ΗΥ352 : ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΟΛΟΓΙΑΣ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

8ο ΦΡΟΝΤΙΣΤΗΡΙΟ

ΔΙΔΑΣΚΟΝΤΑΣ
Αντώνιος Σαββίδης

Περιεχόμενα

- C++11 features
  - Miscellaneous
    - auto & decltype
    - uniform initialization
    - range-for statement
    - enum classes
    - nullptr
    - constexpr
    - user defined literals
    - lambda functions
  - Class related
    - override & final
    - default & delete
    - delegating constructors
    - in-class member initializers
  - Templates
    - template aliases
    - variadic templates
  - Standard Library
    - regular expressions
    - threads

auto & decltype (1/2)

- Το ***auto*** χρησιμοποιείται για να εξάγουμε αυτόματα τον τύπο μιας μεταβλητής από την αρχικοποίησή της
  - Προφανώς η αρχικοποίηση είναι απαραίτητη
- Το ***decltype(****expr****)*** αντιστοιχεί στον τύπος της έκφραση expr
  - Μπορεί να χρησιμοποιηθεί οπουδήποτε χρησιμοποιείται ένας τύπος

```
auto x = 1;           //x is of type int
auto a = x;           //a is also of type int
auto b;               //error: no initializer to deduce type
auto y = 2, z = 3;    //y and z are both of typeint
auto w = "Hello!";    //w is of type const char *
auto z = 4, y = 5.5; //error: one initializer is int, other is double

decltype(5) c;        //c is of type int because 5 is an int
decltype(c) d = 6;    //d is also of type int
std::vector<int> v;
for(std::vector<int>::iterator i = v.begin(); i != v.end(); ++i);
for(auto i = v.begin(); i != v.end(); ++i);//i is of type
                                          //std::vector<int>::iterator
```

auto & decltype (2/2)

- Υπάρχει ένα νέο συντακτικό για να γράψουμε το πρωτότυπο μιας συνάρτησης
  - ***auto*** function_name(*arguments*) -> ***return_type*** {…}
    - Ισοδύναμο με το ***return_type*** function_name(*arguments*) {…}
- Με το νέο συντακτικό, στο return_type μπορούμε να χρησιμοποιήσουμε το decltype και τα ορίσματα της συνάρτησης
  - Αυτό βολεύει ειδικά σε templates που μπορεί να είναι δύσκολο (ή ακόμα και αδύνατο) να εξάγουμε τον επιστρεφόμενο τύπο

Problem

```
template <typename T1, typename T2>
??? add(T1 t1, T2 t2) { return t1 + t2; }
//for T1=int, T2=int,    return value is int
//for T1=int, T2=double, return value is double
SomeType operator+(UserType1 t1, UserType2 t2);
//for T1=UserType1, T2=UserType2, return value is SomeType
```

Solution

```
template <typename T1, typename T2>
auto add(T1 t1, T2 t2) -> decltype(t1 + t2) { return t1 + t2; }
```

Uniform initialization (1/2)

- Η C++03 είχε αρκετά θέματα με τις αρχικοποιήσεις

```
//inconsistent list initialization
string a[] = { "foo", "bar" };      //ok: initialize array variable
vector<string> v = { "foo", "bar"}; //error: does not work here
void f(string a[]);
f( { "foo", "bar" } );              //error: does not work here as well

//inconsistent initialization styles
int a = 2;          //assignment style
int aa[] = { 2, 3 };//assignment style with list
complex z(1,2);     //functional style initialization
x = Ptr(y);         //functional style for conversion/cast/construction

//variable declaration or maybe function declaration?
int a(1);   //variable definition, constructs int with initial value 1
int b();    //function declaration, i.e. int b(void);
int b(foo); //variable definition (if foo is a value) or function
            //declaration (if foo is a type)
```

Uniform initialization (2/2)

