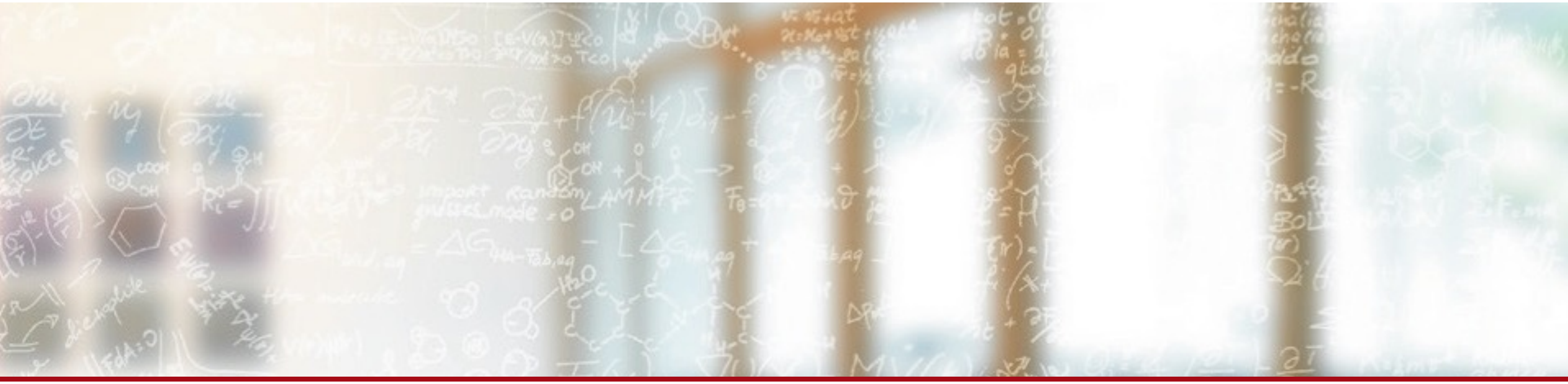
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



Templates

Mauro Bianco

Advanced C++ for HPC

Functions and Classes

- A function is defined as return type, name, and arguments

```
int add(int x, int y) { return x+y; }

int main() { add(65, 35); }
```

- A class is declared as a name after **`class`** or **`struct`**

```
class name1;

struct name2;
```

- `Definition contains type, data and function members`

```
class name1 {};

int main() { name1 x; }
```

Function templates

- A function template <u>is not</u> a function
- Need to be instantiated to be so
- Mental model: substitute type text to template argument

```cpp
template <typename T>
void foo(T x) {
    std::cout << x << "\n";
}

int main() {
    foo<int>(65);
    foo<char>(65);
    foo(3.14159); // Argument Deduction
    foo(std::string("string"));
}
```

Overloads

- Among the options the most specialized is chosen
  - Include ADL available candidates

```
template <typename T>
void foo(T x) {
    std::cout << x << "\n";
}

void foo(std::string const& x) {
    std::cout << "ooh! a string! " << x << "\n";
}

int main() {
    foo<int>(65);
    foo<char>(65);
    foo(3.14159); // Argument Deduction
    foo(std::string("string"));
}
```

Order Matters

```
template <typename T, typename U>
void foo(T, U) {}

int main() {
    foo<std::string, double>("hello", 4.5);
    foo<std::string>("hello", 4.5);
    foo<double>("hello", 4.5);
}
```

cannot convert '"hello"' (type 'const char [6]') to type 'double'

Template Argument Deduction

- To instantiate a template all the arguments must be known!
- Sometime they can be deduced

```
template <typename To, typename From>
To convert(From f);

void g(double d) {
    int i = convert<int>(d);
    char c = convert<char>(d);
    int(*ptr)(float) = convert;
}
```

```
template <typename From, typename To>
To convert(From f);

void g(double d) {
    int i = convert<double, int>(d);
    char c = convert<double, char>(d);
    int(*ptr)(float) = convert;
}
```

