

Contemporary C++: *Learning Modern C++ in a Modern Way*



الماس فناوری ابری پاسارگاد - آلفا

مدرس: سعید امراللهی بیوکی

Agenda 15/24

Session 15. Templates ... Going Deep (Part II)

- C++11 constexpr: Generalized and guaranteed constant expressions
- Templates as a tool for compile-time programming
- Template terminologies
- Fundamental concepts: Instantiation, Specialization, Argument deduction, Two-phase translation
- C++14 template variables
- Return type deduction: The C++11 decltype specifier
- Templates as a tool for compile-time programming
- Q&A

150 min (incl. Q & A)



template **A**rgument deduction



Max Template Function

Prog.

1

```
template<typename T>
inline T max(T const& a, T const& b)
{
    return a > b ? a : b;
}
```

2

```
void f()
{
    int const c = 42;
    int i{0};
    auto m1 = ::max(i, c);
    int& ir = i;
    auto m2 = ::max(i, ir);
    double d = 0.4;
    auto m3 = ::max(c, d); // ERROR: T can be deduced as int or double
    int arr[4];
    auto m4 = ::max(&i, arr); // OK: T is deduced as int*
    std::string s{ "Hello" };
    auto m5 = ::max("Hello", s); // ERROR: T can be deduced as char const[6] or std::string
}
```

- Argument deduction and type conversion
 - automatic type conversions are limited during type deduction
 - Call by reference vs. Call by value
 - Type decay

template **A**rgument deduction cont.



Max Template Function

Prog.

1

```
template<typename T>
inline T max(T const& a, T const& b)
{
    return a > b ? a : b;
}
```

2

```
void f()
{
    // ...
    int const c = 42;
    double d = 0.4;
    auto m3 = ::max(c, d); // ERROR: T can be deduced as int or double
    std::string s{ "Hello" };
    auto m5 = ::max("Hello", s); // ERROR: T can be deduced as char const[6] or std::string
    // ...
}
```

- Explicit type casting
- Explicit Specialization
- Using different template parameters

```
template<typename T1, typename T2, typename RT>
RT max(T1 const& a, T1 const& b);
```

3

```
void f()
{
    int const c = 42;
    double d = 0.4;
    auto m3 = ::max(static_cast<double>(c), d);
    auto m4 = ::max<double>(c, d);
}
```

Type deduction- more examples

```
template<typename T, typename U>
void f(T x, U y);

template<typename T>
void g(T x, T y);
```

 *Pretty Function Test*
Prog.

```
void user()
{
    f(1, 2); // instantiates void f(T, U) [T = int, U = int]
    g(1, 2); // instantiates void g(T, T) [T = int]
    g(1, 2u); // error: no matching function for call to g(int, unsigned int)
}
```

- No automatic type conversion

1

```
template<class T>
inline T const& min(const T& a, const T& b)
{
    return a < b ? a : b;
}
```

3

```
min(static_cast<double>(4), 4.2); // OK
min<double>(4, 4.2); // OK, much better
```

2

```
min(1, 2); // OK: T is int for both arguments
min(4, 4.2) // Error: first T is int, second T is double
```

C class template argument deduction

- Class template argument deduction → CTAD
- Actually, template parameter deduction for constructors



David Vandevor



Mike Spertus

Class template argument deduction- an example

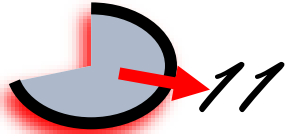
```
template<class T, class U>
struct Pair {
    T first;
    U second;
    Pair(const T& t, const U& u) : first{ t }, second{ u } {}
};

int main()
{
    Pair<int, double> p{ 1, 3.14 }; // C++98/11/14
    Pair p2{ 1, 3.14 }; // C++17
    Pair p3 = p2; // OK
    Pair<int, float> p4{ 1, 3.14 }; // truncation from double to const float
    Pair p5{ 1LL, 3.14f }; // Pair<long long, float>

    return 0;
}
```