- Η C++11 λύνει το πρόβλημα με ένα ενιαίο τρόπο αρχικοποίησης – ***{} initializer lists***
  - Μπορείτε βέβαια να χρησιμοποιήσετε και τα παλιά

```
//consistent list initialization
string a[] = { "foo", "bar" };      //ok: initialize array variable
vector<string> v = { "foo", "bar"}; //now works ok
void f(string a[]);
f( { "foo", "bar" } );              //now works ok as well
//initializer lists work for all cases
int a{2};           //construction
int a = {2};        //construction - assignment initialization
int a = int{2};     //construction–constructor assignment initialization
int aa[] = {2, 3}; //array initialization
complex z{1,2};     //construction
x = Ptr{y};         //conversion/cast/construction
//all variable definitions
int a{1};  //variable definition, constructs int with initial value 1
int b{};   //variable definition, constructs int with initial value 0
int b{foo};//variable definition (foo must be a value, otherwise error)
```

Range-for statement

- Εκτός από το παραδοσιακό for loop, υπάρχει και ένα for loop για να διασχίζουμε ακολουθίες ή συλλογές
  - ***for***(*variable* **:** *range*) *stmt*
- Το range μπορεί να είναι οτιδήποτε προσφέρει begin και end συναρτήσεις που ορίζουν την ακολουθία
  - STL containers, initializer lists, arrays, user defined containers

```
std::vector<int> v;
for(std::vector<int>::iterator i = v.begin(); i != v.end(); ++i)
  std::cout << *i << std::endl; //traditional loop
for(int x : v);
  std::cout << x << std::endl;  //new range-for loop
for(int& x : v)   //the reference allows changing the collection values
  x = x + 1;      //this increments all elements by 1
for(auto& x : v)  //same as above using the new auto feature
  x = x + 1;
for(auto& x : {1,2,3}) //can also use anything with begin&end functions
  std::cout << x << std::endl;
```

enum classes

- Τα παραδοσιακά enums έχουν κάποια προβλήματα
  - implicit conversion σε int
  - κάνουν τα ονόματα ορατά στο εξωτερικό scope οδηγώντας σε πιθανά name clashes
  - δεν επιτρέπουν forward declaration
- Τα enum classes λύνουν αυτά τα προβλήματα

```
enum class Color { red = 0, blue = 1};

Color c1 = 1;                       //error: no int->Color conversion
Color c2 = static_cast<Color>(1); //ok, c2 is 1, i.e. blue
Color c3 = blue;                    //error: blue not in scope
Color c4 = Color::blue;             //ok, typical use
int x = Color::blue;                //error: no Color->int conversion
int y = static_cast<int>(Color::blue); //ok, y is 1

enum class SomeOtherEnum;                       //forward declaration
void foo(SomeOtherEnum value);                  //usage
enum class SomeOtherEnum { value1, value2, }; //definition
```

nullptr

- Το ***nullptr*** είναι αναγνωριστικό (τύπου nullptr_t) που υποδηλώνει το null pointer
  - Δεν είναι αριθμός!

```
void f(int);        //#1
void f(char *);     //#2
f(0);               //which f is called? #1

void g(double);     //#3
void g(char *);     //#4
g(0);               //which g is called? None! error: ambiguous call

f(nullptr);         //#2 (the one with the pointer)
g(nullptr);         //#4 (the one with the pointer)

int x = nullptr;    //error: nullptr is not an int
int *y = nullptr;
int *z = 0;         //old 0 style for null pointer is also valid
assert(y == z);     //the value of the null pointer is the same
```

constexpr

- Το ***constexpr*** δηλώνει ότι μια συνάρτηση μπορεί να κληθεί κατά τη μεταγλώττιση ώστε να υπολογίσει κάποια τιμή που θα είναι διαθέσιμη ως σταθερά
  - Η υλοποίηση μιας τέτοια συνάρτησης έχει αρκετούς περιορισμούς
    - Επιτρέπει μόνο ένα return statement
    - Βέβαια με τον τελεστή ?: και κλήσεις συναρτήσεων μπορούμε να υποστηρίξουμε αναδρομή
  - Η C++14 χαλαρώνει τους περιορισμούς και έχει φιλικότερο συντακτικό
    - loops, τοπικές μεταβλητές, κλπ.

```
int factorial1(int n)
  { return n <= 1 ? 1 : (n * factorial1(n - 1)); }
template<int N> struct S { static const int val = N; };

cout << S<factorial1(5)>::val; //error: factorial1(5) is not constant

constexpr int factorial2(int n)
  { return n <= 1 ? 1 : (n * factorial2(n - 1)); }
cout <<S<factorial2(5)>::val; //ok, will print 120 at runtime
```

