

# Templates

- From macros to templates
- Parameter deduction, instantiation, specialization
- Class templates, partial specialization
- Explicit instantiation
- Dependent types
- Scope resolution, lookup
- Mixins
- CRTP (Curiously Recurring Template Pattern)
- Variadic templates in C++11
- Fold expressions in C++17

# Templates

- Originally Stroustrup planned only Macros
- Side effects are issue with Macros: no types known
- Templates are integrated to C++ type system
- Templates are not functions, they are skeletons
- Parameter deduction
- Instantiation

# Templates

```
template <typename T>
void swap( T& x, T& y)
{
    T temp = x;
    x = y;
    y = temp;
}
template <typename T>
T max( T a, T b)
{
    if ( a > b )
        return a;
    else
        return b;
}
void f()
{
    int i = 3, j = 4, k;
    double f = 3.14, g = 5.55, h;
    k = max(i, j);
    h = max(f, g);
    h = max(i, f);
}
```

# Templates with more types

```
template <class T, class S>
T max( T a, S b)
{
    if ( a > b )
        return a;
    else
        return b;
}
```

```
int f()
{
    int    i = 3;
    double x = 3.14;

    z = max( i, x);
}
```

# No deduction on return type

```
template <class R, class T, class S>
R max( T a, S b, R)
{
    if ( a > b )
        return a;
    else
        return b;
}
z = max( i, x, 0.0);
```

```
template <class R, class T, class S>
R max( T a, S b)
{
    if ( a > b )
        return a;
    else
        return b;
}
z = max<double>( i, x);           // ok, returns 3.14
k = max<long, long, int>( i, x); // converts to long and int
k = max<int, int, int>( i, j);
```

# Template overloading

```
template <class T> T max(T,T);
template <class R, class T, class S> R max(T,S);

template <>
const char *max( const char *s1, const char *s2)
{
    return strcmp( s1, s2) < 0 ? s1 : s2;
}

int i = 3, j = 4, k;
double x = 3.14, z;
char *s1 = "hello";
char *s2 = "world";

z = max<double>( i, x); // ok, max(T,S) returns 3.14
k = max( i, j); // ok, max(T,T)
cout << max( s1, s2); // ok, "world"
```

# Template classes

- All member functions are template
- Lazy instantiation
- Possibility of partial specialization
- Specialization may completely different
- Default parameters are allowed

# Dependent types

- Until type parameter is given, we are not sure on member
- Specialization can change
- If we mean type: **typename** keyword should be used

```
long ptr;
template <typename T>
class MyClass
{
    T::SubType * ptr;    // declaration or multiplication?
    //...
};
template <typename T>
class MyClass
{
    typename T::SubType * ptr;
    //...
};

typename T::const_iterator pos;
```



# Two phase lookup

- There is two phases for template parse and name lookup

```
void bar()
{
    std::cout << "::bar()" << std::endl;
}
```

```
template <typename T>
class Base
{
public:
    void bar() { std::cout << "Base::bar()" << std::endl; }
};
```

```
template <typename T>
class Derived : public Base<T>
{
public:
    void foo() { bar(); } // calls external bar() or error !!
};
```

# Two phase lookup

- There is two phases for template parse and name lookup

```
void bar()  
{  
    std::cout << "::bar()" << std::endl;  
}
```

```
template <typename T>  
class Base  
{  
public:  
    void bar() { std::cout << "Base::bar()" << std::endl; }  
};
```

```
template <typename T>  
class Derived : public Base<T>  
{  
public:  
    void foo() { this->bar(); }    // or Base::bar()  
};
```

# Mixins

- Class inheriting from its own template parameter
- Not to mix with other mixins (e.g. Scala)
- Reversing the inheritance relationship:
  - One can define the Derived class before Base class
  - Policy/Strategy can be injected

```
template <class Base>
class Mixin : public Base { ... };
```

```
class RealBase { ... };
Mixin<RealBase> rbm;
```

```
class Strategy { ... };
Mixin<Strategy> mws;
```

