Chapter 41

Templates

This section is about a shortcut letting us use a variable for a type. We can declare `T val;` and have `T` be a type to be filled-in later. It's used in things like `List<int>` or Unity's `GetComponent<Rigidbody>()`.

There are two versions of it: using it in a class or in a function.

You probably won't use this shortcut in your own code. But there are built-ins using it, and it's nice to understand how they work.

41.1 Template functions

To start with, here's a function that creates an array filled with one number. There's nothing special about it, yet:

```
float[] makeArray(int size, float value) {
  float[] Result = new float[size];
  for(int i=0;i<Result.Length;i++) Result[i]=value;
  return Result;
}

// sample call:
float[] N = makeArray(10, 1.23f); // [1.23, 1.23, 1.23 ... ]
```

An interesting thing is we're not using the "numberness" of the second input. It could be a string or int with no changes.

The template trick lets us actually do that. After the function name, add `<T>`. The angle brackets are required, but the `T` is any name you pick. All of the other `T`'s are filled in for you:

```
T[] makeArray<T>(int size, T value) {
  T[] Result = new T[size];
  for(int i=0;i<Result.Length;i++) Result[i]=value;
```

```
  return Result;
}
```

We can call it with `makeArray<string>(3,"tree");`. That turns all the `T`'s into string's. Where it says `T value` for the input, that's now `string value`. And the `T[]` for the return type is now `string[]`. The answer is `[tree, tree, tree]`. Other sample calls:

```
float[] N = makeArray<float>(10, 1.23f); // same as the 1st example
string[] Fives = makeArray<int>(8, 5); // [5,5,5,5,5,5,5,5]
Cat[] CatList = makeArray<Cat>(3,c); // [c, c, c]
```

Note that the last one, with the cats, will have every entry pointing to that one `c`, since that's how `=` works with pointers.

A way to think of them is that the compiler pre-makes the ones you need. If your program has `makeArray<string>` anywhere, the compiler creates that function from the template, with all the `T`'s filled in with `string`. In other words, this really is a template, used to create several copies of itself with various types.

In `C#` template functions are very limited. Other languages allow more. This, which isn't legal in C#, would check whether 3 values are the same:

```
bool allSame<T>(T a, T b, T c) {
  if(a==b && b==c) return true;
  return false;
}

// sample uses:
bool b1 = allSame<string>("cat", "cat", "dog");
if( allSame<float>(F[0], F[1], F[2]) ) ... // compare three floats from array
```

Notice how it has three inputs, but there's one `<T>`. That's on purpose - `(T a, T b, T c)` says they're three things of the same type – the single type you gave it.

These next two just show the rules. This one requires a type, but never uses it, which is legal but pointless:

```
void sayMoo<T>() { print("moo"); }

// sample calls (both print moo):
sayMoo<Cat>(); // moo
sayMoo<float>(); // moo
```

You can have more than one type - put them in the angle brackets with commas. As usual, if you think of a descriptive name, depending on what the function does, use that. I picked just `T1` and `T2` for this:

```
void printStuff<T1,T2>(T1 a, T2, b, T1 c) {
  print(a+" "+b+" "+c);
}
```

This says the first and last have to be the same types, since they both use `T1`. the middle one could be the same, or could be different. Ex:

```
printStuff<float, string>(3.0f, "goat", 9.3f);
printStuff<string, float>("a", 8, "b");
printStuff<int, int>(1,2,3); // <- T1 same as T2 is legal
```

The loosest form of template functions, which is very not legal in `C#`, is where you use whatever member functions you want, and can use any class that has them.

This takes a list of any type and tries to use `c1.lessThan(c2)` to find the small ones. Any class with a lessThan member function can use it:

```
List<itemType> getSmallItems<itemType>(List<itemType> L, itemType maxVal) {
  List<itemType> Ans = new List<itemType>();
  for(int i=0; i<L.Count; i++) {
    itemType itm=L[i];
    if(itm.lessThan(maxVal)) // using dot-lessThan
      Ans.Add(itm);
  }
  return Ans
}
```

If you called tried to call this with a List of `Cat`, it wouldn't compile. But if you added `bool lessThan(Cat c)` as a member function, it would.

This is a substitute for interfaces and inheritance. You just pick some member functions, like `lessThan` or `a.CompareTo(b)` and write template functions using them. Anyone who wants to use them can add those functions to their class.

41.1.1 Implicit types

If the compiler can guess it, you can often call a template function with the `<T>` left out. For example, in `makeArray(3, 5.6f)` the computer can tell that you meant `makeArray<float>` since you used `5.6f`.

This means that overloads and template functions can blur together.

Suppose you see `doThing("gorilla")` and `doThing(6)`. That could be overloading (they wrote a string and int version of the function.) Or it could be a template, `thing<T>(T a)`, being called with the shortcut (the calls were really `thing<string>("gorilla")` and `thing<int>(6)`).