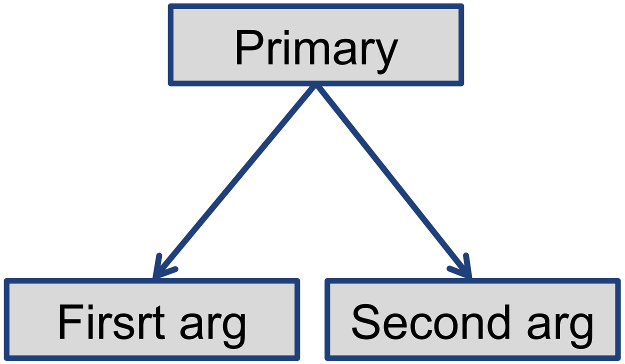
Class templates

- A class template is not a class
- Need to be instantiated
- Only then an object of that type can exist

```
template <typename T>
class templ_name {
    T member;
    T operator()(T a, T b) const {…}
};

int main() {
  using class_name = templ_name<int>;
  class_name x;
}
```

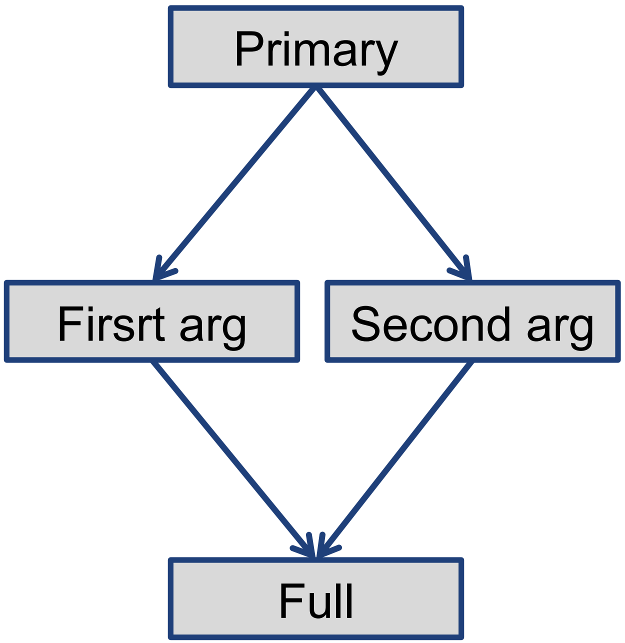
Partial specializations

- What for functions are overloads
- The more specialized is chosen



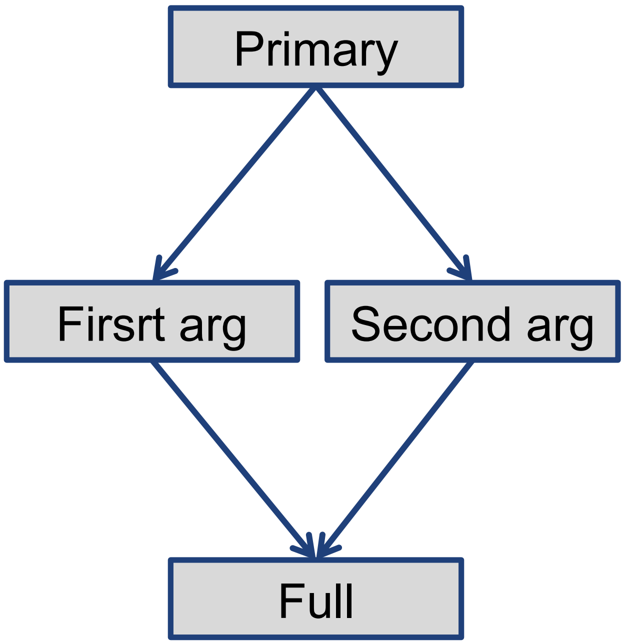

```cpp
template <typename T, typename U>
struct X {}; /* 1 */

template <typename T>
struct X<T, int> {}; /* 2 */

template <typename U>
struct X<float, U> {}; /* 3 */

int main() {




}
```

