40. Function Templates

Objectives

- Write function templates
- Make template function instantiation
- Write function template specialization
- Write header file for templates

We now come to a very important area in modern C++ programming. The concept of templates. There are three types of templates

- Function templates
- Struct templates
- Class templates

For this chapter I'll focus on function templates.

C++ template programming is a very important technique in C++ programming because templates produces code. When you are writing C++ templates, you are not just writing code – you are writing code that produces code.

Modern C++ libraries are frequently written using templates. This is the case for general algorithms and data structures, scientific computation, computer vision, linear algebra, AI, etc.

This is only going to be a short introduction to function templates. C++ templates is a very vast subject, with the C++ standards committee adding new features to templates every few years.

OK. Let's go.

An example of a function template

Function templates are great ... really. It's perfect for people who want to write less code. Let's try an example.

Run this program.

```
#include <iostream>

void println(int x)
{
    std::cout << x << std::endl;
}


int main()
{
    println(1);
    println(3.14);
    println('c');
    return 0;
}
```

When you call `println(3.14)`, C++ will try to find a function that best fit the value passed in (i.e. a `double`). The closest is our `println()` function but it accepts as `int`. So C++ typcast the `double` 3.14 to an `int`, i.e. 3, and call `println()`.

Let's say we do not want C++ to typecast. So here's the program we want:

```
#include <iostream>

void println(int x)
{
    std::cout << x << std::endl;
}


void println(double y)
{
    std::cout << y << std::endl;
}


void println(char z)
{
    std::cout << z << std::endl;
}


int main()
{
    println(1);
    println(3.14);
```

```
    println('c');
    return 0;
}
```

Boy ... what a pain!!! Furthermore the functions all look pretty similar. Smart people hate to do silly things like this: mindless duplication of code.

OK. Before you swear at C++, rewrite this program like this:

```
#include <iostream>

template < typename T >
void println(T x)
{
    std::cout << x << std::endl;
}


int main()
{
    println< int >(1);
    println< double >(3.14);
    println< char >('c');
    return 0;
}
```

WOW!!! There is only one function and it seems to work for all three cases!!!

The “function”

```
template < typename T >
void println(T x)
{
  ...
}
```

is called a **function template**. It's actually not a function. It's like a rubber stamp for producing functions. You should think of the **T** as a **type variable** for the compiler. Here's how the compiler use this `T`. When the compiler see this statement in your program:

```
...
  println< int >(1);
...
```

it (i.e., the compile) says: “Aha! So you need `println< int >`. OK I'll take the rubber stamp `println` (i.e. the function template) and create one with **T** replaced by **int**.”

In other words C++ will insert this into the program:

```
...

void println(int x)
{
  std::cout << x << std::endl;
}

...
```

It might be better to think of the instantiated function as:

```
template <>
void println(int x)
{
  std::cout << x << std::endl;
}
```

This "production of a C++ function" by a function template is called a **function instantiation**. Note that if `println<char *>()` was not called, the following function will ***not*** appear in your program:

```
...

template <>
void println(char * x)
{
  std::cout << x << std::endl;
}

...
```

Note that the function template itself is actually not included in the binary executable code that is executed. It's only used as a rubber stamp by the compiler for creating actual functions.

Writing function templates is sort of like writing **code that produces code**. Specifically a function template can produce functions. This style of programming is called **generic programming**. Templates also lead to something called **metaprogramming**, but I won't get into that because that's beyond the scope of basic programming.

Do you see the power of function templates?

The variable type `T` (usually called a **type parameter**) can appear anywhere in the function template, not just in the function header. Below is an example where the type parameter appears as a return type.

Another thing to remember is that you can use any identifier name for the type parameter. I used `T`. You can use `X` if you like. But the practice is to

use a single uppercase letter.