Often we purposely mix overloads and templates with the same name. Not to be confusing, but because it seemed like the best way to make it feel like one

function, usable by lots of different types.

Programming-wise, sometimes you need to remember the real way. You may call `doThing(c1);` and get an error. The long way should work: `doThing<Cat>(c1);`.

41.2 Template classes

Classes with variable types work roughly the same way. You add `<T>` in the definition, which the user must fill in with a type. Then everywhere inside the class you can use `T`.

41.2.1 Tuple examples

This makes a simple class that can hold two items of any type. As usual, the words that stand for the types don't matter, but it's traditional to use a capital letter if you can't think of anything else:

```
class Pair<S, T> {
  public S val1;
  public T val2;
}
```

This can make any 2-item-holding class by filling in the types:

```
Pair<string, int> p1;
  ...
p1.val1="toad"; p1.val2=5; // a string and an int

Pair<float, float> ff;
 ...
ff.val1=3.2f; ff.val2=2.9f; // both are floats
```

I left out the part where we `new` them. You have to use the full name of the type: `p1 = new Pair<string, int>();` and `ff = new Pair<float, float>();`.

But that shows how the computer thinks these are different. `Pair` isn't a class, yet. It can be used to make lots of classes, with the `<>`'s.

`Pair` is useful. It's commonly called a tuple. It's in lots of languages as a quick way to group two values.

Suppose we want to get the integer and fractional part of a number. We can return them both using a `Pair<int,float>`. This is a regular function:

```
Pair<int,float> getIntAndFraction(float f) {
  // do the math:
  int wholeNum=(int)f;
```

```
  float fraction=f-n;
  // pack in into a Pair:
  Pair<int,float> p = new Pair<int,float>();
  p.val1=wholeNum; p.val2=fraction;
  return p;
}
```

Now we could call `Pair<int, float> nn = getIntAndFraction(4.72f);`. The result would be `nn.val1` is 4 and `nn.val2` is 0.72.

Notice how that is not a template function. It runs using only one exact type (which happens to be a template class).

Sometimes we use tuples as a shortcut for writing a real class. Suppose we want a list of things like: 2 cats, 6 dogs, and so on. We could write a class with name and amount, but `Pair<string,int>` is fine.

We could make an array of them:

```
// An array where each entry is like: (Cat, 2)
Pair<string,int>[] AniCount = new Pair<string,int>[6];
AniCount[0] = new Pair<string, int>();

AniCount[0].val1="ferret"; // <- first item is (ferret, 7)
AniCount[0].val2=7;
```

As a time-saver and neat example, we could make a function that creates pairs. This is a template function, which creates a template class:

```
Pair<S,T> makePair<S,T>(S v1, T v2) {
  Pair<S,T> pp = new Pair<S,T>();
  pp.val1=v1; pp.val2=v2;
  return pp;
}
```

The key to reading it is `makePair<S,T>`. The angle-brackets after the function name are, as usual, the one place we pick the types. Everything else is filled in. But it makes sense when you see it in use:

```
Pair<int, int> p1 = makePair<int,int>(6,9);
// p1.val1 is 6, p1,val2 is 9

Pair<float,string> p2 = makePair<float, string>(4.1f, "cat");
```

The types we write in the brackets determine the required types of the inputs, and the type of the returned Pair.

Since we're giving it the types as input, we get to use implicit types in the call: `makePair("cat", 2);` is a legal shortcut. Here's more of the animal list:

```
AniCount[1] = makePair("cat",2);
AniCount[2] = makePair("rat",12);

print( AniCount[1].val1 ); // cat
```

Backing up, this is a typical case where each part is simple, but the whole thing is confusing since we're doing so much as once: we need a `new`, but `makePair` does it for us; we're using the implied types shortcut in `makePair`, and we're putting a template class in a array.

This next normal function isn't anything new – just more practice. It tells us how many there are of a certain animal:

```
int getAnimalCount(Pair<string,int>[] A, string animal) {
  for(int i=0; i<A.Length; i++)
    if(A[i].val1==animal) return A[i].val2;
  return 0; // couldn't find, means there are 0 of that animal
}
```

This is a pretty simple function. If we had a real class then the `if` would look nicer, like: `if(A[i].name==animal) return A[i].count;`. But with tuples we're always stuck using non-descriptive names like `val1` and `val2`.

41.2.2 More oddball examples

You're allowed to nest template classes. This makes a very crude triple class by using a Pair as the first thing in a pair:

```
Pair<Pair<string,int>,string> Z = new Pair<Pair<string,int>,string>();

Z
------------------
|val1:| val1: frog
|     | val2: 7
|     ---------
|val2: toes
------------------
```

The construction and the picture is no different from any other nested struct, like a Cow holding a Color. But it looks extra bad because of the terrible names. And don't think of it as a Pair inside of a Pair – instead it's a simple class made from Pair, used to make a more complex type of Pair.