User defined literals

- Η C++11 επιτρέπει literals (αριθμούς, χαρακτήρες, strings) με κάποια user-defined κατάληξη που παράγουν user-defined αντικείμενα
  - *type operator "" _suffix (type);*

```
constexpr double operator"" _deg (long double degrees)
  { return degrees * 3.141592/180; } //convert degrees to rads
double x = 90.0_deg;                 //same as x = 1.570796

struct Binary { long val; Binary(long v) : val(v) {} };
Binary operator"" _b (const char* p, size_t n) { //must have these
  long result = 0;                                //exact arguments
  for (size_t i =0; i < n; ++i) {
    if (p[i] != '0' && p[i] != '1')
      throw std::runtime error("invalid binary given");
    result = (result << 1) + p[i] - '0';
  }
  return Binary(result);
}
Binary b = 101100000_b; //b is Binary(352)
```

Lambda functions (1/2)

- Η C++11 υποστηρίζει τοπικές ανώνυμες συναρτήσεις
  - [capture-list](params)^{opt} ***mutable***^{opt} -> return_value^{opt} {/*body*/}
- Το capture list περιέχει μεταβλητές που θέλουμε να είναι ορατές μέσα στη lambda συνάρτηση καθώς και το αν θα είναι ορατές by value ή by reference
  - [&] σημαίνει όλα ορατά by reference
  - [=] σημαίνει όλα ορατά by value (δηλαδή copied)
  - [&x, y, &z] σημαίνει ορατά τα x, z by reference και το y by value
  - [=, &x] σημαίνει όλα ορατά by value και το x by reference
- Το **mutable** χρησιμοποιείται αν θέλουμε να μπορούμε να αλλάξουμε τις μεταβλητές που είναι captured by value
- Αν δεν δηλώσουμε **return_value** τότε εξάγεται από την επιστρεφόμενη τιμή της συνάρτησης (αν υπάρχει)
  - Πολλαπλά return statements με διαφορετικούς τύπους είναι error

Lambda functions (2/2)

- Παραδείγματα

```
std::vector<int> v{2, 4, 3, 1, 5};
std::sort(v.begin(), v.end(), [](int x1, int x2) { return x1 > x2;});
// v is now {5, 4, 3, 2, 1}

int x = 3;
auto iter = std::remove_if(v.begin(), v.end(),
  [x](int n) { return n > x; }); // remove elements greater than 3
v.erase(iter, v.end());          // v is now {3, 2, 1}

char s[] = "Hello World!";
int uppercase = 0;               // to be modified by the lambda
std::for_each(s, s + sizeof(s),  // &uppercase means capture by reference
 [&uppercase] (char c){ if (isupper(c)) uppercase++; }); // can change it
assert(uppercase == 2);

auto add = [](int x, int y) { return x + y; }; // can store in variable
assert(add(1, 2) == 3);

void foo(void (*f)(void)){ f(); }
foo([] { std::cout << "lambda"; }); // can use as function argument
```

override

- Μια συνάρτηση που γίνεται override σε ένα derived class δε χρειάζεται απαραίτητα ειδικό συντακτικό
  - Αυτό μπορεί να δημιουργήσει λάθη
- Το ***override*** χρησιμοποιείται για να γίνει explicit η πρόθεση για override

```
struct Base {
  virtual void f();
  virtual void g() const;
  virtual void h();
  void k(); // not virtual
};
```

```
struct Derived1 : Base {
  void f();           // ok, overrides Base::f()
  void g();           // doesn't override Base::g() (non const), new g
  virtual void h();   // overrides Base::h()
  void k();           // doesn't override Base::k() (not virtual), new k
};
struct Derived2 : Base {
  void f() override;          // ok, overrides Base::f()
  void g() override;          // error: wrong type
  virtual void h() override; // ok, overrides Base::h()
  void k() override;          // error: Base::k() is not virtual
};
```