Exercise. In this example the same type parameter appears as a **return type**. You are given the following program:

```
#include <iostream>

int addOne(int x)
{
  return x + 1;
}

int main()
{
  int a = 1;
  std::cout << addOne(a) << '\n';
  return 0;
}
```

Run it. Easy program right? Now run this:

```
#include <iostream>

int addOne(int x)
{
  return x + 1;
}

int main()
{
  int a = 1;
  std::cout << addOne(a) << '\n';
  std::cout << addOne(3.3) << '\n';
  return 0;
}
```

The 3.3 is passed to the x of `addOne()` which will typecast the value to an integer, i.e. 3. Now rewrite the program so that the `addOne()` function becomes a function template; some test code is included to test your function template. The types that needs to be parameterized (made into a variable) is in bold and larger font.

```
#include <iostream>

int addOne(int x)
{
  return x + 1;
}

int main()
{
  int a = 1;
  std::cout << addOne< int >(a) << '\n';
```

```
  std::cout << addOne< double >(3.3) << '\n';
  std::cout << addOne< char >('c') << '\n';

  int * p = new int;
  std::cout << addOne< int * >(p) << '\n';

  return 0;
}
```

(Note that I'm using the `addOne()` function with `double`, `char`, `int*` values which have the + operator, i.e. if `x` in either an `int`, or a `double`, or a char, or an `int*`, the following makes sense in C++:

```
x + 1;
```

Exercise. In this exercise you write a function template with one type parameter that appears in the function body. Here's one familiar `swap()` function.

```
#include <iostream>

void swap(int & x, int & y)
{
  int t;
  t = x;
  x = y;
  y = t;
}

int main()
{
  int x = 1, y = 42;
  swap(x, y);
  std::cout << x << ' ' << y << std::endl;
  return 0;
}
```

Rewrite it into a function template and test it with the extra code. HINT: The types that should be parameterized are in bold and larger font.

```
#include <iostream>

void swap(int & x, int & y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}

int main()
{
  int x = 1, y = 42;
  swap< int >(x, y);
  std::cout << x << ' ' << y << std::endl;
```

```
  double a = 1.2, b = 3.4;
  swap< double >(a, b);
  std::cout << a << ' ' << b << std::endl;

  return 0;
}
```

Exercise. Complete the following:

```
#include <iostream>

int main()
{
  std::cout << max< int >(1, 3) << '\n';
  std::cout << max< double >(3.1, 1.1) << '\n';
  std::cout << min< int >(1, 3) << '\n';
  std::cout << min< double >(3.1, 1.1) << '\n';
  return 0;
}
```

Another Example

Here's a more substantial example. Recall this:

```
#include <iostream>


bool arrayEqual(int[], int, int[], int);


int main()
{
  int x[] = {1, 2, 3};
  int y[] = {1, 2, 3};
  std::cout << arrayEqual(x, 3, y, 3);
  return 0;
}

bool arrayEqual(int x[], int xSize, int y[], int
ySize)
{
  if (xSize == ySize)
  {
    bool same = true;
    for (int i = 0; i < xSize; i++)
    {
      if (x[i] != y[i])
      {
        same = false;
        break;
      }
    }
    return same;
  }
  else
  {
    return false;
  }
}
```

What if you want to compare arrays of doubles? You need to have another function:

```
bool arrayEqual2(double x[], double y[])
{
  bool same = true;
  for (int i = 0; i < xSize; i++)
  {
    if (x[i] != y[i])
    {
        same = false;
        break;
    }
  }
  return same;
```

```
}
```

Uh oh ... what about comparing array of booleans? Or characters? Etc.

Now try ***this***:

```
#include <iostream>

template < typename T >
bool arrayEqual(T[], int, T[], int);


int main()
{
  int x[] = {1, 2, 3};
  int y[] = {1, 2, 3};
  std::cout << arrayEqual< int >(x, 3, y, 3)
    << std::endl;

  double a[] = {1.2, 3.4, 5.6};
  double b[] = {2.1, 4.3, 6.5};
  std::cout << arrayEqual< double >(a, 3, b, 3)
    << std::endl;

  return 0;
}


template < typename T >
bool arrayEqual(T x[], int xSize, T y[], int ySize)
{
  if (xSize == ySize)
  {
    bool same = true;
    for (int i = 0; i < 3; i++)
    {
      if (x[i] != y[i])
      {
        same = false;
        break;
      }
    }
    return same;
  }
  else
  {
    return false;
  }
}
```

In this example our function template is this:

```
template < typename T >
bool arrayEqual(T x[], int xSize, T y[], int ySize)
{
  ...
}
```

Note that in the body of the function template, x and y are array of elements of type T. The elements of the two arrays appear in the code as

```
bool arrayEqual(T x[], int xSize, T y[], int ySize)
{
  ...
     if (x[i] != y[i])
  ...
}
```

As long as the **!=** makes sense for type **T**, C++ will be able to generate the actual function.

