Templates

- Adding type parameters to a function or class.
- Compiler generates code for each type needed.

```
//Inconvenient way
int max(int a, int b){
  return a > b ? a : b;
}
double max(double a, double b){
  return a > b ? a : b;
}
```

Examples

```
std::vector<int> vi;
std::vector<double> vd;
auto t = std::make_tuple(1,2);
```

- Type arguments between < >.
- Type arguments can be deduced by the compiler.

Syntax

```
/**
 * @brief Function template
 *
 * @tparam T Type parameter
 * @param a First argument
 * @param b Second argument
 * @return T Maximum value
 */
template <typename T>
T max(T a, T b) {
  return a > b ? a : b;
}
```

- tparam tag: Type parameter.

Parametrized types

```
template<typename T>
struct MyType{
  std::vector<T> data;
  T value;
  T get_value(){return value;}
};
MyType<int> mt{{1,2,3}, 10};
MyType<double> mt2{{1.1,2.2,3.3}, 10.1};
mt.get_value();
```

Parametrized types

```
template<typename T>
class Vector{
  T* data;
  size_t size;
public:
  Vector(size_t size): size(size), data(new T[size]){}

  ~Vector(){
    delete[] data;
  }
  //...
};

Vector<int> vi(10);
Vector<double> vd(10);
```

Templated types inside classes

```
template<typename T>
class Vector{
  T* data;
  size_t size;
public:
  Vector(size_t size): size(size), data(new T[size]){}

  ~Vector(){
    delete[] data;
  }

  T& operator[](size_t i){
    return data[i];
  }
};
```

RAII with smart pointers

```
template<typename T>
class Vector{
  std::unique_ptr<T[]> data;
  size_t size;
public:
  Vector(size_t size): size(size), data(new T[size]){}

  T& operator[](size_t i){
    return data[i];
  }
};

Vector<int> vi(10);
vi[0] = 1;
Vector<double> vd(10);
```

Separated declaration/definition

```
template<typename T>
class Vector{
  std::unique_ptr<T[]> data;
  size_t size;
public:
  Vector(size_t size);
  size_t get_size() const;
};
template<typename T>
Vector<T>::Vector(size_t size):
  size(size), data(new T[size]){}
template<typename T>
size_t Vector<T>::get_size() const{
  return size;
}
```

Partial specialization

```cpp
template<typename T, typename U>
struct ValueCast{
  T value;
  U get_value(){return static_cast<U>(value);}
};
template<typename T>
struct ValueCast<T, T>{
  T value;
  T get_value(){return value;}
};
int main(){
  ValueCast<int, double> vc{10}; // Calls generic
  vc.get_value();
  ValueCast<int, int> vc2{10}; // Calls specialized
  vc2.get_value();
}
```

Type deduction

- Compiler can deduce type arguments.

```
auto x = 10; //int
auto y = 10.0; //double
auto w = std::make_tuple(1,2.0);
 ↪  //std::tuple<int,double>
```

Class template deduction

- C++17 allows deduction of class template arguments.

```
template<typename T>
struct MyType{
  T value;
  MyType(T value): value(value){}
};

MyType mt(10); //MyType<int>
MyType mt2(10.0); //MyType<double>
std::vector vi{1,2,3}; //std::vector<int>
```

Nested template arguments

```
std::vector<std::list<int>> vli;
vli.push_back({1,2,3});
vli.push_back({4,5,6});
```

- The >> is not a misplaced input operator.

Function objects

- Functor: A class with overloaded operator().

```
struct MyFunctor{
  int operator()(int a, int b){
    return a + b;
  }
};

MyFunctor f;
f(1,2);
```

Function objects

- Functions that can be passed as arguments.
- std algorithms use function objects.

```
struct LessThan{
  template<typename T>
  bool operator()(const T& a, const T& b){
    return a < b;
  }
};

LessThan lt;
std::vector<int> vi{3,1,2};
std::sort(vi.begin(), vi.end(), lt);
```

Function objects

- Functions that can be passed as arguments.
- std algorithms use function objects.

```
struct LessThan{
  template<typename T>
  bool operator()(const T& a, const T& b){
    return a < b;
  }
};

LessThan lt;
std::vector<double> vd{3.0,1.0,2.0};
std::sort(vd.begin(), vd.end(), lt);
```

Lambda expressions

- Anonymous functions, similar to function objects.
- Syntax: `[capture](args){body}`

```
auto lt = [](double a, double b){return a < b;};

std::vector<double> vd{3.0,1.0,2.0};
std::sort(vd.begin(), vd.end(), lt);
```

Lambda expressions

- Anonymous functions, similar to function objects.
- `auto`: Generic lambda.

```
auto lt = [](auto a, auto b){return a < b;};

std::vector<double> vd{3.0,1.0,2.0};
std::sort(vd.begin(), vd.end(), lt);


std::vector<int> vi{3,1,2};
std::sort(vi.begin(), vi.end(), lt);
```

Lambda expressions

- Anonymous functions, similar to function objects.
- [&]: Local names can be accessed by reference.

```
int x = 10;
auto ltx = [&](auto a){return a < x;};
if(ltx(5))
  std::cout << "5 is < than x\n";
else
  std::cout << "5 is >= than x\n";
```

Lambda expressions

- Anonymous functions, similar to function objects.
- [&]: Local names can be accessed by reference.

```
int x = 10;
auto ltx = [=](auto a){return a < x;};
std::vector<int> v(1000);
std::default_random_engine rng(1234);
std::generate(v.begin(), v.end(),
              [&](){return rng()%100;});
auto count = std::count_if(v.begin(), v.end(),
                           ltx);
std::cout<<count;
```

Functions as arguments

```
using std::cout;

template<typename C, typename Oper>
void for_each(C& c, Oper op){
  for(auto& x: c)
    op(x);
}

std::vector vi{1,2,3};
auto print = [](auto x){cout<<x<<" ";};
for_each(vi, print);
```