C++11/14

Programming Techniques HS15

auto

```
// the good old days
std::vector<double> vec;
for(std::vector<double>::iterator it = vec.begin(); it != vec.end(); ++it) {
    …
}

// typedef made it better
typedef std::vector<double> my_vec;

my_vec vec2;
for(my_vec::iterator it = vec2.begin(); it != vec2.end(); ++it) {
    …
}

// auto solution (not best c++11 solution for loops)
for(auto it = vec.begin(); it != vec.end(); ++it) {
    …
}
```

auto

```
// the good old days
std::map<std::string, double> mymap;
…
std::pair<std::string, double> p = *mymap.begin();
std::pair<std::string, float> p_oops = *mymap.begin(); // maybe not intended…

// typedef made it better
typedef std::map<std::string, double> mymap_t;

mymap_t mymap2;
…
mymap_t::value_type p2 = *mymap2.begin();

// auto solution
auto p2 = *mymap2.begin();
```

auto

```
// the good old days
std::vector<double> vec

// type-shortcuts
uint size = vec.size(); //hmmm, its probably an uint ?! sure, what can go wrong?

// correct, but too much typing:
std::vector<double>::size_type size = vec.size();

// auto solves the problem again
auto size = vec.size();
```

How does `auto` work?

templates

```cpp
int num = 10;
int & num_r = num;
int const num_c = 10;
int const & num_cr = num_c;

// pass by value
fct(num);           // int
fct(num_r);         // int
fct(num_c);         // int
fct(num_cr);        // int

// pass by reference (or pointer)
fct_ref(num);       // int &
fct_ref(num_r);     // int &
fct_ref(num_c);     // int const &
fct_ref(num_cr);    // int const &

// pass by universal reference
fct_uref(num);      // int &
fct_uref(10);       // int &&
```

```cpp
template<typename T>
void fct(T t_val) {
    PRINT_TYPE_OF(t_val);
}

template<typename T>
void fct_ref(T & t_ref) {
    PRINT_TYPE_OF(t_ref);
}

template<typename T>
void fct_uref(T && t_uref) {
    PRINT_TYPE_OF(t_uref);
}
```

auto

```
int num = 10;
int & num_r = num;
int const num_c = 10;
int const & num_cr = num_c;

// by value
auto a0 = num;          // int
auto a1 = num_r;        // int
auto a2 = num_c;        // int
auto a3 = num_cr;       // int

// reference (or pointer)
auto & a0ref = num;     // int &
auto & a1ref = num_r;   // int &
auto & a2ref = num_c;   // int const &
auto & a3ref = num_cr;  // int const &

// universal reference
auto && a0uref = num;   // int &
auto && a1uref = 10;    // int &&
```

```
template<typename T>
void fct(T t_val) {
    …
}

template<typename T>
void fct_ref(T & t_ref) {
    …
}

template<typename T>
void fct_uref(T && t_uref) {
    …
}
```

auto uses the rules of template type deduction

(there is one small exception / irrelevant for now)

template type deduction

```
int num = 10;
int & num_r = num;
int const num_c = 10;
int const & num_cr = num_c;

// by value
auto a0 = num;          // int
auto a1 = num_r;        // int
auto a2 = num_c;        // int
auto a3 = num_cr;       // int
```

ignores cv-qualifier

ignores ref-ness

```
// reference (or pointer)
auto & a0ref = num;     // int &
auto & a1ref = num_r;   // int &
auto & a2ref = num_c;   // int const &
auto & a3ref = num_cr;  // int const &
```

ignores ref-ness

```
// universal reference
auto && a0uref = num;   // int &
auto && a1uref = 10;    // int &&
```

c++11 magic

covered in later lecture

simpler loops

```
std::vector<double> vec;

// auto solution from earlier
for(auto it = vec.begin(); it != vec.end(); ++it) {
    …
}

// nice c++11 loops
for(double val: vec) {
    …
}

// even nicer auto loops
for(auto val: vec) {
    …
}
```

why **decltype**

```cpp
// first try
template<typename T>
T mean_try1(T a, T b) {
    return (a + b) / 2.0;
}

// bad integral types / res = 1
auto res1 = mean_try1(1, 2);
```

```cpp
// take 2
template<typename T>
double mean_try2(T a, T b) {
    return (a + b) / 2.0;
}

// bad for floating types with
// higher or lower precision that double
// either truncation or waste of space
auto res2 = mean_try2(float(1), float(2));
```