C++ class template argument deduction- more examples

```
std::tuple<int, double> t{42, 3.14};  
auto t = std::make_tuple(42, 3.14);  
  
return std::tuple<int, double>{42, 3.14};  
return std::make_tuple(42, 3.14);
```



```
std::tuple t{42, 3.14};  
auto t = std::make_tuple(42, 3.14);  
  
return std::tuple{42, 3.14};  
return std::make_tuple(42, 3.14);
```

```
std::mutex mtx;  
std::lock_guard<std::mutex> lg{mtx};
```



```
std::timed_mutex timed_mtx;  
std::unique_lock ul{timed_mtx};
```


Default template argument

- default template argument
 - default class template argument
 - default function template argument

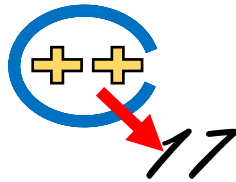
- The map container

```
template<typename Key, class V, typename Compare = std::less<Key>>
class map {
    // ...
};
```

- Other examples from standard library

```
// C++ standard library
template<class T, class Container = deque<T>> class stack;
template<class T, class Container = vector<T>> class priority_queue;
template< class IntType = int> class uniform_int_distribution;
```

```
template<typename Source = int>
std::string convert_to_string(Source arg)
{
    std::stringstream ss;
    ss << arg;
    return ss.str();
}
```



- std::to_string()

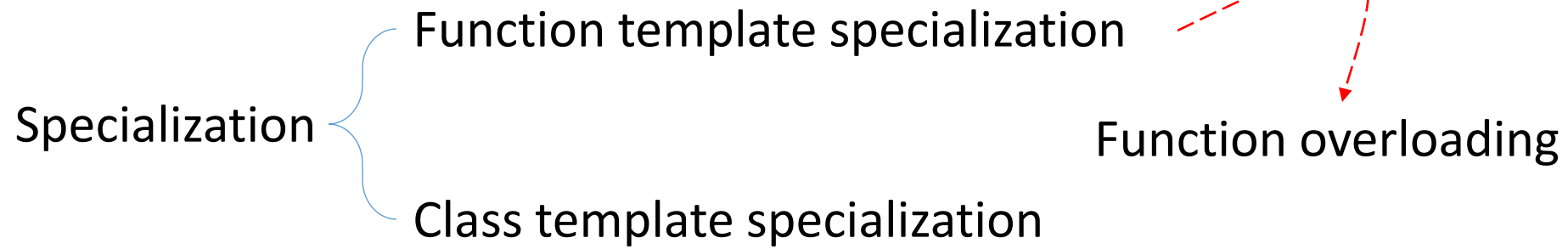
Non-type template parameters

- Value template arguments
 - In addition to type arguments, a template can take value arguments.

```
template<typename T, int N>
struct Buffer {
    constexpr int size() { return N; }
    T elem[N];
    // ...
};
```

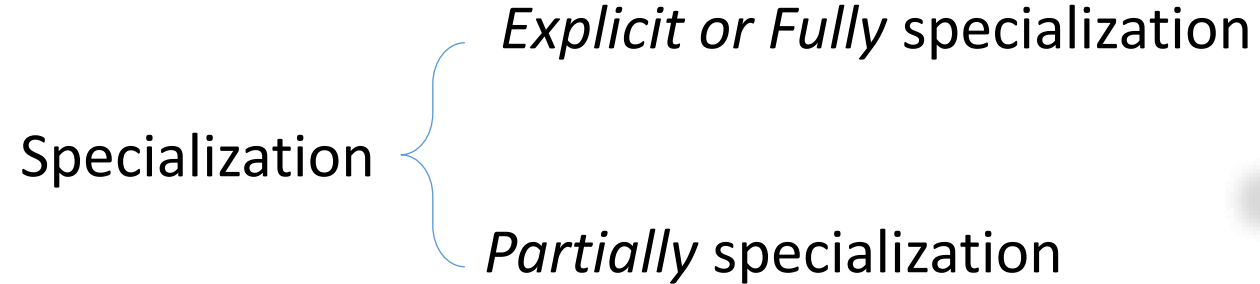
```
Buffer<char,1024> glob; // global buffer of characters (statically allocated)
void fct()
{
    Buffer<int,10> buf; // local buffer of integers (on the stack)
    // ...
}
```

template **S**pecialization



- Specialization templates allows you to *optimize* implementation for certain types or to *fix a misbehavior* of certain types for an instantiation of the class template.

