41. Structure templates

Objectives

- Write structure templates
- Instantiate structure templates by creating structure variables

Structure template

Once you understand the concept of a function template, it's not too surprising when you see a structure template:

```
template < typename T >
struct vec2
{
  T x, y;
};

int main()
{
  vec2< int > p = {1, 2};
  std::cout << p.x << ',' << p.y << std::endl;

  vec2< double > q = {1.1, 2.2};
  std::cout << q.x << ',' << q.y << std::endl;

  return 0;
}
```

Neat right? And of course we can use them both at the same time when you have template functions supporting structure templates. In this case you will see a templatized structure parameter.

```
#include <iostream>

template < typename T >
struct vec2
{
  T x, y;
};


template < typename T >
void println(const vec2< T > & p)
{
  std::cout << p.x << ',' << p.y << std::endl;
}


int main()
{
 vec2< int > p = {1, 2};
 println< int >(p);

 vec2< double > q = {1.1, 2.2};
 println< double >(q);

 return 0;
}
```

Exercise. Add this function template to your program above.

```
#include <iostream>

template < typename T >
struct vec2
{
  T x, y;
};


template < typename T >
void println(const vec2< T > & p)
{
  std::cout << p.x << ',' << p.y << std::endl;
}


template < typename T >
vec2< T > add(const vec2< T > & p,
              const vec2< T > & q)
{
  vec2< T > r;
  r.x = p.x + q.x;
  r.y = p.y + q.y;
  return r;
}


int main()
{
  vec2< int > p = {1, 2};
  println< int >(p);

  vec2< double > q = {1.1, 2.2};
  println< double >(q);

  return 0;
}
```

Read the code carefully. What does it do? Test this function.

Exercise. Here's a structure template:

```
template < typename T >
struct Array
{
    T x[1000];
    int length;
```

```
};
```

Write a function `init()` so that `init(a)` will set `a.length` to 0. Write another function `append()` so that `append(a, x)` will "add x to a". Write a function `println()` so that `println(a)` will print the values in `a` i.e., `a.x[0]`, `a.x[1]`, ..., `a.x[a.length - 1]`.

Template parameters need not be types. They can be integers too. Try this ...

Exercise. Here's a structure template:

```
template < typename T, int N >
struct Array
{
    T x[N];
    int length;
};
```

Here's a struct variable created using the above:

```
Array< int, 1000 > a; // a.x is size 1000 int array
```

Write a function `init()` so that `init(a)` will set `a.length` to 0. Write another function `append()` so that `append(a, x)` will "add x to a". Write a function `println()` so that `println(a)` will print the values in `a` i.e., `a.x[0]`, `a.x[1]`, ..., `a.x[a.length - 1]`.

Header Files

Here is the most important rule for writing function templates: the whole function must be in the header file.

This is very different from your regular functions where you put the prototype in the header file and the actual definition of the function in a cpp file.

(Note that struct templates, like structs, should also be in the header file.)

Make sure you remember that or you will get an error when you compile your program.