why **decltype**

```
// the main problem!
template<typename T>
??? mean_try1(T a, T b) {
    return (a + b) / 2.0;
}
// the compiler knows what (a + b) / 2.0 is but won't tell us
// lets try to get it out...
```

why decltype

```
// take 3
template<typename T, bool is_integral>
struct mean_trait {
    typedef double type;
};

// use partial template specialization
template<typename T>
struct mean_trait<T, false> {
    typedef T type;
};

// isn't it a thing of beauty...
template<typename T>
typename mean_trait< T
                   , std::is_integral<T>::value // c++11 <type_traits>
                   >::type mean_try3(T a, T b) {
    return (a + b) / 2.0;
}

// works now for the built-in types
// but doesn't for my_int since it is not known by std::is_integral
auto try3 = mean_try3(my_int(1), my_int(2)); // returns my_int(1);
```

```
struct my_int {
    my_int(int in);

    int x;
};
my_int operator+(my_int a, my_int b) {
    return a.x + b.x;
}
double operator/(my_int a, double d) {
    return a.x / d;
}
```

where is **decltype**

```
// take 4, generic
// (you'll love decltype after this)
template<typename T>
struct use_double {
    static T t;

    static char check(T);
    static double check(double);

    enum {value =
        (sizeof(check((t+t)/double(2)))
      == sizeof(double))};
};
// full specialization
// to avoid return value overload
template<>
struct use_double<double> {
    enum {value = true};
};
```

```
template<typename T, bool use_double>
struct mean_trait_chooser {
    typedef double type;
};

template<typename T>
struct mean_trait_chooser<T, false> {
    typedef T type;
};

// better_mean_trait (not nicer though...)
template<typename T>
struct better_mean_trait {
    typedef
        typename
            mean_trait_chooser<
                    T
                  , use_double<T>::value
        >::type
    type;
};
```

where is **decltype**

```
// take 4, generic
// (you'll love decltype after this)
template<typename T>
struct use_double {
    static T t;

    static char check(T);
    static double check(double);

    enum {value =
        (sizeof(check((t+t)/double(2)))
      == sizeof(double))};
};
// full s
// to avo
template<
struct us
    enum
};
```

```
.)
template<typename T>
ser<
e<T>::value
};
```

```
struct my_int {
    my_int(int in);

    int x;
};
my_int operator+(my_int a, my_int b) {
    return a.x + b.x;
}
double operator/(my_int a, double d) {
    return a.x / d;
}
```

```
typename better_mean_trait<T>::type mean_try4(T a, T b) {
    return (a + b) / 2.0;
}

auto try4 = mean_try4(my_int(1), my_int(2));
// returns double(1.5)
// fails if (T+T)/double is neither T nor double
```

decltype

```
// take 5, finally there's a way to ask the compiler: decltype
template<typename T>
decltype((T() + T()) / double()) mean_try5(T a, T b) {
    return (a + b) / 2.0;
}

auto try5 = mean_try5(my_int(1), my_int(2)); // works perfectly

// but what if T has no default constructor T()?
// use std::declval to "fake" an instance (or the nice *(T*)(0) construction)
template<typename T>
decltype((std::declval<T>() + std::declval<T>()) / double()) mean_try5b(T a, T b) {
    return (a + b) / 2.0;
}
// but now it's ugly again… lets fix that
```

decltype

```
// take 6, with trailing return type (nicer syntax, identical to mean_try5)
template<typename T>
auto mean_try6(T a, T b) -> decltype((a + b) / 2.0) {
    return (a + b) / 2.0;
}

// do we really need to type the same expression twice?

// not with c++14 return value deduction
template<typename T>
decltype(auto) mean_try7(T a, T b) {
    return (a + b) / 2.0;
}

// side-note: auto alone is possible, but would not be sufficient in
// a general case, thats why one needs decltype(auto) (c++14). It makes
// auto deduce the exact type and not use "template type deduction“.
// Because writing:
// 1) auto           will always be a non reference to whatever returns
// 2) auto &         will always give you a reference to what returns
// 3) auto &&        returns also always a reference
// 4) decltype(auto) is the exact type of whatever is returned (ref & non-ref)
```

