נושאים מתקדמים בתכנות מונחה עצמים
תרגיל/מעבדה 2

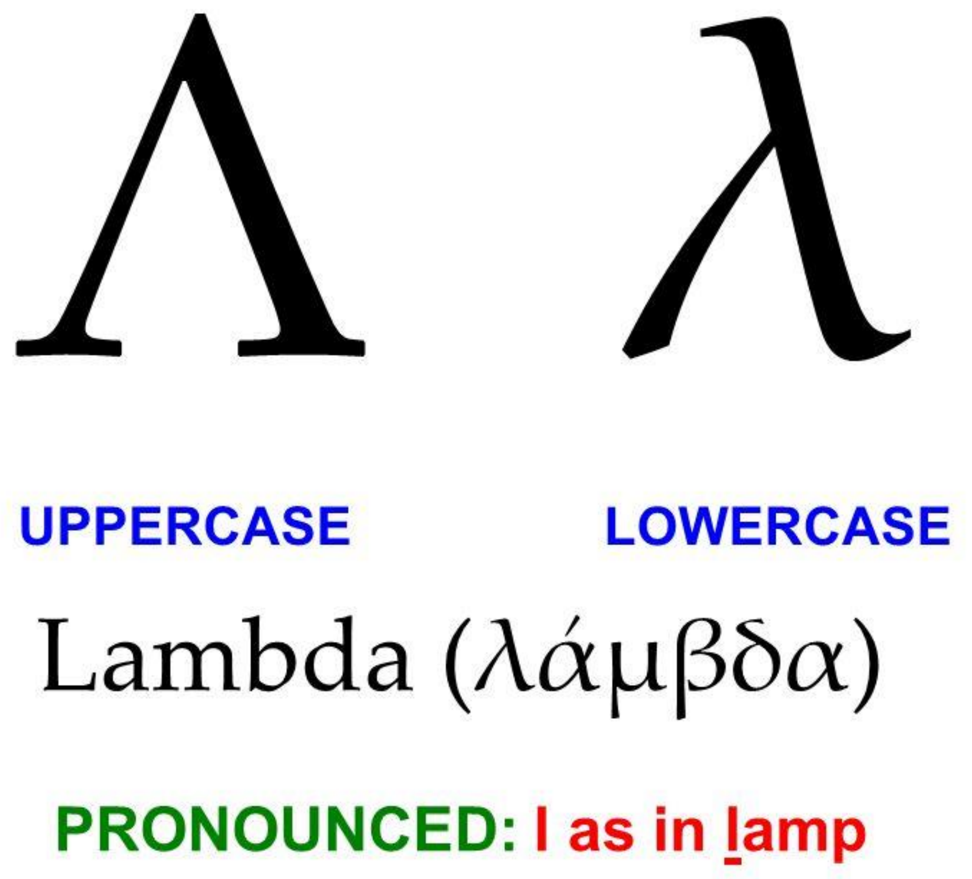
פרופ' עפר שיר

ofersh@telhai.ac.il

החוג למדעי המחשב

תל-חי המכללה האקדמית

מטרת התרגיל: λ



Lambda expressions

Source: B. Stroustrup, *The C++ Programming Language*, 4th ed., 2013

Lambda Expressions (Functions)

A lambda expression consists of a sequence of parts:

• A possibly empty capture list, specifying what names from the definition environment can be used in the lambda expression's body, and whether those are copied or accessed by reference. The capture list is delimited by **`[]`** .

• An *optional* parameter list, specifying what arguments the lambda expression requires. The parameter list is delimited by **`()`** .

• An *optional* **`mutable`** specifier, indicating that the lambda expression's body may modify the state of the lambda (i.e., change the lambda's copies of variables captured by value) .

• An *optional* **`noexcept`** specifier.

• An *optional* return type declaration of the form **`->`** type .

• A body, specifying the code to be executed. The body is delimited by **`{}`** .

