- **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*
  - **Example: A CoinMoney type**
    - *Illustrate using these concepts to define a programmer-defined data type that can be used with the same ease and power as a built-in data type like int or double*
    - *E.g. in world of vending machines, money is not just a dollar value, but consists of quantities of coins.*
    - *We want to keep track of amounts of money that come in the form of coins. E.g. we can have $1 in coins, but having it in four quarters is different from having it in 20 nickels.*
    - *Let's define a class of objects that represent collections of coins, and can be manipulated like ordinary numerical variables.*
      - E.g.
        - if m1 is 2 dimes and 3 nickels, and m2 is a quarter, then
        - m3 = m1 + m2;
        - results in m3 being a quarter, 2 dimes, and 3 nickels.
      - Can call functions with such variables
        - compute_something (m1, m2, m3)
      - Can do easy I/O, just like with ints
      - cout << "m1 = " << m1 << endl;
    - *Can do part of this with a structure in C, and so will start there, because a C++ class is based on the idea of a structure in C*
    - *MUST BE SURE YOU UNDERSTAND EXACTLY WHAT IS HAPPENING*
  - **CoinMoney_C_struct**
    - *Example code is CoinMoney_C_struct.c*
      - keeping C I/O for this one case
    - *The declaration says how we want memory laid out, but doesn't *define* a particular variable, or ask the compiler to arrange to allocate memory yet.*
      - declaration - information for the compiler

        - when we want one of these, this is the amount and layout of memory that we want
      - definition - committ to the memory space for one of these things
        - compiler, please make arrangement to set aside the memory and allow me to refer to it with the variable name
    - *Each CoinMoney object will correspond to three ints worth of space in memory.*
    - *The individual ints are called "members" and each has a name whose SCOPE is the struct definition - nickels inside CoinMoney is different from some other variable named nickels*
    - *if declare a struct Coinmoney variable m1, then get three ints of memory, can refer to all three with the name "m1"*
      - m1.nickels is contents of &m1+0
      - m1.dimes is contents of &m1 + 4
      - m1.quarters is contents of &m1 + 8
    - *in main function, declare/define three CoinMoney variables, memory is allocated on the stack for them.*
      - m1, m2, m3
    - *set members of m1*
    - *print it out by using it as an argument to the print function - come back to that*
    - *assign one to another*
      - the members of m1 get copied into the corresponding members of m2
    - *skip down to value function*
      - pushes a copy of m1 on the stack
      - function accesses it the same way .dimes means add 4 to where struct object starts
    - *print function*
      - push copy of argument on the stack
      - prints contents of members
      - push a copy of the parameter onto the stack
      - value function gets it, returns value
    - *add function*
      - two arguments get copied onto the stack
      - create another object, set its members to the sum, return the object, meaning its value gets copied into the caller's m3.
    - *CONCLUSION: in a struct CoinMoney object behaves something like an built-in-type variable, but can only do the following with it:*
      - Use it as a function argument and parameter
      - Return it from a function
      - Assign them to each other
      - Access their data members with the dot operator.
    - *CLASSES in C++ are a lot like this, but much more powerful.*
- **CoinMoney_class**
  - *Example code is CoinMoney_class.cpp*
  - *NOW that have basic ideas, look at a class-style definition of CoinMoney*
  - *Does all the things Winston describes in Ch 9 -14.*

- *Overview first, then see how they work in more detail.*
- *main routine*
  - declare a CoinMoney variable m1, with the constructor to set its values
  - declare two CoinMoney variables m2, m3, the default constructor initializes them to our default values.
  - See how member functions are called with a dot operator
    - pick out member variable, likewise, pick out member function
    - m1 - compiler knows it is a CoinMoney object, so knows that m1.print is calling the print function that was defined as part of the CoinMoney class
  - print out the results, see how m1, m2 was initialized
  - can't set values directly, have to use writer functions to change m2
  - gets its value with m1.value() member function
  - assign m2 = m1, like a C struct, member variables are copied, one to the other
  - call add function - works same way as C version, gets m3 back wiith the sum.
- *Go to the declaration, Skip down to private: section first. Same data members/member variables as the C version.*
- *These are marked as private, meaning that the compiler will not allow you to refer to these from outside the class.*
- *back to beginning, declare some of members to be public: compiler will allow you to refer to these outside the class*
- *Constructors - default, with arguments - initializes values for you automatically*
  - When you define - allocate memory for- one of these, the compiler puts in a call to this function for you to do whatever you want
  - forgetting to initialize variables is a big source of bugs - can control how it should be done, compiler makes it happen "automagically"
- *reader and writer functions, as in Winston, but I like to call them get and set, instead of read and write*
  - allows code outside the class to read and modify the member variables
  - later show why this makes a lot of sense
  - reader functions are marked "const" - do not modify the object
    - const member functions promise not to modify the data in the object
  - writer functions, of course, do modify the object
- *a member function that computes the value - does it the same way as in struct version*
  - also a const function
- *Another member function that prints out the value, using cout <<*
  - also a const function
- *a non-member function for add - can't access private members*
- *looks the same as the C version, except uses reader functions to get the values*
- *HERE'S WHAT OFTEN CONFUSES PEOPLE. WHY NO DOT OPERATOR HERE?*
  - KEY CHARACTERISTIC OF A MEMBER FUNCTION: Member variable names, and other member functions, can be directly referred to no need to name the class or use the dot operator.
  - When a member function executes, it is applying to an individual object - the member variables and functions are "automagically" the ones belonging to that object, and so don't have to be designated.

