CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 7
September 19, 2018

Reference Types (cont.)

Reference Types (cont.)

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.

A review of definitions

- An **object** is a block of memory into which data can be stored along with a type.
- The **type** of an object tells the storage size and interpretation of its contents.
- The **R-value** of an object is the sequence of bytes stored in it.
- The **L-value** of an object is a unique label for the object. It is often represented by a machine address.
- A **reference** is an L-value along with its type.
- An object might or might not have a **name**. If it does, the name is **bound** to a reference.

LHS and RHS contexts

- The meaning of a name or reference depends on the context in which it appears.
- The right hand side of an assignment statement is said to be **RHS context**. A name appearing there evaluates to the R-value of the object that it references.
- The left hand side of an assignment statement is said to be **LHS context**. A name appearing there evaluates to the L-value of the object that it references.

Example

`int x = 3` creates an object on the stack of type `int`, stores the number 3 in it, and gives it the name "`x`".

Let `0x1234` be the address of the newly-created object `x`.

- The L-value of `x` is `0x1234`;
- The R-value of `x` is `3`;
- `x` itself names the reference `(0x1234, int)`.

In the expression `y = x+1`, the name `x` appears in RHS context.
Its R-value, `3`, is fetched from `x` and used by the `+` operator.
The name `y` appears in LHS context.
Its L-value is where the result of `x+1` is stored.

Pointers

A **pointer** is a special kind of R-value that embeds a reference.

The prefix operator `*`, applied to a pointer, returns the reference embedded in the pointer. This operation is called **following the pointer**.

A pointer that embeds a reference of type `T` is said to have type `T*`.

If `x` is a reference of type `T`, then the prefix operator `&` can be applied to `x` to produce a pointer to `x`.

The type of `&x` is `T*`. Thus, `*&x` is an alias for `x`.

Pointer objects

- A **pointer object** of type `T*` is an object that can store pointers of type `T*` as its R-values.
- The star operator `*p` applied to a pointer object `p` first fetches the R-value of `p` which is a pointer. It then follows that pointer and returns its embedded reference.
- This returned reference can be used like any other object. For example, if `p` has type `int*`, then `(*p) = 17` stores 17 into the reference returned by `*p`, which will have type `int`.

Examples Presented in Class

Several examples were presented in class on the blackboard.

Hand-drawn pictures used boxes to represent objects, hex numbers to represent L-values, numbers inside boxes to represent primitive R-values, and arrows starting inside one box and pointing to another to represent pointers.

Anyone who missed class is encouraged to borrow class notes from someone who attended.

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 `const`).
- Once a reference is bound to an object, it cannot be changed to refer to another object. Pointer objects can be be assigned a different pointer at any time (unless declared to be `const`).
- A reference is always associated with a fixed piece of storage. By way of contrast, a pointer object can contain the special value `nullptr`, which is a special pointer that can be compared for equality but not be followed.

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	An R-value consisting of a machine address.
Pointer object	An object into which a pointer can be stored.
Reference	A typed L-value.
Identifier	A name which is bound to a reference.

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 pointer object `x` of type `T*`, creates a dynamically-allocated object of type `T`, and stores a pointer to it in `x`.
`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]`.