For instance you can compare characters using !=. Therefore you can

```
...
  char s[] = {'a', 'b', 'c'};
  char t[] = {'a', 'b', 'c'};
  std::cout << arrayEqual<char>(x, 3, y, 3)
    << std::endl;
...
```

Exercise. Why does this program not work? Which statement causes the problem?

```
#include <iostream>

template < typename T >
void printProduct(T x, T y)
{
  std::cout << (x * y) << std::endl;
}

int main()
{
  int a = 2;
  int b = 3;
  printProduct< int >(a, b);

  double c = 1.2;
  double d = 3.4;
```

```
  printProduct< double >(c, d);

  int * e = new int;
  int * f = new int;
  printProduct< int * >(e, f);
}
```

(Run this with your compiler and read the error message.)

Exercise. Convert the following program to one containing a function template:

```
#include <iostream>

int max (int x[], int size)
{
  int m = x[0];
  for (int i = 1; i < size; i++)
  {
    if (m < x[i]) m = x[i];
  }
  return m;
}

int main()
{
  int x[10] = {3, 1, 5, 6, 2, 8, 9, 3, -2, 3};
  std::cout << max(x, 10) << std::endl;
  return 0;
}
```

And test it with this:

```
#include <iostream>

int main() {
  int x[10] = {3, 1, 5, 6, 2, 8, 9, 3, -2, 3};
  std::cout << max< int >(x, 10) << std::endl;

  int z[8] = {3, 1, 5, 6, 2, 8, 9, 3};
  std::cout << max< int >(z, 8) << std::endl;

  double y[8] = {3.1, 1.8, 5.2, 6.7,
                 8.4, 9.1, 3.6, -2.7};
  std::cout << max< double >(y, 8) << std::endl;

  return 0;
}
```

Template parameters

In all previous examples we have only one parameter. And the parameter is used as a type. You can have as many template parameters as you like. You can actually use value parameters. Try these examples.

```
#include <iostream>

template < typename T, typename S >
void spam(T x, S y)
{
  std::cout << x << ", " << y << std::endl;
}

int main()
{
  spam<int, double>(1, 2.2);
  spam<double, int>(1.1, 2);
  spam<double, double>(1.1, 2.2);
  spam<int, int>(1.1, 2.2);
  return 0;
}
```

(Of course for the last function there is a typecast from the two doubles 1.1 and 2.2 into integers).

And here's an example where a template parameter is actually is an **int** value:

```
#include <iostream>

template < typename T, int S >
T eggs(T x)
{
  return x + S;
}

int main()
{
  std::cout << eggs<int, 1>(42) << std::endl;
  std::cout << eggs<int, 11>(42) << std::endl;
  std::cout << eggs<double, 1>(2.2) << std::endl;
  std::cout << eggs<char, 3>('a') << std::endl;

  return 0;
}
```

So now you know that a template parameter is declared with the word **"typename"** if it is a type and if it is an integer value you use the word **"int"**.

Exercise. Can you have a template parameter that is a char? A double?