# Liskov substitutional principle

- Barbara Liskov 1977: object of subtype can replace object of base
- `Mixin<Derived>` is not inherited from `Mixin<Base>`

```
class Base { ... };  
class Derived : public base { ... };
```

```
template <class T> class Mixin : public T { ... };
```

```
Base          b;  
Derived       d;
```

```
Mixin<Base>    mb;  
Mixin<Derived> md;
```

```
b = d          // OK  
mb = md;       // Error!
```

# Liskov substitutional principle

- Barbara Liskov 1977: object of subtype can replace object of base
- `Mixin<Derived>` is not inherited from `Mixin<Base>`

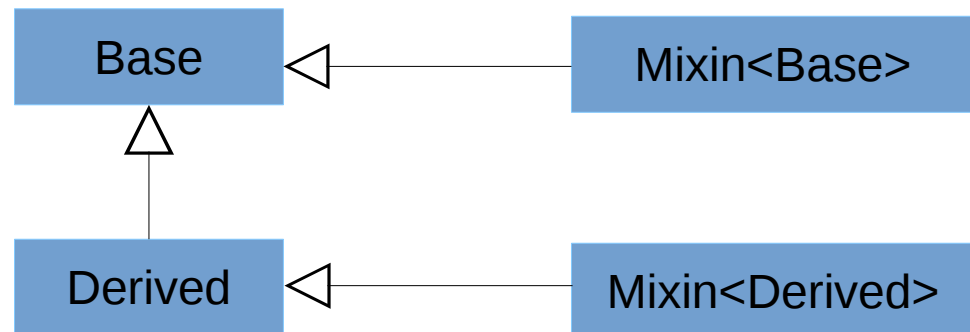
```
class Base { ... };  
class Derived : public Base { ... };
```

```
template <class T> class Mixin : public T { ... };
```

```
Base      b;  
Derived  d;
```

```
Mixin<Base>  mb;  
Mixin<Derived> md;
```

```
b = d      // OK  
mb = md;   // Error!
```



# Liskov substitutional principle

- Barbara Liskov 1977: object of subtype can replace object of base
- `Mixin<Derived>` is not inherited from `Mixin<Base>`

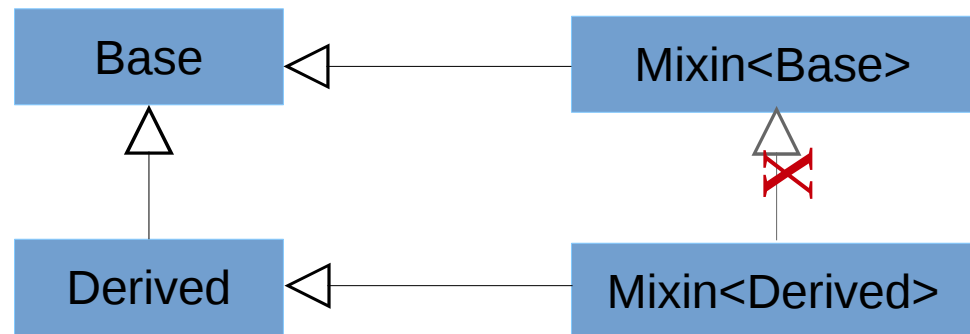
```
class Base { ... };  
class Derived : public Base { ... };
```

```
template <class T> class Mixin : public T { ... };
```

```
Base      b;  
Derived  d;
```

```
Mixin<Base>  mb;  
Mixin<Derived> md;
```

```
b = d      // OK  
mb = md;   // Error!
```



# Constraints

- No concept checking in C++ (yet)

```
class Sortable { void sort(); };
```

```
// be careful with lazy instantiation
```

```
template <class Sortable>
```

```
class WontWork : public Sortable
```

```
{
```

```
public:
```

```
    void sort()
```

```
    {
```

```
        Sortable::srot(); // !!misspelled
```

```
    }
```

```
};
```

```
void client()
```

```
{
```

```
    WontWork<HasSortNotSrot> w; // still compiles
```

```
    w.sort() // syntax error only here!
```

```
}
```

# Curiously Recurring Template Pattern (CRTTP)

- James Coplien
- Static polymorphism

```
template <typename T>
struct Base
{
    // ...
};

struct Derived : Base<Derived>
{
    // ...
};
```