C class template specialization



Full Generic

Partial Specialized

Full Specialized

- Examples

```
template<typename T1, typename T2>
class MyClass { // primary class template
    // ...
};

template<> // explicit specialization
class MyClass<string, float> {
    // ...
};

template<typename T> // partial specialization
class MyClass<T, T> {

};

template<typename T> // partial specialization
class MyClass<bool, T> {
    // ...
};
```

```
template<class T>
class Stack {
    // ...
};

template<>
class Stack<T*> {

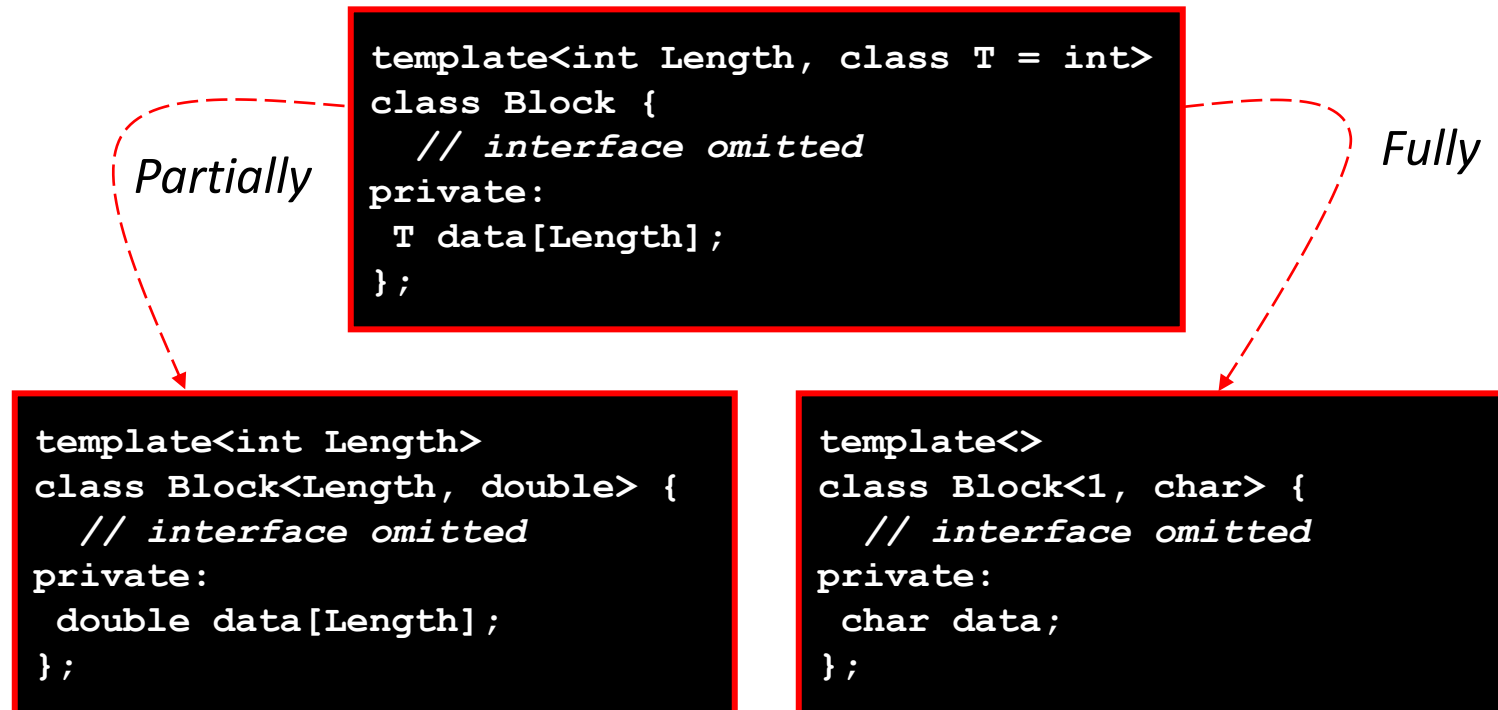
};

template<int*>
class Stack<int*> {

};
```

Class template specialization- more examples

- Example: POOMA library
- Los Alamos National Laboratory:
POOMA Toolkit → Nuclear Fusion and Fission research



Function template specialization

- Function template specialization

```
template<class T> T sqrt(T);  
template<class T> complex<T> sqrt(complex<T>);  
double sqrt(double);  
void f(complex<double> z)  
{  
    sqrt(2);           // sqrt<int>(int)  
    sqrt(2.0);         // sqrt<double>  
    sqrt(z);           // sqrt<double>(complex<double>)  
    sqrt<>(2.0);        // sqrt<double>(double)  
}
```

- You can specialize a single member function, once you have done so, you can no longer specialize the whole class.

