Object-Oriented Principles and Practice / C++

Alice E. Fischer

April 20, 2015

A Pointer is not an Object

C++ provides two ways to create objects:

- An objects can be create on the run-time stack by declaring it in any block of code.
  - **ClassName varName(parameters);**
  - Stack objects are freed when the declaring-block ends.
  - If the declaration is in a loop, an object will be created and freed each time around the loop.
- An object can be allocated dynamically.
  - **ClassName* t = new ClassName(parameters)**;
  - A pointer to the new object is stored in the pointer variable `t`.
  - This object will exist until we execute **delete t**

Arrays of Objects and Non-Objects

- An array of objects can be created, either dynamically or on the stack.
  - **ClassName varName[length];**
  - **ClassName* t = new ClassName[length];**
  - The array will be initialized by the `default` constructor.
  - There is no way to pass parameters to the default constructor.
- An array of object-pointers can be created, either dynamically or on the stack.
  - **ClassName* varName[length];**
  - **ClassName** t = new (ClassName*)[length];**
  - When we do this, we expect to create objects later, using `new`, and plug the pointers into the empty array slots.
- In both cases, **delete[]** is used to free the arrays.

One Function, Multiple Methods

Suppose we wish to construct a Date.

- There are potentially many ways to make a Date.
  - We might want the the input to be (Mon, day, yr)
  - We might want a simpler format: (Month, day)
  - We might want the user to enter a string: ("April 15, 2010")
  - We might want today's date. (No parameters needed).
- We could define multiple Date constructors:
  - `Date( int month, int day, int year);`
  - `Date( int month, int day);`
  - `Date( char* MDY);`
  - `Date()`
- All OK, since the four methods have different parameter lists.

One Method, Multiple Prototypes

C++ gives us a shortcut: optional parameters with default values.
Chapter 7, page 23

- Only one method is needed to create the first three prototypes:
  Date(int month, int day, int year=2010);
- Default parameters must always be on the right end of the list.
- If three arguments are given in the the constructor call, they will override the default values.
- If two arguments are given, the year will default to 2010.
- We could keep this combined method and the last two individual methods.
- Or we could use some slightly tricky coding to eliminate the last two cases also.

An enum constant is not an int

A C++ enum type is a real type. Enum constants are not synonyms for ints.

- `enum dayName { UNK, MON, TUE, WED, THR, FRI, SAT, SUN };`
- These are named constants, not strings.
- It is often a good idea to include a constant in your enum to represent "unknown".
- You can create enum-type variables and expressions: `dayName today = MON;`
- You often want a parallel array of strings for output": `"Unknown", "Monday", "Tuesday" ...`

Values, References, and Pointers

Simple Variables, L-values and R-values

Pointers

References

R-value References

L-values and R-values

Programming language designers have long been bothered by the asymmetry of assignment.

`x = 3` is a legal assignment statement.

`3 = x` is not legal.

Expressions are treated differently depending on whether they appear on the left or right sides of an assignment statement.

Something that can appear on the left is called an *L-value*.

Something that can appear on the right is called an *R-value*.

Intuitively, an L-value is the address of a storage location – some place where a value can be stored.

An R-value is a thing that can be placed in a storage location. R-values are sometimes called *pure data values*.

Simple variable declaration

The declaration `int x = 3;` says several things:

1. All values that can be stored in `x` have type `int`.
2. The name `x` is *bound* (when the code is executed) to a storage location adequate to store an `int`.
3. The `int` value 3 is initially placed in `x`'s storage location.

The L-value of `x` is the address of the storage location of `x`.

The R-value of `x` is the object of type `int` that is stored in `x`.

Simple assignment

The assignment statement `x = 3;` means the following:

1. Get an R-value from the right hand side (`3`).
2. Get an L-value from the left hand side (`x`).
3. Execute the current definition of = using the L-value and the R-value.
4. If `operator=` has not been redefined, this action puts the R value from step 1 into the storage location whose address was obtained from step 2.

Automatic dereferencing

Given

```
int x = 3;
int y = 4;
```

Consider

```
x = y;
```

This is processed as before, except what does it mean to get an R-value from `x`?

Whenever an L-value is presented and an R-value is needed, *automatic deferencing* occurs.

This means to go the storage location specified by the presented L-value (`&y`) and get its R-value (`4`). Then the assignment takes place as before.