# Curiously Recurring Template Pattern (CRTTP)

- James Coplien 1995
- F-bounded polymorphism 1980's
- Operator generation, `Enable_shared_from_this`, Static polymorphism

```
template <typename T>
struct Base
{
    // ...
};

struct Derived : Base<Derived>
{
    // ...
};
```

# Operator generator

```
class C
{
    // ...
};

bool operator<(const C& l, const C& r)
{
    // ...
};

inline bool operator!=(const C& l, const C& r) { return l<r || r<l; }
inline bool operator==(const C& l, const C& r) { return ! (l<r); }
inline bool operator>=(const C& l, const C& r) { return r < l; }
// ...
```

# Operator generator

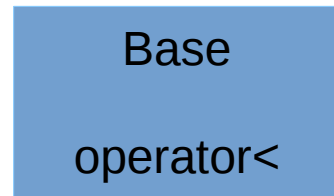
```
class C
{
    // ...
};

bool operator<(const C& l, const C& r)
{
    // ...
};

inline bool operator!=(const C& l, const C& r) { return l<r || r<l; }
inline bool operator==(const C& l, const C& r) { return ! (l<r); }
inline bool operator>=(const C& l, const C& r) { return r < l; }
// ...
```

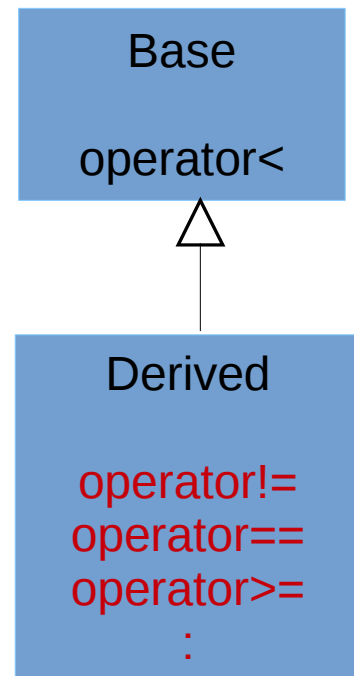
- We want automatically generate the operators from operator<