Constant expressions

- C++ has always had the concept of *constant expressions*.

```
const double PI = 3.14; // don't change the content of PI
char* hi = "Hello, world"; // "Hello, world" is constant character
const char* hi = "Hi"; // char to constant data
char* const hello = "Hello"; // constant pointer to character
const char* const Hawaii = "Aloha"; // constant pointer to constant character
class Point {
    int x, y;
    const int z; // constant data member
public:
    Point(int xx, int yy) : x(xx), y(yy), z(0) {}
    int GetX() const { return x; } // constant member function
    int GetY() const { return y; }
    void SetX(int xx) { x = xx; } // non-constant member function
    // ...
};
const Point Center(0, 0); // constant object
void f(const Point); // copy semantics: don't change the content of argument
void g(const Point&); // reference semantic: don't change the
```

- `const`'s primary function is to express the idea that an object is not modified through an interface.
- constant advantages: Excellent optimization opportunities for compilers, more maintainable code

Generalized and guaranteed constant expressions

1.

- The constant expressions may be or may be not evaluated at compile-time.
- Example: array bound

```
// array_bound.cpp  
int get_five() { return 5; }  
long long LA[get_five() + 10]; // error: array bound is not an integer constant
```

```
// GCC 4.6.0 under Ubuntu 10.10 32-bit  
$ g++ -c -std=c++98 array_bound.cpp  
error: array bound is not an integer constant
```

2.

- The problem is from the point of theoretical view, `get_five()` isn't constant expression.
- Generalized and guaranteed constant expressions (`constexpr`) extend compile time evaluation to functions and variables of user-defined types.

```
// array_bound.cpp  
constexpr int get_five() { return 5; }  
long long LA[get_five() + 10]; // OK  
fill(LA, LA + 15, 42LL);
```

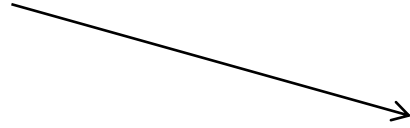
Generalized and guaranteed constant expressions cont.

- Literal types or classes

- Constexpr functions



Exactly one return
statement



The return type should
be literal type

```
constexpr void f(int x) // error: return type is void
{ / ... / }
```

Constexpr: examples

```
constexpr double abs(double x) { return x < 0 ? -x : x; }
constexpr double PI = 3.14;
class Point {
public:
    constexpr Point() : x_{0}, y_{0} {}
    constexpr Point(int x, int y) : x_{x}, y_{y} {}

    constexpr int x() { return x_; }
    constexpr int y() { return y_; }
    void x(int xx) { x_ = xx; }
    void y(int yy) { y_ = yy; }
private:
    int x_;
    int y_;
};
int main()
{
    constexpr double abs_val[] = { abs(1.0), abs(2.0), abs(-3.14) };
    Point p;

    return 0;
}
```

Light-weight
Abstractions



Constexpr Point
Prog.

- A function declared constexpr is implicitly an inline function.
- Godbolt.org

C const vs. constexpr

- The concept of immutability
- `const` means roughly “I promise not to change this value.” This is used primarily to specify interfaces so that data can be passed to functions using pointers and references without fear of it being modified. The compiler enforces the promise made by `const`. The value of a `const` may be calculated at run time.
- `constexpr` means roughly “to be evaluated at compile time.” This is used primarily to specify constants, to allow placement of data in read-only memory (where it is unlikely to be corrupted), and for performance. The value of a `constexpr` must be calculated by the compiler.
- The *constexpr* primary function is to extend the range of what can be computed at compile time, making such computation type-safe.
- The *constexpr* mechanism
 - Provides more general constant expressions
 - Allows constant expressions involving user-defined types → ROM
 - Provides a way to guarantee that an initialization is done at compile time.

Variable templates

- Until variable templates were introduced in C++14, parametrized variables were typically implemented as either static data members of class templates or as constexpr function templates returning the desired values.