A relatively simple example

```
class Modulo_print {
     ostream& os;
     int m;
public:
   Modulo_print(ostream& s, int mm) :os(s), m(mm) {}
   void operator()(int x) const
           { if (x%m==0) os << x << '\n'; }
};
```

(I)

```
void print_modulo(const vector<int>& v, ostream& os,
int m)
     // output v[i] to os if v[i]%m==0
{
  for_each(v.begin(),v.end(),
  [&os,m](int x) { if (x%m==0) os << x << '\n'; } );
}
```

(II)

הערות

- ביטוי ה-λ ואובייקט הפונקציה שקולים.
- רשימת ה-capture, **[&os,m]**, דורשת הפעלת בנאים; **os** נשמר כרפרנס, **m** נשמר כעותק.
- ביטוי ה-λ אינו מחזיר ערך, לכן הוא **void**
- ביטוי ה-λ הינו **const** כברירת מחדל, אחרת עליו לציין **mutable**
- ניתן לשכתב את (II) ללא λ בתחביר C++11 מודרני:

```
void print_modulo(const vector<int>& v, ostream& os,
int m)
{
  for_each(v.begin(),v.end(),Modulo_print{os,m} );
}
```

Naming a lambda expression

ניתן לתת שם לביטוי ה-λ באמצעות התחביר הבא:

```
void print_modulo(const vector<int>& v, ostream& os, 
int m)
{
auto Modulo_print = [&os,m] (int x) { if (x%m==0) os << 
x << '\n'; };

  for_each(v.begin(),v.end(),Modulo_print );
}
```

Capture

If we want to access local names, we have to say so or get an error:

```
void f(vector<int>& v) {
     bool sensitive = true;
     // ...
     sort(v.begin(),v.end(),
          [](int x, int y) { return sensitive ? x<y
: abs(x)<abs(y); }  // ERROR: can't access sensitive
     );
}
```

- **`[]`**: an empty capture list. This implies that no local names from the surrounding context can be used in the lambda body. For such lambda expressions, data is obtained from arguments or from nonlocal variables.

- **`[&]`**: implicitly capture by reference. All local names can be used. All local variables are accessed by reference.

- **`[=]`**: implicitly capture by value. All local names can be used. All names refer to copies of the local variables taken at the point of call of the lambda expression.

- **`[capture-list]`**: explicit capture; the capture-list is the list of names of local variables to be captured (i.e., stored in the object) by reference or by value. Variables with names preceded by **`&`** are captured by reference. Other variables are captured by value. A capture list can also contain **`this`** as an element.

- **`[&, capture-list]`**: implicitly capture by reference all local variables with names not mentioned in the list. The capture list can contain **`this`**. Listed names cannot be preceded by **`&`**. Variables named in the capture list are captured by value.

- **`[=, capture-list]`**: implicitly capture by value all local variables with names not mentioned in the list. The capture list cannot contain **`this`**. The listed names must be preceded by **`&`**. Variables named in the capture list are captured by reference.

Lambda and **this**

We can include class members in the set of names potentially captured by adding **this** to the capture list.

```
class Request {
     function<map<string,string>(const map<string,string>&)>
oper;
     map<string,string> values;
     map<string,string> results;
public:
     Request(const string& s);
     void execute() {
          [this](){results=oper(values);}
     }
};
```

`mutable` Lambdas

- Usually, we don’t want to modify the state of the function object, so by default we can’t.
- Not to be confused with modifying the state of some variable *captured by reference*.
- For the unlikely scenario that we do, we can declare the lambda mutable:

```
void algo(vector<int>& v)
{
     int count = v.size();
     std::generate(v.begin(),v.end(),
          [count]()mutable{ return --count; }
     );
}
```

The `--count` decrements the **copy** of v’s size stored in the closure!

Call and Return

Most rules for lambdas are borrowed from the rules for functions and classes. However, two irregularities should be noted:

1. If a lambda expression does not take any arguments, the argument list can be omitted. Thus, the minimal lambda expression is `[]{}`.
2. A lambda expression’s return type can be deduced from its body.

- If a lambda body does not have a **return**-statement, the lambda’s return type is **`void`**.
- If a lambda body consists of just a single **return**-statement, the lambda’s return type is the type of the **return**’s expression.
- If neither is the case, we have to explicitly supply a return type.

```
void g(double y) {
  [&]{ f(y);}  // return type is void

  auto z1 = [=](int x){ return x+y; }
// return type is double

  auto z2 = [=]()->int { if (y) return 1; else return 2; }
// explicit return type

}
```

The Type of a Lambda

- A lambda that captures nothing can be assigned to a pointer to function of an appropriate type:

```
double (*p1)(double) = [](double a) { return sqrt(a); };
double (*p2)(double) = [&](double a) { return sqrt(a);};
      // ERROR: the lambda captures
double (*p3)(int) = [](int a) { return sqrt(a);};
      // ERROR: argument types do not match
```