# Operator generator in Derived

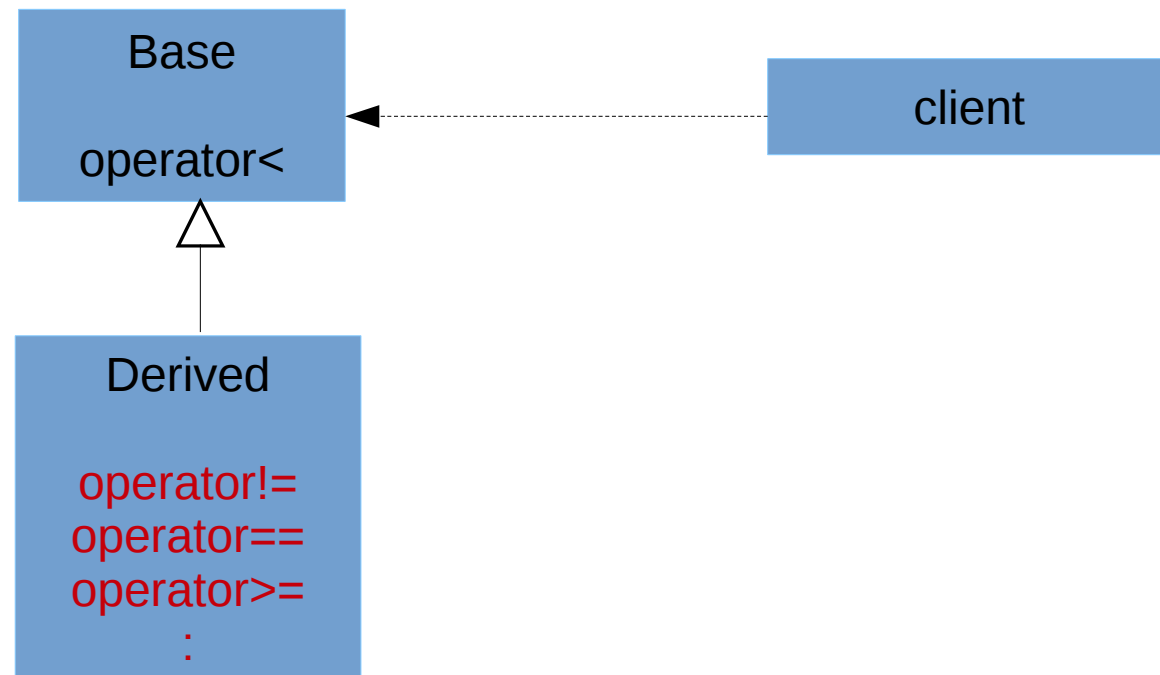


Base  
operator<

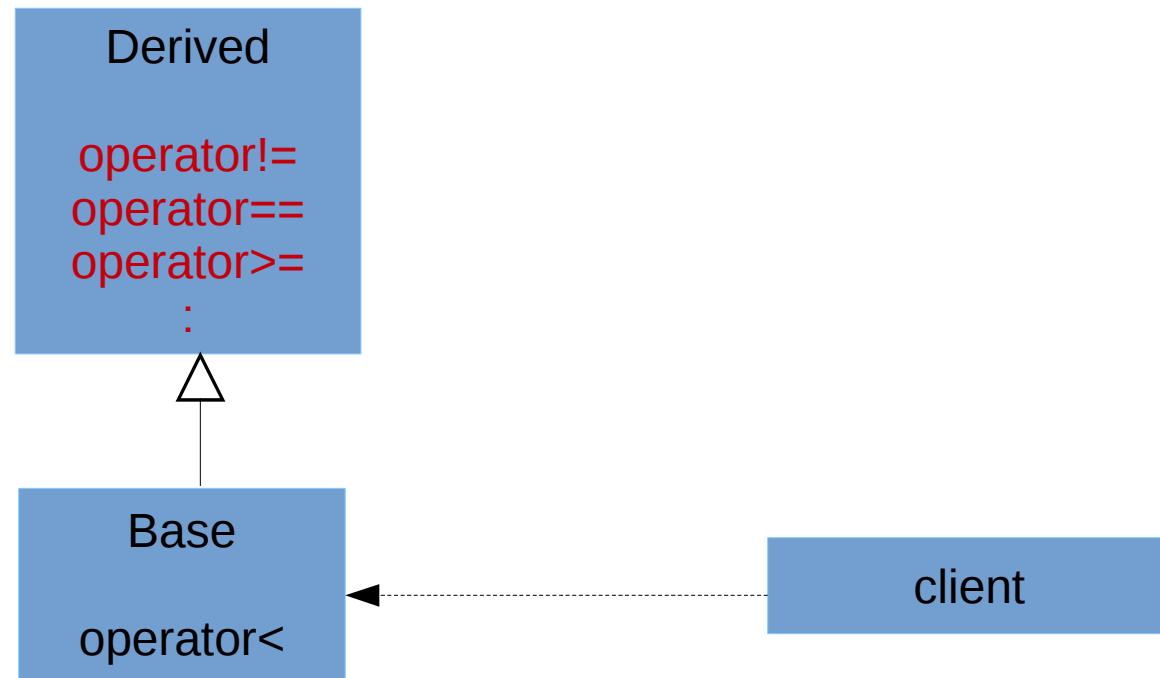
# Operator generator in Derived



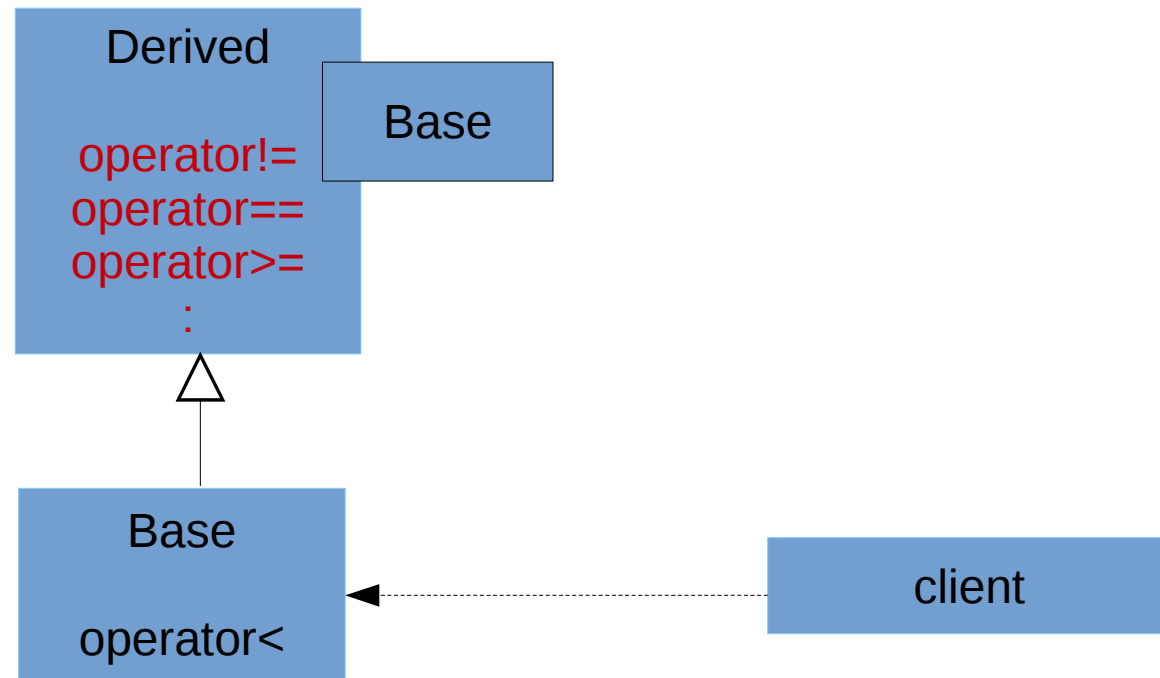
# Operator generator in Derived



# Operator generator in Base



# Operator generator in Base





# Enable shared from this

```
#include <memory>
#include <cassert>

class Y : public std::enable_shared_from_this<Y>
{
public:

    std::shared_ptr<Y> f()
    {
        return shared_from_this();
    }
};

int main()
{
    std::shared_ptr<Y> p(new Y);
    std::shared_ptr<Y> q = p->f();
    assert(p == q);
    assert(!(p < q || q < p)); // p and q must share ownership
}
```

# Object counter

```
template <typename T>
struct counter {
    static int objects_created;
    static int objects_alive;
    counter() {
        ++objects_created;
        ++objects_alive;
    }
    counter(const counter&) {
        ++objects_created;
        ++objects_alive;
    }
protected:
    ~counter() // objects should never be removed through pointers of this type
    {
        --objects_alive;
    }
};
template <typename T> int counter<T>::objects_created( 0 );
template <typename T> int counter<T>::objects_alive( 0 );

class X : counter<X> { ... };
class Y : counter<Y> { ... };
```

# Polymorphic chaining

```
class Printer
{
public:
    Printer(ostream& pstream) : m_stream(pstream) {}

    template <typename T>
    Printer& print(T&& t) { m_stream << t; return *this; }

