## Automatic Type Deduction: decltype

Let's take a look at what decltype does.

#### WE'LL COVER THE FOLLOWING ^

- decltype vs auto
- Rules

# decltype vs auto

The decltype keyword was also introduced in C++11, though its functionality differs from auto. decltype is used to determine the type of an expression or entity.

Here is the correct format:

### decltype(expression)

We can use auto to create variables, but decltype returns the type of an expression containing variables.

### Rules #

- If the expression is an *lvalue*, decltype will return a reference to the data type to the expression
- If the expression is an *rvalue*, decltype will return the data type of the value

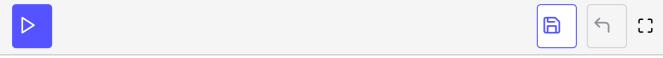
```
#include <iostream>
int main() {
  int i = 1998; // Rvalue
  decltype(i) i2 = 2011; // Same as int i2 = 2011

decltype((i)) iRef = i2; // (i) is an lvalue, reference returned
```

```
std::cout << "iRef: " << iRef << std::endl;
std::cout << "i2: " << i2 << std::endl;

std::cout << std::endl;

iRef = 2012;
std::cout << "iRef: " << iRef << std::endl;
std::cout << "i2: " << i2 << std::endl; // iRef is a reference after all
}</pre>
```



In line 7, the parentheses around i indicate that this is an expression instead of a variable. Hence, decltype computes int& instead of int.

decltype is not used as often as auto. It is useful with templates that can deduce the type of a function.

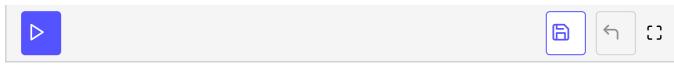
Here's another example of decltype in action:

```
#include <iostream>
#include <vector>
int func(int, int){ return 0; }
int main(){
                                              // int
 decltype(5) i = 5;
 int& intRef = i;
                                                   // int&
 decltype(intRef) intRefD = intRef;
                                             // int&
                                              // int*
 int* intPoint = &i;
 decltype(intPoint) intPointD = intPoint;
                                              // int*
 const int constInt = i;
                                              // const int
 decltype(constInt) constIntD = constInt;  // const int
 static int staticInt = 10;
                                             // static int
 decltype(staticInt) staticIntD = staticInt; // static int
 const std::vector<int> myVec;
 decltype(myVec) vecD = myVec;
                                       // const std::vector<int>
 auto myFunc = func;
                                              // (int)(*)(int, int)
 decltype(myFunc) myFuncD = myFunc;
                                             // (int)(*)(int, int)
 // define a function pointer
 int (*myAdd1)(int, int) = [](int a, int b){ return a + b; };
 // use type inference of the C++11 compiler
 decltype(myAdd1) myAdd2 = [](int a, int b){ return a + b; };
  std::cout << "\n";</pre>
```

```
// use the function pointer
std::cout << "myAdd1(1, 2) = " << myAdd1(1, 2) << std::endl;

// use the 2 variable
std::cout << "myAdd2(1, 2) = " << myAdd2(1, 2) << std::endl;

std::cout << "\n";
}</pre>
```



We can see how decltype deduces the types of different entities, including the function pointer in line 32.

In the next lesson, we'll learn how to use decltype and auto together.