final

- Το ***final*** keyword μπορεί να χρησιμοποιηθεί σε
  - μια virtual συνάρτηση ώστε να μη μπορεί να γίνει override
  - σε μια κλάση ώστε να μη μπορεί να κληρονομηθεί

```
struct X { virtual void f(); };
struct Y : X { virtual void f() final; }; // disallow further override
struct Z : Y { virtual void f(); }; // error: cannot override final Y:f()

struct S { void f() final; }        // error: f is not virtual

class MemoryManager final {  // Singleton class, so disable inheritance.
private:
     MemoryManager(void);
     MemoryManager(const MemoryManager&);
     MemoryManager& operator=(const MemoryManager&);
public:
     MemoryManager& Instance();
     // other methods
};
```

default & delete

- Με το ***default*** δηλώνουμε ότι θέλουμε τη default υλοποίηση κάποιας μεθόδου
  - π.χ. constructor, assignment operator
- Με το ***delete*** δηλώνουμε ότι μια μέθοδος δε θέλουμε να ορίζεται για την κλάση μας

```
class X {
  X& operator=(const X&) = default; //Default assignment operator
  X(int x) {}
  X(void) = default; //Default empty constructor – same as the one the
};      //compiler would generate if we didn't provide any constructor

class Y {
  Y& operator=(const Y&) = delete;  //Disallow assignment
  Y(const Y&) = delete;             //Disallow copy construction
}; //overall, Y cannot be copied

struct Base { void f(); };
struct Derived : Base { void f() = delete; } //No f for Derived
Derived d; d.f(); //error: f is deleted
((Base&)d).f(); //ok, calls Base:f()
```

Delegating constructors

- Στη C++03 κοινή λειτουργικότητα σε constructors πρέπει να μπει σε ξεχωριστή συνάρτηση και να κληθεί από κάθε constructor ξεχωριστά
- Στην C++11 ένας constructor μπορεί να καλέσει άλλο

```
class X {
  int x;
  void do_something(int val);
public:
  //C++03
  X(int _x) : x(_x) { do_something(x); } //assignment to x and call to
  X() : x(0) { do_something(x); }        //do_something is repeated
  X(const char* s) : x(atoi(s)) { do_something(x); } //in all ctors
  //C++11
  X(int _x) : x(_x) { do_something(x); } //code written once here
  X() : X(0) {}                        //delegate to constructor X(int)
  X(const char* s) : X(atoi(s)) {} //delegate to constructor X(int)
};
```

In-class member initializers

- Στη C++11 μπορούμε να αρχικοποιήσουμε members μιας κλάσης απευθείας στη δήλωσή τους
  - Αν τα αρχικοποιήσει και κάποιος constructor υπερισχύει η αρχικοποίηση του constructor

```
class X {
  int a = 1, b = 2;
  std::string str = "Hello!";
public:
  X(int _a, int _b) : a(_a), b(_b) {} //str is default initialized
  X() : {}                            //all members are default initialized
  X(const std::string& s) : str(s) {} //a and b are default initialized
};

X x1{0, 1};      //a = 0, b = 1, str = "Hello!"
X x2;            //a = 1, b = 2, str = "Hello!"
X x3{"World!"}; //a = 1, b = 2, str = “World!"
```

Template aliases

- Στη C++11 μπορούμε να δηλώσουμε templates που συμπεριφέρονται σαν άλλα templates (δηλαδή aliases) ενδεχομένως ορίζοντας κάποια από τα ορίσματά τους
  - Δε μπορούν να γίνουν specialized ως κανονικά templates
- Aliases μπορούμε να έχουμε και χωρίς templates (*type aliases)*, σαν εναλλακτικό συντακτικό για το typedef

```
template<class T> struct MyAlloc {};
template<class T>
using Vec = std::vector<T,MyAlloc<T>>; //vector using custom allocator

template<class T>
using ptr = T*;   //'ptr<T>' is now an alias for T*
ptr<int> x;       //same as int* x

typedef std::list<int> IntList1; //Traditional list type decl
using IntList2 = std::list<int>; //same effect with new syntax

typedef void (*FuncPtr1)(int); //Traditional function pointer type decl
using FuncPtr2 = void (*)(int);//same effect with the new syntax
```

Variadic templates (1/3)