Exercise. Remember this example:

```
#include <iostream>

template < typename T >
T max(T x[], int size)
{
  T m = x[0];
  for (int i = 1; i < size; i++)
  {
    if (m < x[i]) m = x[i];
  }
  return m;
}

int main()
{
  int x[10] = {3, 1, 5, 6, 2, 8, 9, 3, -2, 3};
  std::cout << max< int >(x, 10) << std::endl;

  double y[8] = {3.1, 1.8, 5.2, 6.7,
                 8.4, 9.1, 3.6, -2.7};
  std::cout << max< double >(y, 10) << std::endl;

  return 0;
}
```

Modify the template function so that the size of the array is passed in as a template parameter instead of a function parameter. In other words your `main()` should look like this:

```
#include <iostream>

...

int main()
{
  int x[10] = {3, 1, 5, 6, 2, 8, 9, 3, -2, 3};
  std::cout << max< int, 10 >(x) << std::endl;

  int z[8] = {3, 1, 5, 6, 2, 8, 9, 3};
  std::cout << max< int, 8 >(x) << std::endl;

  double y[8] = {3.1, 1.8, 5.2, 6.7,
                 8.4, 9.1, 3.6, -2.7};
  std::cout << max< double, 8 >(y) << std::endl;

  return 0;
}
```

What is the difference between the old template function and the new in terms of the code they produced? Specifically how many actual functions were created by C++ for the above `main()` by the original function template and by the new template function?

So now you know you can have either this

```
template < typename T>
int max(T x[], int size)
{
    ...
}
```

Or this:

```
template < typename T, int SIZE >
int max(T x[])
{
   ... SIZE appears in code ...
}
```

The big question is … what's the difference between the two? Why does C++ allow integer template parameters?

Well for the first version, an example of a function call is this:

```
max< int >(x, 10);
```

For the second version , an example of a function call is this:

```
max< int, 10 >(x);
```

In the second case, the **`int 10`** is not passed into the function.

Review the **very important** pages in the first chapter on functions where I did a trace. When you call a function, in your CPU, the program has to put the arguments for a function call into the function call stack. The first version has to put two values into the stack. The second version has to put one value into the stack:

```
max< int >(x, 10);
```

```
max< int, 10 >(x);
```

See it? This is the benefit.

However there's a disadvantage. Recall that if you make the following function call in your code:

```
max< int, 10 >(x);
```

then your compiler will instantiate (i.e., create) the functions

```
template <>
int max< int, 10 >(int x[])
{
    ...
}
```

If you also have

```
max< double, 20 >(y);
```

then your compiler will also instantiate

```
template <>
int max< double, 10 >(int x[])
{
    ...
```

```
}
```

This means that the second version will generate a large executable binary code.

Exercise. Can integer template parameter accept values from a variable? Say something like this:

```
int x[] = {2, 3, 5, 7, 11};
int i = 5;
max< int, i >(x);
```

Exercise. Write a function template that computes the maximum of the values from one pointer address up to but not including the second pointer address.

```
int x[] = {2, 3, 5, 7, 11};
std::cout << max< int >(&x[0], &x[5]);
```

Exercise. IMPORTANT!!! Create a simple experiment to verify that a function template parameter can accept a pointer type.

Header Files

Here's a very common gotcha. (I see this all the time.)

When you organize your template functions, you should remember to include the **body of the function template in the header file** not a cpp file. That's because function templates are inline functions.

Exercise. Write a header file `array.h` so that the following works:

```
#include "array.h"

int main()
{
  int x[] = {5, 3, 1, 2, 4};
  int xSize = sizeof(x) / sizeof(int);
  int y[] = {5, 3, 1, 2, 4};
  int ySize = sizeof(x) / sizeof(int);

  println< int >(x, xSize);
  println< int >(y, ySize);
  bubbleSort< int >(x, xSize);
  println< int >(x, xSize);
  println< int >(y, ySize);

  double a[] = {5.5, 3.3, 1.1, 2.2, 4.4};
  int aSize = sizeof(a) / sizeof(double);
  double b[] = {5.5, 3.3, 1.1, 2.2, 4.4};
  int bSize = sizeof(b) / sizeof(double);

  println< double >(a, aSize);
  println< double >(b, bSize);
  bubbleSort< double >(a, aSize);
  println< double >(a, aSize);
  println< double >(b, bSize);

  return 0;
}
```

Automatic resolution

Sometimes you don't have to tell C++ which “version” of the function to use. Here's a previous exercise:

```
#include <iostream>

template < typename T >
void println(T x)
{
  std::cout << x << std::endl;
}

int main()
{
  println<int>(1);
  println<double>(3.14);
  println<char>('c');
  return 0;
}
```

Modify and run this instead:

```
#include <iostream>

template < typename T >
void println(T x)
{
    std::cout << x << std::endl;
}

int main()
{
    println(1);
    println(3.14);
    println('c');
    return 0;
}
```

You see, when you call

```
println(1);
```

this matches the version with

```
template <>
void println(int x)
{
    std::cout << x << std::endl;
}
```

and deduce that the `T` is `int`.