Lines using it (to finish the picture):

```
Z.val1.val1="frog";
Z.val1.val2=7;
Z.val2="toes";
```

For real, we'd probably not do this, using either a special class (with better variable names and useful member functions.) Or we'd at least have `Triple<string,int,string>`. But maybe we need to frequently split out the first two, so having them in a Pair is handy.

When you see lots of nested template classes, it can be difficult to know whether there was a good reason to do it.

Here's another simple function using a Pair to return two values. You give it a letter and it searches your array for the first word with it. It returns the index of that word *and* the index in the word where the letter was:

```
Pair<int, int> findWithLetter(string[] S, letter ch) {
  Pair<int,int> Answer = new Pair<int,int>();
  Answer.val1=-1; // not found
  for(int i=0; i<S.Length; i++) {
    int pos=S[i].IndexOf(ch);
    if(pos>=0) { Answer.val1=i; Answer.val2=pos; break; }
  }
  return Answer;
}
```

Some sample calls, to explain how it works:

```
string[] S1 = {"goat", "pony", "rat"};
Pair<int,int> Ans = findWithLetter(S1,'t'); // returns [0,3]
Ans = findWithLetter(S1,'n'); // returns [1,2]
```

Notice how this function only finds letters in lists of strings. It's not a template function.

Member functions in template classes are allowed to use the pre-set types. They don't need extra `<>`'s after the name. Here's a working `set` function for Pair:

```
class Pair<S, T> {
  public S val1; // no change
  public T val2;

  public void set(S v1, T v2) { val1=v1; val2=v2; }
}
```

`set` says that you have to use it with the types you said you were going to use. Some sample use:

```
Pair<string,int> hh = new Pair<string,int>();
hh.set("hamster",4); // <- hh knows it takes string and int

Pair<float, float> ff = new Pair<float,float>();
ff.set(4, 9.876f); // <- ff knows it takes 2 floats
```

All-in-all: template classes have you set the types once, in the declaration. Everywhere inside, even member functions, you can use the "type variables". They'll be filled in with the real types.

Moving on, suppose we want to sideways combine two lists. We want [3, 8, 9] combined with [cat, owl, ant] to get a single list with [(3,cat), (8,owl), (9,ant)].

The input lists can be of any types, which means we need a template function. Obviously the new list will use a Pair to make the items. It will get its types from the original 2 arrays.

Here's the side-by-side array combiner. Remember that we only fill in `<S,T>` after `zipArrays`. The rest of the `S`'s and `T`'s are copied from those:

```
Pair<S,T>[] zipArrays<S,T>(S[] L1, T[] L2) {
  // use length of shorter array:
  int len=L1.Length; if(L2.Length<L1.Length) len=L2.Length;
  Pair<S,T>[] Result = new Pair<S,T>[len]; // result array
  for(int i=0;i<len;i++) {
    Result[i]=new Pair<S,T>();
    Result[i].val1=L1[i]; // copy from the side-by-side array boxes ...
    Result[i].val2=L2[i]; // ...into val1 and vals2
  }
  return Result;
}
```

To review, that's a template function, which uses it's inputs to define a specific template class. To make it worse, it can guess `<S,T>` from the array inputs. A sample call:

```
int[] N={3,8,9};
string[] W={"cat","owl","ant"};

Pair<int,string> NW = zipArrays(N, W); // really zipArrays<int,string>(N,W)
print( NW[2].val2 ); // ant
```

Despite how messy it looks, it's pretty cool to be able to jam just any two arrays side-by-side like that.

This final function is the backwards version of that (well, sort of). It takes an array of some Pair and gives you the first part, cut out from it (it could turn `NW`, above, back into `N`):

```
// input is a Pair array. Output is a simple array of all val1's:
S[] ripOutFirst<S,T>(Pair<S,T> anArray) {
  S[] Ans = new S[anArray.Length];
  for(int i=0; i<anArray.Length; i++)
    Ans[i] = anArray[i].val1; // only 1st values go into the answer
  return Ans;
}
```

Hopefully that first line makes a little sense now. The input is a Pair array, of any types. The output is a simple array, using part 1 of the Pair (since the S's match, not the T's).

41.3 Templates and container classes

A class that just holds items, acting as a nicer or better array, is usually called a *container* class. We've used `List<int>` before. It's just an ordinary template container class.

