- **Operator Overloading, Templates.**
  • H: *A Summary of Operator Overloading*, then read • S 18 and • S 19. Skip 19.2.5, 19.2.6. Skim 19.3.
  • S 23 Templates. Skim 23.5.2, skip 23.5.2.1, 23.5.2.2, 23.7.1

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;

- **Basic Templates**
- **Intro**
  - **C++ is a strongly typed language - there is strict set of rules on what types that variables can have, and when one type can be used as another type.**
    - *e.g. conversion rules:*
      - my_int = my_double;
      - my_int = sqrt (int_var);
      - Thing * = pointer_to_gizmo; // illegal
  - **C++ is also statically typed - types of variables are known and fixed at compile time.**
    - *Enables compiler to generate very fast and efficient code*
    - *Most programming languages work this way.*
  - **Compare to LISP**
    - *Lisp is a language that is dynamically typed; every "variable" can have any kind of value at all - numbers, strings, lists, even code (since code is a list of expressions).*
    - *Every value is actually an object that carries its type with it - so at run time, every operator or function knows what to do with it; it it turns out to be the wrong type, you get a run-time error*
      - Example - playing around with variable values in lisp
        - ```
          (defun example()
          (let (x y z)
          (setq x 5)
          (print x)
          (setq y 10)
          (print y)
          (setq z (+ x y))
          (print z)
          ;;(setq z (append x y)) ;; comment out

          (setq x (list 'a 'b))
          (print x)
          (setq y (list 15 "foo"))
          (print y)
          (setq z (append x y))
          (print z)

          (setq z (+ x y))

          ))

          ;output:

          5
          10
          15
          ```
          ```

> Error: value 5 is not of the expected type LIST.
> While executing: CCL::APPEND-2
> Type Command-. to abort.

commenting out append of numbers
5
10
15
(A B)
(15 "foo")
(A B 15 "foo")
> Error: value (A B) is not of the expected type NUMBER.
> While executing: CCL::+-2
> Type Command-. to abort.

Can't add lists, etc

- *But this run-time checking can be very slow. Statically typed is faster*

- **But strong and static typing has a serious pitfall - impossible to use the same code to work on different types**
  - *Example of how clumsy this can be:*
    - ```
      void swap (int& a, int& b)
      {
          int temp = a;
          a = b;
          b = temp;
      }
      ```
- **Will this work for doubles?**
  - *swap(double_var1, double_var2);*
  - *A conversion from double to int is allowed (though it loses information)*
  - *But the fuction can't be called, because a reference to an int can't be set to refer to a double - same concept as disallowed pointer conversions.*
- **Will this work for C strings?**
  - *No, because pointers will not be converted to integers*
- **What about for string objects?**
  - *No - compiler will reject because an string can't be converted into an integer*
- **Have to write a different version of swap for every type - what a pain!**
- **Code will be fast and efficient, but are we doomed to writing it out over and over again?**
- **Generic Programming and Templates**
  - *Concept of generic programming - writing code that applies to all kinds of types, and letting compiler modify it as needed for the type we want.*
      ```

- *in C++, this is done with TEMPLATES*
  - you write the code using a template, and specifying a TYPE PARAMETER (one or more)
  - The compiler generates the appropriate code for the TYPE PARAMETER when it is needed
- *Concept of the template:*
  - A recipe for the compiler to follow to generate some code for you.
  - Both function templates and class templates

- **C++ templates can be extremely sophisticated**
  - *Std. Lib. uses them very heavily - almost all templates, in fact*
  - *Very fancy template programming is now the cutting-edge concept ...*
  - *But simple use of templates is easy and worth knowing*
    - For your own code
    - To help understand how to use Std. Library code

Function templates

Function template approach:

- *You define the function template*
- *When your code uses the function, the compiler generates the suitable definition of the function - INSTANTIATING the template*
- ***Compiler deduces the relevant types from the type used in the arguments of the call***
  - **A key feature of function templates - very useful in a variety of ways**
  - **Class templates have to have the types explicitly specified!**

- **function templates can be useful - StdLib is full of them, for handy & often used things - later.**

