C++11

initializer lists

is used to access values in an initialization list, which are themselves of type const T.

```
auto il = { 10, 20, 30 };  // the type of il is an initializer_list
```

constructors taking only one argument of this type are called initializer-list constructors

```
struct myclass {
  myclass (int,int);
  myclass (initializer_list<int>);
  /* definitions ... */
};

myclass foo {10,20};  // calls initializer_list ctor
myclass bar (10,20);  // calls first constructor
```

Note that initializer-list constructors take precedence when they are used

uniform initialization

we can now also initialize the standard library containers in the following ways:

```
std::vector<std::string> v = { "xyzzy", "plugh", "abracadabra" };
std::vector<std::string> v({ "xyzzy", "plugh", "abracadabra" });
std::vector<std::string> v{ "xyzzy", "plugh", "abracadabra" }; // "Uniform initialization"
```

auto

with the keyword auto C++ can do automatic type deduction

```
int fun() { return zero; }

int& fun2() {return zero;}

int* fun3() { return &zero;}

auto x = 0;                 // OK, x is of type int
const auto y = 0.;          // OK, y is of type const double -- note however the conversion due to cout below
//y = 5.;                   // Error
//auto z;                   // Error!
auto s = std::string("Hello, world!");      //OK
auto p = &y;                // OK -- pointer to constant double

auto iarr = {1,2,3,4};  // OK -- int[4]

auto f = fun();             // OK --  f is of type int
auto f2 = fun2();           // p2 is of type int, not int& -- use auto& f2bis = fun2() instead
auto pf = fun3();           // pf is of type int*
```

advantage: avoid long (templated) typenames and typedefs; avoid difficult type deductions doing yourself

how: the rules are essentially the same as in template argument deduction

certain pitfalls possible: proxy class, vector<bool>

decltype

is related to auto : it inspects the type of an entity or expression

```
int i = 10;
decltype(i) j = i * 5;        // j is an int
```

recall the ‘hack’ for the addition of two different functions with templates?

here is a simple way in C++11:

```
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u) { // return type depends on template parameters
  return t+u;
}
```

it uses a late specified return type

in the examples related to this week’s lecture you can find an example of how to add integers, or complex numbers this week. (it also contains some advanced trick)

see later also for lambda functions where decltype is often the only solution

range-based for loop

for loops can now be range based:

```
// from cppreference.com
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {0, 1, 2, 3, 4, 5};

    for(const int &i : v) // access by const reference
        std::cout << i << ' ';
    std::cout << '\n';

    for(auto i: v) // access by value, the type of i is int
        std::cout << i << ' ';
    std::cout << '\n';

    for(auto&& i: v) // access by reference, the type of i is int&
        std::cout << i << ' ';
    std::cout << '\n';

    for(int n: {0, 1, 2, 3, 4, 5}) // the initializer may be a braced-init-list
        std::cout << n << ' ';
    std::cout << '\n';

    int a[] = {0, 1, 2, 3, 4, 5};
    for(int n: a)  // the initializer may be an array
        std::cout << n << ' ';
    std::cout << '\n';

    for(int n: a)
        std::cout << 1 << ' '; // the loop variable need not be used
    std::cout << '\n';

}
```

random number generators

A random number class (and other mathematical functions) from the boost libraries have been included in the standard library. It is defined in the header <random>

```
// exponential_distribution -- based on the example from cplusplus.com
#include <iostream>
#include <random>

int main()
{
  const int nrolls=10000;  // number of experiments
  const int nstars=100;    // maximum number of stars to distribute
  const int nintervals=10; // number of intervals

  unsigned int seed = 10;
  std::cout << "Enter seed : ";
  std::cin >> seed;
  std::mt19937 generator(seed);     // Mersenne-Twister random number generator
  std::exponential_distribution<double> distribution(3.5);      // exponential distribution
  std::uniform_real_distribution<double> rnd(0.0, 1.0);         // uniform real distribution

  std::cout << "\na random number : " << rnd(generator) << "\n";

  int p[nintervals]={};

  for (int i=0; i<nrolls; ++i) {
    double number = distribution(generator);
    if (number < 1.0) ++p[int(nintervals*number)];
  }

  std::cout << "exponential_distribution (3.5):" << std::endl;
  std::cout << std::fixed; std::cout.precision(1);

  for (int i=0; i<nintervals; ++i) {
    std::cout << float(i)/nintervals << "-" << float(i+1)/nintervals << ": ";
    std::cout << std::string(p[i]*nstars/nrolls,'*') << std::endl;
  }

  return 0;
}
```

