Basic C++

Templates

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

Templates

- From macros to templates
- Parameter deduction, instantiation,specialization
- Class templates, partial specialization
- Two phase lookup
- 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
- Unconstrained generics (but Concepts since C++20)
- Templates are not functions, they are skeletons
- Parameter deduction + Instantiation
- Definitions are placed in header files

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);
}
```

How function calls resolved

- First: check for non-templates with exact parameter match
- Second: check for non-templates with exact parameter match
  - Parameter deduction for all parameters with no conversion
  - Choose the most specific template
  - If successful, instantiate specialization
- Third: check for non templates with parameter conversion

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

```cpp
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); // 4
    h = max(f,g); // 5.55
    h = max(i,f); // ?
}
```

```cpp
template <typename T>
void swap( T& x, T& y)
{
    T temp{std::move(x)};
    x = std::move(y);
    y = std::move(temp);
}
template <typename T>
constexpr const T& max(const T& a,const T& b)
{
    if ( a < b )
        return b;
    else
        return a;
}
```

Templates with more types

```
template <class T, class S>
T max( T a, S b)   // is this ok?
{
    if ( a > b )
        return a;
    else
        return b;
}

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

  z = max( i, x);     // z == 3.0
}
```

Templates with more types

```
template <class R, class T, class S>
R max( T a, S b)    // is this ok?
{
    if ( a > b )
        return a;
    else
        return b;
}

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

  z = max( i, x);     // compile error: no deduction on return type
}
```

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); // works, but...

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 long(i) and int(x)
k = max<int, int, int>( i, j);          // too complex notation
```

No deduction on return type

```
template <class T, class S>
std::common_type<T,S>::value max( T a, S b) // since C++11
{
    if ( a > b )
        return a;
    else
        return b;
}


template <class T, class S>
auto max( T a, S b)          // since C++14: return type deduction
{
    if ( a > b )
        return a;
    else
        return b;
}
```

Template specialization

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

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

Template overloading

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

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

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

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

Template overloading

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

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

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

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

Class templates

- All member functions are templates
- Lazy instantiation
- Possibility of partial specialization
- Specialization(s) may completely different
- Default parameters are allowed

Class template

```
// complex.h

#ifndef COMPLEX_H
#define COMPLEX_H

template <typename T>
struct complex_t
{
    T re;
    T im;
};

template <typename T>
bool operator==(complex_t<T> c1, complex_t<T> c2)
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_H
```

Class template

```
// complex.h

#ifndef COMPLEX_H
#define COMPLEX_H

template <typename T>
struct complex_t
{
    T re;
    T im;
};

template <typename T>
bool operator==(complex_t<T> c1, complex_t<T> c2)
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_H
```

```
#include “complex.h”

bool f(complex_t<double> par)
{
    complex_t<double> c{1.,3.14};
    return c == par;
}
```

Class template

```
// complex.h

#ifndef COMPLEX_H
#define COMPLEX_H

template <typename T=double>
struct complex_t
{
    T re;
    T im;
};

template <typename T=double>
bool operator==(complex_t<T> c1, complex_t<T> c2)
{
    return c1.re == c2.re && c1.im == c2.im;
}

#endif // COMPLEX_H
```

```
#include “complex.h”

bool f(complex_t<> par)
{
    complex_t<> c{1.,3.14};
    return c == par;
}
```

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;
```

Dependent types

- There are a few exceptions, where **typename** is not needed
- Before C++20
  - Inheritance
  - Member initialization ids
- Since C++20
  - Using declaration
  - Data member declaration
  - Function parameters
  - Default argument of template
  - Type of casts

```
template <typename T>
// Before C++20
class MyClass : T::X  // base class
{
   int i{T::val}; // member initializ.

