More on types in C++

- useful additional types/features we can make use of:
- **static** local variables: maintain their values across calls
- **typedefs**: allow us to assign a name to a new data type
- **enums**: allow us to create named sets of integers or characters
- **auto**: gets the compiler to automatically determine the correct data type to use in a variable declaration
- **references**: allow us to create an alias for a given data item

Static local variables

- local variables in functions are created and initialized 'fresh' with each call to the function
- by preceeding a variable declaration with the keyword static it gets moved to an external storage space, is initialized once, and maintains updates to its value across function calls

```
void foo(int x)
{
   int i = 1;
   cout << i;
   i++;
}
// every call to foo starts with a new copy of i,
// always prints out 1
```

```
void foo( )
{
   static int i = 1;
   cout << i;
   i++;
}
// in first call i is 1, prints 1
// in second call i is 2, prints 2
// in third call i is 3, etc
```

typedefs

- typedefs allow us to create and use a name for a data type
  - can use the name when declaring variables, parameters, or constants, instead of repeating the full type specification (can be handy for complex types)
  - can improve readability by using the type name to clarify the purpose of variables, constants, parameters
- syntax is generally
  - typedef *theactualtype ourchosenname ;*

typedefs for readability

- suppose we need to store/use temperatures in our program, represented as floats
- we can create a new type, named temperature

```
typedef float temperature;
```

- we can then create variables of that type, e.g.

```
temperature x, y, z;
```

- they can be used just like any other floats, but the type name makes their purpose a bit clearer

typedefs for multi-dim arrays

- suppose we have a two-dim array where R and C specify the number of rows and columns
- we can create a new type name for it, e.g. Table:

```
typedef float Table[R][C];
```

- we can now declare two dimensional arrays of this size using table, e.g.

```
Table t;
```

- we can then use the new 2d array normally, e.g.

```
t[x][y] = 12;
```

enumerated types: enum

- used to create named subsets of integers or characters
- give a name for the subset, and a name to each individual value

```
enum WeekDays { Mon=1, Tue=2, Wed=3, Thu=4, Fri=5 };
```

- the name of the subset can be used as a type

```
WeekDays today, anotherDay;
```

- the names for the individual values can be used like constants

```
if (today == Mon) {
   cout << “I do not like working on Mondays”;
}
```

references (&)

- references can create an alias for an existing data item

```
int x = 10;
int &foo = x;  // foo can be used as an alias for x
```

  - references must be initialized at point of declaration
  - safer than pointers since cannot be wild or null
  - easier to use than pointers since no symbol needed to dereference
- compiler implements using memory addresses/pointers
- used when we want a second named access to something (as in pass by reference) without actually fully replicating the item

auto (compiler determines type)

- we can use auto instead of an explicit type when declaring variables, the compiler infers correct type to use based on the initialization of the variable

```
auto x = foo(something); // determines type of x from the return type of foo
```

- **upside**: easier, can be more readable than repeating some complex type specifications
- **downside**: can obscure what's happening when the reader (e.g. someone maintaining/debugging the code) can't easily see what the variable's type will really be