

# Templates

Programação (L.EIC009)

Eduardo R. B. Marques, DCC/FCUP

# Outline

- Motivation
- Function templates
- **struct** template types

# Introduction

# Motivation

The same code is often useful for distinct types, e.g., common algorithms (sorting, searching, ...) and data structures (lists, sets, ...).

C++ allows the definition of **templates** for functions and data types. Templates can define **generic code that is instantiated for multiple types**.

The same concerns are addressed in other programming languages, e.g., generic types in Java.

## Use of template

C++ allows `template` definitions of the form

```
template <typename T>  
. . .
```

where `T` stands for a type variable to instantiate. The compiler generates appropriate code per each distinct instantiation.

Among other possibilities (we will just cover the most basic cases), we can have more than one type:

```
template <typename T, typename U>  
. . .
```

Note: `class` may be used instead of `typename` interchangeably, i.e.

```
template <class T>  
. . .
```

# Template functions

## Example 1

Definition:

```
// Calculate the absolute difference of a and b
template <typename T>
T abs_diff(T a, T b) {
    return a > b ? a - b : b - a;
}
```

Use:

```
int i = abs_diff(1, 2);
double d = abs_diff(3.4, 1.2);
cout << i << ' ' << d << '\n';

1 2.2
```

## Example 1 (cont.)

In the example, T is identified as a template type. It can then be used as a “generic type variable” for the return type, parameters, or local variables.

```
template <typename T>
T abs_diff(T a, T b) {
    return a > b ? a - b : b - a;
}
```

Alternative definition making use of a local variable:

```
template <typename T>
T abs_diff(T a, T b) {
    T r;
    if (a > b) r = a - b;
    else r = b - a;
    return r;
}
```



## Example 1 (cont.)

For each use of the template function, the compiler deduces a concrete type for `T` and checks if its use is appropriate. Each valid use of the template leads to specially generated code (per each type required).

```
template <typename T>
T abs_diff(T a, T b) {
    ...
}

int i = abs_diff(1, 2); // T <- int
double d = abs_diff(3.4, 1.2); // T <- double
```

## Example 1 (cont.)

Compilation fails when it is not possible to infer a concrete type for T:

```
template <typename T>
T abs_diff(T a, T b) {
    ...
}
```

```
cout << abs_diff(1, 2.3); // T <- int or double ?
```

```
abs_diff.cpp:11:11: error: no matching function
for call to 'abs_diff'
```

```
cout << abs_diff(1, 2.3) << '\n';
```

```
^~~~~~
```

```
abs_diff.cpp:2:3: note: candidate template ignored:
deduced conflicting
```

```
types for parameter 'T' ('int' vs. 'double')
```

```
T abs_diff(T a, T b) {
```

```
^
```

```
1 error generated.
```

## Example 1 (cont.)

Compilation also fails if the inferred concrete type does not comply with the required functionality for the template code. In the following case, the compiler

```
template <typename T>
T abs_diff(T a, T b) {
    ...
}

. . .
struct point2d { int x; int y; };
point2d a {0, 0}, b {1, 2};
point2d c = abs_diff(a, b)
```

```
abs_diff.cpp:3:12: error: invalid operands
to binary expression ('point2d' and 'point2d')
    return a < b ? b - a : a - b;
           ~ ^ ~
```

```
abs_diff.cpp:14:15: note: in instantiation
of function template specialization 'abs_diff<point2d>'
```

## Example 2

Template functions can also be used with array arguments:

```
template <typename T>
bool contains(const T array[], int n, T value) {
    for (int i = 0; i < n; i++) {
        if (array[i] == value)
            return true;
    }
    return false;
}

int ia[] { 1, 2, 3};
bool b1 = contains(ia, 3, 2);

char ca[] = { 'a', 'b', 'c' };
bool b2 = contains(ca, 3, 'x');
```