Here's part of the `List` class written out:

```
class List<T> {
  T[] theArray; <- the entire point of this class is a nice array front-end
  int size=0; // how much of the array we're really using

  public void Insert(int index, T newItem) { ... }
  public int IndexOf(T item) { ... }
  public T ElementAt(int index) { ... } // <- I made this one up
  public void RemoveAt(int index) { }
  ...
}
```

Nothing here is new. `List<T>` says there's no class named `List` – you need to supply a type to complete it.

Inside we know the `T`'s are all copies of the one you declared it with: `T[] theArray;` is an array of string's if you declared `List<string> W;`.

The member functions use the regular template class rule: they know the `T` from when you made it. When you read them, every `T` means "whatever type is in the List."

Another fun created template container class is a Map, which `C#` calls a dictionary. It takes two types. First, an example:

```
Dictionary<string, float> AniWt = new dictionary<string,float>();
AniWt["zebra"] = 5.5f;
AniWt["wombat"] = 3.25f;

print( AniWt["zebra"] ); // 5.5
```

If you read the tooltip, it says `<TKey, TValue>`. Those names are a hint: we can use the first type as a look-up (keys is another word for that), to get the stored values.

From our old rules, we know the two types are allowed to be the same, so we can map animals to the sounds they make:

```
Dictionary<string, string> AniSound= new dictionary<string,string>();
AniSound["duck"] = "quack";
AniCount["cow"] = "moo";

// example of a real look-up:
if(AniCount.ContainsKey("unicorn"))
  print( "Unicorn says " + AniSound["unicorn"] );
else
  print("Unicorn is silent");
```

Maps are strange containers. They don't remember the order you added things, and are terrible if you want to look at everything with a loop. Their only use is for look-ups.

Even if you have int's for the look-up, Maps are good when you have numbers from all over, but not too many. Suppose we have a few hundred ships, with ID numbers from negative to positive a billion:

```
Dictionary<int,string> D = new Dictionary<int,string>();
D[4]="frigate"; // so far an array could do this
D[100278]="yacht";
D[-1093]="pinnace";

// look up an ID:
string shipName="none";
if(D.ContainsKey(shipID)) shipName=D[shipID];
```

That last line is a quick, relatively simple way to check a ship ID. In this case, a Dictionary is better than an array.

Template container classes can have nested templates. This converts the previous "array of name/count" example into a List of Pairs:

```
List<Pair<string,int>> ACount = new List<Pair<string,int>>();
ACount.Add( makePair("cat",2) ); //<- can re-use makePair
ACount.Add( makePair("ferret",7) );
```

You could read `List<Pair<string,int>>` as "a list of string-int pairs".

Here's a more complicated, but semi-common, example of nested template classes. It lets you look up any word and get a list of numbers for it:

```
Dictionary<string,List<int>> AA = new Dictionary<string,List<int>>();

// Make a simple [1,3,8] list:
List<int> ff = new List<int>(); ff.Add(1); ff.Add(3); ff.Add(8);

AA["farm"]=ff; // "farm" is [1,3,8]
```

We can even play compound type tricks with this. `AA["farm"].Count` is 3, and `AA["farm"][2]` is 8 (the last thing in the list).

41.4 Unity template functions

We've been using `GetComponent<Renderer>()` in Unity3D as the first part of color changing. That's obviously a template function. A review of it in use:

```
Renderer rr = GetComponent<Renderer>();
rr.material.color = Color.red;

// with a script we wrote:
catMoveScript cs = cat1.GetComponent<catMoveScript>(); // look-up
c1.name="Soxs"; // now use it
```

It works in a strange way. There's a secret list of `Componenet`'s. Pretend it's named `C`. Component is a base class, and Renderer, scripts and anything else we might search for, is a subtype.

`GetComponent<T>` search the list, checking for the subtype you asked for:

```
// template function: no inputs, returns type you asked for, or null:
T GetComponent<T>() {
  // look at every item in component list:
  foreach(Component c in C) {
    // is it the correct type:
    T cc = c as T; // dynamic cast
    if(cc!=null) return cc;
  }
  return null;
}
```

This is the first template function we've seen that also does stuff with inheritance.

The other common Unity function using a template is Instantiate. A review, this uses Instantiate with 3 different types:

```
public GameObject treePrefab;
public Transform rockPrefab;
public Rigidbody fallingRockPrefab; // hooked up through the rigidbody

void Start() {
  GameObject newTree = Instantiate(treePrefab);
  Transform newRock = Instantiate(rockPrefab);
  RigidBody newFallrock = Instantiate(fallingRockPrefab);
}
```

The way it does this is with a template for the type, used implicitly. The heading for Instantiate:

```
T Instantiate<T>(T copyMe) { ... }
```

When you call, `Instantiate(rockPrefab)`, which is a Transform, the system figures out you meant to use `Instantiate<Transform>`, which means it knows to return a `Transform`. There's more sneaky stuff going on inside, but templates are how it's magically able to return the type you gave it.