

# Function and `class` templates

P. Ronchese

Dipartimento di Fisica e Astronomia “G.Galilei”

Università di Padova

“Object oriented programming and C++” course

## Similar operations

Similar operations can be defined for different types

All the following can be defined for all kind of numbers, `int`, `float` and so on:

- basic operations `+`, `-`, `*`, `/`
- absolute value
- power
- ...

More complex objects could share more complex operations

## Generic types

C++ allows writing code with “generic” object types, performing the same operations with objects having different types

A lot of code replication can be avoided

- Work economy in code writing
- Ease of maintenance
- Error reduction

## Function templates declaration and definition

A function can take arguments of generic type by declaring it a “function template”

```
template<class T>
void perm(T& x, T& y, T& z) {
    // make a cyclic permutation of x, y, z
    T t = x;
    x = y;
    y = z;
    z = t;
    return;
}
```

The compiler takes care of calling the proper instance

- any number of template arguments with any name can be used
- `template<typename T>` can be used as well

## Template vs. non-template functions

### Actual code generation

- A template function is just what its name says: a “template”.
- The actual code for the specific types must be generated.
- The compiler must have access to the function definition when the code generation for a specific type is needed (at least once).
- Difference between template and non-template function:
  - non-template functions can be simply declared before being called, and defined in a different translation unit,
  - template functions must be defined in the same translation unit where it is used (exceptions possible but deprecated).
- This can lead to multiple definitions of the same function, but that's allowed as all definitions come from the same template.

## Template functions usage

Best practice: include the full definition with the declaration

```
template<class T>
void perm(T& x, T& y, T& z) {
    ...
}
int main() {
    int i;
    int j;
    int k;
    ...
    perm(i, j, k);
    ...
}
```

## Type consistency

The consistency of a code template with the actually used types is performed only when an instance of the template is created

```
template<class T>
void perm(T& x,T& y,T& z) {
    cout << x.size() << " "
         << y.size() << " "
         << z.size() << endl;
    // make a cyclic permutation of x,y,z
    T t=x;
    x=y;
    y=z;
    z=t;
    return;
}
```

Works only if used with T having a function `size()` .

## Trailing return type

C++11 only

A template function return type can depend on the arguments type

```
template<class T, class S>
auto mult(const T& x,
          const S& y) -> decltype(x*y) {
    return x*y;
}
```

## Variable number of template parameters

C++11 only

Variadic template:  
a template accepting any number of template arguments

```
template<class T>
void print(T t) {
    cout << t << endl;
    return;
}
template<class T, class ... R>
void print(T t, R ... r) {
    cout << t << ' ';
    print(r...);
    return;
}
```

## Fold expressions

C++17 only

Recursive handling of variadic parameters:  
fold expressions

```
template<class ... T>
void print(T t, R ... r) {
    ((cout << t << ' '), ...) << endl;
    return;
}
```

## Class templates declaration and definition

Most flexibility is achieved by the use of `class` templates

A previously seen example (“FloatArray”) can be easily used for other types than `float` by making it a `class` template.

```
template<class T> // T replaces "float"
class Array {     // everywhere
public:          // in the class definition
    Array(int n);
    Array(const Array<T>& a);
    ~Array();
    ...
private:
    int eltn;
    T* cont;
    void copy(const Array<T>& a);
};
```

## Class templates implementation

The specification `template<class T>` must be included in the implementation of constructor(s), destructor, functions and operators

```
template<class T>
Array<T>::Array(int n) :
    eltn(n),
    cont(new T[eltn]) {
}
template<class T>
Array<T>::~~Array() {
    delete[] cont;
}
...
```

The implementation must come before the `class` is used:  
it's usually coded in "implementation files" (`.hpp`)  
included at the end of the header (when not inlined)

## Template class usage

In the declaration of objects with template type, the template parameter(s) must be specified

The usage is then exactly the same

```
#include "Array.h"
int main() {
    int i;
    int n=5;
    Array<float> a(n);
    for(i=0;i<n;++i) a[i]=i*3.2;
    for(i=0;i<n;++i) cout << i << " "
                        << a[i] << endl;
    return 0;
}
```

## Template specialization

Function and `class` templates allow the automatic replication of the same code with different types (or integer quantities), but some special situation could require a special handling

Template specialization: the implementation of a template `class` (or one of its functions) for a specific type is provided in addition to the generic one

```
template<>
Array<bool>::Array(int n) :
    eltn(n),
    cont(new bool[eltn]) {
    cout << "create an Array of bool" << endl;
}
```

A (possibly empty) `template` specification is needed all the same

## Nested class templates

A class template could contain a nested class, being referred from outside the enclosing class .

The compiler could be unable to know if a member of a class template is a type or an object: it assumes it's an object, unless a `typename` is specified.

```
template <class T> class A {
public:
    class C {
    };
};
template <class T> class B {
public:
    void f(A<T>* p) {
        typename A<T>::C* c; // pointer to "A<T>::C"
    };
};
```

## typename specification

```

template <class T> class A;
template <class T> class B {
public:
    void f(A<T>* p) {
        // by default A<T>::C is an object
        // and A<T>::C * c is a multiplication
        typename A<T>::C * c;
        // adding a "typename"
        // A<T>::C is declared being a type
    };
};

```

A **typename** specification is required in a declaration if:

- there's a template parameter
- there's a scope resolution operator `::`

"typename" can be used in place of "class" in template declarations: `template <typename T> class A`

## Non-type template arguments declaration

Template arguments can be not only types, but also integers, or anything similar to an `int` (`char`, `enum` ...)

Example: a fixed-size array

```
template<class T, unsigned int N>
class FixedSizeArray {
public:
    FixedSizeArray();
    FixedSizeArray(
        const FixedSizeArray<T,N>& a);
    ~FixedSizeArray();
    ...
private:
    T* cont;
    void copy(const FixedSizeArray<T,N>& a);
};
```

## Non-type template arguments implementation

```
template<class T, unsigned int N>
FixedSizeArray<T,N>::FixedSizeArray():
    cont(new T[N]) {
};

template<class T, unsigned int N>
FixedSizeArray<T,N>::copy(
    const FixedSizeArray<T,N>& a) {
    if(cont==a.cont) return;
    delete[] cont;
    cont=new T[N];
    T* pr=a.cont+N;
    T* pl= cont+N;
    while(pl>cont) *--pl=*--pr;
    return;
};
```

## “Higher level” template classes

Template parameter can be a template itself

A blank is required between two angular parentheses

```
Array< Array<int> > aai;
```

C++11 only

The blank can be omitted

```
Array<Array<int>> aai;
```