Type Inference in C++ (auto and decltype)

Type Inference refers to automatic deduction of the data type of an expression in a programming language. Before C++ 11, each data type needs to be explicitly declared at compile time, limiting the values of an expression at runtime but after the new version of C++, many keywords are included which allows a programmer to leave the type deduction to the compiler itself.  
With type inference capabilities, we can spend less time having to write out things compiler already knows. As all the types are deduced in compiler phase only, the time for compilation increases slightly but it does not affect run time of the program.

**auto keyword**

The auto keyword specifies that the type of the variable that is being declared will be automatically deducted from its initializer. In case of functions, if their return type is auto then that will be evaluated by return type expression at runtime.

|  |
| --- |
| // C++ program to demonstrate working of auto  // and type inference  #include<bits/stdc++.h>  using namespace std;    int main()  {      auto x = 4;      auto y = 3.37;      auto ptr = &x;      cout <<  typeid(x).name() << endl           <<  typeid(y).name() << endl           << typeid(ptr).name() << endl;        return 0;  } |

Run on IDE

Output :

i

d

Pi

We have used typeid for getting the type of the variables. typeid is an operator which is used where dynamic type of an object need to be known. typeid(x).name() return shorthand name of the data type of x, for example, it return i for integers, d for doubles, Pi for the pointer to integer etc. But actual name returned is mostly compiler dependent. You can read more about typeid [here](http://en.cppreference.com/w/cpp/language/typeid).

A good use of auto is to avoid long initializations when creating iterators for containers.

|  |
| --- |
| // C++ program to demonstrate that we can use auto to  // save time when creating iterators  #include<bits/stdc++.h>  using namespace std;    int main()  {      // Create a set of strings      set<string> st;      st.insert({"geeks", "for", "geeks", "org"});        // 'it' evaluates to iterator to set of string      // type automatically      for (auto it = st.begin(); it != st.end(); it++)          cout << \*it << " ";        return 0;  } |

Run on IDE

Output :

for geeks org

**Note :** auto becomes int type if even an integer reference is assigned to it. To make it reference type, we use auto &.

// function that returns a 'reference to int' type

int& fun() { }

// m will default to int type instead of

// int& type

auto m = fun();

// n will be of int& type because of use of

// extra & with auto keyword

auto& n = fun();

**decltype Keyword**

It inspects the declared type of an entity or the type of an expression. Auto lets you declare a variable with particular type whereas decltype lets you extract the type from the variable so decltype is sort of an operator that evaluates the type of passed expression.  
Explanation of above keyword and their uses is given below

|  |
| --- |
| // C++ program to demonstrate use of decltype  #include<bits/stdc++.h>  using namespace std;    int fun1()   { return 10;  }  char fun2()  { return 'g'; }    int main()  {      // Data type of x is same as return type of fun1()      // and type of y is same as return type of fun2()      decltype(fun1()) x;      decltype(fun2()) y;        cout <<  typeid(x).name() << endl;      cout <<  typeid(y).name() << endl;        return 0;  } |

Run on IDE

Output:

i

c

Below is one more example to demonstrate the use of decltype

|  |
| --- |
| // Another C++ program to demonstrate use of decltype  #include<bits/stdc++.h>  using namespace std;  int main()  {     int x = 5;       // j will be of type int : data type of x     decltype(x) j = x + 5;       cout << typeid(j).name();       return 0;  } |

Run on IDE

Output:

i

c

**A program that demonstrates use of both auto and decltype.**  
Below is a [C++ template](http://geeksquiz.com/templates-cpp/) function min\_type() that returns minimum of two numbers. The two numbers can be of any integral type. The return type is determined using type of minimum of two.

|  |
| --- |
| // C++ program to demonstrate use of decltype in functions  #include <bits/stdc++.h>  using namespace std;    // A generic function which finds minimum of two values  // return type is type of variable which is minimum  template<class A, class B>  auto findMin(A a, B b) -> decltype(a < b ? a : b)  {      return (a < b) ? a : b;  }    // driver function to test various inference  int main()  {      // This call returns 3.44 of doubale type      cout << findMin(4, 3.44) << endl;        // This call returns 3 of int type      cout << findMin(5.4, 3) << endl;        return 0;  } |

Run on IDE

Output :

i3.44

3

Bogotobogo Tutorial:

Type Inference (auto)

In C++03, we must specify the type of an object when we declare it. Now, C++11 lets us declare objects without specifying their types.

auto a = 2; // a is an interger

auto b = 8.7; // b is a double

auto c = a; // c is an integer

Also, the keyword **auto** is very useful for reducing the verbosity of the code. For instance, instead of writing

for (std::vector<int>::const\_iterator it = v.begin(); it != v.end(); ++it)

we can use the shorter version:

for (auto it = v.begin(); it != v.end(); ++it)

#include <vector>

int main()

{

std::vector<int> a, b;

std::vector<int> v{ 1, 2, 3, 4, 5 };

for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)

a.push\_back(\*it);

for (auto it = v.begin(); it != v.end(); ++it)

b.push\_back(\*it);

}

The keyword **decltype** can be used to determine the type of an expression at compile-time. For example:

#include <vector>

int main()

{

const std::vector<int> v(1);

auto a = v[0]; // a has type int

decltype(v[1]) b = 1; // b has type const int&, the return type of

// std::vector<int>::operator[](size\_type) const

auto c = 0; // c has type int

auto d = c; // d has type int

decltype(c) e; // e has type int, the type of the entity named by c

decltype((c)) f = c; // f has type int&, because (c) is an lvalue

decltype(0) g; // g has type int, because 0 is an rvalue

}

Note that the type denoted by **decltype** can be different from the type deduced by **auto**.

Range-based for loop

C++11 extends the syntax of the for statement to allow for easy iteration over a range of elements.

This form of for will iterate over each element in the list. It will work for C-style arrays, initializer lists, and any type that has begin() and end() functions defined for it that return iterators.

All of the standard library containers that have begin/end pairs will work with the range-based for statement.

#include <iostream>

#include <vector>

int main()

{

std::vector<int> a, b;

std::vector<int> v{ 1, 2, 3, 4, 5 };

// C++03

for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)

a.push\_back(\*it);

// C++11

for (auto it = v.begin(); it != v.end(); ++it)

b.push\_back(\*it);

for (int item : a)

std::cout << item << " "; // read only access

std::cout << std::endl;

for (auto &item; : b) {

item \*= 10;

std::cout << item << " "; // read only access

}

}

Output:

1 2 3 4 5

10 20 30 40 50