נושאים מתקדמים בתכנות מונחה עצמים

תרגיל/מעבדה 5

פרופ' עפר שיר

ofersh@telhai.ac.il

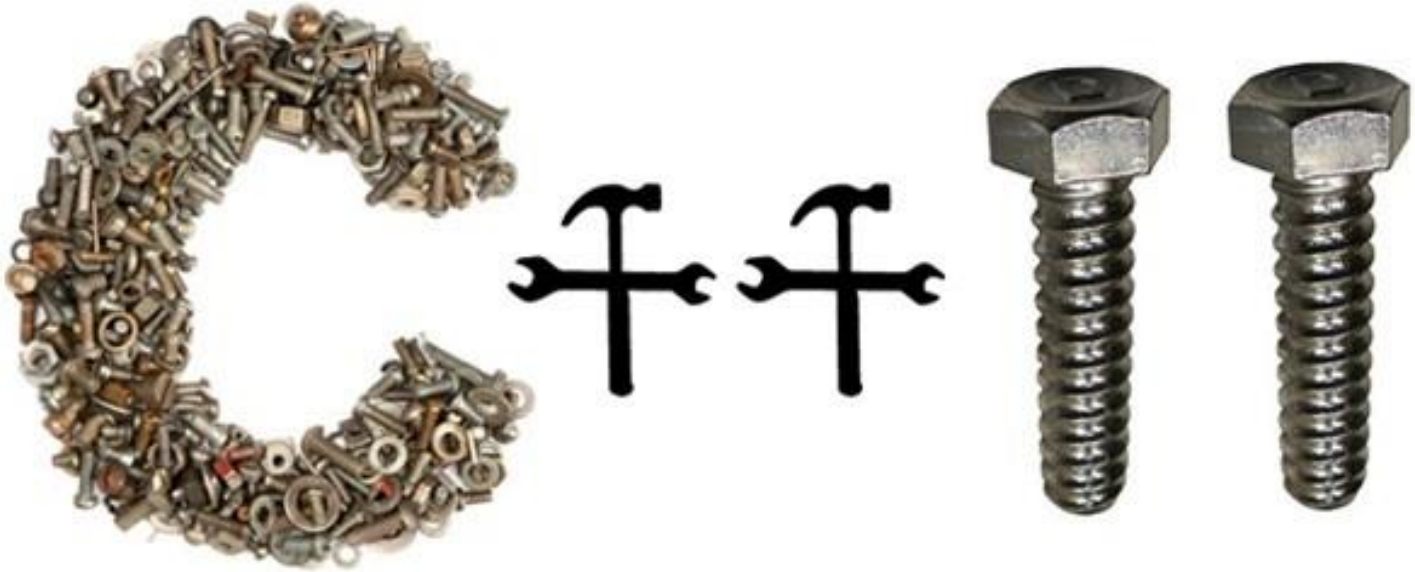
החוג למדעי המחשב

תל-חי

המכללה האקדמית

מטרת התרגיל

- C++0x features and syntax
- C++0x practice



C++11 Features

auto, decltype()

Deducing a Type: **`auto`** and **`decltype()`**

C++11 מספקת שני מנגנונים להסקה אוטומטית של סוג הטיפוס מתוך ביטוי נתון:

- **`auto`** להסקה אוטומטית מתוך איתחול
- **`decltype(expr)`** להסקה שאיננה טריוויאלית, כלומר מתוך מאתחל פשוט – כגון ערך החזרה של פונקציה או טיפוס מופע של מחלקה.