   // Since C++20
   using TX = T::X; // using
   T::X  member;    // data member
   void f( T::X param); // func param
};
```

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(); }  // compile error or calls external bar()
};
```

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()
};
```

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();
  }
};
template <typename T>
void execute( std::vector<T*> v) {
  for( auto ptr : v ) ptr->interface();
}

struct Derived1 : Base<Derived1> {
  void implementation();
};
struct Derived2 : Base<Derived2> {
  void implementation();
};
std::vector<Base<Derived1>*> v1; /* ... */ execute(v1);
std::vector<Base<Derived2>*> v2; /* ... */ execute(v2);
```

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

- Concept checking only since C++20
- Lazy instantiation

```
template <class Sortable>
class MyMixin : public Sortable
{
public:
    void f()  // will use Sortable::sort
    {
        this->sort();  // lookup delayed due to two phase lookup
    }
};

struct X { void srot(); };  // not sortable

void client()
{
  MyMixin<X> w;  // this line still compiles
  w.f()          // syntax error only here!
}
```

Curiously Recurring Template Pattern (CRTP)

- James Coplien
- Static polymorphism

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

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

Curiously Recurring Template Pattern (CRTP)

- 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

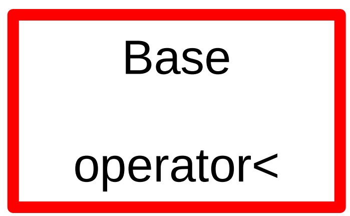
```
class C
{
  // ...
};

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

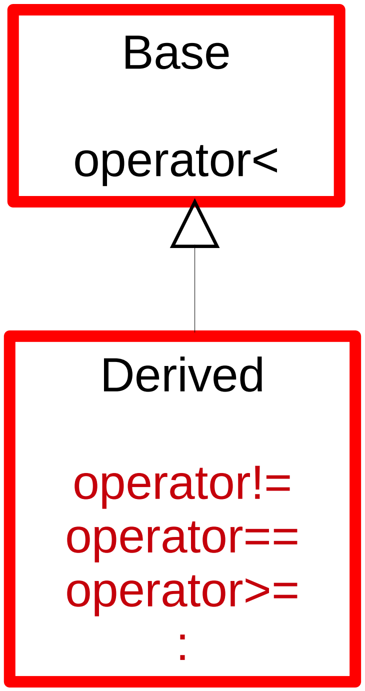
```
class C
{
  // ...
};