- Static data member

```
// my_pi.h
template<typename T>
struct PhysicsConstants {
    static const double pi;
};
// my_pi.cpp
template<typename T>
const double PhysicsConstants<T>::pi = 3.14159;

void f()
{
    auto p = PhysicsConstants<char>::pi;
    // ...
}
```

- Constexpr functions

```
// my_pi.h
template<typename T>
constexpr auto Pi_value()
{
    return 3.14159;
};

void f()
{
    auto p = Pi_value<char>();
    // ...
}
```


Variable Template Emulation Test

Variable templates

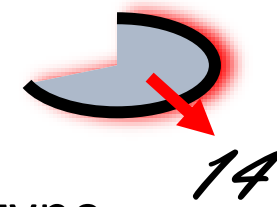
- When we use a type, we often want constants and values of that type.
- Since C++14, variables also can be parameterized by a specific type.
- Templated variable

```
template<typename T>
constexpr T pi = T(3.141592653589793238462643383);

template<class T>
T circular_area(T r) // function template
{
    return pi<T> * r * r; // pi<T> is a variable template instantiation
}
```



Variable Template Test
Prog.



Gabriel Dos Reis

Variable templates- another example

- Static data member

```
template<typename T>
class GetSize {
    static const int len = sizeof(T);
};

int length = GetSize<double>::len; // use GetSize
```

VS.

- Variable template

```
template<typename T>
constexpr int sizeof_var = sizeof(T);

int length = sizeof_var<double>; // use size
```


Variable templates specialization



Fibonacci Variable Template
Prog.

Decltype- the type of an expression

- Decltype is declaration type specifier.
- **decltype(E)** is the type ("declared type") of the name or expression **E** and can be used in declarations.
- Given a name or an expression, decltype tells you the name's or the expression's type.

```
int i = 0; // decltype(i) is int
const int i = 0; // decltype(i) is const int
bool f(const Widget& w); // decltype(w) is const Widget&
                        // decltype(f) is bool(const Widget&)

struct Point {
    int x, y; // decltype(Point::x) is int
}; // decltype(Point::y) is int
Widget w; // decltype(w) is Widget
if (f(w)) // decltype(f(w)) is bool
    // ...
```

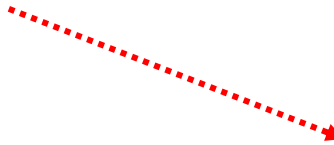
```
int i = 0;
decltype(i) j = 1;
```

Decltype- Some examples

- Using decltype as a declaration type specifier:

```
void f(const vector<int>& a, vector<float>& b)
{
    typedef decltype(a[0] * b[0]) Tmp;
    for (int i = 0; i<b.size(); ++i) {
        Tmp* p = new Tmp(a[i] * b[i]);
        // ...
    }
    // ...
}
```

- Using decltype as a return type specifier:
 - Sometimes a return type simply cannot be expressed in the usual manner



```
// Function template to return product of two  
// values of unknown types:  
template<typename T, typename U>  
??? product(const T &t, const U &u) {  
    return t * u;  
}
```

```

// Function template to return product of two
// values of unknown types:
template<typename T, typename U>
auto product(const T &t, const U &u) -> decltype(t * u) {
    return t * u;
}

```

```

template<class T, class U>
auto operator+(const Matrix<T>& a, const Matrix<U>& b) ->Matrix<decltype<T{} + U{}>>; // declaration

template<class T, class U>
auto operator+(const Matrix<T>& a, const Matrix<U>& b) ->Matrix<decltype<T{} + U{}>> // definition
{
    Matrix<decltype(T{} + U{})> res;
    for (int i = 0; i != a.rows(); ++i)
        for (int j = 0; j != a.cols(); ++j)
            res(i, j) = a(i, j) + b(i, j);
    return res;
}

```

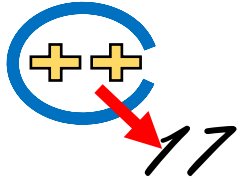
- Very high-qualified code!

Auto vs. decltype

- C++ provides two mechanisms for deducing a type from an expression:
 - **auto**: for deducing a type of an object from its initializer
 - **decltype**: for deducing the type of something that is not a simple initializer, such as the return type for a function or the type of a class member.
- If you just need the type for a variable that you are about to initialize **auto** is often a simpler choice. You really need **decltype** if you need a type for something that is not a variable, such as a return type.