std::function

```
#include <functional> // std::function

// simpson returns from PT I: integrate fct form a to b
double simpson( std::function<double(double)> fct
              , double a
              , double b
              , unsigned int N);

// std::function takes everything with the correct
// call signature. It’s a generalized function pointer
```

two argument function

```cpp
#include "simpson.hpp"
#include <iostream>

// a function with two variables
double exp_ax(double a, double x) {
    return std::exp(a * x);
}

int main() {
    // where do we set a?
    std::cout << simpson(exp_ax, 0, 1, 100) << std::endl;
    // does not compile since exp_ax has signature
    // double(double, double) and not double(double)

    return 0;
}
```

global variable

```cpp
// global variable "solution"
#include "simpson.hpp"
#include <iostream>

// an ugly global variable
double a;

// the function to be integrated
double exp_a_glob(double x) {
    return std::exp(a * x);
}

int main() {
    a = 3.4;
    std::cout << simpson(exp_a_glob, 0, 1, 100) << std::endl;

    return 0;
}
```

function object

```
// a function object for exp(a*x)
#include "simpson.hpp"
#include <iostream>

class exp_fct_obj {
    public:
    // set the parameter a in the constructor
    exp_fct_obj(double a) : a_(a) {}

    // the function call operator calculates the function
    double operator()(double x) {
        return std::exp(a_ * x);
    }

    private:
    double a_; // the fixed parameter a
};

int main() {
    double a = 3.4;
    std::cout << simpson(exp_fct_obj(a), 0, 1, 100) << std::endl;

    return 0;
}
```

std::bind

```cpp
#include "simpson.hpp"
#include <iostream>
#include <functional> // for std::bind

double exp_ax(double a, double x) {
    return std::exp(a * x);
}

int main() {
    using namespace std::placeholders; // for _1, _2

    double a = 3.4;

    // bind one argument: _1, _2, ... are used for
    // unbound arguments of the resulting function
    auto exp_bind_a = std::bind(exp_ax, a, _1);

    std::cout << simpson(exp_bind_a, 0, 1, 100) << std::endl;

    return 0;
}
```

lambdas

```
// lambda functions
#include "simpson.hpp"
#include <iostream>

double exp_ax(double a, double x) {
    return std::exp(a * x);
}

int main() {
    double a = 3.4;

    // create a lambda function
    // [=] indicates that all variable
    // used inside the lambda are passed by value
    auto exp_a_lambda = [=](double x){ return exp_ax(a, x); };

    std::cout << simpson(exp_a_lambda, 0, 1, 100) << std::endl;

    // lambda in function
    std::cout << simpson([=](double x){ return exp_ax(a, x); }, 0, 1, 100);

    return 0;
}
```

std::function

```
// put multiple function-like objects in the same vector

… uses code from previous examples …

int main() {
    using namespace std::placeholders;

    double a = 3.4;

    std::vector<std::function<double(double)>> fct;

    fct.push_back(exp_a_glob);                   // normal function
    fct.push_back(exp_fct_obj(a));               // function object
    fct.push_back(std::bind(exp_ax, a, _1)); // function via bind
    fct.push_back([=](double x){ return exp_ax(a, x); }); // lambda

    for(auto f: fct) // simpler loops
        std::cout << simpson(f, 0, 1, 100) << std::endl;

    return 0;
}
```

lambdas

```
// return-type is void
auto hello_world = [](){ print_hello_world(); };
hello_world();

// return-type is deduced by
// "template deduction rules" to be double
auto exp_a_lambda = [=](double x){ return exp_ax(a, x); };

int val = 0;
int Y = 6;
// return-type specified with trailing return type
auto add_Y  = [&](int & in) ->void {in += Y;};
auto add_Y2 = [&](int & in) ->void {in += Y;};

add_Y(val); // adds 6

// Y was captured per reference [&]
Y = 1;
add_Y(val); // now only adds 1

// sidenote: each lambda has it's own unique type
PRINT_TYPE_OF(add_Y)  // main::{lambda(int&)#1}
PRINT_TYPE_OF(add_Y2) // main::{lambda(int&)#2}
```

lambdas

- The `[ ]` indicate a lambda function, and how variables from the enclosing scope should be used (captured) inside the lambda

