Basic Templates

Intro
- C++ is a strongly typed language: there is a strict set of rules on what types 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; if 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
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 function 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 a 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:
      ```cpp
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 example - 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)
• **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 ordinary 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.

• **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.
      
      • 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:*
      
      • template<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, but
    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