Partial specializations: Partial order




```
template <typename T, typename U>
struct X {};

template <typename T>
struct X<T, int> {};

template <typename U>
struct X<float, U> {};



int main() {
    X<char, double> a; // primary template
    X<char, int> b; // Specialization 1
    X<float, double> c; // specialization 2
    X<float, int> d; // ????
}
```

Partial specializations: Partial order




```
template <typename T, typename U>
struct X {};

template <typename T>
struct X<T, int> {};

template <typename U>
struct X<float, U> {};

template <>
struct X<float, int> {};

int main() {
    X<char, double> a; // primary template
    X<char, int> b; // Specialization 1
    X<float, double> c; // specialization 2
    X<float, int> d; // Now OK!
}
```

Pattern Matching

```
template <typename T, typename U>
struct X {}; /* 1 */

template <typename W, typename T, typename U>
struct X<W, X<T,U>> {}; /* 2 */

template <typename T>
void foo(X<T,T>) {}

int main() {

   X<int, X<int, float>> b;
   X<int, X<char, X<int,void>>> c;
   X<double, double> a;
   foo(a); // foo(c)?
}
```

Specialization

Primary

Pattern Matching

```
template <typename T, typename U>
struct X {}; /* 1 */

template <typename W, typename T, typename U>
struct X<W, X<T,U>> {}; /* 2 */

template <typename T>
void foo(X<T,T>) {}

template <typename T, typename U>
void foo(X<T,U>) {}

int main() {
    X<double, double> a;
    X<int, X<char, X<int,void>>> c;

    foo(a); foo(c);
}
```

Default template arguments

- Template argument can be defaulted
- From C++11 this is possible on function templates
- Only the arguments on the right **if not deduced**

```
template <typename T, typename Result=char>
Result foo(T x) {
    return static_cast<Result>(x);
}

int main() {
    cout << foo(65) << "\n";
    cout << foo<int>(65) << "\n";
    cout << foo<int, int>(65.3) << "\n";
}
```

Default Template arguments for classes

```
template <typename T=double, int size=10>
class my_container {};

int main() {
    my_container<> x;
    // x is a my_container of 10 doubles
    // <> are needed since
    // my_container is a template!

    my_container<100> y; // ERROR
}
```

- This is true before C++17
- In C++>=17 `my_container x;` is ok
  - Type deduction in constructors

Default Template Arguments and Specializations

```
template <typename T=double, int size=10>
struct my_container {};

template <typename T>
struct my_container<T, 10> {};

template <typename T>
struct my_container<T, 15> {};

int main() {
    my_container<char, 10> z;
    my_container<float,30> u;
    my_container<int, 15> v;
    my_container<int> y;
    my_container<> x;

}
```

Always search primary template first!

<T, 10> specialization

Primary

<T, 15> specialization

<T, 10> specialization

<double, 10> specialization

Non type template arguments

- Integral values can be used as template arguments
  - char, int, and even pointers

```
template <int I>
struct static_int {
    static constexpr int value = I;
};


int main() {
    std::cout << static_int<5>::value;
}
```

SFINAE

- When looking for specialization some substitution may fail

```
template <typename T, typename U = int>
struct X {};

template <typename T>
struct X<T, typename T::extra_type> {};
```

Specialization in Action

```
template <typename T, typename U = int>
struct X {};

template <typename T>
struct X<T, typename T::extra_type> {};

struct A { using value_type = int; };

struct B { using extra_type = int; };

struct C { using extra_type = float; };

struct D { using extra_type = char; };

int main() {
    X<A> a;
    X<B> b;
    X<C> c;
    X<B,char> b1;
    X<D,char> d;
}
```

SFINAE: primary template

Specialization

Primary: `C::extra_type` not **int**

Primary: `B::extra_type` not char

Specialization: extra_type is char

SFINAE: std::enable_if

```
template <typename T, typename VOID = void>
struct A;

template <typename T>
struct A<T, typename std::enable_if<
                std::is_same<T, int>::value, void>::type
       > {
    A() { std::cout << "int!\n"; }
};


template <typename T>
struct A<T, typename std::enable_if<
                !std::is_same<T, int>::value, void>::type
       > {
    A() { std::cout << "not int\n"; }
};

A<float> x;
```