`[ ]`	Capture nothing (or, a scorched earth strategy?)
`[&]`	Capture any referenced variable by reference
`[=]`	Capture any referenced variable by making a copy
`[=, &foo]`	Capture any referenced variable by making a copy, but capture variable foo by reference
`[bar]`	Capture bar by making a copy; don't copy anything else
`[this]`	Capture the this pointer of the enclosing class

c++14 lambdas

```
// one can use auto for parameter
auto print_lambda = [=](auto x){ std::cout << x << std::endl; };

// mimics a template function
template<typename T>
void print_fct(T x) {
    std::cout << x << std::endl;
}
```

new **using** functionality

```
// normally used for namespaces
using namespace std;

// suppose a project uses a int and double vector
typedef std::vector<double> d_container_type;
typedef std::vector<int>    i_container_type;

// lets say we want to change the container to list
// we have to change two (possibly many more) typedefs...
typedef std::list<double> d_container_type;
typedef std::list<int>    i_container_type;

// with c++11 we can change typedef with using
// nicer syntax, especially for function pointer typedefs
typedef void (*FP)(double, double);
using FP2 = void (*)(double, double);


using d_container_type = std::vector<double>;
using i_container_type = std::vector<int>;
```

using

```
// but the real power of using lies in
// the possibility to template it
template<typename T>
using container_type = std::vector<T>;

using d_container_type = container_type<double>;
using i_container_type = container_type<int>;


// if I want to change vector to list now
// only one using needs to be changed
template<typename T>
using container_type = std::list<T>;
```

using should be preferred to **typedef**

using

```
/ side-note: remember "typename" if you have dependent types in "typedef"?
template<typename T>
struct echo_type {
    typedef T type;
};

template<typename T>
struct dependent_demo {
    typedef          echo_type<T>::type type; // will not compile!
    typedef typename echo_type<T>::type type; // typename is needed
};

// no need for this with using, since the compiler knows it's a type
template<typename T>
using echo_type_t = typename echo_type<T>::type; // hide typename ... ::type here

template<typename T>
struct dependent_demo {
    using type = echo_type_t<T>; // much much nicer to use...
};
```

<type_traits>

```
#include <type_traits>

// ... so nice in fact, c++14 introduces new type traits
template<typename T>
struct type_trait_demo {
    using old_way = typename std::remove_const<T>::type;
    using new_way =          std::remove_const_t<T>;
};

// we will encounter the c++11 type_traits library in a later lecture again
```

<random>

```
#include <random>   // c++11
#include <iostream>

int main() {
    // create an engine
    std::mt19937 mt;

    // create distributions
    std::uniform_int_distribution<int>     uint_d(0, 10);
    std::uniform_real_distribution<double> ureal_d(0., 10.);
    std::normal_distribution<double>       normal_d(0., 4.);
    std::exponential_distribution<double>  exp_d(1.);

    // create random numbers:
    std::cout <<   uint_d(mt) << std::endl;
    std::cout <<  ureal_d(mt) << std::endl;
    std::cout << normal_d(mt) << std::endl;
    std::cout <<    exp_d(mt) << std::endl;

    // check the reference for all distr & engines!

    return 0;
}
```

std::mem_fn

```
struct my_int {
    my_int(int in);
    void set_x(int x_new);

    int x;
};

int main() {
    my_int a(1);
    a.set_x(2);
    // pass method address
    auto free_set_x = std::mem_fn(&my_int::set_x);

    // sometime it is useful to have a free function
    free_set_x(a, 2);

    // we can even bind the instance a to the function
    auto free_set_x_in_a = std::bind(free_set_x, a, _1);
    free_set_x_in_a(2);

    return 0;
}
```

nullptr

```
// overload the function foo
void foo(char *);
void foo(int);

// what is called?
int main() {
    foo(0);              // calls second foo
    foo(NULL);           // ambiguous since decltype(NULL) == long
    foo(nullptr);        // calls first foo

    decltype(nullptr); // is decltype(nullptr) / very convenient
}
```

always use **nullptr** to initialize an empty pointer

NULL and **0** for pointers belong to old c++

constexpr

```
#include <array> //should be used instead of int a[10]

// the size needs to be knows during compile-time
std::array<int, 10> a;

// this fails
int const N = 10;
std::array<int, N> b;
```

constexpr

```cpp
#include <array> //should be used instead of int a[10]

// the size needs to be knows during compile-time
std::array<int, 10> a;

// this fails
int n = 10;
int const N = n;
std::array<int, N> b;

// this works (the compiler checks if a const is known at compile-time)
int const N1 = 10;
// maybe N1 is known, maybe it isn't, let's try
std::array<int, N1> c;

// this works
int constexpr N2 = 10;
// N2 is guaranteed to be known during compile-time
std::array<int, N2> d;
```

constexpr

```
constexpr int add(int a, int b) {
    return a + b;
}

int constexpr N = 10;

// the function is executed during compile-time ! if all its arguments
// are constexpr and the result used in a constexpr context

std::array<int, add(1, N)> d;

// a constexpr function behaves like a normal function during runtime
int n1;
int n2;

// read two numbers
std::cin >> n1 >> n2;
std::cout << add(n1, n2) << std::endl;
```