    template <typename T>
    Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
    ostream& m_stream;
};

void f()
{
    Printer{myStream}.println("hello").println(500); // works!
}
```

# Polymorphic chaining

```
class Printer
{
public:
    Printer(ostream& pstream) : m_stream(pstream) {}

    template <typename T>
    Printer& print(T&& t) { m_stream << t; return *this; }

    template <typename T>
    Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
    ostream& m_stream;
};

class ColorPrinter : public Printer
{
public:
    ColorPrinter() : Printer(cout) {}
    ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};

void f()
{
    Printer{myStream}.println("hello").println(500); // works!
}
}
```

# Polymorphic chaining

```
class Printer
{
public:
    Printer(ostream& pstream) : m_stream(pstream) {}

    template <typename T>
    Printer& print(T&& t) { m_stream << t; return *this; }

    template <typename T>
    Printer& println(T&& t) { m_stream << t << endl; return *this; }
private:
    ostream& m_stream;
};

class ColorPrinter : public Printer
{
public:
    ColorPrinter() : Printer(cout) {}
    ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};

void f()
{
    Printer{myStream}.println("hello").println(500); // works!
    // compile error                               v here we have Printer, not ColorPrinter
    ColorPrinter().print("Hello").SetConsoleColor(Color.red).println("Printer!");
}
```

