- **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**
    - *Swap as a template: T (can be anything) is TYPE PARAMETER*
      - ```
        template <typename T>
        void swap (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:*
      - swap(my_int1, my_int2);
      - compiler will generate the code:
        - ```
          void swap (int&a int& b)
                {int temp = a;
                a = b;
                b = temp;
                }
          ```
    - *If you write:*
      - swap(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
      - swap(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";
  - swap (s1, s2); //?? allowed?
  - char * p1 = s1;
  - char * p2 = s2;
  - swap (p1, p2); ???
    - this swaps the pointers, but not the strings!
  - how would you swap the contents of the two strings?
    - swap(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. swap(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)

- **Class templates**
  - **See Smart_array example**
  - **A class template is a class definition in which member variables have parameterized types**
    - *e.g. Ordered_array of Player *, String*
    - *e.g. List of doubles, Strings, Ordered_arrays, 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 */}
};
```

-
- **Important issues about Class templates**
  - **The name of a template class:**
    - *classname<typeparameter>*
    - *classname<sometype> when instantiated*
    - *e.g. Ordered_array was originally a non-template class that was a smart array of ints*
    - *now, a template class Ordered_array instantiated with ints is named:*
    - *Ordered_array<int>*
    - *must use this name everywhere we would have used the plain name before.*
  - **How about class templates that use other class templates : no problem:**
    - ```
      template <typename DT>
       class Thing {
              DT data_var;
              Ordered_array<DT> data_array;
      };
      ```
  - **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 <class T> class Thing {
          void foo() {
                  blah;
                  blah;
                  }
          };
          ```
      - definition outside:
        - ```
          template <class T> class Thing {
          void foo();
          };

          void Thing<T>::foo() {
                  blah;
                  blah;
                  }
          ```
  - **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_array<> template, include Ordered_array.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
  - Future compilers may make it better - "export" keyword was supposed to help
    - But actually, looking like export is not as good an idea as everybody was expecting!
- **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