module07

Jump to bottom

qingqingqingli edited this page on Jan 25, 2021 · 3 revisions

Table of contents

- Introduction to templates
- From C - parametric macros
- Templates
- Default types
- Template specialization
- Resources

introduction to templates

- Templates are parameterized by one or more template parameters, of three kinds: `type template parameters`, `non-type template parameters`, and `template template parameters`.

- **Function template** is a blueprint for defining a family of functions. The compiler uses a function template to generate a function definition when necessary. A `function definition` that is generated from a template is an instance or an instantiation of the template. A `function template` is a parametric function definition, where a particular function instance is created by one or more parameter values. The compiler generates each template instance once.

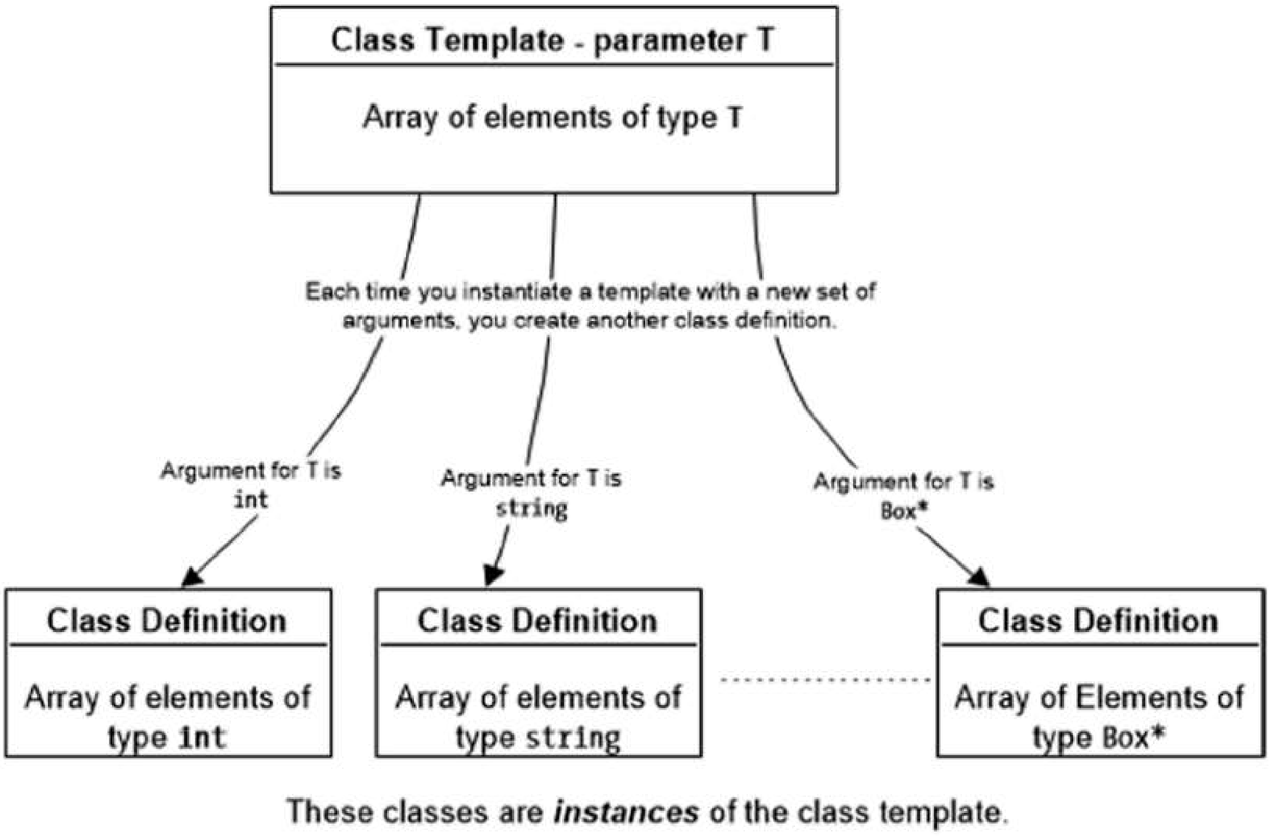
- Need to particularly careful when using pointer types as template arguments, as you could pass the address, instead of dereferenced value as a parameter.

- **Template specialization** defines a behaviour that is different from the standard template. The definition of a template specialization must come after a declaration or definition of the original template. The specialization must also appear before its first use. Otherwise, the program won't compile.

- It is possible to overload a function / template with another function / template.

- **Class templates** are templates the compiler can use to create classes. Class templates are `a powerful mechanism for generating new class types automatically`. A significant portion of the Standard Library is built entirely on the ability to define templates, particularly the `Standard Template Library`, which includes many class and function templates.