SFINAE: std::enable_if

```
template <typename T, typename VOID = void>
struct A;

template <typename T>
struct A<T, std::enable_if_t<std::is_same_v<T, int>, void>
       > {
    A() { std::cout << "int!\n"; }
};


template <typename T>
struct A<T, std::enable_if_t<!std::is_same_v<T, int>, void>::type
       > {
    A() { std::cout << "not int\n"; }
};

A<float> x;
```

Possible implementations enable_if

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T>
{
    using type = T;
};
```

```
template<bool B, class T = void>
struct enable_if;

template<class T>
struct enable_if<true, T>
{
    using type = T;
};
```

Class Template Type Deduction (C++17)

```
template <typename T>
class A {
    T x;
public:
    A(T x) : x{x} {}
};

int main() {
    A<int> x(3);
    A y(3); // A<int>
}
```

```
template <typename F>
struct B {
    F f;
    B(F&& f) : f{std::move(f)} {}

    template <typename ...Args>
    void call(Args&&... args) {
        f(std::forward<Args>(args)...);
    }
};


int main() {
    auto f = [](int i, int j) {cout << i+j << "\n"};
    B<decltype(f)> a{std::move(f)};




}
```

Class Template Type Deduction (C++17)

```
template <typename T>
class A {
    T x;
public:
    A(T x) : x{x} {}
};

int main() {
    A<int> x(3);
    A y(3); // A<int>
}
```

```
template <typename F>
struct B {
    F f;
    B(F&& f) : f{std::move(f)} {}

    template <typename ...Args>
    void call(Args&&... args) {
        f(std::forward<Args>(args)...);
    }
};


int main() {
    auto f = [](int i, int j) {cout << i+j << "\n"};
    B<decltype(f)> a{std::move(f)};

    B b{[](int i, int j) {cout << i+j << "\n";}};

    a.call(3,4);
    b.call(3,4);
}
```

Class Template Type Deduction (C++17)

```
template <typename T>
class A {
    T x;
public:
    template <typename U>
    A(T x, U, int) : x{x} {}
};

int main() {
    A<int> x(3, 3.4, 7);
    A y(3, 3.4, 7); // A<int>
    A z{y}; // A<int>
}
```

```
template <class T, class U>
A<T> make_A(T a, U x, int y) {
    return A<T>{a, x, y};
} // Fictional function template

template <class T>
A<T> make_A(A<T> a) {
    return A<T>{a}; // copy
} // Fictional function template

int main() {
    A<int> x(3, 3.4, 7);
    auto y = make_A(3, 3.4, 7);
    auto z = make_A(y);
}
```

Classes and Meta-Programming

- Kinds of members
  - [Static] Function
  - [Static] Data
  - Constexpr function
  - Static const/Constexpr data
  - Type (nested type names)
- Meta-programming is manipulating types
  - And static const/constexpr values
- The main mechanism is using class templates
- Single applications rarely need TMP
- Useful when building abstraction layers
  - E.g., header-only libraries

Step Back

- Type members are possible
- Access like static members
  - `X::type_t<U>`
- Visibility rules as normal
- Constexpr variables visible at translation unit level

```
class X {
public/private/protected:
    using type = …;

    template <typename T, …>
    using type_t = …;

    static const int a = 10;
    static constexpr int b =10;

    X(…);

    void member(…);
};
```

A simple example

- A complex way to say
  - `movl    $120, %esi`

https://godbolt.org/z/WvEhGTaoM

```
template <unsigned char N>
struct factorial {
   static constexpr unsigned value =
                    N * factorial<N-1>::value;
};

template <>
struct factorial<1> {
   static constexpr unsigned value=1;
};


int main() {
   std::cout << factorial<5>::value << "\n";
}
```

A Convention for TMP

- TMP is still an “accident” in C++
- Boost::MPL conventions partially adopted by ISO C++
- A meta-function returning a type has a public `::type`
- A meta-function returning a value has a public `::value`
- Or both

