- **Lecture Outline - Basics of Classes, Struct vs Class, Operator overloading, How members really work**
- **Basic Concepts**
  - **Idea: User defined type is basis of OOP;**
    - *Built-in types like int, double*
      - declare them: int i, j; double x;
      - operate on them: i = i + j;
      - use them as function arguments, parameters, and return values: foo(i) i = foo(j); int foo (int ii) { ...}
      - value of argument is copied into space on stack for parameter
      - return value is held temporarily and then copied back to caller's variable
    - *A type*
      - represents a certain kind of data
      - a certain amount of memory required to represent the data
      - can be used in certain ways, with certain operators
    - *A user-defined type is a custom-made type that you define in order to represent things in the problem domain*
    - *An example (see examples on website for actual code)*
      - CoinMoney - a class that keeps track of money represented in coins - e.g. quarters, nickels, dimes, etc.
      - Sketch implementation:
        - class CoinMoney {
          public:
          CoinMoney () :
          nickels(0), dimes(0), quarters(0) {}
          CoinMoney (int nickels_, int dimes_, int quarters_) :
          nickels(nickels_), dimes(dimes_), quarters(quarters_) {}
          // other public functions
          private:
          int nickels;
          int dimes;
          int quarters;
          };
- **How a class is like a struct, and in fact a struct is a class!**
  - **a C++ class is based on a C struct - what about structs in C++**
  - **"struct" and "class" keywords can be used interchangeably. The only difference is:**
    - *"class" - all members are private by default*
    - *"struct" - all members are public by default*
  - **in customary usage:**
    - *use "struct" only where you want all members to be public*
      - because that is how the class should work anyway
      - especially if only data members - same way you use a struct in C
      - "POD" class - "plain old data"
    - *use class everywhere else*

        - rule of thumb: Make all members private except for the public interface
- **Declaring and defining - member functions and variables**
    - **Compiler must be told about a class before you can refer to it. Provide class declaration before code refers to objects or members of the class.**
    - **But within a class declaration, the code can refer to member variables or member functions before the compiler has seen those members.**
        - *As if the compiler overviews the whole class declaration before, notes the member variables and functions, then goes through an compiles the member function code.*
        - *For this process, all it needs to digest the class declarations is the member variable names and types, and the member function prototypes.*
    - **Can define the functions themselves either inside or outside the class declaration.**
        - *class Thing {*
          void foo ()
          { the code } // defined inside
          };
        - *but if define function outside, have to tell the compiler which class the function belongs to, using the "scope resolution" operator, ::*
        - *class Thing {*
          void foo(); // function prototype
          };

          void Thing::foo()
          { the code}
        - *the function definition can appear anywhere later in the file, or another file altogether (the usual case) since the class declaration and member function prototype tell the compiler everything necessary to compile code that uses the class.*
- **Operator Overloading -**
    - **In C++, every operator in the language has a function-like name:**
        - *operator+, operator*, operator<<*
    - **You can assign a different meaning to the operator for a user-defined type by defining the function with the appropriate arguments.**
        - *E.g. CoinMoney m1, m2;*
        - *m1 + m2*
        - *CoinMoney operator+ (CoinMoney m1, CoinMoney m2) --- l.h.s., r.h.s*
    - **You are not allowed to overload the operators for built-in types**
        - *Can't change what it means to add two numbers!*
        - *Too much mischief!*
    - **Usual custom - make operator functions member functions when possible**
        - *First parameter has to be an object of the class type*
        - *If you define this function as a member function, the first argument is implicit*
            - CoinMoney operator+ (CoinMoney m2) --- r.h.s. only
            - l.h.s. argument must be this type of object.
        - *so x + m1 can't be a member function, m1 + x can be.*
    - **If LHS object is not of the class type, the operator function can't be a member function.**