Template example

- *Swapem as a template: T (can be anything) is TYPE PARAMETER*
  - ```
    template <typename T>
    void swapem (T& a, T& b)
    {
            T temp = a;
            a = b;
            b = temp;
    }
    ```
- *these days, new "typename" keyword often used instead of "class" in the template declaration header*
  - the template parameter is the name of a type - always - and might not be a class type!
- *compiler must see the template definition first - before your use of it in code*
  - defined at top level of a file
  - often put in a header file
- *If you write:*
  - swapem(my_int1, my_int2);
  - compiler will generate the code:
    - ```
      void swapem (int&a int& b)
      {
              int temp = a;
              a = b;
              b = temp;
      }
      ```
- *If you write:*
  - swapem(str1, str2); // str1 and str2 are string

- compiler will generate the code:
  - void swap (string& a string& b)
    {
        string temp = a;
        a = b;
        b = temp;
    }

- **Advantage:**
  - *You get to have the benefits of strong static typing*
    - Compiler error checks and warnings
    - Fast run speed
  - *But don't have to write repetitious code.*
- **Additional detail about function templates:**
  - *Can have more than one type parameter:*
    - template <class T1, class T2>
      void print_both(T1 a, T2 b)
      { cout << a << b << endl;}
    - if you write print_both(my_char, my_double);
    - compiler will create and call:
      - void print_both(char a, double b);
  - *After compiler instantiates the template, subject to normal rules of compilation and execution: code must be correct and make sense;*
    - For example
      - suppose class Thing does not have a public assignment operator
      - swapem(thing1, thing2) would fail to compile as a result because the assignment statements would be illegal
        - code example:

```
template <class T>
void swapem(T &a, T &b){
   T temp = a;
   a = b;
   b = temp;
}

class Thing {
public:
     Thing(int i_, char c_) : i(i_), c(c_) {}
     int i;
     char c;
     friend ostream& operator<< (ostream&, const Thing&);
private:
```

```
    Thing& operator= (const Thing& rhs);
};

ostream& operator<< (ostream& oss, const Thing& t)
{
    oss << '[' << t.i << ", " << t.c << ']';
    return oss;
}

int main(){
   Thing thing1(1,'A'), thing2(2, 'B');
   cout << "thing1: " << thing1 << ", thing2: " << thing2 << endl;
   swap(thing1, thing2);
   cout << "thing1: " << thing1 << ", thing2: " << thing2 << endl;
   return 0;
}
```

- main.cpp:19: error: 'Thing& Thing::operator=(const Thing&)' is private

- some template error messages can be confusing, though - lots of room for improvement in current compilers!
- g++ is actually among the better ones - parse it apart patiently - it tells you everything

- Other exaple - what does the instantiated code actually do?
  - char s1[20] = "Hello";
  - char s2[20= " Goodbye";
  - swapem (s1, s2); //?? allowed?
  - char * p1 = s1;
  - char * p2 = s2;
  - swapem (p1, p2); ???
    - this swaps the pointers, but not the strings!
  - how would you swap the contents of the two strings?
    - swapem(char * s1, char * s2); ???

- **What rules does the compiler follow to instantiate vs. when to use other overloaded functions:**
  - *First, compiler looks for exact type match with non-template function*
    - e.g. swapem(char * s1, char * s2);
  - *Second, a directly applicable template*
  - *Third, do ordinary argument conversions on a non-template function*
    - e.g. print_both(int, int)

A simple variadic template example

- **A function template that will call another function based on any number of arguments**
- **Basis for some very useful templates, like std::make_shared**
-
```
// a function template that calls the relevant function
// named "f" that accepts the supplied arguments as a parameter pack
template<typename... Args>
void callit(Args... args)
{
    cout << "\nin callit with parameter pack of size " << sizeof...
(args) << endl;
    f(args...); // the version of f that matches the parameter pack
}

// a set of overloaded functions
void f(int i)
{
    cout << "f(int) called with " << i << endl;
}

void f(int i, int j)
{
    cout << "f(int, int) called with " << i << ' ' << j << endl;
}

void f(int i, double d)
{
    cout << "f(int, double) called with " << i << ' ' << d << endl;
}

void f(int i, const string& s)
{
    cout << "f(int, string) called with " << i << ' ' << s << endl;
}

void f(int i, int j, double d, const string& s)
{
    cout << "f(int, int, double, string) called with "
        << i << ' ' << j << ' ' << d << ' ' << s << endl;
}

int main()
{
    callit(42);
    callit(1, 2);
    callit(1, 2.2); // int double is preferred overload
    callit(1, string("Hello"));
    callit(1, "Hello"); // string literal converts to string variable
    callit(1, 2, 3.14, "Goodbye");
//  callit(1, 2, 3); // compile fails because there is no matching
function
```