```
template <Arguments…>
struct meta_function {
    using type = … ;
    static constexpr … value = … ;
};
```

- Type members with arbitrary names are called **traits**

std::integral_constant

```
template<class T, T v>
struct integral_constant {
    typedef T value_type;
    static constexpr value_type value = v;
};


//use:
static_assert(integral_constant<int, 7>::value == 7, “Error”)
```

Building abstractions: std::rank example

- Type of `T[3][4]` is `(T[3])[4]`

```
template<class T>
struct rank
    : public integral_constant<size_t, 0>
{};

template<class T, size_t N>
struct rank<T[N]> // (int[3])[4] => T[4] where T = int[3]
    : public integral_constant<size_t, rank<T>::value + 1>
{};

template<class T>
struct rank<T[]>
   : public integral_constant<size_t, rank<T>::value + 1>
{};
```

An Example with Types: If on Types

- If (boolean value) then type1, else type2
- A shorter version

```
template <bool Pred, typename T1, typename T2>
struct select_first {
    using type = T2;
};

template <typename T1, typename T2>
struct select_first<true, T1, T2> {
    using type = T1;
};
```

The false case is also the primary template

- But why do we need this?

One (Maybe) Silly Example

```
template <bool WithRef>
typename select_first<WithRef, int&, int>::type
with_ref (typename select_first<WithRef, int&, int>::type x) {
    x += 1;
    return x;
}

int main() {
    int x = 1;
    with_ref<true>(x);
    with_ref<false>(x);
}
```

Alias templates

- Typedefs on steroids!
  - Still typedefs
- `using integer_type = int; // just a typedef`
- `template <typename T> using type = vector<T>;`
  - `type<double> x(100)`
- Many template arguments and defaults are allowed

```
template <template <typename, typename> class T,
        typename U,
        template <typename> class Alloc = std::allocator>
using my_container = T<U, Alloc<U>>;

my_container<vector, int> x(100);
```

Alias templates

```
template <typename T>
using my_vec = std::vector<T, my_allocator<T>>;

template <typename T>
std::size_t size_of(my_vec<T> const& v) { return v.size(); }
```

Alias templates

```
template <typename T>
using my_vec = std::vector<T, my_allocator<T>>;

template <typename T>
std::size_t size_of(my_vec<T> const& v) { return v.size(); }

template <typename T>
std::size_t size_of(std::vector<T, my_allocator<T>> const& v) {
    return v.size();
}
```

Ambiguous

Alias templates

```
template <typename T>
using my_vec = std::vector<T, my_allocator<T>>;

template <typename T>
std::size_t size_of(my_vec<T> const& v) { return v.size(); }







template <template <typename, typename> class V>
void do_nothing(V<int, my_allocator<int>> const&) {}

template <template <typename> class V>
void do_nothing(V<int> const&) {}

int main() {
    do_nothing(my_vec<int>(23));
}
```

Never picked up
for my_vec

Variadic Templates

- A **template parameter pack** accepts zero or more arguments!
- Using … to express packs

```cpp
template <typename ...Ts>
void foo(Ts ...args) {}

template <typename ...Ts>
class A {};
```

```cpp
template <typename ...Ts>
void foo(Ts ...args) {
    function(args...);
    pattern(args)...;
    function(&args...);
}
```

Parameter pack

Pack expansion: arg0, arg1, arg2

Produces a comma separated list: `pattern(arg0), pattern(arg1), pattern(arg2),…`

`function(&arg0, &arg1…);`

Recursion in action

```
void pretty_print(std::ostream& s) {
    s <<"\n";
}

template <typename T, typename ...Ts>
void pretty_print(std::ostream& s, T first, Ts ...values) {
    s << " {" << first << "} ";
    pretty_print(s, values...);
}

int main(){
    pretty_print(std::cout, 3.2, "hello", 42, "world");
}
```

https://godbolt.org/z/QTU3Uz

Tuples

- Sequences of element of arbitrary types

