Basic C++

7

Dr. Porkoláb Zoltán Károly

gsd@inf.elte.hu

http://gsd.web.elte.hu

Lambda functions

- Terminology
- How it is compiled
- Capture by value and reference
- Mutable lambdas
- Use of this
- Init capture and generalized lambdas in C++14
- Constexpr lambda and capture *this and C++17

Functor

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

using namespace std;

struct PrinterFunctor
{
  void operator()(int n) const { cout << n << " "; }
};

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), PrinterFunctor());
    cout << endl;
    return 0;
}

$ g++ l.cpp
$ ./a.out
0 1 2 3 4 5 6 7 8 9
```

Lambda

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

using namespace std;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
    cout << endl;
    return 0;
}

$ g++ l.cpp
$ ./a.out
0 1 2 3 4 5 6 7 8 9
```

Lambdas are mapped to functors

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; } );
    cout << endl;
    return 0;
}
```

Lambdas are mapped to functors

```
// [](int n) { cout << n << " "; }

struct LambdaFunctor
{
    void operator() (int n) const { cout << n << " "; }
};

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), LambdaFunctor() );
    cout << endl;
    return 0;
}
```

Lambda

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

using namespace std;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
    cout << endl;
    return 0;
}

$ g++ l.cpp
$ ./a.out
0 1 2 3 4 5 6 7 8 9
```

Lambda introducer with opt. capture

Lambda parameter declaration

Optional return type
in form: -> type

Lambda terminology

- Lambda expression

  [ ] (int n) {   }

- Closure object
  - Runtime object created from lambda expression
  - Copy constructable (but not copy assignable)
  - Can be stored in std::function
  - May hold captured variables
- Closure class
  - The type of the closure object
  - Deleted default constructor and copy assignment operator

Explicit return type

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    deque<double> dd;
    deque<int>    di;

    transform(v.begin(), v.end(), front_inserter(dd),
                              [](int n) -> double  { return n / 2.0; } );

    transform(v.begin(), v.end(), back_inserter(di),
                              [](int n) -> int     { return n / 2.0; } );

    for_each(dd.begin(), dd.end(), [](double n) { cout << n << " "; });
    cout << endl;
    for_each(di.begin(), di.end(), [](double n) { cout << n << " "; });
    cout << endl;

    return 0;
}
```

```
4.5 4 3.5 3 2.5 2 1.5 1 0.5 0
0 0 1 1 2 2 3 3 4 4
```

Can contain multiple statements

```
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    for_each(v.begin(), v.end(), [](int n) {
             cout << n ;
             if ( n % 2 )
                 cout << ":odd ";
             else
                 cout << ":even ";
                         });
    cout << endl;
    return 0;
}
0:even 1:odd 2:even 3:odd 4:even 5:odd 6:even 7:odd 8:even 9:odd
```

Capture

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int x = 0;
    int y = 0;

    cin >> x >> y;  // read 3 6

    v.erase( remove_if(v.begin(),v.end(),[x,y](int n) { return x < n && n < y; }),
             v.end()
           );

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });

    cout << endl;
    return 0;
}

3 6
0 1 2 3 6 7 8 9
```

Capture by value

Capture by value

```
[x,y](int n) { return x < n && n < y; }

struct LambdaFunctor
{
public:
    LambdaFunctor(int a, int b) : m_a(a), m_b(b) { }
    bool operator()(int n) const { return m_a < n && n < m_b; }
private:
    int m_a;
    int m_b;
};

// ...

 v.erase( remove_if(v.begin(),v.end(),LambdaFunctor(x,y)), v.end());
```

copy

The x and y parameters are copied and being stored in the function object. We cannot modify the captured values because the **operator**() in functor is **const**. It is a real copy, therefore the modification of x and y is not reflected inside the lambda.