nullptr

0 and are NULL deprecated for pointers (they really are integers). Instead, nullptr should be used for null pointers, for instance in initialization. nullptr_t is the the type of the null pointer literal, in a beautiful way of a cyclical definition

```
// based on cppreference.com
#include <cstddef>
#include <iostream>

template<class F, class A>
void Fwd(F f, A a)              // template forward function
{
    f(a);
}

void g(int* i)
{
    std::cout << "Function g called\n";
}

void f(std::nullptr_t nullp)
{
   std::cout << "null pointer overload\n";
}


int main()
{
    g(NULL);            // Fine
    g(0);               // Fine

    Fwd(g, nullptr);    // Fine
//  Fwd(g, NULL);    // ERROR: No function f(int)
    f(nullptr);      // Fine
    return 0;
}
```

note that it is legal to delete a nullptr, unlike 0 or NULL

tuples

tuples pack objects — possibly of different type — together in a single object, just like pair objects do for pairs. Important functions are make_tuple, get, tie and ignore

```
// tuple example from cplusplus.com
#include <iostream>     // std::cout
#include <tuple>        // std::tuple, std::get, std::tie, std::ignore

int main ()
{
  std::tuple<int,char> foo (10,'x');
  auto bar = std::make_tuple ("test", 3.1, 14, 'y');

  std::get<2>(bar) = 100;                                    // access element

  int myint; char mychar;

  std::tie (myint, mychar) = foo;                            // unpack elements
  std::tie (std::ignore, std::ignore, myint, mychar) = bar;  // unpack (with ignore)

  mychar = std::get<3>(bar);

  std::get<0>(foo) = std::get<2>(bar);
  std::get<1>(foo) = mychar;

  std::cout << "foo contains: ";
  std::cout << std::get<0>(foo) << ' ';
  std::cout << std::get<1>(foo) << '\n';

  return 0;
}
```

variadic templates

C++11 allows you to use variadic templates. They can be instantiated with any number of template arguments

```
// for a class or struct
template<class ... Types> struct Tuple {};      // the std::tuple is defined with variadic templates

// for a function
template<class ... Types> void f(Types ... args) {};    // uses ... to unpack

// example 1
template<typename T>                            // variadic templates are very often used recursively
T adder(T v) {
  return v;
}

template<typename T, typename... Args>
T adder(T first, Args... args) {
  return first + adder(args...);
}
```

constexpr

means that it is possible to evaluate the value or the function at compile time

This implies const for objects:

```
int i;                                // non-const
//constexpr auto ArraySize1 = i;      // error, i  not known at compile time
//std::array<int, ArraySize1> m1;     // error, same problem
constexpr auto ArraySize2 = 10;       // OK
std::array<int, ArraySize2> m2;       // OK
```

For functions it implies inline. If an argument is not known at compile time, the function evaluates at runtime, as usual.

```
constexpr
int fac(const int n) noexcept {
  return n==1? 1 : n *fac(n-1);
}

std::array<int, fac(4)> m3;          // OK
```

(in C++11 *only a return executable statement* is allowed; starting from C++14 more flexibility is allowed: local variables and loops; the conditional ? : operator is a single statement!)

defaulted and deleted constructors

remember the trick of putting the copy constructor and destructor in the private part of a class? C++11 offers a better solution:

```
struct noncopyable
{
  noncopyable() =default;   // needed because explicit writing of the (albeit deleted) copy constructor
                            // prevents the default constructor; this can be fixed by defaulting it with
                            // no performance penalty
  noncopyable(const noncopyable&) =delete;  // deleted copy construction; calling it results in compile-time error
  noncopyable& operator=(const noncopyable&) =delete;   // deleted assignment
};
```

Default behavior is simple syntax, it corresponds to the automatically generated code by the compiler. It can only be applied to member functions and can have no default arguments

you can delete special and normal member (and non-member) functions to prevent them from being defined or called. In particular, automatic code generation by the compiler is prohibited.

```
template<typename T>
void fun_int_only(T) = delete;

void fun_int_only(int) { return;}       // repeat for const int, int&,... if needed
```

std::array

is a container equally efficient to C's T[n]. They have fixed size and do not manage memory through an allocator. The size is a template parameter. A special feature is that they can be treated as tuple objects (get, tuple_size,tuple_element)

```
int main ()
{
  std::array<int,10> myarray;

  myarray.fill(10);

  // print content
  std::cout << "myarray contains:";
  for (unsigned int i=0; i<10; i++)
    std::cout << ' ' << myarray[i];
  std::cout << '\n';

  std::tuple_element<0,decltype(myarray)>::type myelement;  // int myelement
  myelement = std::get<2>(myarray);
  std::get<2>(myarray) = std::get<0>(myarray) + 5;
  std::get<0>(myarray) = myelement;

   for ( auto it = myarray.cbegin(); it != myarray.cend(); ++it )
    std::cout << ' ' << *it;   // cannot modify *it
  std::cout << "\n";

  return 0;
}
```

hash table

C++11 has unordered associative containers

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap

Internally, they order their elements using hash tables such that fast access is provided (using std::hash , which can be overloadeded)