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<

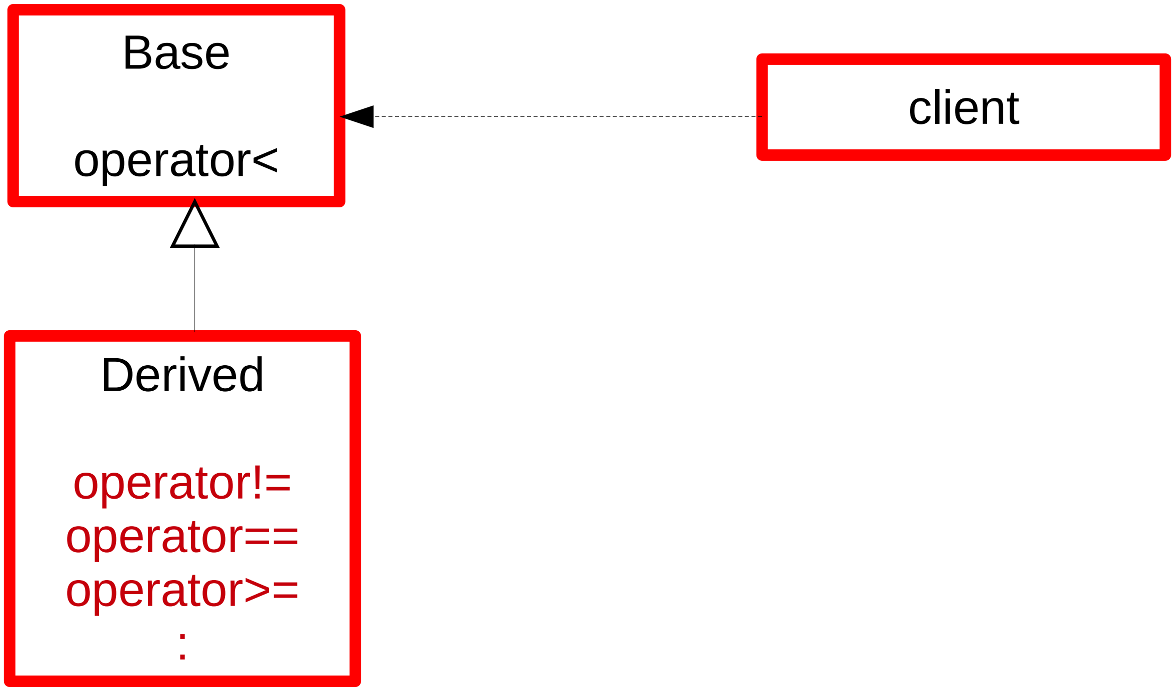
Operator generator in Derived



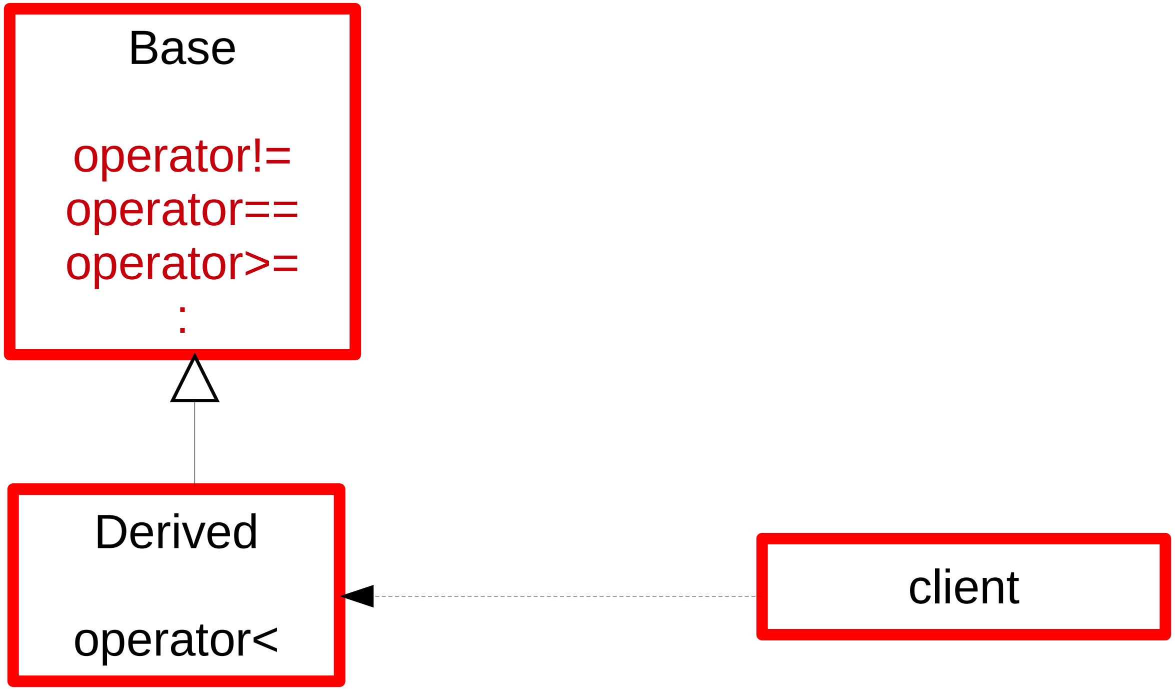
Operator generator in Derived



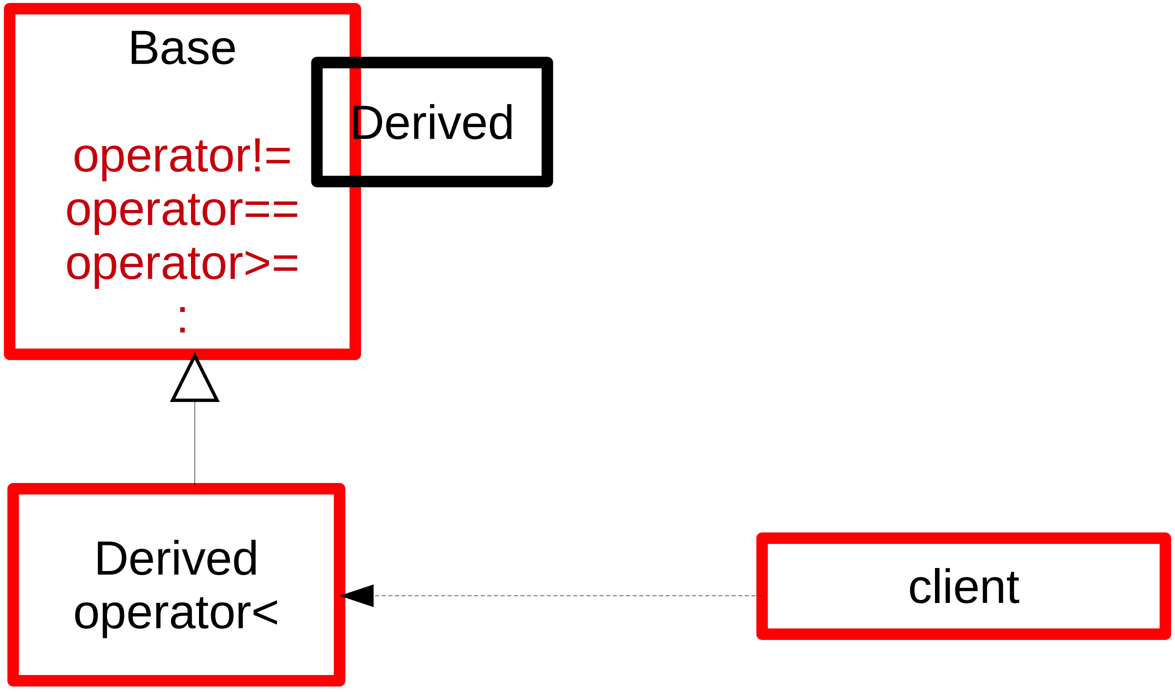
Operator generator in Derived



Operator generator in Base



Operator generator in Base



Enable shared from this

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

class Y
{
public:

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

Enable shared from this

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

class Y
{
public:

    std::shared_ptr<Y> f()   // BAD!!!
    {
        return shared_ptr<Y>(this);  // ???
    }
};

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

Enable shared from this

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

class Y
{
public:
    Y() : ptr_to_me(std::shared_ptr<Y>(this)) { }
    std::shared_ptr<Y> f()
    {
        return ptr_to_me;  // ???
    }
private:
    std::shared_ptr<Y> ptr_to_me;
};

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

Enable shared from this

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

class Y
{
public:
    Y() : ptr_to_me(std::shared_ptr<Y>(this)) { }
    std::shared_ptr<Y> f()  // BAD!!!
    {
        return ptr_to_me;  // ???
    }
private:
    std::shared_ptr<Y> ptr_to_me;
};

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

Enable shared from this

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

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

    std::shared_ptr<Y> f()  // OK
    {
        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 share ownership
}
```

Enable shared from this

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

template <typename Y>
class std::enable_shared_from_this<Y>
{
public:
  enable_shared_from_this(...)
  {
    // ...
    if (ptr != nullptr && ptr->weak_this.expired())
     {
       ptr->weak_this = std::shared_ptr<std::remove_cv_t<Y>>(
                                  *this,
                                  const_cast<std::remove_cv_t<U>*>(ptr));
     }
    // ...
private:
  mutable std::weak_ptr<Y>    weak_this;
}
```