```
//  callit(string("Hello")); // compile fails because there is no
matching function
}

/* output:

in callit with parameter pack of size 1
f(int) called with 42

in callit with parameter pack of size 2
f(int, int) called with 1 2

in callit with parameter pack of size 2
f(int, double) called with 1 2.2

in callit with parameter pack of size 2
f(int, string) called with 1 Hello

in callit with parameter pack of size 2
f(int, string) called with 1 Hello

in callit with parameter pack of size 4
f(int, int, double, string) called with 1 2 3.14 Goodbye
*/
```

-

Common example of variadic templates

- ```
  /* Demonstrate a simple use of variadic templates to implement
  a simple function that outputs all of its arguments which can be
  any number and type. Also shown is a simple wrapper that calls the
  same function given any number and type of arguments.
  */

  #include <iostream>
  using namespace std;


  // an no-parameter "last" version of print in the "recursive"
  instantiation
  void print()
  {
      cout << endl;
  }

  // a parameter-taking "last" version which prints a '$' before its output
  // to show when it gets called.
  // if present, this one is called in preference to the no-parameter
  version
  // so comment it out to see the no-parameter version at work.
  template <typename T>
  void print(const T& arg)
  {
      cout << '$' << arg << endl;
  }


  // the "recursive" version of print
  // it prints the size of the args list between < and > as part of the
  output
  template <typename T, typename... Ts>
  void print(const T& firstArg, const Ts&... args)
  {
   //  cout << '<' << sizeof...(args) << '>' << firstArg;
      cout << firstArg << ' ';
      print(args...);
  }

  // a demonstration of a wrapper for a variadic template
  template<typename... Args>
  void zap(Args... args)
  {
      print(args...);

  }

  int main()
  {
      print(7.5, "hello", 42);
  ```
  ```

```
    zap("hello", 42, 7.5, "zap");
}

/* output with no-parameter version of "last" print:
7.5 hello 42
hello 42 7.5 zap
*/

/* output with no-parameter version of "last" print,
with the sizeof... output present in the "recursive" version
notice how the sizeof... is zero at the end

<2>7.5<1>hello<0>42
<3>hello<2>42<1>7.5<0>zap
*/


/* output with parameter version of "last" print,
with the sizeof... output present in the "recursive" version
showing '$' followed by last element:

<3>7.5<2>hello<1>42$
<4>hello<3>42<2>7.5<1>zap$
*/
```

Class templates

- **A class template is a class definition in which member variables have parameterized types**
  - *e.g. Ordered_list of Player *, String*
  - *e.g. List of doubles, Strings, Ordered_lists, etc.*
- **Class templates are extremely useful for container classes**
  - *Gives generic but type-safe containers*
  - *Java has a quasi-template concept as a result - but not statically typed.*
- **How to create a class template:**
  - *Build a class that has oridinary member variable data types*
  - *Make sure it works right.*
  - *Change the relevant data types to template type parameters.*
  - *Instantiate by giving the types*
  - *There you go!*
- **micro example of class template:**
  - *start with*
    - ```
      class Thing {
              int x;
              double y;
              void defrangulate() {/* incredibly complex code */}
              };
      ```
  - *After fully debugging it, change to*
    - ```
      template <typename T1, typename T2>
      class Thing {
              T1 x;
              T2 y;
              void defrangulate() {/* incredibly complex code */}
              };
      ```
  - *use by:*
    - Thing<int, double> thing1;
      - compiler generates:
      - ```
        class Thing {
                int x;
                double y;
        ```
        ```

```
void defrangulate() {/* incredibly complex code */}
};
```

- Thing<String, Item> thing2;
    - compiler generates:
    - ```
      class Thing {
              String x;
              Item y;
              void defrangulate() {/* incredibly complex code */}
              };
      ```