The unordered_map consists of <key, value> pairs:

```
// unordered_map::insert -- from cplusplus.com
#include <iostream>
#include <string>
#include <unordered_map>

int main ()
{
  std::unordered_map<std::string,double>
              myrecipe,
              mypantry = {{"milk",2.0},{"flour",1.5}};

  std::pair<std::string,double> myshopping ("baking powder",0.3);

  myrecipe.insert (myshopping);                        // copy insertion
  myrecipe.insert (std::make_pair<std::string,double>("eggs",6.0)); // move insertion
  myrecipe.insert (mypantry.begin(), mypantry.end());  // range insertion
  myrecipe.insert ( {{"sugar",0.8},{"salt",0.1}} );    // initializer list insertion

  std::cout << "myrecipe contains:" << std::endl;
  for (auto& x: myrecipe)
    std::cout << x.first << ": " << x.second << std::endl;

  std::cout << std::endl;

  std::unordered_map<std::string,double>::hasher hfun = myrecipe.hash_function();
  for (auto& x: myrecipe) std::cout << hfun(x.first) << std::endl;

  return 0;
}
```

static_assert

performs compile-time assertion checking

checks a bool expression and provides an error message if not satisfied

```
static_assert(2+2==5, "wrong sum");
```

```
tst.cpp: In function 'int main()':
tst.cpp:28:3: error: static assertion failed: wrong sum
   static_assert(2+2==5, "wrong sum");
```

lambda functions

to have quick, inlined functions without having to write a named function

basic syntax:

```
#include <iostream>

using namespace std;

int main()
{
    auto func = [] () { cout << "Hello, world!\n"; };
    func(); // now call the function
}
```

- the *capture specification* [] tells the compiler we create a lambda function
- next are the *arguments* () : in this case there are none
- the compiler (in this example) automatically determines the *return type* (none here)
- then follows the *function body* {…}
- with auto we can avoid having to write a function pointer explicitly
- the function is called in the next line

the full syntax is:

```
[ captures ] (parameters) -> returnTypesDeclaration { lambdaStatements; }
```

so by writing [] () -> int { /* … */ }; you specify the return type to be int

for_each

lambda functions can be used efficiently with the STL using for_each defined in <algorithm> :

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

using namespace std;

int main() {
  vector<int> v;
  v.push_back( 1 );
  v.push_back( 2 );

  for ( auto it = v.begin(); it != v.end(); ++it ) {
    cout << *it << endl;
  }

  for_each( v.begin(), v.end(), [] (int val) {
    cout << val << endl;
  } );
}
```

- in for_each(first, last, f) a function object f is applied to every iterator in the range first to last
- has the right end condition
- is as efficient as the usual for loop

lambda functions

next examples (similar as before)

```
auto sum = [] (int i, int j) { return i+j;};
std::cout << sum(2,4) << std::endl;

std::vector<int> student_grades {20, 40, 67, 99, 13, 42, 65, 81, 82, 35, 79, 20, 4, 96, 54, 49, 35, 67, 10, 39 };

auto nr_passed = std::count_if(student_grades.begin(), student_grades.end(), [] (int val) { return (val >= 50);});
std::cout << "number of passed students is : " << nr_passed << " or " << static_cast<double>(nr_passed*100)/student_grades.size() << "
    percent. \n";
```

how to pass a parameter from outside the lambda function body? the second snippet shows how to do this, where threshold is passed by value (ie a copy is made)

```
auto nr_almost_passed = std::count_if(student_grades.begin(), student_grades.end(), [] (int val) { return (val >= 40 && val < 50);});
std::cout << "number of almost-passed students is : " << nr_almost_passed << " or " << static_cast<double>(nr_almost_passed*100)/student_grades.
    size() << " percent. \n";

int threshold = 45;
auto nr_almost_passed2 = std::count_if(student_grades.begin(), student_grades.end(), [threshold] (int val) { return (val >= threshold && val < 50);
    });
std::cout << "number of almost-passed students is : " << nr_almost_passed2 << " or " << static_cast<double>(nr_almost_passed*100)/student_grades.
    size() << " percent. \n";
```

we can equivalently do this as follows: with [=] all parameters found inside the lambda function body are passed by value

```
auto nr_almost_passed3 = std::count_if(student_grades.begin(), student_grades.end(), [=] (int val) { return (val >= threshold && val < 50);});
std::cout << "number of almost-passed students is : " << nr_almost_passed3 << " or " << static_cast<double>(nr_almost_passed*100)/student_grades.
    size() << " percent. \n";
```

sometimes we also want to change the value of a parameter inside the lambda function. with [&] all values are passed by reference:

```
int sum = 0;
int count = 0;
int ref_val = 0;
// we want to know the average of all grades excluding all marks below 5

for_each(student_grades.begin(), student_grades.end(), [&, ref_val] (int val) { if (val >= ref_val) { sum += val; count++;}} );
std::cout << "average grade : " << static_cast<double>(sum) / count << " for " << count << " students.\n";
```

lambda closures

these are the rules for lambda closures:

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

capturing by reference is not without dangers and can lead to dangling references (see textbooks)

unique_ptr

- is an example of a smart pointer
- has little to no overhead compared to bare pointers
- they take ownership of a resource and have hence sole responsibility for deleting the resource at some point: no 2 unique_ptr instances can manage the same resource
- this will occur when the unique_ptr is destroyed or ownership changes due to assignment or unique_ptr::reset is called
- copy constructor is deleted
- defined in <memory>

```cpp
// from cppreference.com
#include <iostream>
#include <memory>

struct Foo
{
    Foo()      { std::cout << "Foo::Foo\n";  }
    ~Foo()     { std::cout << "Foo::~Foo\n"; }
    void bar() { std::cout << "Foo::bar\n";  }
};

void f(const Foo &)
{
    std::cout << "f(const Foo&)\n";
}

int main()
{
    std::unique_ptr<Foo> p1(new Foo);  // p1 owns Foo
    if (p1) p1->bar();

    {
        std::unique_ptr<Foo> p2(std::move(p1));  // now p2 owns Foo
        f(*p2);                  // dereferencing of unique_ptr

        p1 = std::move(p2);  // ownership returns to p1
        std::cout << "destroying p2...\n";
    }

    if (p1) p1->bar();

    // Foo instance is destroyed when p1 goes out of scope
}
```

std::function

this class template is a general-purpose polymorphic function wrapper to store, copy and invoke functions, lambda expressions, std::bind, pointers to member functions, …

it replaces function pointers

```cpp
// from cppreference.com
#include <functional>
#include <iostream>

struct Foo {
    Foo(int num) : num_(num) {}
    void print_add(int i) const { std::cout << num_+i << '\n'; }
    int num_;
};

void print_num(int i)
{
    std::cout << i << '\n';
}

struct PrintNum {
    void operator()(int i) const
    {
        std::cout << i << '\n';
    }
};
```

```cpp
int main()
{
    // store a free function
    std::function<void(int)> f_display = print_num;
    f_display(-9);

    // store a lambda
    std::function<void()> f_display_42 = []() { print_num(42); };
    f_display_42();

    // store the result of a call to std::bind
    std::function<void()> f_display_31337 = std::bind(print_num, 31337);
    f_display_31337();

    // store a call to a member function
    std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
    const Foo foo(314159);
    f_add_display(foo, 1);

    // store a call to a member function and object
    using std::placeholders::_1;
    std::function<void(int)> f_add_display2= std::bind( &Foo::print_add, foo, _1 );
    f_add_display2(2);

    // store a call to a member function and object ptr
    std::function<void(int)> f_add_display3= std::bind( &Foo::print_add, &foo, _1 );
    f_add_display3(3);

    // store a call to a function object
    std::function<void(int)> f_display_obj = PrintNum();
    f_display_obj(18);
}
```

override and final

the keyword override ensures that a function is virtual and overrides a virtual function of the base class

```
// from cppreference.com
struct A
{
    virtual void foo();
    void bar();
};

struct B : A
{
    //void foo() const override; // Error: B::foo does not override A::foo
                                 // (signature mismatch)
    void foo() override; // OK: B::foo overrides A::foo
    //void bar() override; // Error: A::bar is not virtual
};
```

the keyword final ensures that a function is virtual and may not be overridden by a derived class

type traits

Type traits defines a compile-time template-based interface to query or modify the properties of types.

http://en.cppreference.com/w/cpp/types#Type_traits_.28since_C.2B.2B11.29

regular expressions

are a standardized way to express patterns to be matched against sequences of characters

```
#include <iostream>
#include <string>
#include <regex>

int main()
{
  // Simple regular expression matching
  std::string fnames[] = {"foo.txt", "bar.txt", "baz.dat", "student"};
  std::regex txt_regex("[a-z]+\\.txt");

  for (const auto &fname : fnames) {
    std::cout << fname << ": " << std::regex_match(fname, txt_regex) << '\n';
  }
}
```

see the documentation:

http://www.cplusplus.com/reference/regex/

the grammar rules for regular expressions follow the ECMAScript

move semantics

see examples

read your textbook for lvalue, rvalue, prvalue, xvalue, glvalue

rvalue reference: T&&

topics not covered

- concurrency and the c++ memory model
- shared_ptr, weak_ptr
- threads : thread, atomic, mutex, …
- operator new and placement new
- move semantics, rvalue
- regex