- Η C++11 υποστηρίζει templates με μεταβλητό πλήθος παραμέτρων
- Οι πολλαπλές παράμετροι ονομάζονται *parameter packs* και μπορούν να έχουν τη μορφή
  - *typename… Args* – πολλαπλοί τύποι
  - *type… Args* – πολλαπλές τιμές του συγκεκριμένου τύπου
    - π.χ. int... Args
  - template<params> class… Args – πολλαπλά template templates parameters

```
template<typename... Types> struct Tuple {};
Tuple<> t0;           // Types contains no arguments
Tuple<int> t1;        // Types contains one argument: int
Tuple<int, float> t2; // Types contains two arguments: int and float

template<typename... Types> void f(Types... args);
f();       //calls f(void): args contains 0 args
f(1);      //calls f(int) : args contains 1 args: int
f(2, 1.0); //calls f(int, double): args contains 2 args: int and double
```

Variadic templates (2/3)

- Τα parameter packs γίνονται expand μέσα στο template

```
E1 e1; E2 e2; E3 e3; //E1, E2, E3 are types, e1, e2, e3 are variables

template<typename... Types>
void f(Types... args) { //function formals, expanded to f(E1, E2, E3)
  f(args...);  //function arguments, expanded to f(e1, e2, e3);
  f(&args...); //support pattern as well: expanded to f(&e1, &e2, &e3);
  f(g(args)...); //another pattern: expanded to f(g(e1), g(e2), g(e3));
}
f(e1, e2, e3);

template<class A, class B, class...C>
class X : public C... {//Base class specifier, X inherits from E1,E2,E3
  X(const C&... args) : C(args)... {} // Initializer list
  //template argument list
  Tuple<A, B, C...> t1;  //expands to Tuple<A, B, E1, E2, E3>
  Tuple<C..., A, B> t2;  //expands to Tuple<E1, E2, E3, A, B>
  Tuple<A, C..., B> t3;  //expands to Tuple<A, E1, E2, E3, B>
  static const int count = sizeof...(C); //sizeof... operator, count=3
}
X<E1, E2, E3> x;
```

Variadic templates (3/3)

- Παράδειγμα: type-safe printf

```
printf("%s = %f", std::string("pi"), 3.14159); // we want such calls
void printf(const char* s){
  while (s && *s) {
    if (*s == '%' && *(++s) != '%') // %% represents a plain %
      throw runtime_error("invalid format: missing arguments");
    std::cout << *s++;
  }
}
template<typename T, typename... Args>
void printf(const char* s, T value, Args... args) {
  while (s && *s) {
    if (*s == '%' && *(++s) != '%') {
      std::cout << value;         // use first non-format argument
      return printf(++s, args…); // recurse with remaining arguments
    }
    std::cout << *s++;
  }
  throw std::runtime error("extra arguments provided to printf");
}
```

Regular expressions

- Η C++11 παρέχει υποστήριξη στο standard library για regular expressions στο header <regex>

```
#include <regex>
std::regex integer("(\\+|-)?[[:digit:]]+"); //optional sign, and
                                            //one or more digits
for (const auto& str : {"123", "-456", "7.89", "abc"}){
  if (std::regex_match(str, integer))       //check if str is an integer
    std::cout << str << " is an integer" << std::endl;
  else
    std::cout << str << " is not an integer" << std::endl;
}
std::string str = "str 123 with 456 integers 789";
std::smatch match; //structure to hold match information
while (std::regex_search(str, match, integer)){//search str for matches
  std::cout << match.str() << std::endl;
  str = match.suffix().str(); //continue to match remaining text
}
```

Threads

- Η C++11 παρέχει υποστήριξη στο standard library για threads, mutexes, locks, condition variables
  - Header files <thread>, <mutex>, <condition_variable>

```
#include <thread>
#include <mutex>
#include <functional>    // For std::bind.
int amount = 0;
std::mutex amount_mutex; // Mutex object.

void deposit(int val) {
  std::lock_guard<std::mutex> lock(amount_mutex); // Lock mutex for the
  amount += val;                                  // current scope.
}
void test() {
 std::thread t1(std::bind(deposit,100)); // New thread, calls deposit(100).
 std::thread t2([] { deposit(200); });   // New thread, calls deposit(200).
 t1.join();                              // Wait for threads to finish.
 t2.join();
 assert(amount == 300); // After both threads finish, amount should be 300.
}
```