Pointer values

- A pointer is a primitive object that denotes a machine address. It is frequently diagrammed as an arrow.
- The pointer itself is an R-value, which can be stored in a pointer variable.
- The referent of a pointer is the object stored at that machine address.
- The type of a pointer is the type of its referent, followed by $*$ Example: If `y` is a simple integer variable, then the type of a pointer to `y` is `int*`
- We say the pointer *references* `y` or *points at* `y` .

Following a pointer

To *follow* a pointer means to obtain the L-value it points at.

- The basic operator for following a pointer is unary `*`.
- We say that `E` points to `*E`.
- `*` is the inverse of `&`. It takes a R-value-pointer and returns the L-value of its referent.
- Some expressions evaluate to R-value-pointer results. If `E` is a such an expression, then `*E` is the L-value encapsulated by the pointer that results from evaluating `E`.
- If `E` has type `T*`, then the values stored in `*E` have type `T`.

Pointer creation

- Pointers (R-values) are created by applying the unary operator & to an object that has an L-value.
- Example: If `y` has type `int`, then the expression `&y` is a pointer of type `int*` that references `y`.
- More generally, if `x` has type `T`, then the expression `&x` yields a pointer (R-value) of type `T*` that references `x`.
- In C and C++, the name of an array, without [], is translated as a reference to the first slot in the array.
- In C and C++, the name of a function, without (), is translated as a reference to the entry point of the function. Parentheses are required to denote a function call.

Pointer variables

Variables into which pointers can be stored are called (not surprisingly) *pointer variables*.

A pointer variable is no different from any other variable except for the types of values that can be stored in it.

- `int* q` declares `q` to be a variable into which pointers of type `int*` can be stored.
- If `x` is an integer variable, then the assignment `q = &x` creates a pointer to `x` and stores it in `q`.

Just as we often conflate "integer" and "integer variable", it is easy to confuse "pointer" with "pointer variable". Try to use these words clearly!

Pointer assignment

Pointers can be assigned to pointer variables.

- If `p` and `q` are pointer variables of the same type, then `p = q;` is an assignment statement.
- It has the same interpretation as any other assignment, i.e., fetch the (pointer) value from `q` and store it in `p`.
- Example: Suppose we execute `q = &x;` `p = q;`
  In the second assignment, the L-value `q` is dereferenced to an R-value, which is then placed in `p`.
- Note: The automatic dereference of `q` results in the R-value pointer stored in `q`. It does NOT follow that pointer to get `x`.
- Now `p` contains a copy of the pointer stored in `q`, and both of these pointers reference `x`.

Pointer example

```
int x = 3;
int y = 4;
int* p;
int* q;
int* r;

p = &x;          // p points to x.
*p = 5;          // Now x==5.
q = p;           // p and q both point to x.
*q = *p + 1;     // Now x==6.
```

Common mistake – dangling pointer

```
*r = x+y;        // What's wrong here?
```

Pointer declaration syntax

A word of warning

`int x, y;` is shorthand for `int x; int y;` but

`int* p, q;` is not same as `int* p, int* q`.

Rather, it means `int* p; int q;`.

For this reason, many authors put the `*` next to the variable instead of with the name.

My approach is different: declare one variable per line, with an appropriate comment.

Spacing around the star doesn't matter, but logically it belongs with the type.

Reference types

Recall: Given `int x`, two types are associated with `x`: an L-value (its machine address) and an R-value (the number stored there).

C++ exposes the difference through reference types and variables.

A *reference* is the address of an object.

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

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;    // Must have an initializer.
y = 4;         // Now x == 4.
```

The meaning of `int& y = x;` is that `y` becomes another name for the L-value `x`. Since `x` names an L-value, `y` names the same L-value.

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 storage locations.

```
int ages[10];              // ages of members of a family
int& father = ages[0];     // give name to first element
int& mother = ages[1];     // give name to second element
int& kids = ages[2];       // this family has 8 kids.
int& last = ages[9];       // last kid in family

// use p to scan through the kid-part of the array
int* p;
for (p=&kids; p!=&last; p++) {...}
```

R-value References

An *r-value reference* type is written with two ampersands: `myType&&`.

This provides an additional type in the family pure value, variable, reference, pointer that is useful for defining efficient processing in class templates.

It opens up the possibility of defining different semantics for moving and for copying and object.

R-value references allow programmers to avoid logically unnecessary copying. They are primarily meant to aid in the design of higher performance and more robust libraries. We will return to this subject in a few weeks.