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& 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();
  }
};
template <typename T>
void execute( std::vector<T*> v) {
  for( auto ptr : v ) ptr->interface();
}

struct Derived1 : Base<Derived1> {
  void implementation();
};
struct Derived2 : Base<Derived2> {
  void implementation();
};
std::vector<Base<Derived1>*> v1; /* ... */ execute(v1);
std::vector<Base<Derived2>*> v2; /* ... */ execute(v2);
```

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>
T 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";

}
```

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";

}
```

Parameter packs

- Pattern(Args)… is replaced by Pattern(Arg1), Pattern(Arg2),...
- Recursive processing of pack

```
template <class... Ts>
void f(Ts... args)      // ---> void f(int arg1, char arg2)
{
  g(args...);       // Pattern ==   args -----> g(  arg1,   arg2);
  g(++args...);     // Pattern == ++args -----> g(++arg1, ++arg2);
  g(1, ++args...);  // Pattern == ++args -----> g(1, ++arg1, ++arg2);
  g(++args..., 9);  // Pattern == ++args -----> g(++arg1, ++arg2, 9);
  g(++args)...;     // Pattern == g(++args) --> g(++arg1), g(++arg2);
  h(g(const_cast<const Ts*>(&args))...);
                    // Pattern == g(cons_cast<const Ts*>(&args) --->
                    //     h( g(const_cast<const int*>(&arg1)),
                    //        g(const_cast<const char*>(&arg2)) );
  h(g(args...)+args...); // nested:
                 // inner: h(g(arg1,arg2) + args...);
                 // outer: h(g(arg1,arg2)+arg1, g(arg1,arg2)+arg2 );
}
f(1,'a');
```

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>
T 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";

}
```

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>
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";

}
```

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::vetor<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_insert_iterator(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
}
```

Class template argument deduction

- Possible traps

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

int main()
{
  std::vector v1{1,2};
  std::vector v2(v1.begin(), v1.end());
  std::vector v3{v1.begin(), v1.end()};

  std::cout << v1[0] << v2[0] << '\n'; //<< v3[0] << '\n';
  return 0;
}

// vector<T>::vector<T>(std::initializer_list<T>);
// <Iter> vector<T>::vector<T>(Iter b, Iter e);
//       T := Iter::value_type
```

Class template argument deduction

- Possible traps
- C++17 : The Biggest Traps - Nicolai Josuttis [C++ on Sea 2019]

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

void f(std::set<int> coll)
{
  std::vector v1{1,2};     // vector<int>   2 elements
  std::vector v2{1};       // vector<int>   1 elements
  std::vector v3(3,4);     // vector<int>   3 elements
  std::vector v4(8);       // error: can’t deduct T
  std::vector v5(8,"");    // error: vector<const char[1]> deduced
  std::vector v6(coll.begin(), coll.end()); // vector<int> deduced
  std::vector v7{coll.begin(), coll.end()}; // vector<iter> deduced
  std::vector v8{"hello", "word"}; // vector<const char *> deduced
  std::vector v8("hello", "word"); // vector<It=const char *> deduced
}
```

Class template argument deduction

- Possible traps

```
#include <iostream>
#include <type_traits>
#include <optional>

int main()
{
  std::optional  maybe_int(1);
  std::optional  maybe_maybe_int(maybe_int);
  std::cout <<
    std::is_same_v<decltype(maybe_int), decltype(maybe_maybe_int)>
              << '\n';
  return 0;
}
```

Class template argument deduction

- Possible traps

```
#include <iostream>
#include <type_traits>
#include <optional>

int main()
{
  std::optional  maybe_int(1);
  std::optional  maybe_maybe_int(maybe_int);
  std::cout << std::boolapha <<
    std::is_same_v<decltype(maybe_int), decltype(maybe_maybe_int)>
            << '\n';
  return 0;
}

true
```