    - OUTSIDE THE CLASS: m1.nickels or m1.print()
        - Compiler: I know it is CoinMoney nickels/print for m1 because m1 is a CoinMoney
    - INSIDE THE CLASS: nickels or print()
        - Compiler: I know it is CoinMoney nickels/print because this function is a CoinMoney because it was declared to be a member of the CoinMoney class!
- **Digression: Basic I/O stream classes**
    - *cin, cout are objects. cin is an input stream istream object, cout is an output stream ostream object*
        - cin.get(), cin.clear() - a member function of istream class
        - others?
            - cout.setf(.....) a member function of ostream class
            - cout. precision(.....) a member function of ostream class
    - *in fact, they are global variables, declared in the <iostream> header files*
    - *more member functions later when we deal with file streams*
- **CoinMoney_class_why_private**
    - *Example code is CoinMoney_why_private.cpp*
    - *calculating value very differently - must guarantee that data members are not changed without class functions knowing about it.*
    - *This is why making data members private and providing reader/writer functions is recommended*
    - *Note funny business about const*
        - a fine point - but if don't get into, have to explain more
        - notice how get_value now modifies data in the object - the caching variables
        - so, get_value shouldn't be const anymore, right?
        - But: getting the value is conceptually const - the outside world doesn't know, and can't tell, whether the object was changed or not!
            - no reader/writer for the caching variables!
        - But again: if we say get_value is NOT const (but all the other read-only functions are), then this is a strong suggestion that get_value changes the object in relevant ways.
        - What to?
        - Very reluctantly, add something to the language - everything is there for a very good reason that was argued about for some time!
        - Handle this special case of const member function with a special key word, "mutable"
            - relatively new to the language
            - allows this member variable to be modified, even if the member function is marked "const"
            - outside the class, get_value doesn't change the object - not in anyway that the outside world is supposed to be able to detect, so it really is conceptually a const function
                - but to do its work, has to modify certain things in the object that conceptually are invisible to the outside world
        - if don't or can't use "mutable", then this and possibly other member functions can't be "const"
            - Can get into awkward situations - which is why "mutable" is included here
        - Projects in this course probably won't use "mutable" - tends to show up when some kind of caching sort of thing is being done; rare otherwise.
        -

- **Using constructors to create objects with values directly**
  - *Example code is CoinMoney_using_ctors.cpp*
  - *assumes either previous example - just shows add function*
  - *Three versions of the add function*
  - *First, declares a CoinMoney object as variable sum, then set its members to sum.*
  - *Second, declares sum but uses constructor to set its members at the same time.*
    - remember how constructor was defined and used?
  - *Third, looks odd - but very convenient*
    - remember that in C++ you can declare a variable at the point where you first use it.
    - in some situations, the variable doesn't need to have a name ... this looks like a call to a constructor function, but actually it is a declaration of an unnamed CoinMoney object, gets initialized, and then serves immediately as the value for the return statement - gets copied back, then it's gone.
    - Often used when there is no point to bothering with a variable with a name.
    - All three forms do the same thing; second and third are more efficient.
      - First - allocate memory, do the default initialization, then change values
      - Second, third allocate memory, but initialize immediately with actual values

- **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

- **Operator Overloading -**
  - **Example code is CoinMoney_overloaded_add.cpp**
  - **assumes previous examples - just shows add function**
  - **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.**
    - *m1 + m2*
    - *CoinMoney operator+ (CoinMoney m1, CoinMoney m2) --- l.h.s., r.h.s*
  - **You are not allowed to overload the operator for built-in types**
    - *Can't change what it means to add two numbers!*

- *Too much mischief!*

- **Friend Functions - CoinMoney_friend_add.cpp**
  - **Example code is CoinMoney_friend_add.cpp**
  - **non-member functions do not have access to private data**
  - **normally, use reader/writer access functions**
  - **sometimes having to use 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
  - **in a class declaration, you can declare a function to be a "friend" of a class - grant access to private members.**
  - **several design issues around this, but for now use simple case where justified:**
  - **friend <function prototype> in the class declaration**
- **Usual custom - make functions member functions**
  - **Example code is CoinMoney_member_add.cpp**
  - **don't need to declare friends as much - use only when advantage**
  - **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.**
- **Overloading the output operator**
  - **Example code is CoinMoney_output_op.cpp**
  - **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.