Capture by reference

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum,x,y](int n) { if (x < n && n < y) sum += n; });

    cout << "sum = " << sum << endl;

    return 0;
}

2 7
sum = 18
```

Mutable lambda

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum,x,y](int n) { if (x-- < n && n < y++) sum += n; });

    cout << "sum = " << sum << endl;

    return 0;
}

$ g++ -std=c++11 -Wall -pedantic lambda4.cpp
lambda4.cpp: In lambda function:
lambda4.cpp:18:62: error: decrement of read-only variable ‘x’
     std::for_each(v.begin(),v.end(),[&sum,x,y](int n) { if (x--<n && n<y++) sum+=n; });
                                                              ^~
lambda4.cpp:18:73: error: increment of read-only variable ‘y’
 std::for_each(v.begin(),v.end(),[&sum,x,y](int n) { if (x--<n && n<y++) sum+=n; });
                                                                     ^~
```

Mutable lambda

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum,x,y](int n) mutable { if (x-- < n && n < y++)
                                                                                sum += n; });

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = 2
y = 7
```

Globals are not captured

```
int   x = 0;
int   y = 0;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum,x,y](int n) mutable { if (x-- < n && n < y++)
                                                                       sum += n; });

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

$ g++ -std=c++11 -Wall -pedantic lambda4.cpp
lambda5.cpp: In function main:
lambda5.cpp:19:43: warning: capture of variable ‘x’ with non-automatic storage duration
lambda5.cpp:19:45: warning: capture of variable ‘y’ with non-automatic storage duration
```

Globals are not captured

```
int   x = 0;
int   y = 0;

int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int sum = 0;

    cin >> x >> y;  // read 2 7

    for_each(v.begin(),v.end(),[&sum](int n) mutable { if (x-- < n && n < y++)
                                                                    sum += n; });
    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = -8
y = 15
```

Capturing summary

- No capture

  [ ]
- By value

  [x,y]   [=]
- By reference

  [&x, &y]   [&]
- Mixed:

  [=, &x, &y]   [&, x, y]
- Only automatic lifetime (local) variables could be captured
- Constness is preserved on capture
- Global variables, static members, heap storage can be used if visible but they are not captured

Capturing *this*

```
struct X
{
    int s;
    vector<int> v;
    void print() const
    {
        for_each(v.begin(), v.end(), [](int n) { cout << n*s << " "; });
    }
};

int main()
{
    X x;
    x.s = 2;
    for(int i = 0; i < 10; ++i)
        x.v.push_back(i);

    x.print();
    return 0;
}
```

```
$ g++ l10.cpp
l10.cpp: In lambda function:
l10.cpp:15:60: error: ‘this’ was not captured for this lambda function
```

Capturing *this*

```
struct X
{
    int s;
    vector<int> v;
    void print() const
    {
        for_each(v.begin(), v.end(), [this](int n) { cout << n*s << " "; });
    }
};

int main()
{
    X x;
    x.s = 2;
    for(int i = 0; i < 10; ++i)
        x.v.push_back(i);

    x.print();
    return 0;
}
```

```
0 2 4 6 8 10 12 14 16 18
```

Capturing *this*

```
struct X
{
    int s;
    vector<int> v;
    void print() const
    {
        int s = 9;
        for_each(v.begin(), v.end(), [this,s](int n) { cout << n*s << " "
                                                    << this->s << " "; });
    }
};

int main()
{
    X x;
    x.s = 2;
    for(int i = 0; i < 10; ++i)
        x.v.push_back(i);

    x.print();
    return 0;
}
```

```
0 2 9 2 18 2 27 2 36 2 45 2 54 2 63 2 72 2 81 2
```

Capturing *this*

- The **this** not captured by default
- The **this** is always captured by value
- [=] implicitly captures **this**
- Since C++17 ***this** can be captured (by value)
- Capturing **this** can be dangerous
  - Storing a non-smart pointer
  - Lifetime may already finished when lambda function is called

Capturing *this*

```
std::function<void (int)> f;  // global

struct X
{
  X(int i) : ii(i) {}
  int ii;
  void addLambda()
  {
    f = [=](int n) { if (n == ii) cout << n;  // [=] captures this if needs
                     else         cout << ii; // this->ii, indicates capturing this
                   };
  }
};

int main()
{
  {
    std::unique_ptr<X> up = std::make_unique<X>(4);
    up->addLambda();
    f(4);
  }  // object pointed by "up" destroyed here

  f(4);     // Likely aborts! The captured this points to already dead object
  return 0;
}
```

Copying lambda

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    cin >> x >> y;  // read 2 7


    for_each(v.begin(),v.end(),f);

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 0
x = 2
y = 7
```

Copying lambda

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;
    // auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    for_each(v.begin(),v.end(),f);