ההסקה פשוטה: המנגנונים הללו פשוט מעבירים את טיפוס הביטוי שכבר ידוע לקומפיילר.

auto

- משתנים המוגדרים באמצעות **auto** מקבלים את הטיפוס של הביטוי המאתחל אותם:

```
auto x1 = 10; // x1: int
std::map<int, std::string> m;
auto i1 = m.begin(); //i1: std::map<int,string>::iterator
```

- ניתן להוסיף אינדיקטורים **const** או מצביע/רפרנס:

```
const auto *x2 = &x1; // x2: const int*
const auto& i2 = m; //i2: const std::map<int,std::string>&
```

- לקבלת **const_iterator**, עשו שימוש בפונקציות האוספים החדשות **cbegin()** , **cend()** , **crbegin()** , **crend()**:

```
auto ci = m.cbegin();
// ci: std::map<int, std::string>::const_iterator
```

auto

עבור משתנים שאינם מוגדרים מפורשות להיות **רפרנס**:

- אינדיקטורים **const** ברמה הגבוהה אינם נחשבים
- מערכים ושמות פונקציה עוברים רדוקציה למצביעים:

```
const std::list<int> li;
auto v1 = li; // v1: std::list<int>
auto& v2 = li; // v2: const std::list<int>&
float data[BufSize];
auto v3 = data; // v3: float*
auto& v4 = data; // v4: float (&)[BufSize]
```

ראו גם את הקוד בשקף הקודם, בו האינדיקטור **const** לא היה ברמה הגבוהה, אלא ברמת ה-**auto**

auto

- ניתן להגדיר מספר משתנים, כל עוד הסקת הטיפוס זהה:

```
void foo(std::string& s) {
  auto temp = s, *pOrig = &s; // temp: std::string,
      . . .                   // pOrig: std::string*
}
auto i = 10, d = 5.0; // ERROR !
```

- התחביר הכפול של איתחול ישיר/העתקה תקף גם כאן:

```
auto v1(expr);
auto v2 = expr;
```

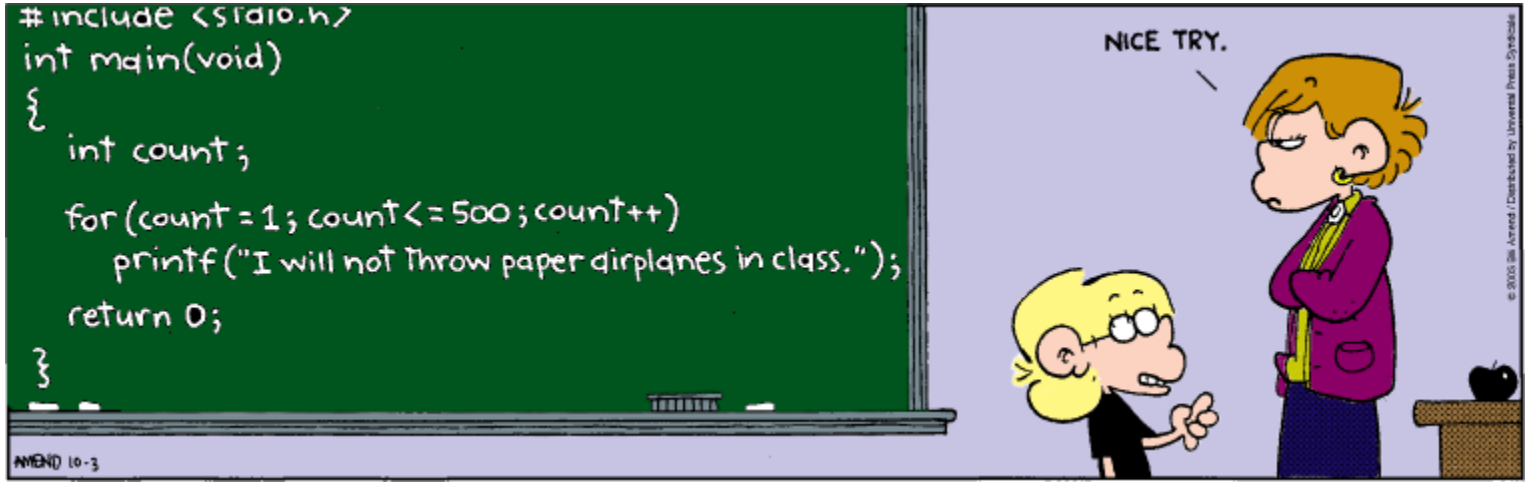
decltype()