Explicit specialization- an example

```
template<class T, class U>
auto add(const T& t, const U& u) -> decltype(t + u)
{
    return t + u;
}
```



```
int main()
{
    add<int, int>('x', 3.14); // [T = int, U = int]
    add<int>('x', 3.14); // [T = int, U = double]
    add<>('x', 3.14); // [T = char, U = double]
    add('x', 3.14); // [T = char, U = double]

    return 0;
}
```

improving the **V**ector class

- The Vector class so far:

```
class Vector {  
    int size_  
    int* elem_  
public:  
    /*  
     * Essential operations  
     */  
    Vector();  
    explicit Vector(int /* size */);  
    Vector(const Vector&);  
    Vector& operator=(const Vector&);  
    Vector(Vector&&);  
    Vector& operator=(Vector&&);  
    ~Vector();  
    int size() const; // the current size  
    int operator[](int) const; // access element: getter  
    int& operator[](int); // access element: setter  
};
```

- How do we change the size of a vector (change the number of elements?)
- How do we catch and report out-of-range vector element access?
- How do we specify the element type of a vector as an argument?

improving the **V**ector class- generic vector

- Someone who wants a vector is unlikely always to want a vector of **integers**. A vector is a general concept, independent of the notion of integer numbers. Consequently, the element type of a vector ought to be represented independently.

```
template<class T>
class Vector {
    int size_;
    T* elem_;
public:
    /*
     * Essential operations
     */
    Vector();
    explicit Vector(int /* size */);
    Vector(const Vector&);
    Vector& operator=(const Vector&);
    Vector(Vector&&);
    Vector& operator=(Vector&&);
    ~Vector();
    int size() const; // the current size
    T operator[](int) const; // access element: getter
    T& operator[](int); // access element: setter
};
```

Zero-overhead principle

- The zero-overhead principle is a C++ design principle that states:
 - You don't pay for what you don't use.
 - What you do use is just as efficient as what you could reasonably write by hand.

In general, this means that no feature should be added to C++ that would impose any overhead, whether in time or space, greater than a programmer would introduce without using the feature.

Two-phase translation

- Templates are compiled twice:
 1. At the point of definition: Without instantiation, the template code itself is checked for correct syntax.
 2. At the point of use: at the time of instantiation, the template code is checked against the template arguments.
- A template is not instantiated unless its definition is actually needed.
- Generic code issues
 - The location of the template
 - The locations where the template is used
 - The locations where the template arguments are defined



- How C++ compilers handle template instantiation?



- On-demand

templates: code Composition

- The composition offered by templates is type-safe (no object can be implicitly used in a way that disagrees with its definition).

```
template<class T>
class Stack {
    std::vector<T> elems;
public:
    void push(const T&);
    void pop();
    T peek();
private:
    bool empty() const {
        return elems.empty();
    }
};
```

An Example- Pair

```
template<class T1, class T2>
struct Pair {
    T1 first;
    T2 second;
public:
    Pair() : first(T1()), second(T2()) {}
    Pair(T1 f, T2 s) : first(f), second(s) {}
    Pair& operator=(const pair<U1, U2>& p);
    template<class U1, class U2> pair& operator=(const pair<U1, U2>& p); // template member function
};
```

```
class IPAddress {
    // ...
};

class MacAddress {
    // ...
};

class Point {
    // ...
};
```

```
void use_pair()
{
    std::string first_name, last_name;
    using pair<std::string, std::string> = PersonalInfo;
    using pair<MacAddress, IPAddress> = NICAddress;
    using pair<Point, Point> = Line;

    Line my_line{point{0, 0}, Point{1, 1}};
    NICAddress my_host{MacAddress("1234.AFE1.01FF"),
                      IPAddress{"192.168.1.10"}};
    PersonalInfo mine{"Saeed" "Amrollahi"};
}
```

Thanks for your patience ...

A man who asks a question is a fool for minute,
The man who does not ask, is a fool for a life.
- Confucius

Learning to ask the right (often hard) questions is an essential part of learning to think as a programmer.

- Bjarne Stroustrup *programming Principles and Practice Using C++, page 4.*

There is no stupid question, but there is stupid answer.
- Howard Hinnant