Exercises. I have two function templates, f and g, that accepts a pointer and prints the value stores in the pointer and prints the value that the pointer is pointing to.

```
int * p = new int;
f(p);
g(p);
```

Because of automatic resolution, I don't have to specify the type when I call these two functions. The two functions print:

```
0x10d92c0 0
0x10d92c0 0
```

Of course f and g has to also work for

```
double * q = new double;
f(q);
g(q);
```

Although these two functions have the same output, they are actually different.

- How would you implement f and g so that they are different?
- In what scenario would you use one way of implementation and not the other?

Gotcha

You notice that when I define a template function I write this:

```
template < typename T >
bool arrayEqual(T x[], int xSize, T y[], int ySize)
{
    ...
}
```

which is really the same as this:

```
template <typename T>
bool arrayEqual(T x[], int xSize, T y[], int ySize)
{
    ...
}
```

In other words there is no space after < and before >. You should follow the first format. Why? Because, besides functions, several other "things" can be "templatized", including types. You can end up calling a template function with a template type like this:

```
someFunction< someType<int> >(...);
```

(Don't worry too much about this: I'll be talking about type templates later.) So what will happen is that if you don't have a practice of putting spaces around a type parameter? You might end up with code like this:

```
someFunction<someType<int>>(...);
```

And now you have a problem because the >> as in

```
someFunction<someType<int>>(...);
```

is the input operator!!! (Also known as the stream insertion operator.) To make matters worse, the error message you get in such cases tend to be unintelligible.

If you know what you're doing you may leave out the spaces. Otherwise it's best to stay with the first format: Always have a space after < and before > when used in the context of templates.

(ASIDE: The new C++0x standard, published since 2011, stipulates that a 2011-compliant C++ compiler should parse nested templates correctly. So something like `someFunction<someType<int>>(…)` should be fine for such a compiler. But to be absolutely safe, I suggest you follow the practice as stated above and use a space to separate the two >>. That way your C++ code will compile on both new and old compilers.)

Non-deducible context and ambiguity

There are times when your C++ compiler won't be able to deduce the correct type for your template parameter T.

Try to compile this:

```
template < typename T >
T f(int x)
{
 return x;
}
int main()
{
    f(42);
    return 0;
}
```

In this case, T can be **int**, But note that it can also be **double** since C++ allows automatic typecasting from **int** to **double**.

In these cases, you have to explicit specify the substitution for **T**:

```
template < typename T >
T f(int x)
{
 return x;
}
int main()
{
    f< int >(42);
    return 0;
}
```

Template specialization

There are times when you want “special cases” for your function templates. For instance look at this example:

```
#include <iostream>

template < typename T, int SIZE >
T sum(T x[])
{
  T sum = 0;
  for (int i = 0; i < SIZE; i++)
  {
    s += x[i];
  }
  return s;
}

int main()
{
  int x[10] = {3, 1, 5, 6, 2, 8, 9, 3, -2, 3};
  std::cout << sum< int, 1 >(x) << std::endl;
  std::cout << sum< int, 2 >(x) << std::endl;
  std::cout << sum< int, 10 >(x) << std::endl;

  double y[8] = {3.1, 1.8, 5.2, 6.7,
                 8.4, 9.1, 3.6, -2.7};
  std::cout << sum< double, 1 >(y) << std::endl;
  std::cout << sum< double, 2 >(y) << std::endl;
  std::cout << sum< double, 10 >(y) << std::endl;

  return 0;
}
```

The function template sum does work. But clearly when the number of terms to add is small, it’s silly waste of time to do the summation in a loop. For instance when the size is 1, you can just return x[0]. When the size is 2, it’s x[0] + x[1]. In both of these cases, if you unroll the for-loop, you would save on declaring int variable i with initialization of 0, you save on checking i < size, you save on ++i. There’s also a cost in the CPU of going from the bottom of the loop to the top of the loop. (This will be clarified in CISS360.) For the case of size = 1, using a for-loop will make the function call about 5 times slower!