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = 2
y = 7
```

Copying lambda

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;
    // auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };

    for_each(v.begin(),v.end(),f);
    for_each(v.begin(),v.end(),f);  // 2nd time

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}
```

```
2 7
sum = 88
x = 2
y = 7
```

Copying lambda

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; }; // copy constr

    for_each(v.begin(),v.end(),f);  // copy constr






    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = 2
y = 7
```

Copying lambda

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; }; // copy constr

    for_each(v.begin(),v.end(),f);  // copy constr

    // f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };  op= deleted




    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 44
x = 2
y = 7
```

Copying lambda

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    int   x = 0;
    int   y = 0;
    int sum = 0;

    cin >> x >> y;  // read 2 7
    auto f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; }; // copy constr

    for_each(v.begin(),v.end(),f);  // copy constr

    // f = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; };  op= deleted

    const auto& f2 = [=,&sum](int n) mutable { if( x-- < n && n < y++ ) sum += n; }; // ref

    for_each(v.begin(),v.end(),f2);  // copy constr

    cout << "sum = " << sum << endl;
    cout << "x = " << x << endl;
    cout << "y = " << y << endl;
    return 0;
}

2 7
sum = 88
x = 2
y = 7
```

Nullary lambdas

```
int main()
{
    vector<int> v;
    int i = 0;

    generate_n(back_inserter(v), 10, [&] { return i++; } );

    for_each(v.begin(), v.end(), [](int n) { cout << n << " "; });
    return 0;
}
```

```
0 1 2 3 4 5 6 7 8 9
```

Conversion to function pointer

```
int main()
{
    vector<int> v;
    for(int i = 0; i < 10; ++i)
        v.push_back(i);

    deque<double> dd;
    deque<int>    di;

    double (*fp1)(int) = [](int n) -> double  { return n / 2.0; };   // no capture
    int    (*fp2)(int) = [](int n) -> int     { return n / 2.0; };   // no capture
    void   (*fp3)(double) = [](double n) { std::cout << n << " "; }; // no capture

    std::transform(v.begin(), v.end(), front_inserter(dd), fp1);
    std::transform(v.begin(), v.end(), back_inserter(di),  fp2);

    for_each(dd.begin(), dd.end(), fp3);
    cout << endl;
    for_each(di.begin(), di.end(), fp3);
    cout << endl;

    return 0;
}
```

```
4.5 4 3.5 3 2.5 2 1.5 1 0.5 0
0 0 1 1 2 2 3 3 4 4
```

IIFE – Immediately Invoked Function Expression

```
/* const */ int i = some_default_value; // can't do it const
                                        // since value depends
if(someConditionIstrue)                 // on some condition.
{
    // Do some operations and calculate the value of i;
    i = // some calculated value;
}
int x = i; // use i

// But unfortunately in this case there is no way to guarantee
// it is used as a constant, so now if some one comes and does
i = 10; // This is valid
==========================================================

const int i = [&]{

    int i = some_default_value;

    if(someConditionIstrue)
    {
        // Do some operations and calculate the value of i;
        i = // some calculated value;
    }
    return i;

} (); // note: () invokes the lambda!
```

Generalized lambdas in C++14

```
auto L = [](const auto& x, auto& y){ return x + y; };

means:

struct /* anonymous */
{
    template <typename T, typename U>
    auto operator()(const T& x, U& y) const // N3386 Return type deduction
    {
        return x + y;
    }
} L;
```

Generalized lambdas in C++14

```
int main()
{
    auto my_lambda = [](auto a, auto b) { return a < b; };

    float af = 1.5, bf = 2.0;
    int ai = 3, bi = 1;
    std::string as = “Hello”, bs = “World”;

    std::cout << "Float: "   << my_lambda(af, bf) << std::endl;
    std::cout << "Integer: " << my_lambda(ai, bi) << std::endl;
    std::cout << "String: "  << my_lambda(as, bs) << std::endl;

    return 0;
}
```

Init capture in C++14

```
auto up = std::make_unique<X>();

auto func = [up = std::move(up)] { return up->f(); }
```

```
   up is a member     outer up is   here we use the member up
inside the lambda      captured       inside the lambda
```

