# Type Inference

# AUTO AND DECLTYPE

The keywords *auto* and *decltype* are alternate methods for type deduction.

- Abstraction:  abstracting the type. This can make the code more readable.
- Reduced redundancy:  removing redundancy improves code quality (DRY).
- Easy refactoring: without redundancy, the code is more easily refactored and flexible.

# AUTO

- The use of *auto* limited to declaring variables.
- The variable type is not late binding but early binding, implied, and static.
- Difficult to write some code without *auto*, such as templates and undocumented types.

```
std::vector<vector<int>> *v = new std::vector<vector<int>>();
```

or

```
auto myvector = new std::vector<vector<int>>();
```

# AUTO LIMITATIONS

Cannot use *auto* in the following context:

- Function parameters
- Member variables
- static types

# AUTO EXAMPLE

The following example demonstrates the *auto* keyword to declare a vector and than an iterator.  Do you recall the syntax to declare an iterator for a vector of integers?  Auto saves the day and deduces the correct type.

```
int main() {
    auto myvector = vector<int>();
    auto it = myvector.begin();
    myvector.insert(it, 200);

    return 0;
};
```

# DECLTYPE

- The **decltype** keyword is more general purpose than **auto**.
  - It is not limited to capturing the type specification of variables.
  - **decltype** can deduce the type of expressions without evaluating the expression.
  - The result of **decltype** can be used as a substitute for a type name.

```
int main() {
  const int size = 5;
  decltype(size) resize = 10; // resize set to const int
  resize = 20; // Error: expression not a modifiable lvalue
};
```

# TEMPLATED EXAMPLE

- Demonstrates a trailing return type.
- Defer establishing the return type until after parameters are stated.
- The return type can then be derived from the parameter types.

```
template<typename L, typename R>
auto multiply(L l, R r)->decltype(l*r)
{
	return l*r;
}
```