- **The name of a template class:**
    - *classname<typeparameter>*
    - *classname<sometype> when instantiated*
    - *e.g. Ordered_list was originally a non-template class that was a smart array of ints*
    - *now, a template class Ordered_list instantiated with ints is named:*
    - *Ordered_list<int>*
    - *must use this name everywhere we would have used the plain name before.*
- **Defining class template member functions**
    - *Every member function of a class template is a function template!*
        - Even for ordinary classes, you can have member functions that are template functions!
            - Occasionally *very* handy!
    - *Member functions defined inside the class declaration - no problem, same as non-template classes*
    - *Member functions defined outside the class declaration -*
        - Class name becomes the template class name in template form:
    - *Simple example:*
        - definition inside
            - ```
              template <typename T> class Thing {
              void foo() {
                      blah;
                      blah;
                      }
              };
              ```
        - definition outside:

- ```
  template <typename T> class Thing {
  void foo();
  };

  void Thing<T>::foo() {
          blah;
          blah;
          }
  ```

- **How about class templates that use other class templates : no problem:**
  - ```
    template <typename T>
     class Thing {
            T data_var;
            list<T> data_list
    };
    ```

- **How about default parameters for class member functions that are templated types? Can do:**
  - ```
    template <typename T>
     class Thing {
            Thing (SomeType initial_value = Gizmo<T>)  // as long as SomeType can be initialized with a Gizmo
    };
    ```

- **How about member functions that have an additional template type parameter? Can do, just a nested sort of template declaration:**
  - *Looks odd, but it is correct*
    - ```
      // define inside the class declaration:
      template <typename T>
       class Thing {
              template <typename OT>
              void foo(OT ot)
              {
                      blah;
                      blah;
              }
      };

      // define outside the class declaration:
      template <typename T>
       class Thing {
              template <typename OT>
              void foo(OT ot);
      };
      template <typename T>
      template <typename OT>
      void Thing<DT>::foo(OT ot)
      {
              blah;
              blah;
      }
      ```

- **Template Magic Trick #1 Using a function template to infer types in creating a class template**
  - *Suppose we have*

- template <typename T1, typename T2>
  class Thing {
  public:
  Thing (T1 x_, T2 y_) : x(x_), y(y_) {}
  private:
  T1 x;
  T2 y;
  };

- *We want to instantiate it as an unnamed object with int, double and initialize it, say to give it to another function. Have to write:*
  - foo(Thing<int, double> (42, 3.14));
- *Writing out the class instantiation parameters can be inconvenient, but can't be avoided with class template - we have to specify the types. However, suppose we write the following function template:*
  - template <typename T1, typename T2>
    Thing<T1, T2> make_Thing(T1 t1, T2 t2)
    {
    return Thing<T1, T2> t(t1, t2);
    }
- *Now we can create and initialize our template class object and let the compiler deduce what T1 and T2 are from the function arguments:*
  - foo(make_Thing(42, 3.14));
- *Common pattern in the Standard Library: a function template that uses type deduction of parameters to instantiate and return a class template object - many facilities come in pairs of templates: the instantiating function and the class object.*
- *e.g.*
  - std::make_pair<int_var, my_string> creates and returns std::pair<int, std::string> initialized with int_var and my_string.

What about static member variables?

- *Yes, you can have them, but …*
  - ```
    template <typename T>
    class Thing {
         T x;
         static int counter;
         void defrangulate() {/* incredibly complex code */}
         };
    ```
  - ```
    template<typename T>
    int Thing<T>::counter = 0;  // initialize it
    ```
- *There is a different static int counter for each instantiation of T ! - Shared with the same T Things, but not shared between different T Things.*