```
auto func = [up = std::make_unique<X>()] { return up->f(); }

auto func = [x = std::as_const(x)] { ... }     // make x const inside the lambda
```

Init capture example

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

int main()
{
  std::vector<int> v;
  int i = 0;

  auto f = [cnt = 0](int n) mutable { std::cout << ++cnt << ":" << n << " "; };

  std::generate_n( std::back_inserter(v), 10, [&] { return i++; } );

  std::for_each( v.begin(), v.end(), f);
  std::for_each( v.begin(), v.end(), f);
  return 0;
}
```

```
1:0 2:1 3:2 4:3 5:4 6:5 7:6 8:7 9:8 10:9 1:0 2:1 3:2 4:3 5:4 6:5 7:6 8:7 9:8 10:9
```

Constexpr lambda in C++17

```
#include <iostream>

int main()
{
    constexpr auto multi = [](int a, int b){ return a * b; };

    static_assert(multi(3,7) == 21, "3x7 == 21");
    static_assert(multi(4,5) == 15, "4x5 != 15");

    return 0;
}
```

Constexpr lambda in C++17

```
template<typename Range, typename Func, typename T>
constexpr T SimpleAccumulate(const Range& range, Func func, T init)
{
    for (auto &&elem : range)
    {
        init += func(elem);
    }
    return init;
}
int main()
{
    constexpr int t[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    constexpr int x = 2;
    constexpr int y = 7;

    constexpr auto f = [x,y](int n) { return (x < n && n < y) ? 0 : n; };
    constexpr int sum = SimpleAccumulate( t, f, 0);


    static_assert( 27 == sum );

    std::cout << "sum = " << SimpleAccumulate( t, f, 0) << '\n';
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    return 0;
}
```

Constexpr lambda in C++17

```
template<typename Range, typename Func, typename T>
constexpr T SimpleAccumulate(const Range& range, Func func, T init)
{
    for (auto &&elem: range)
    {
        init += func(elem);
    }
    return init;
}
int main()
{
    constexpr int t[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    constexpr int x = 2;
    constexpr int y = 7;

    // constexpr auto f = [x,y](int n) { return (x < n && n < y) ? 0 : n; };
    constexpr int sum = SimpleAccumulate( t,
                             [x,y](int n) { return (x < n && n < y) ? 0 : n; };, 0);

    static_assert( 27 == sum );

    std::cout << "sum = " << SimpleAccumulate( t, f, 0) << '\n';
    std::cout << "x = " << x << '\n';
    std::cout << "y = " << y << '\n';
    return 0;
}
```

Capture *this in C++17

```
struct my_struct
{
  int x;
  int y;
  void value();
};

void my_struct::value()
{
  [=, this](){};      // C++17 error: = captures this by default, ok since C++20
  [=, *this](){};     // captures my_struct by value since C++17
  [this, *this](){}; // ok since C++20: repeating this in capture
}
```

C++20

- Allow [=,this]
- Pack expression in init capture   [...args = std::move(args)]
- Capture for structured bindings
- Template lambdas (with concepts)
- Default constructible and assignable lambdas (if no state)
- Lambdas in unevaluated context (pl. sizeof)
- *this is captured by reference if captured implicitly by [=] or [&]
- *this captured by [=] is deprecated

std::function

- General purpose function wrapper
- Can store, copy and invoke
  - Pointer to function
  - Functor
  - Lambda expression
  - Bind expression
  - Member function
  - Pointer to data member
- Pretty expensive template construct

std::function

```
#include <functional>
#include <cmath>     // std::sin

double fahr2cels(double x){ return 5./9.*(x-32); }  // function

struct half  // functor
{
    double operator()(double x) { return x/2 ;}
};
void f(int choice)
{
    auto lambda = [](double x) { return 2*x; };  // lambda

    std::function<duble(double)> func;   // empty

    switch(choice)
    {
      case 1: func = std::sin;  break; // from <cmath>
      case 2: func = lambda;    break; // lambda
      case 3: func = half;      break; // functor
      case 4: func = fahr2cels; break; // function pointer
    }
    if ( func )        // otherwise std::bad_function_call is thrown
    {
        std::cout << func(0.5) << '\n';
    }
}
```

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

```
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); // ?
}
```

```
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 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);
```

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";

}
```

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)
```

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

```cpp
#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;
}
```