- A class template is a `parameterized` type - a recipe for creating a family of class types, using one or more parameters. It is not a class, but just a recipe for creating classes, because this is the reason for many of the constraints on how you define class templates.



- There are many applications for class templates but they are perhaps most commonly used to define **container classes**. These are classes that can contain sets of objects of a given type, organized in a particular way. In a container class the organization of the data is independent of the type of objects stored.

- Instantiation of a class template doesn't instantiate any of its member functions unless they are also used. At link time, identical instantiations generated by different translation units are merged.

From C - parametric macros

- macros have limitations with the edge effect

```
#include <stdio.h>

int max_int (int x, int y) {return (x>=y ? x : y);}
float max_float (float x, float y) {return (x>=y ? x : y);}
```

```
char max_char (char x, char y) {return (x>=y ? x : y);}

int foo(int x) {printf("Long computing time\n"); return x; }

#define max(x, y) (((x) > = (y))? (x) : (y)) // parametric macro
// cpp (C pre-processor) will find all the defines
// when it has an edge effect, it can be messy

int main(void)
{
        int a = 21;
        int b = 42;

        printf("Max of %d and %d is %d\n", a, b, max_int(a, b));
    printf("Max of %d and %d is %d\n", a, b, max(a, b));

    float c = -1.7f;
    float d = 4.2f;

    printf("Max of %f and %f is %f\n", c, d, max_float(c, d));
    printf("Max of %f and %f is %f\n", c, d, max(c, d));

        char e = 'a';
        char f = 'b';

        printf("Max of %c and %c is %c\n", e, f, max_int(e, f));
        printf("Max of %c and %c is %c\n", e, f, max(e, f));

        //*but .....
        printf("Max of %d and %d is %d\n", a, b, max_int(foo(a), foo(b)));
        printf("Max of %d and %d is %d\n", a, b, max(foo(a), foo(b))); // it can be problem

        return (0);
}
```

- In C, `void *` is an option to take in different types of data. A lot of dereferencing is required when you run a big program, which can influence program performance

```
struct list_s {
        void * content;
        size_t size; // needs to know the size to move
        struct list * next;
} list_t;

list_t* list_new(void * content, size_t size);
void list_delete(list ** list);
```

templates

- We need to ask the compiler to instantiate our template. There are two ways: explicit instantiation and implicit instantiation

```
#include <iostream>

template<typename T> // tell the compiler that we're writing a template

T const & max(T const & x, T const & y) { // use the address and not a copy will save space
	return (x >= y? x : y); // apart from a scala type, it could also used with instanc
}

int foo(int x) {
	std::cout << "Long computing time" << std::endl;
	return x;
}

int main(void)
{
	int a = 21;
	int b = 42;

	std::cout << "Max of " << a << " and " << b << " is ";
	std::cout << max<int>(a, b) << std::endl; // explicit instantiation -> this is pref
    std::cout << "Max of " << a << " and " << b << " is ";
    std::cout << max(a, b) << std::endl; // implicit instantiation -> it might not work for

    float c = -1.7f;
    float d = 4.2f;

    std::cout << "Max of " << c << " and " << d << " is ";
    std::cout << max<float>(c, d) << std::endl; // explicit instantiation
    std::cout << "Max of " << c << " and " << d << " is ";
    std::cout << max(c, d) << std::endl; // implicit instantiation

	char e = 'a';
	char f = 'z';

	std::cout << "Max of " << e << " and " << f << " is ";
	std::cout << max<char>(e, f) << std::endl; // explicit instantiation
	std::cout << "Max of " << e << " and " << f << " is ";
    std::cout << max(e, f) << std::endl; // implicit instantiation

    // no problem here

    int ret = max<int>(foo(a), foo(b)); // explicit instantiation -> it will not be macros
	std::cout << "Max of " << a << " and " << b << " is ";
	std::cout << ret << std::endl;

	return 0;
}
```

- Compilers can also write template for classes and structures.

> template for structure

```
#include <iostream>

template<typename T>
class List {

public:
    List<T>(T const & content) {
        // etc...
    }

    List<T>(List<T> const & list) {
        //etc...
    }

    ~List<T>(void) {
        //etc...
    }

    //etc...

private:
        T * _content; // it works the same without *
    List<T> * _next;
};

/*******************************************************/

int main(void)
{
        List<int> a(42);
        List<float> b(3.14f);
        List<List<int>> c(a); // A list of list of integers

        //etc...

        return 0;
}
```

