

Template Variations

Version 1: Dr. Ofir Pele

Version 2: Dr. Erel Segal-Halevi

Template Variations (folder 1)

A template can receive several arguments:

```
template <typename T1, typename T2>
struct pair {
    T1 first;
    T2 second;
};

pair<string,int> func() {return {"hi",2};}

int main() {
    pair<string,int> a {"hello", 3};
    auto b = func();
}
```

Template Variations (folder 1)

A template can receive constant integral arguments:

```
template<typename T, int Size>
class array { T m_values[Size];
public:
    // operator[] , operator<<
    // static size constant ...
};

array<char,1024> arr1;
// Array<char> arr1(1024); // dynamic
// char arr1[1024];        // static
```

Template Specialization

Template function specialization (folder 2)

Example application:

- General swap – uses operator=.
- Specific swap for a "Buffer" class –
 swaps the size and the pointer (see folder 2).

Template class specialization (folder 3)

```
template <typename T> class Test {  
    public: Test() {    cout << "General"; }  
};
```

```
template <> class Test <int> {  
    public: Test() {    cout << "Specialized"; }  
};
```

```
int main() {  
    Test<int> a;    // Specialized  
    Test<char> b;  // General  
    Test<float> c; // General  
}
```

Template class specialization (folder 3)

Example application:

- We have a general `vector<T>`
- We create a specific `vector<bool>` to reduce memory space – save 8 bools in one char.
- See folder 3

Template class specialization (folder 4)

Example application:

- We have a template function that should only work for numeric arguments.
- We create a *class* to tell us whether a type is numeric.
- We create a compiler error using the `static_assert` keyword.
- See folder 4

Template class specialization (folder 4)

Example application:

- We have a template function whose return-type should change based on the template type.
- We create a *class* that keeps a field with the required return type.
- We get the return type with the `decltype` keyword.
- See folder 4.

Template Meta-Programming

Template Meta-Programming

// primary template computes 3 to the Nth

```
template<int N> class Pow3 { public:  
    static const int result =  
        3*Pow3<N-1>::result;  
};
```

// full specialization to end recursion

```
template<> class Pow3<0> { public:  
    static const int result = 1;  
};
```

```
int main() {  
    cout << Pow3<1>::result<<"\n"; //3  
    cout << Pow3<5>::result<<"\n"; //243  
    return 0;  
}
```

Template Meta-Programming (folder 5)

Goal: Numerically calculate and plot the n-th derivative of an arbitrary function.

Steps:

- 1) **rgb.hpp** – class for creating a ppm picture file (see week 7), and plotting a "function-like object" (=functor).
- 2) **functors_demo.cpp** – demonstrates plotting various functors and lambda expressions.
- 3) **derivative.hpp** – the derivative template.
- 4) **animate_demo.cpp** – function animation.

Summary: Polymorphism vs. Templates

- Templates **compilation time** is much longer than using inheritance.
- Using templates enlarges the code size.
- Compilation errors can be very confusing.
- Templates **running time** is much faster than using inheritance.
- Combined with compiler optimizations, templates can reduce runtime overhead to zero.

Longer compilation time is not always a bad thing (from xkcd):