- *Must define as a non-member function.*
- *non-member functions do not have access to private data*
- *How do you access the member variable values?*
  - normally, use reader/writer access functions
  - sometimes having to use public accessors to get at private data is inconvenient, or not right
    - e.g. maybe this is the only place you need to - why clutter things?
    - or data does need to be completely private and should NOT have any reader/writter functions
      - but in this particular case you need to be able to get to it from outside the class
  - *Friend Functions*
    - in a class declaration, you can declare a function to be a "friend" of a class - grant access to private members.
    - friend <function prototype> anywhere in the class declaration
  - *Friend classes*
    - This class declares another class as a friend
    - Its member functions have access to the private members of the class
    - friend class <classname> anywhere in the class declaration
- **Overloading the output operator**
  - *Extremely handy. Output your own objects however you want:*
    - e.g. cout << m1 << endl
  - *How to do it:*
    - ostream& operator<< (ostream& os, CoinMoney m)
      - {
        - os << m.nickels << " nickels, " << m.dimes << " dimes, "
          - << m.quarters << " quarters, totaling $" << m.value();
        - return os;
      - }
    - Four points!
      - Can't be a member function! Left-hand parameter isn't the right type!
      - first parameter must be declared as REFERENCE to OSTREAM
      - returned type must be declared as REFERENCE to OSTREAM
      - function MUST return its ostream parameter!
  - *Digress a bit.*
    - cout is an object from the class ostream, initialized at program start, to output to the console.
    - ostream class defined in library <iostream.h>, <iostream>
    - operator<< has been overloaded for all of the built in types
    - basic form of << overload:
    - ostream& operator<< (ostream& os, type x) {
      - code for outputting characters
      - return os;

        - }
    - accepts a reference to an ostream object, and returns the reference, for two reasons
        - cascaded I/O
            - cout << x << y; ==> ((cout << x) << y)
            - each << operates on an ostream lhs, other objecr rhs, returns the ostream, so next can work on it.
        - reference because you don't want to copy the ostream object - lots of internal state information would be lost -
        - "pass through" reference argument. Same object returnd (alias for it) as passed in.
- **Overloading the input operator**
    - *In handout*
    - *Can overload the input operator the same way, but less common*
        - have to use reference parameter for the object
        - istream& operator>> (istream& is, CoinMoney& m);
        - cout << "Enter values for x, nickels, dimes, quarters, and i" << endl;
        - cin >> x >> m1 >> i;

Constructors

- **Compiler guarantees that constructor will be called when an object comes into being, and destructor when it ceases to exist.**
    - *void foo()*
      {
      CoinMoney m; // compiler will insert a constructor call
      . . .
      return;
      } // m goes out of scope, compiler will insert a destructor call
    - *void foo()*
      {
      *CoinMoney * p = new CoinMoney; // new will do a constructor call*
      . . .
      delete p; // delete will do a destructor call
- **What do constructors do with member variables?**
    - *If you assign them in a constructor function, then that's that.*
    - *If not, two cases:*
        - member variable is built-in type like double, int, etc - nothing
        - member variable is a user-defined type, then its DEFAULT CONSTRUCTOR is run
        - member variables are constructed in the order they appear in the class declaration!
- **Short-hand - constructor initializers**
    - *CoinMoney() : nickels(0), dimes(0), quarters(0) {}*
    - *CoinMoney(int n, int d, int q) : nickels(n), dimes(d), quarters(q) {}*
    - *Requirement : Start using now for all simple initializations*
    - *Optional for simple member variables, but essential for other things*
- **What happens during construction if a member variable is a user-defined type?**
    - *First, what happens if it is a built-in type, like int?*

- *Answer: nothing, unless you do something in the constructor*
- *But if member variable is a class type, its constructor will be called*
- *e.g. vending machine class*
  - class VendingMachine {
    public:
    VendingMachine()
     {}
     // other members

    private:
    CoinMoney coinbox;
    };
  - if you construct a VendingMachine, what happens to the CoinMoney variable?
    - VendingMachine v;
      {...}
    - answer: default constructed - all zeros, in our example
  - what if you want something else? VendingMachine constructor can arrange it
  - VendingMachine() : coinbox(1,1,1) {...} // compiler will call CoinMoney ctor with those parameters
  - VendingMachine(int n, int d, int q) : coinbox(n, d, q) {...}
- *THE FOLLOWING WILL NOT WORK, BUT EVERYBODY TRIES THEM ONCE!*
- *VendingMachine(int n, int d, int q) :*
  - {
    // creates a local variable which is then ignored
    CoinMoney coinbox(n, d, q);
    // OR
    // creates an un-named local Coinmoney object which is then tossed away
    CoinMoney (n, d, q);

    // this works, but why bother with it when ctor initializer will work for you?
    coinbox = CoinMoney(n, d, q);
    }
- **Default function parameters in constructors**
  - *Provides another way to define the default constructor*
    - the default constructor is one that can be CALLED with no arguments
      - CoinMoney int n = 0, int d = 0, int q = 0) : nickels(n), dimes(d), quarters(q) {}