```
tuple<int, string, float> t(10,"A",3.14);
auto tup = make_tuple("Hello", 42, string("World"));
const char* ptr;
int x;
string str;
tie(ptr, x, str) = tup;
```

Syntax and tuples

Special sizeof

```
template <typename ...T>
struct A {
    static constexpr int count = sizeof...(T);
};
```

Pack expansion

Pack expansion with pattern

```
template <typename ...T>
struct B {
    using tuple = std::tuple<T...>;
    using tuple_of_vector = std::tuple<std::vector<T>...>;
    using std::tuple<std::pair<T, std::vector<T>>...>;

    tuple _data;

    B(T... args) : _data(args...) {}
};
```

Pattern for zipping

Structured Bindings (C++17)

```
template <class ...Ts>
std::tuple<Ts...> values(Ts&&... as) {
    return std::make_tuple(std::forward<Ts>(as)...);
}

int main() {

    int a[2] = { 3,4 };

    auto [x, y] = a;
    std::cout << x << ", " << y << "\n";

    auto [u, v, w] = values(3.4, std::string{"hello"}, 8);
    std::cout << u << ", "
              << v << ", "
              << w << "\n";
}
```

C++17 Fold Expressions

- More elaborate pack-expansions (*<u>reductions on packs</u>*)
- Unary right fold: $(E \odot \ldots) \rightarrow E_1 \odot (\ldots \odot (E_{N-1} \odot E_N))$
- Unary left fold: $(\ldots \odot E) \rightarrow ((E_1 \odot E_2) \odot \ldots) \odot E_N$
- Binary right fold: $(E \odot \ldots \odot I) \rightarrow E_1 \odot (\ldots \odot (E_{N-1} \odot (E_N \odot I)))$
- Binary left fold: $(I \odot \ldots \odot E) \rightarrow (((I \odot E1) \odot E2) \odot \ldots) \quad \odot EN$

- Available operators (all binary):
- `+`, `-`, `*`, `/`, `%`, `^`, `&`, `|`, `=`, `<`, `>`, `<<`, `>>`, `+=`, `-=`, `*=`, `/=`, `%=`, `^=`, `&=`, `|=`, `<<=`, `>>=`, `==`, `!=`, `<=`, `>=`, `&&`, `||`, `,`, `.*`, `->*`
- There are already very clever examples of use of these

Fold Expressions: Binary Left

```
string operator+(string a, string b) {
    return string("+" + a.s + b.s);
}

template <int ...Vs>
string concat1() {
    return (string("ciao") + ... + paren(Vs));
}

std::cout << concat1<1,2,3,4,5>() << "\n";
```

- `((((("ciao" + paren(1)) + paren(2)) + paren(3)) + paren(4)) + paren(5))`
- `(((( "+ciao(1)" + paren(2)) + paren(3)) + paren(4)) + paren(5))`
- `((( "++ciao(1)(2)" + paren(3)) + paren(4)) + paren(5))`
- `(( "+++ciao(1)(2)(3)" + paren(4)) + paren(5))`
- `( "++++ciao(1)(2)(3)(4)" + paren(5))`
- `"+++++ciao(1)(2)(3)(4)(5)"`

Fold Expressions: Binary Right

```
string operator+(string a, string b) {
    return string("+" + a.s + b.s);
}

template <int ...Vs>
string concat2() {
    return (paren(Vs) + ... + string("ciao"));
}

std::cout << concat2<1,2,3,4,5>() << "\n";
```

- `paren(1) + (paren(2) + (paren(3) + (paren(4) + (paren(5) + "ciao"))))`
- `paren(1) + (paren(2) + (paren(3) + (paren(4) + "+(5)ciao")))`
- `paren(1) + (paren(2) + (paren(3) + "+(4)+(5)ciao"))`
- `paren(1) + (paren(2) + "+(3)+(4)+(5)ciao")`
- `paren(1) + "+(2)+(3)+(4)+(5)ciao”`
- `"+(1)+(2)+(3)+(4)+(5)ciao"`

Fold Expressions: Unary Left