- השימוש ב-auto הוא כאשר יש בידינו מאתחל מותאם.
- לעיתים, בפרט ב**תכנות גנרי**, נרצה הסקה של טיפוס הביטוי ללא משתנה מאותחל.
- במקרים כאלו קשה, עד בלתי-אפשרי, לבטא את סוג הטיפוס אשר תלוי, למשל, בפרמטר התבנית.

```
int& foo(int& i);
float foo(float& f);
template <class T>
auto transparent_forwarder(T& t) -> decltype(foo(t))
{ return foo(t); }
```

decltype()

```
const int&& foo();
const int bar();
int i;
struct A { double x; };
const A* a = new A();
decltype(foo()) x1; // type is const int&&
decltype(bar()) x2; // type is int
decltype(i) x3; // type is int
decltype(a->x) x4; // type is double
```



C++11 Features

Range-Based for Loops

C++0x **`for`** Statements in Containers

תחביר חדש וקומפקטי המיועד למעבר על אוספים:

```
for (range_declaration : range_expression)
  loop_statement
```

- `range_declaration`: a named variable, whose type is the type of the element of the sequence, represented by `range_expression`, or a reference to that type.
- `range_expression`: a suitable sequence (with `begin()` and `end()`).

Range-Based **for** Loops

- דוגמה בסיסית למעבר על אוסף מטיפוס ווקטור:

```
std::vector<int> v;
…
for (int i : v) std::cout << i;
// iteratively print every element in v
```

- משתנה האיטרציה יכול להיות גם מטיפוס רפרנס:

```
for (int& i : v) std::cout << ++i;
```

- אינדיקטורים כגון **auto,const** מתקבלים:

```
for (auto i : v) std::cout << i; // same as above
for (auto& i : v) std::cout << ++i; // ditto
```

Relevance

- התחביר תקף בתקן החדש לגבי כל טיפוס נתונים התומך במושג הטווח (*range*) באמצעות **`begin()`**, **`end()`**.
- כלומר, בהינתן אובייקט **`obj`** מטיפוס **`T`**, הביטויים **`begin(obj)`**, **`end(obj)`** תקפים.
- הספריה הסטנדרטית מגדירה ב-**`<iterator>`** פונקציות תבנית גלובליות:

```
template<typename C>
    auto begin(C& c) -> decltype(c.begin());
template<typename C>
    auto begin(const C& c) -> decltype(c.begin());
template<typename C>
    auto end(C& c) -> decltype(c.end());
template<typename C>
    auto end(const C& c) -> decltype(c.end());
```

Implementation

הביטוי

```
for (iterVarDeclaration : expression)
  statementToExecute
```

יהיה שקול ברמת העיקרון למימוש הבא:

```
{
  auto& range = expression;
  for (auto b = begin(range), e = end(range);
  b != e;
  ++b ) {
    iterVarDeclaration = *b;
    statementToExecute
  }
}
```

הערה לסיכום: השפה **איננה** מציעה לולאות מקבילות עבור do או while.

Structures without `begin()` `end()`

```
template<typename T>
Iterator<My_container<T>> begin(My_container<T>& c){
    return Iterator<My_container<T>>{&c[0]};        // iterator to first element
}

template<typename T>
Iterator<My_container<T>> end(My_container<T>& c) {
    return Iterator<My_container<T>>{&c[0]+c.size()}; // iterator to last element
}
```

It is assumed for simplicity that an underlying data-structure `c` is accessible via `c[0]` and `c[0]+size()`; usage of getters is possible. Otherwise, it is also possible to assume something along the lines of accessing `c.head` and `c.tail`.

C++0x Practice