Const correctness with class members

- **Make member functions const if they don't change the state of the object.**
- **Member variables can be const, but if so, they can only be initialized in the constructor initializer list!**
- **Rare case: a member function is logically const - looks const to the outside world - but does its work by modifying certain private members. For example, saving computation time by caching a result internally.**
  - *Mark member function as const*
  - *Mark modified member variable as mutable - can be modified by a const member function.*
  - *THIS IS RARE - DO NOT USE IN THIS COURSE.*

- **How do member functions really work?**
  - **Three questions:**
    - *Where are the member functions?*
    - *How does compiler know or represent which member function goes with what class?*
    - *How do you get access to the data members without the dot operator?*
  - **Member functions are actually just ordinary functions after the Compiler gets through with them.**
  - **Compiler essentially rewrites your member functions in the process of compiling them.**
    - *Very first C++ actually did exactly that ... translated early C++ into C which was then compiled.*
  - **first, class objects occupy a piece of memory**
  - **Remember using a pointer together with a struct - as in P1**
    - *common pattern - function has a first parameter that is a pointer to the data in memory that represents the object - the function works on that object*
    - *CoinMoney * ptr;*
    - *ptr = address of a piece of memory*
    - *ptr->dimes // (*ptr).dimes*
    - *access the int that lies at address in ptr + 4 bytes.*
  - **Member functions actually have an implicit parameter, "this", a pointer to the piece of memory that the current object occupies. Compiler compiles this member function**
    - ```
      double value()  // in CoinMoney class
      {
            return (5 * nickels + 10 * dimes + 25 * quarters) / 100.;
      }
      ```
  - **as if you had written this function:**
    - ```
      double value(CoinMoney * const this)
      {
            return (5 * this->nickels + 10 * this->dimes + 25 * this->quarters) / 100.;
      }
      ```
  - **in non member code, expression invoking the member function gets rewritten as:**
    - *x = m1.value(); ==> x = value(&m1);*
    - *or if CoinMoney * ptr;*
    - *ptr = &m1;*
    - *ptr->value() ==> value(ptr);*
  - **Likewise, in member function, invoking another member function gets rewritten:**
    - *compute_value();*
    - *compute_value(this);*
  - **You can use the "this" pointer variable yourself - will see uses of it later.**
    - *this == a const pointer to the current object, with type <this class> * const*
    - **this == the current object itself - dereferencing the pointer.*
    - *BUT don't use it if you don't have to - just duplicating what the compiler does, wastes time, error prone, looks ignorant*
  - **How does the compiler keep track of which function goes with what class?**
    - *Overloading mechanism:*

- *suppose class Gizmo { int value() {....} };*
- *CoinMoney's value has signature:*
  - value (CoinMoney * this)
- *Gizmo's value has signature:*
  - value (Gizmo * this)
- *Different signatures, different functions!*

- **How does overloading work? How does it tell those apart?**
  - *Normally, every function has to have a unique name*
    - linkers assume it is so ...
  - *Most C++ implementations actually rewrite the function name to make it include the signature!*
  - *value_9CoinMoneyp*
  - *value_5Gizmop*
  - *called "name mangling" - same old linker logic can be used*
  - *Most implementations try to hide the this parameter and the mangled name from you - supposed to be under the hood.*
    - no standard mangling scheme - up to compiler
    - but you will see it from time to time - read through the gobbledegook - you can usually figure out which class and which function is involved

- **Inline functions - why reader and writer functions don't hurt.**
  - **C++ compiler has capability of doing function "in lining"**
    - *can save time and space*
    - *body of a function is inserted at the point of the call*
    - *no function call overhead*
    - *total size of code imay be larger, but code is faster*
  - **by default, functions defined inside a class declaration will be inlined (if the compiler can do it)**
  - **by custom, only simple functions have their definitions in class declaration**
    - *e.g. readers and writers - they will almost certainly be inlined, so no function call overhead!*
  - **complicated function defined outside the class declaration**
    - *put function declaration - prototype - in the class declaration*
      - double value();
      - still a member function, because that is where it is declared
    - *define function outside using "scope resolution operator"*
    - *double CoinMoney::value() {...}*
  - **often want to set "don't inline" compiler option while debugging**
    - *e.g. there may be no separate statements to step through or set a break point on*