```
string operator+(string a, string b) {
    return string("+" + a.s + b.s);
}

template <int ...Vs>
string concat3() {
    return (... + paren(Vs));
}

std::cout << concat3<1,2,3,4,5>() << "\n";
```

- `((((paren(1) + paren(2)) + paren(3)) + paren(4)) + paren(5))`
- `((("+(1)(2)" + paren(3)) + paren(4)) + paren(5))`
- `(("++(1)(2)(3)" + paren(4)) + paren(5))`
- `("+++(1)(2)(3)(4)" + paren(5))`
- `"++++(1)(2)(3)(4)(5)"`

Fold Expressions: Unary Right

```
string operator+(string a, string b) {
    return string("+" + a.s + b.s);
}

template <int ...Vs>
string concat4() {
    return (paren(Vs) + ...);
}

std::cout << concat4<1,2,3,4,5>() << "\n";
```

- `paren(1) + (paren(2) + (paren(3) + (paren(4) + paren(5))))`
- `paren(1) + (paren(2) + (paren(3) + "+(4)(5)"))`
- `paren(1) + (paren(2) + "+(3)+(4)(5)")`
- `paren(1) + "+(2)+(3)+(4)(5)”`
- `"+(1)+(2)+(3)+(4)(5)"`

Folding Expressions with Associative Operators

```
string operator+(string a, string b) {
    return string(a.s + "+" + b.s);
}
```

- Binary Left: “`ciao+(1)+(2)+(3)+(4)+(5)`”
- Binary Right: “`(1)+(2)+(3)+(4)+(5)+ciao`”
- Unary Left: “`(1)+(2)+(3)+(4)+(5)`”
- Unary Right: “`(1)+(2)+(3)+(4)+(5)`”

An example 1/6

- `my_container` is not a template
  - `my_container` is not flexible

```
struct my_container {
    using value_t = int;
    using container_t = vector<value_t>;
    container_t C;

    my_container(size_t s) : C(s) {}
};


 my_container my_c(42);
```

An example 2/6

- Customizing the basics

```
template <typename VT>
struct my_container {
    using value_t = VT;
    using container_t = vector<value_t>;
    container_t C;

    my_container(size_t s) : C(s) {}
};


 my_container my_c<int>(42);
```

An example 3/6

- Customizing the allocator

```
template <typename VT, typename Allocator = allocator<VT>>
struct my_container {
    using value_t = VT;
    using container_t = vector<value_t, Allocator>;
    container_t C;

    my_container(size_t s) : C(s) {}
};


my_container<int> my_c(42);
my_container<int, std::allocator<int>> my_c(42);
my_container<int, std::allocator<double>> my_c(42);
```

An example 4/6 – Template Template Arguments

- Avoiding redundancies

```
template <typename VT,
          template <typename> class Allocator = allocator>
struct my_container {
    using value_t = VT;
    using container_t = vector<value_t, Allocator<value_t>>;
    container_t C;

    my_container(size_t s) : C(s) {}
};


my_container<int> my_c(42);
my_container<int, std::allocator> my_c2(42);
```

An example 5/6

- Being completely explicit

```
template <typename Container>
struct my_container {
    using value_t = typename Container::value_type;
    using container_t = Container;
    container_t C;

    my_container(size_t s) : C(s) {}
};


my_container<vector<int, allocator<int>>> my_c(42);
```

An example 6/6

- Customizing the whole

```
template <typename VT,
          template <typename, typename> class CT = vector,
          template <typename> class Allocator = allocator>
struct my_container {
    using value_t = VT;
    using container_t = CT<value_t, Allocator<value_t>>;
    container_t C;

    my_container(size_t s) : C(s) {}
};


my_container<int> my_c(42);
my_container<int, vector> my_c2(42);
my_container<int, vector, allocator> my_c3(42);
```