# Polymorphic chaining

```
template <typename ConcretePrinter>
class Printer
{
public:
    Printer(ostream& ostream, ostream& pstream) : m_stream(pstream) {}

    template <typename T>
    ConcretePrinter& print(T&& t) { m_stream << t;
        return static_cast<ConcretePrinter&>(*this); }

    template <typename T>
    ConcretePrinter& println(T&& t) { m_stream << t << endl;
        return static_cast<ConcretePrinter&>(*this); }

private:
    ostream& m_stream;
};

class ColorPrinter : public Printer<ColorPrinter>
{
public:
    ColorPrinter() : Printer(cout) {}
    ColorPrinter& SetConsoleColor(Color c) { /* ... */ return *this; }
};

void f()
{
    Printer{myStream}.println("hello").println(500); // works!
    ColorPrinter().print("Hello").SetConsoleColor(Color.red).println("Printer!");
}
```

# Static polymorphism

- When we separate interface and implementation
- But no run-time variation between objects

```
template <class Derived>
struct Base
{
    void interface()
    {
        // ...
        static_cast<Derived*>(this)->implementation();
        // ...
    }
    static void static_funcion()
    {
        Derived::static_sub_funcion();
    }
};
struct Derived : Base<Derived>
{
    void implementation();
    static void static_sub_function();
};
```

# Using (C++11)

- Typedef won't work well with templates
- Using introduce type alias

```
using myint = int;
template <class T> using ptr_t = T*;

void f(int) { }
// void f(myint) { }    syntax error: redeclaration of f(int)

// make mystring one parameter template
template <class CharT> using mystring =
    std::basic_string<CharT, std::char_traits<CharT>>;
```



# Variadic templates (C++11)

- Type pack defines sequence of type parameters
- Recursive processing of pack

```
template<typename T>
T sum(T v)
{
    return v;
}
template<typename T, typename... Args> // template parameter pack
T sum(T first, Args... args)         // function parameter pack
{
    return first + sum(args...);
}

int main()
{
    double lsum = sum(1, 2, 3.14, 8L, 7);

    std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
    std::string ssum = sum(s1, s2, s3, s4);
}
```

# Variadic templates (C++11)

- Type pack defines sequence of type parameters
- Recursive processing of pack

```
template<typename T>
T sum(T v)
{
    return v;
}
template<typename T, typename... Args> // template parameter pack
std::common_type<T,Args...>::type sum(T first, Args... args)
{
    return first + sum(args...);
}

int main()
{
    double lsum = sum(1, 2, 3.14, 8L, 7);

    std::string s1 = "x", s2 = "aa", s3 = "bb", s4 = "yy";
    std::string ssum = sum(s1, s2, s3, s4);
}
```

# Mixin reloded (C++11)

- Variadic templates make us possible to define variadic set of base

```
struct A {};  
struct B {};  
struct C {};  
struct D {};
```

```
template<class... Mixins>  
class X : public Mixins...  
{  
public:  
    X(const Mixins&... mixins) : Mixins(mixins)... { }  
};
```

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

# Class template deduction (C++17)

- The compiler can deduce template parameter(s) from
  - Declaration that specifies initialization
  - New expression
  - Function-style cast expressions

```
// examples from cppreference.com
```

```
std::pair p(2,4.5) // C++11: std::pair<int,double>(2,4.5)  
std::vector v = { 1, 2, 3, 4}; // std::vector<int>
```

```
template <class T> struct A { A(T,T); };  
auto y = new A{1,2}; // A<int>{1,2}
```

```
std::mutex mtx;  
auto lck = std::lock_guard(mtx); // std::lock_guard<std::mutex>(mtx)  
std::copy_n(v1,3, std::back_inserter(v2)); // back_inserter(v2)
```

# Class template deduction (C++17)

- Automatic and User defined deduction guideline

```
// example from cppreference.com
template <class T> struct Container
{
    Container(T t) {}
    template<class It> Container(It beg, It end);
};

template<class It>
Container(It beg, It end) ->
    Container<typename std::iterator_traits<It>::value_type>;

int main()
{
    Container c(7) // ok T=int, using automatic guide
    std::vector<double> vd = { 3.14, 4.14, 5.14 };
    auto c = Container(v.begin(), v.end()); // ok, T=double using guide
    Container d = {5,6}; // error
}
```