## Example 3

Several template functions are defined in the C++ library. For instance `sort` in header `<algorithm>` can be used to sort arrays and also (to be discussed later) container objects.

```
#include <iostream>
#include <algorithm>
using namespace std;

int main() {
    int a[7] = { 1, 5, 6, 0, 3, 4, 2 };
    sort (a, a + 7);
    for (int v : a) { cout << v << ' '; }
    cout << '\n';
    return 0;
}
```

0 1 2 3 4 5 6

## Example 3 (cont.)

A variant of `sort` also takes a comparison function (pointer) as argument to express the ordering of elements:

```
// Comparison function
bool less_than(time_of_day a, time_of_day b) {
    return a.h < b.h || (a.h == b.h && a.m < b.m);
}

int main( ) {
    time_of_day t[4] = { { 12, 30 }, { 8, 10 },
                        { 23, 30 }, { 11, 12 } };
    sort(t, t + 4, less_than);
    for (time_of_day v : t)
        cout << (int) v.h << ':' << (int) v.m << ' ';
    cout << '\n';
    return 0;
}
```

8:10 11:12 12:30 23:30

`struct` template types

## Example 1

Definition of a template point with XY coordinates - the template argument T is used as the type for member fields:

```
template <typename T>
struct point2d {
    T x;
    T y;
};
```

Use:

```
point2d<int> icoords = { 1, 2 };
point2d<double> dcoords = { 1.5, -1.3};
point2d<double> dpa[4] = { {1.5, 2.5},
                           {-2.5, 4.21},
                           {3.1, -6.3},
                           {4.1, 8.2}};
```



## Example 1 (cont.)

We can define functions combined with template `struct` types:

```
template <typename T>
point2d<T> midpoint(const point2d<T> arr[], int n) {
    point2d<T> m = { 0, 0 };
    for (int i = 0; i < n; i++) {
        m.x += arr[i].x; m.y += arr[i].y;
    }
    m.x = m.x / n; m.y = m.y / n;
    return m;
}

template <typename T>
point2d<T> mul(T f, const point2d<T>& a) {
    return { f * a.x, f * a.y };
}
```

## Example 1 (cont.)

Use of the previous functions:

```
point2d<int> ipoint = { 1, 2 };
point2d<double> dpoint = { 1.5, -1.3 };
ipoint = mul(2, ipoint);
dpoint = mul(2.5, dpoint);

. . .
point2d<double> dpa[4] = { { 1.5, 2.5 },
                          { -2.5, 4.2 },
                          { 3.1, -6.3 },
                          { 4.1, 8.2 } };
point2d<double> dmid = midpoint(dpa, 4);
```

## Example 2 - std::pair

std::pair is defined in the C++ library by the <utility> header as

```
template <typename U, typename V>
struct pair {
    U first;
    V second;
};
```

Use:

```
#include <pair>
using namespace std;
. . .
pair<int, double> x { 1, 2.5 };
pair<double, int> y { 2.5, 1};
pair<const char*, time_of_day> z { "Hello", { 23, 59 } };

// make_pair can also be used
pair<int, double> x2 = make_pair(-1, 2.4);
```

## Use of typedef and auto

Template types are often verbose.

Type aliases introduced via `typedef` are sometimes convenient for more succinct code:

```
typedef point2d<int> ipoint2d;
typedef pair<const char*, const char*> spair;
. . .
ipoint2d x { 1, 2 };
spair y { "Hello", "World" };
```

So is `auto` to avoid writing types altogether:

```
auto x = make_pair(1, 2);
auto y = make_pair("Hello", "World");
```

## Example 3

The `simple_vector` type from previous classes can be defined as a template (see provided code for complete definition):

```
template <typename T>
struct simple_vector {
    T* elements;
    int capacity;
    int size;
};

template<typename T>
int size(const simple_vector<T>* sv) {
    return sv->size;
}

template<typename T>
T get(const simple_vector<T>* sv, int i) {
    return sv->elements[i];
}
```