CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 11
October 5, 2016

References

Reference types

Recall: Given `int x`, two types are associated with `x`: an L-value (the reference to `x`) and an R-value (the type of its values).

C++ exposes this distinction through *reference* types and declarators.

A *reference type* is any type `T` followed by `&`, i.e., `T&`.

A reference type is the internal type of an L-value.

Example: Given `int x`, the name `x` is bound to an L-value of type `int&`, whereas the values stored in `x` have type `int`

This generalizes to arbitrary types `T`: If an L-value stores values of type `T`, then the type of the L-value is `T&`.

Reference declarators

The syntax `T&` can be used to declare names, but its meaning is not what one might expect.

```
int x = 3;      // Ordinary int variable
int& y = x;     // y is an alias for x
y = 4;          // Now x == 4.
```

The declaration must include an initializer.

The meaning of `int& y = x;` is that `y` becomes a name for the L-value `x`.

Since `x` is simply the name of an L-value, the effect is to make `y` an alias for `x`.

For this to work, the L-value type (`int&`) of `x` must match the type declarator (`int&`) for `y`, as above.

Use of named references

Named references can be used just like any other variable.

One application is to give names to otherwise unnamed objects.

```
int axis[101];              // values along a graph axis
int& first = axis[0]   ;    // give name to first element
int& last = axis[100];      // give name to last element
first = -50;
last = 50;

// use p to scan through the array
int* p;
for (p=&first; p!=&last; p++) {...}
```

Reference parameters

References are mainly useful for function parameters and return values.

When used to declare a function parameter, they provide call-by-reference semantics.

```
int f( int& x ){...}
```

Within the body of `f`, `x` is an alias for the actual parameter, which must be the L-value of an `int` location.

Reference return values

Functions can also return references.

```
int& g( bool flag, int& x, int& y ) {
    if (flag) return x;
    return y;
}
...
g(x<y, x, y) = x + y;
```

This code returns a reference to the smaller of `x` and `y` and then sets that variable to their sum.

Custom subscripting

Suppose you would like to use 1-based arrays instead of C++'s 0-based arrays.

We can define our own subscript function so that `sub(a, k)` returns the L-value of array element `a[k-1]`.

`sub(a,k)` can be used on either the left or right side of an assignment statement, just like the built-in subscript operator.

```
int& sub(int a[], int k) { return a[k-1]; }
...
int mytab[20];
for (k=1; k<=20; k++)
    sub(mytab, k) = k;
```

Constant references

Constant reference types allow the naming of pure R-values.

```
const double& pi = 3.14159265358979323846264338327950;
```

Actually, this is little different from

```
const double pi = 3.14159265358979323846264338327950;
```

In both cases, the pure R-value is placed in a read-only object, and `pi` is bound to its L-value.

Comparison of reference and pointer

- A reference (L-value) is the result of following a pointer.
- A pointer is only followed when explicitly requested (by `*` or `->`).
- A reference name is bound when it is created. Pointer objects can be initialized at any time (unless declared to be read-only using `const`).
- Once a reference is bound to an object, it cannot be changed to refer to another object. Pointer objects can be changed to point to another object at any time using assignment (unless declared to be read-only).
- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.

Concept summary

Concept	Meaning
Object	A block of memory and its contents.
L-value	The machine address of an object.
R-value	The value stored in an object.
Pointer value	An R-value consisting of a machine address.
Pointer object	An object into which a pointer value can be stored.
Identifier	A name in a program which is bound to an L-value.

Type summary

Let `T` be any type.

Concept	Type	Meaning
Object	`T`	L-value has type `T&`, R-value has type `T`.
L-value	`T&`	The object at its address has type `T`.
R-value	`T`	The type of the data value is `T`.
Pointer object	`T*`	L-value has type `T*&`, R-value has type `T*`.
L-value of ptr obj	`T*&`	The object at its address has type `T*`.
Pointer R-value	`T*`	The type of the data value is `T*`.

Declaration syntax

`T x;`	Binds `x` to the L-value of a new object of type `T`.
`T& x=y;`	Binds `x` to the L-value of `y` which has type `T&`.
`T* x = new T;`	Binds `x` to the L-value of a new object of type `T*` and initializes its value with a pointer to a new dynamically-allocated object of type `T`.
`T* y;`	Binds `y` to a new uninitialized object of type `T*`.

Storing a list of objects in a data member

A common problem is to store a list of objects of some type `T` as a data member `li` in a class `MyClass`.

Here are six ways it can be done:

1. `T li[100];`	`li` is *composed* in `MyClass`.
2. `T* li[100];`	`li` is *composed* in `MyClass`. Constructor does loop to store `new T` in each array slot.
3. `T* li;`	Constructor does `li = new T[100];`.
4. `T** li;`	Constructor does `li = new T*[100];` then does loop to store `new T` in each array slot.
5. `vector<T> li;`	Uses Standard `vector` class. `T` must be copiable.
6. `vector<T*> li;`	Constructor does loop to store `new T` into each vector slot.

How to access

Here’s how to acces element 3 in each case:

1.	`T li[100];`	`li[3]`.
2.	`T* li[100];`	`*li[3]`.
3.	`T* li;`	`li[3]`.
4.	`T** li;`	`*li[3]`.
5.	`vector<T> li;`	`li[3]`.
6.	`vector<T*> li;`	`*li[3]`.