Important issues about Class templates

Major Practical Issue: How the compiler processes templates

- *Compiler must see the complete template definition for every translation unit that makes use of the template.*
- *Standard practice: put the complete template definition in a header file.*
  - Both classes and member functions of those classes
  - Compiler/linker work together to avoid/handle duplicated definitions with templates
  - E.g. to use Ordered_list<> template, #include Ordered_list.h
- *Potentially very awkward - header files can get very long.*
  - Standard Library - iostream is actually a monster set of templates - almost all of the I/O library is actually being read in, in near source form
  - Why - makes it easy for the same code to be used for both normal and wide characters!
    - Not of a lot of use to us, though!
  - It is possible to separate code into .h and .cpp files, but is not done very often, and is not as flexible - see Stroustrup p.696 ff
    - for example, put declaration in .h, function definitions in .cpp followed by explicit instantiations, compile the .cpp along with all other .cpp.
    - I've done this: Is only a good solution when you know the possible instantiations in advance:
    -
  - Future compilers may make it better - "export" keyword was supposed to help
    - But actually, export is not as good an idea as everybody was expecting!
- *Basic distinction: point of instantiation versus when instantiated.*
  - The point of instantiation is where your code requires an template to be instantiated.
  - However, compiler processes all of the code in the translation unit, then instantiates the templates, then compiles those.
  - It usually reports errors at the point of instantiation, but it is happening after the non-template code has been compiled.
  - Allows for use of incomplete types at the point of instantiation if they become complete types later in the translation unit.

- **Typedef and type aliases with templates**
  - *using mytype = existing_type;*
  - *usually equivalent to a typedef, but more flexible with templates:*
    - tempplate<typename T>
      using Vector = std::vector<T>;
    - /* template <typename T>
      typedef Ordered_list<T> myOL; // error typedef can't be a template

```
*/
   template <typename T>
   using myOL = Ordered_list<T>;

   template <typename T>
   myOL<T> foo(myOL<T> x)
   {return x;};
```

- **Dependent types - occasional issue**
  - *Suppose you are writing a template with T as the type parameter*
    - template <typename T>
  - *and somewhere in the middle of it you refer to "foo" that is in the type given by T*
    - T::foo
  - *the type of foo depends on T - it is a dependent type.*
  - *What is foo? Compiler can't tell just from T::foo because it doesn't know what T is yet.*
  - *On certain occasions, the compiler will complain because of the ambiguity. Usually foo should be the name of a type embedded in T (like a nested class or a typedef). Compilers used to just assume it, bu it could be something else - like a static variable or a member function.*
  - *If the compiler is confused, and foo is the name of a type, you need to tell the compiler with the typename keyword:*
    - typename T::foo
    - "foo" is the name of a type declared within the scope of T
- **Library implementers: Preventing code bloat for template classes containing pointers**
  - *Code bloat: every template instantiation is a complete copy of the code, differing only in the type declarations.*
  - *E.g. Linked_list<Thing>, LInked_list<int>, Linked_list<char*> - 3 "copies" of the same code*
  - *Can't do anything about this, but there is a special case for pointer types:*
  - *E.g. Linked_list<Thing*>, LInked_list<int*>, Linked_list<char*>*
  - *Instead of 3 copies differing only by pointer type, implement in terms of void*, with casts to/form the actual type*
  - *How its done (sketch - many details and members left out) - provide these three templates in this order to the compiler*
    - First, provide the complete base template - the normal template
      -
        ```
        template<typename T>
        class Linked_list {
        void insert(const T& datum);

        class Iterator {
        ```

```
T& Iterator::operator* ()
     {return node->datum;}
     };

private:
     class Node {
          T datum;
     };
};
```

- Then specialize the whole template for void* - compiler picks this for Linked_list<void*> instead of the base template

- ```
  template<>
  class Linked_list<void*> {

  void insert(void* datum);

  class Iterator {
       void*& operator* ()
            {return node->datum;}
       };

  private:
       class Node {
            void* datum
       };

  };
  ```

- Then partially specialize for prointer types, and implement in terms of the void* instantiation. with casts to/from the T* type and void*

- ```
  template<T*>
  class Linked_list {
  public:
  void insert(T* datum)
       {vplist.insert (static_cast<void*>(datum));

  class Iterator {
       T*& operator* ()
            {    …
            return static_cast<T*>the_datum  // very sketched
            }
  private:
       Linked_list<void*> vplist;
  };
  ```

- Note: If member functions are inline, the “delegation” of calls to vplist takes no run-time.

- Advantages:
  - All Linked_lists of pointer type share the same run-time code (the Linked_list<void*>).
  - Less code bloat if you have a lot of containers of different pointer types.
- Disadvantages:

- Lots of code near-duplication (more or less whole template for void* specialization)
- the casting to/from void* can get very complex when taking into account: const interators, const containers, and that T might be a const type (e.g. T* is const char *).
- Definitely a job for professional library writers!!!