- Or just name it using `std::function` :

```
void f(string& s1, string& s2) {
  function<void(char* b, char* e)> rev =
    [&](char* b, char* e) { if (1<e-b)
    { swap( *b, *--e); rev(++b,e); } };
      rev(&s1[0],&s1[0]+s1.size());
      rev(&s2[0],&s2[0]+s2.size());
}
```

decltype()

- לעיתים, בפרט **בתכנות גנרי**, נרצה הסקה של טיפוס הביטוי ללא משתנה מאותחל.
- קיים שימוש בביטוי **auto** – אבל רק כאשר יש בידינו מאתחל מותאם: **auto k = 7;**
- למשל, פונקציה המחשבת סכום של שתי מטריצות, שבעיקרון יכולות להחזיק טיפוסים שונים; טיפוס החזרה יוגדר ע"פ תוצאת החיבור:

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

Lambda practice

Rewrite using Lambda expressions

```
#include<iostream>
#include<list>
#include<algorithm>
#include<iterator>
using namespace std;
struct Moore {
  bool operator() (int x, int y) { return ((x%4) > (y%4)) ; }
} Roger;

int main(void) {
  const int SIZE = 10;
  int array [ SIZE ] = { 2, 6, 4, 8, 12, 10, 16, 14, 20, 18 } ;
  list<int> L (array, array+SIZE);
  L.sort( Roger );
  ostream_iterator<int> out (cout, " ");
  copy(L.begin(), L.end(), out);
  return 0;
}
```

```
#include<vector>
#include<list>
#include<iterator>
#include<iostream>
using namespace std;
template<typename S, typename T, typename P>
T boo(S i, S f, T t, P p) {
  while(i != f) {
    if (p(*i)) *t++ = *i;
    i++;
  }
  return t;
}
class Boogie {
public: bool operator() (int x) { return !(x == 0) && !(x & (x - 1)); }
};
const int Nmax=10;
int main(void) {
  vector<int> v;
  for (int i=0; i<Nmax; i++) v.push_back(i);
  list<int> l(Nmax);
  boo( v.begin(), v.end(), l.begin(), Boogie() );
  for (list<int>::iterator i=l.begin(); i!=l.end(); i++)
    cout << *i << ' ';
  cout << endl;
  return 0;
}
```

```
#include <iostream>
#include <map>
#include <iterator>
using namespace std;
template <typename T> class HBO {
public: bool operator()(const T& a, const T& b) const {
      return a > b;
   }
};
typedef std::multimap< double, string, HBO<double> > mmh;
int main(void) {
   mmh tvs;
   tvs.insert( mmh::value_type(8.5,string("Olive Kitteridge")) );
   tvs.insert( mmh::value_type(9.4,string("The Sopranos")) );
   tvs.insert( mmh::value_type(9.4,string("The Wire")) );
   tvs.insert( mmh::value_type(9.1,string("Game of Thrones")) );
   tvs.insert( mmh::value_type(8.7,string("Boardwalk Empire")) );
   tvs.insert( mmh::value_type(9.4,string("True Detective")) );
   mmh::const_iterator iter = tvs.begin();
   unsigned short list = 4;
   while (list--) {
      cout << iter->first << '\t'<< iter->second << '\n';
      iter++;
   }
   return 0;
}
```

```
#include <iostream>
#include <vector>
#include <algorithm>
template<class T, class U>
struct ACDC {
   bool operator()(const std::pair<T,U>& a, const std::pair<T,U>& b) const {
      return *a.first < *b.first; }
 };
template<class IterIn, class IterOut>
void rock_n_roll(IterIn first, IterIn last, IterOut out) {
   std::vector< std::pair<IterIn,int> > s(last-first);
   for(int i=0; i < s.size(); ++i)
      s[i] = std::make_pair(first+i, i);

   std::sort(s.begin(), s.end(), ACDC<IterIn,int>());

   for(int i=0; i < s.size(); ++i, ++out)
      *out = s[i].second;
 }
int main(void) {
   int a[10] = { 15,12,13,14,18,11,10,17,16,19 };
   std::vector<int> b(10);
   rock_n_roll(a, a+10, b.begin());
   for(int i=0; i<10; ++i)
      std::cout << b[i] << ' ';
   std::cout << std::endl;
   return 0;
}
```