C++ allows you to specify special cases for a function template. These are called **template specializations**.

Run this:

```
#include <iostream>

template < typename T, int SIZE >
T sum(T x[])
{
```

```
  ...
}

template <>
int sum< int, 0 >(int x[])
{
  return x[0];
}

int main()
{
  ...
}
```

I've added a specialization of the **sum** function template for the case when **T = int** and **SIZE = 0**.

Exercise. Add 3 more specializations to the above sum function template:

- **T = int** and **SIZE = 2**
- **T = double** and **SIZE = 1**
- **T = double** and **SIZE = 2**

Insert print statements into the **sum** function template and its specialization to verify that you are indeed calling the right function.

The above specialization is called a **full specialization**. For templates, there's a concept of **partial specialization**. The syntax would be something like this:

```
template < typename T >
T sum< T, 0 >(int x[])
{
  return x[0];
}
```

i.e., only `SIZE` has been specialized to `0`. However the above **does not compile** because **currently partial specialization is not supported for function templates**. (However it is supported for struct templates and class templates – see later chapters for struct and classes.)

This is a bummer since if we do have partial specialization for function templates, we'll be writing less code.

Also, note that the specialized function

```
template <>
int sum< int, 0 >(int x[])
{
  return x[0];
}
```

Is not a template anymore. It's a regular function. Therefore for multi-file compilation,

- You can put that in a cpp file and put the prototype in the header file, or
- You can inline the function and put the above in the header file.

The standard practice for creating specializations is to create them for frequently used cases where you want speed. For instance, you probably do NOT want to have specializations for `int sum< int, 0 >, int sum< int, 1 >, int sum< int, 2 >, …, int sum< int, 99 >`!!!

Exercise. Test putting your **`sum`** function templates and its 4 specializations into a header file as inline function. Make sure you can compile your program.

Exercise. Next, try putting the prototypes of the 4 specializations into a heaer file and implementations (i.e. bodies) of the 4 specializations in the cpp file. Make sure your program compiles.

Exercise. Write a `max` function template with specializations similar to the `sum` function template.

```
template < typename T, int SIZE >
int max(T x[])
{
    ...
}
```

Exercise. Write a `bubblesort` function template with specializations similar to the `sum` function template.

Exercise. Write a `binarysearch` function template with specializations similar to the `sum` function template.

Exercises. You can model a 2D point as an array of two numbers: {2, 3}. In general you want to model points of dimension 1, 2, 3, 4, …. In physics, engineering, computer vision, etc, it's common to have points with coordinate which are **`float`**s or **`double`**s. Technically speaking I'm talking about "vectors" and not "points". One common operation is to add

vectors of the same dimension. For instance

{2, 3} + {1, 6} = {2+1, 3+6}

Write a function template so that you can run the following:

```
double p0[2] = {2, 3};
double q0[2] = {1, 6};
double sum0[2];
vec_add< double, 2 >(sum0, p0, q0);
vec_println(sum0); // prints <3, 9> and newline

float p1[2] = {2.1, 3.2};
float q1[2] = {1.3, 6.4};
float sum1[2];
vec_add< float, 2 >(sum1, p1, q1);
vec_println(sum1); // prints <3.4, 9.6> and newline
```

Your `vec_add` and `vec_println` should work for any dimension. After you are done, write some specializations. The most common case is when the dimension is 1, 2, 3, 4.

Exercise. Write a template function with specialization(s) to print an array.

```
int x[] = {2, 3, 5};
array_println< int >(&x[0], &x[3]);
                // prints {2, 3, 5} and '\n'

double y[] = {2.1, 3.1, 5.1};
array_println< double >(&y[0], &y[3]);
                // prints {2.1, 3.1, 5.1} and '\n'

char z[] = "abcde";
array_println< char >(&z[0], &z[5]);
                // prints abcde and '\n'
```