Class template argument deduction

- Possible traps
- https://akrzemi1.wordpress.com/2018/12/09/deducing-your-intentions/

```
#include <iostream>
#include <type_traits>
#include <optional>

int main()
{
  std::optional  maybe_int(1);                   // intention: wrap
  std::optional  maybe_maybe_int(maybe_int);  // intention: copy
  std::cout <<
    std::is_same_v<decltype(maybe_int), decltype(maybe_maybe_int)>
            << '\n';
  return 0;
}

template <class T> auto __deduce(T) -> optional<T>;
template <class T> auto __deduce(const optional<T>&) -> optional<T>;
```

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<>         x0{};
  X<A,B,C,D> xx{a,b,c,d};
  X<A,A>      xa{a,a} // Error: duplicate base type ‘A’ invalid
}
```

std::visit

```cpp
using var_t = std::variant<int, long, double, std::string>; // the variant to visit
template<class T> struct always_false : std::false_type {}; // helper type for the visitor

int main()
{
    std::vector<var_t> vec = {10, 15l, 1.5, "hello"};
    for(auto& v: vec)
    {
        // 1. void visitor, only called for side-effects (here, for I/O)
        std::visit([](auto&& arg){std::cout << arg;}, v);

        // 2. value-returning visitor, demonstrates the idiom of returning another variant
        var_t w = std::visit([](auto&& arg) -> var_t {return arg + arg;}, v);

        // 3. type-matching visitor: a lambda that handles each type differently
        std::cout << ". After doubling, variant holds ";
        std::visit([](auto&& arg) {
            using T = std::decay_t<decltype(arg)>;
            if constexpr (std::is_same_v<T, int>)
                std::cout << "int with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, long>)
                std::cout << "long with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, double>)
                std::cout << "double with value " << arg << '\n';
            else if constexpr (std::is_same_v<T, std::string>)
                std::cout << "std::string with value " << std::quoted(arg) << '\n';
            else
                static_assert(always_false<T>::value, "non-exhaustive visitor!");
        }, w);
    }
}
```

std::visit

```
using var_t = std::variant<int, long, double, std::string>; // the variant to visit

// helper type for the visitor
template <class... Ts> struct overloaded : Ts... { using Ts::operator()...; };
template <class... Ts> overloaded(Ts...) -> overloaded<Ts...>;

int main()
{
    std::vector<var_t> vec = {10, 15L, 1.5, "hello"};

    for (auto& v: vec)
    {
        // type-matching visitor: a class with 4 overloaded operator()'s
        std::visit(overloaded {
            [](int arg) { std::cout << arg << ' '; },
            [](long arg) { std::cout << arg << ' '; },
            [](double arg) { std::cout << std::fixed << arg << ' '; },
            [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; },

        }, v);
    }

}
```

```
10 15 1.500000 “hello”
```

std::visit

```
using var_t = std::variant<int, long, double, std::string>; // the variant to visit

// helper type for the visitor
template <class... Ts> struct overloaded : Ts... {
    using Ts::operator()...;
    consteval void operator()(auto) { static_assert(false, "Unsupported type"); }  // C++23
};
// since C++20 no need for deduction guide for aggregates
// since C++23 static_assert(false) generates error only when the template instantiated

int main()
{
    std::vector<var_t> vec = {10, 15L, 1.5, "hello"};

    for (auto& v: vec)
    {
        // type-matching visitor: a class with 3 overloaded operator()'s
        std::visit(overloaded {
            [](int arg) { std::cout << arg << ' '; },
            [](long arg) { std::cout << arg << ' '; },
            [](double arg) { std::cout << std::fixed << arg << ' '; },
            [](const std::string& arg) { std::cout << std::quoted(arg) << ' '; },
        }, v);
    }

}
```



```
10 15 1.500000 "hello"
```

Fold expressions (C++17)

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

(pack op …)	unary right fold	E1 op (…op(En-1 op En))
(pack op … op init)	binary right fold	E1 op (…op(En-1 op (En op i)))
(… op pack)	unary left fold	((E1 op E2) op…) op En
(init op … op pack)	binary left fold	(((i op E1) op E2) op…) op En

```
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;
}
```