<int>>" to "list<int, allocator<int>> &" discards qualifiers
```

Constant Methods

In previous examples, we saw the compiler would not allow us to modify a constant object instance:

```cpp
const list<int> &z = zeroTerminated();
// z.push_back(13); // syntax error
cout << z.size(); // OK
```

How does the compiler know it is safe to call `size()` but not `push_back()`?

- `size()` is declared to be a constant method.

Declaring a Constant Method

Append the keyword `const` to the method name, after the parameter list:

```cpp
class Bar { public:
    void addInt(int i);
    int size(void) const;
private:
    list<int> L;
};
```

Implementing a Constant Method

The `const` keyword also has to be used in the implementation of the method:

```cpp
int Bar::size(void) const {
    return L.size();
}
```
Using Constant Objects

- You can invoke a `const` method for constant and non-constant object instances
  ```cpp
  Bar x;
  x.addInt(2);
  cout << "x.size = " << x.size() << endl;
  ```
- You can only invoke `const` methods on constant objects
  ```cpp
  const Bar y = x;
  // y.addInt(13);  // syntax error
  cout << "y.size = " << y.size() << endl;
  ```

Class Design Considerations

- If a method does not modify an object, declare it `const` so it can be used for `const` objects
- If you do not specify methods to be `const`, users will not be able to define `const` instances of your class

Next

- Unix shells and tcsh