C++14 **constexpr**

```
// c++11 only allows one return-statement in a constexpr function
// ...but we know how to use recursion
constexpr int pow(int a, int b) {
    return b == 0 ? 1 : pow(a, b - 1) * a;
}
// "cond ? a : b" is short for "if(cond) return a; else return b;"

// c++14 relaxes the constrains for constexpr functions
constexpr int pow(int a, int b) {
    int res = 1;
    for(int i = 0; i < b; ++i) {
        res *= a;
    }
    return res;
}
```

delegating constructor

```
// c++98 problems
class myclass {
    public:
    myclass(): c_num_(10), num_(42) {}

    myclass(int nr): c_num_(10), num_(nr) {}

    myclass(int nr1, int nr2): c_num_(10), num_(nr1 + nr2) {}

    // we write three times c_num_(10)...

    private:
    int c_num_;
    int num_;
};
```

delegating constructor

```
// c++11 delegating ctors
class myclass {
    public:
    // delegates to myclass(int)
    myclass(): myclass(42) {}

    // only one ctor needs to init members
    myclass(int nr): c_num_(10), num_(nr) {}

    // delegates to myclass(int)
    myclass(int nr1, int nr2): myclass(nr1 + nr2) {}


    private:
    int c_num_;
    int num_;
};
```

in-class member initializer

```
// c++11 even better: in-class member initializer
class myclass {
    public:
    // delegates to myclass(int)
    myclass(): myclass(42) {}

    // only one ctor needs to init members
    myclass(int nr): num_(nr) {}

    // delegates to myclass(int)
    myclass(int nr1, int nr2): myclass(nr1 + nr2) {}


    private:
    int c_num_ = 10; // can overwritten by ctor (but doesn't have to)
    int num_;
};
```

default & delete

```
// the old way of disabling special functions
class no_copy_class {
    private:
    // define copy constructor private -> copy not possible
    no_copy_class(no_copy_class const & rhs) {
    }
};

// the c++11 way of deleting special functions
class no_copy_class {
    public:
    // you can enable the default behavior of special function
    no_copy_class() = default;

    // this disables the compiler-generated default-ctor
    no_copy_class(int a);

    // it is public information that this class is non-copyable
    no_copy_class(no_copy_class const & rhs) = delete;

    // you can mark any function as deleted
    void some_fct() = delete;
};
```

override & final

```
struct base {

    virtual void fct1() const; // not pure (=0)
    virtual void fct2() const;
    virtual void fct3() const final; // no one writes a better version!
};

struct derived final: public base {

    // forgot const, no override, but no compile error
    void fct1();
    // compiler will fail since base::fct2 with this signature is not found
    void fct2() override;
    // works
    void fct2() const override;
    // fails since base::fct3 final
    void fct3() const override;
};

// fails since derived is final
struct no_chance: public derived {
};
```

standard types

```
uint  a; // is it 32 or 64 bit?!
         // may depend on the machine and OS
         // ambiguity is not our friend


// c++11 introduces
uint8_t  a0;
uint16_t a1;
uint32_t a2;
uint64_t a3;

int8_t  a4;
int16_t a5;
int32_t a6;
int64_t a7;
```

large features covered later

- smart pointer
- rvalue reference / universal reference
- move semantics / perfect forwarding / `noexcept`
- variadic templates

further features

- scoped enums
- universal initialization / `std::initializer_list`
- `static_assert`
- `std::tuple`
- `<regex>`, `<chrono>`, `<ratio>`
- `<thread>`, `<mutex>`, `<future>`
- and more…

for further information

- cplusplus.com or cppreference.com reference
- Book: Scott Meyers: Effective Modern C++
- http://www.slideshare.net/adankevich/c11-15621074