default types

- `tpp` file can be used as a naming convention for templates
- `Default type` means if I don't tell you what the type is, the compiler can assume that it's this type

```
Template<typename T = float>

class Vertex {
public:
        Vertex (T const & x, T const & y, T const & z): _x(x), _y(y), _z(z) {}
        ~Vertex(void){}

        T const & getx(void) const {return this->_x};
    T const & gety(void) const {return this->_y};
    T const & getz(void) const {return this->_z};

    // etc...

private:
        T const _x;
    T const _y;
    T const _z;

    Vertex(void);
};

template<typename T>
std::ostream & operator<<(std::ostream & o, Vertex<T> const & v) {
        std::cout.precision(1);
        o << setiosflag(std::ios::fixed);
        o << "Vertex( ";
        o << v.getX() << ", ";
        o << v.getY() << ", ";
        o << v.getZ();
        o << " )";
        return o;
}

/*******************************************************/

int main(void)
{
        Vertex<int> v1(12, 23, 34);
        Vertex<> v2(12, 23, 34); // 12, 23, 34 will be implicitly converted to floats

        std::cout << v1 << std::endl;
    std::cout << v2 << std::endl;

    return 0;
}
```

```
Vertex( 12, 23, 34);
Vertex( 12.0, 23.0, 34.0);
```

template specialization

- Full or partial template specialization are the same as overload. `Partial specializations` are only allowed for class templates.
- A `class template` by itself is not a type, or an object, or any other entity. No code is generated from a source file that contains only template definitions. **In order for any code to appear, a template must be instantiated**: the template arguments must be provided so that the compiler can generate an actual class (or function, from a function template).

```
/**************FULL SPECIALISATION*********************/

template <typename T, typename U>
class Pair {

public:
        Pair<T, U>(T const & lhs, T const & rhs) : _lhs(lhs), _rhs(rhs) {
                std::cout << "Generic template" << std::endl;
                return;
        }

        ~Pair<T, U>(void) {}

        T const & fst(void) const {return this->lhs;}
    U const & snd(void) const {return this->rhs;}

private:

        T const & _lhs;
    U const & _rhs;

    Pair<T, U>(void);
};

/**************PARTIAL SPECIALISATION*********************/

template <typename U>
class Pair<int, U> { // syntax is different here

    public:
    Pair<T, U>(int & lhs, T const & rhs) : _lhs(lhs), _rhs(rhs) {
        std::cout << "Int partial specialization" << std::endl;
        return;
    }

    ~Pair<T, U>(void) {}

    int     & fst(void) const {return this->lhs;}
    U const & snd(void) const {return this->rhs;}

    private:

    int     & _lhs;
```

```
    U const & _rhs;

    Pair<T, U>(void);
};

/*******************************************************/

template <>
class Pair<bool, bool> { // syntax is different here

public:
    Pair<bool, bool>(bool lhs, bool rhs) : _lhs(lhs), _rhs(rhs) {
        std::cout << "bool/ bool full specialization" << std::endl;
        this->_n = 0;
        this->_n |= static_cast<int>(lhs) << 0; // the first bit
                this->_n |= static_cast<int>(rhs) << 1; // the second bit
        return;
    }

    ~Pair<bool, bool>(void) {}

        bool fst(void) const {return (this->_n & 0x01);}
        bool snd(void) const {return (this->_n & 0x02);}

private:

    int     _n;

    Pair<bool, bool>(void);
};

/*******************************************************/

template<typename T, typename U>
std::ostream & operator<<(std::ostream & o, Pair<T, U> const & p) {
    o << "Pari(" << p.fst() << ", " << p.snd() << " )";
    return o;
}

std::ostream & operator<<(std::ostream & o, Pair<bool, bool> const & p) {
    o << std::boolalpha << "Pari(" << p.fst() << ", " << p.snd() << " )"; // std::boolalpha
        return o;
}

/*******************************************************/

int main(void){
        Pair<int, int> p1(4, 2); // if one parameter matches, it will use the partial speci
        Pair<std::string, float> p2(std::string "Pi", 3.14f);
        Pair<float, bool> p3(4.2f, true);
        Pair<bool, bool> p4(true, false);

        std::cout << p1 << std::endl;
```

```
        std::cout << p2 << std::endl;
        std::cout << p3 << std::endl;
    std::cout << p4 << std::endl;

    return 0;
}
```

```
Int partial specialization
Generic template
Generic template
bool/ bool full specialization
Pair(4, 2)
Pair(Pi, 3.14f)
Pair(4.2f, 1)
Pair(true, false)
```

resources

- [template classes]
- [class template - cppreference]

Pages 11

Find a page...

- Home
- module00
- module01
- module02
- module03
- module04
- module05
- module06
- module07
- module08
- Object Oriented Programming Intro

Clone this wiki locally

```
https://github.com/qingqingqingli/CPP.wiki.git
```