# Fold expressions (C++17)

- Reduces (folds) a parameter pack over a binary operator
- Syntax

<code>( pack op ... )</code>	unary right fold	<code>E1 op (...op(En-1 op En))</code>
<code>( pack op ... op init )</code>	binary right fold	<code>E1 op (...op(En-1 op (En op i)))</code>
<code>( ... op pack )</code>	unary left fold	<code>((E1 op E2) op...) op En</code>
<code>( init op ... op pack )</code>	binary left fold	<code>(((i op E1) op E2) op...) op En</code>

```
template <typename... Args>  
bool all(Args... args) { return ( ... && args); }
```

```
int main()  
{  
    bool b = all( i1, i2, i3, i4);  
    // = ((i1 && i2) && i3) && i4;  
}
```

# Examples: variadic template

```
#include <sstream>
#include <iostream>
#include <vector>

template <typename T>
std::string to_string_impl(const T& t)
{
    std::stringstream ss;
    ss << t;
    return ss.str();
}

std::vector<std::string> to_string()
{
    return {};
}

template <typename P1, typename ...Param>
std::vector<std::string> to_string(const P1& p1, const Param&... params)
{
    std::vector<std::string> s;
    s.push_back(to_string_impl(p1));

    const auto remainder = to_string(params...);
    s.insert(s.end(), remainder.begin(), remainder.end());
    return s;
}

int main()
{
    const auto vec = to_string("hello", 1, 4.5);
    for (const auto& x : vec )
        std::cout << x << std::endl;
}
```

# Examples: variadic template

```
#include <sstream>
#include <iostream>
#include <vector>

template <typename T>
std::string to_string_impl(const T& t)
{
    std::stringstream ss;
    ss << t;
    return ss.str();
}

std::vector<std::string> to_string()
{
    return {};
}

template <typename P1, typename ...Param>
std::vector<std::string> to_string(const P1& p1, const Param&... params)
{
    return { to_string_impl(params)... }; // std::initializer_list
    std::vector<std::string> s;
    s.push_back(to_string_impl(p1));

    const auto remainder = to_string(params...);
    s.insert(s.end(), remainder.begin(), remainder.end());
    return s;
}

int main()
{
    const auto vec = to_string("hello", 1, 4.5);
    for (const auto& x : vec )
        std::cout << x << std::endl;
}
```



# Examples: variadic template

```
#include <sstream>
#include <iostream>
#include <vector>

template <typename ...Param>
std::vector<std::string> to_string(const Param&... params)
{
    const auto to_string_impl = [] (const auto& t) { // generic lambda C++14
        std::stringstream ss;
        ss << t;
        return ss.str();
    };

    return { to_string_impl(params)... }; // std::initializer_list
}

int main()
{
    const auto vec = to_string("hello", 1, 4.5);
    for (const auto& x : vec )
        std::cout << x << std::endl;
}
```

# Examples: fold expressions

```
#include <iostream>

template <typename ...T>
auto sum(T... t)
{
    typename std::common_type<T...>::type result{};
    std::initializer_list<int>{ (result += t, 0)... };
    return result;
}

int main()
{
    std::cout << sum(1,2,3.0,4.5) << std::endl;
}
```

# Examples: fold expressions

```
#include <iostream>

template <typename ...T>
auto sum(T... t)
{
    typename std::common_type<T...>::type result{};
    std::initializer_list<int>{ (result += t, 0)... };
    return ( t + ... ); // from C++17 e.g. clang-3.8
}

int main()
{
    std::cout << sum(1,2,3.0,4.5) << std::endl;
}
```

# Examples: fold expressions

```
#include <iostream>

template <typename ...T>
auto sum(T... t)
{
    typename std::common_type<T...>::type result{};
    std::initializer_list<int>{ (result += t, 0)... };
    return ( t + ... ); // from C++17 e.g. clang-3.8
}

template <typename ...T>
auto avg(T... t)
{
    return ( t + ... ) / sizeof...(t); // from C++17
}

int main()
{
    std::cout << sum(1,2,3.0,4.5) << std::endl;
    std::cout << avg(1,2,3.0,4.5) << std::endl;
}
```