- **Final example of CoinMoney class - pretty complete**
  - **Example code is CoinMoney_overloaded_ops.cpp or CoinMoney_final.cpp**
  - **All of declaration shown here**
  - **Look at main function**
- **More about constructors**
  - **What do constructors do with member variables?**
    - *If you assign them in a constructor function, then thats what.*
    - *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) {}*
    - *Recommendation : Use now*
    - *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*
    - *e.g. vending machine class*
      - class VendingMachine {
        - VendingMachine()
          - default constructor
        - stuff

        - CoinMoney coinbox;
    - if you construct a Merchant, and don't do anything with the CoinMoney member variable, what happens?
        - VendingMachine()
            - {...}
        - answer: default constructed - all zeros, in our example
    - what if you want something else? VendingMachine constructor can arrange it
    - VendingMachine() : coinbox(1,1,1)
        - {...}
    - 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 and initializes a local "coinbox" which is then tossed away
        - CoinMoney coinbox(n, d, q);
        -
        - // creates an un-named local Coinmoney object which is then tossed away
        - CoinMoney (n, d, q);
        -
        - // compiler doesn't know what you are doing here - syntax is all wrong
        - coinbox(n, d, q);
    - }
- *Best is with a constructor initializer*

- **More about declaring and defining - ordinary functions**
    - **Reminder about declaring and defining regular functions and variables**
    - **You must tell the compiler about a variable before it will accept code that refers to it. Usually declare and define at same point in the code**
        - *int i; // i is an integer, commit to memory space for it now*
        - *later - cases where variable will be declared without defining it also.*
    - **Remember function prototype is a declaration - tells the compiler everything needed to compile code containing a call to the function.**
        - *int foo(double, char);*
    - **Actual function code is the definition - here is what is to be compiled and stored into memory for this function.**
        - *int foo(double, char)*
        - *{*
            - ...
        - *}*

- **Must tell the compiler about the function before it will accept code containing a call of it. You can provide the declaration first, then code calling it, then the definition itself wherever you want, or just the definition first.**
  - *Actually, can even put the function definition in another file altogether, if you wan t- will do later - have to set things up right*

- **More about declaring and defining - member functions and variables**
  - **Class declarations have a somewhat different set of rules.**
  - **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
      };

      void Thing::foo()
      { the code}
    - *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 laster 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.*

- **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 ...*
  - **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.*
- **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*
  - **normal function call**
    - *double foo(double y) { return (y * y + 3.14);}*
    - *....*
    - *x = foo(z) + 5.83;*
    - *push copy of z onto stack*
    - *push function return info on stack*
    - *branch to foo*
    - *compute y * y + 3.14, save in register*
    - *pop stack*
    - *branch to return address*
    - *calculate register value + 5.83*
  - **inline call**
    - *inline double foo(double y) { return (y * y + 3.14);}*
    - *...*
    - *x = foo(z) + 5.83;*
    - *just as if you wrote:*
    - *x = z * z + 3.14 + 5.83;*
    - *calculate z * z + 3.14 + 5.83 (can optimize)*
  - **It is up to the compiler to decide whether it can inline - function can be too complicated**
    - *or impossible - e.g. recursive function!*
  - **so you *request* inline compilation - do if possible**
  - **by default, functions defined inside a class declaration will be inlined (if the compiler can do it)**
  - **by custom, only simple functions put 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, cause 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*
- **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;
        - }
      - CoinMoney int n = 0, int d = 0, int q = 0) : nickels(n), dimes(d), quarters(q)
        - {
          - nickels = n;
          - dimes = d;
          - quarters = q;
        - }
- **OTHER TOPICS**
  - **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) {
        - int n, d, q;
        - is >> n >> d >> q;
        - m.set_nickels(n);
        - m.set_dimes(d);
        - m.set_quarters(q);
        - return is;
        - }
      - cout << "Enter values for x, nickels, dimes, quarters, and i" << endl;
      - cin >> x >> m1 >> i;