CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 6
September 21, 2010

Functions and Methods (continued)

How should one choose the parameter type?

Parameters are used for two main purposes:

- To send data to a function.
- To receive data from a function.

Sending data to a function: call by value

For sending data to a function, call by value copies the data whereas call by pointer or reference copies only an address.

If the data object is large, call by value is expensive of both time and space and should be avoided.

If the data object is small (eg., an `int` or `double`), call by value is cheaper since it avoids the indirection of a reference.

Call by value protects the caller's data from being inadvertantly changed.

Sending data to a function: call by reference or pointer

Call by reference or pointer allows the caller's data to be changed. Use `const` to protect the caller's data from inadvertane change.

Ex: `int f( const int& x )` or `int g( const int* xp )`.

Prefer call by reference to call by pointer for input parameters.

Ex: `f( 234 )` works but `g( &234 )` does not.

Reason: 234 is not a variable and hence can not be the target of a pointer.
(The reason `f( 234 )` *does* work is a bit subtle and will be explained later.)

Receiving data from a function

An output parameter is expected to be changed by the function.

Both call by reference and call by value work.

Call by reference is generally preferred since it avoids the need for the caller to place an ampersand in front of the output variable.

Declaration: `int f( int& x )` or `int g( int* xp )`.

Call: `f( result )` or `g( &result )`.

The implicit argument

Every call to a class function has an *implicit argument*, which is the name written before the function call.

```
class MyExample {
private:
   int count;    // data member
public:
   void advance(int n) { count += n; }
   ...
};
...
MyExample ex;
ex.advance(3);
```

Increments `count` by 3.

`this`

The implicit argument is passed as a pointer.

In the call `ex.advance(3)`, the implicit argument is `ex`, and a pointer to `ex` is passed to `advance()`.

The implicit argument can be referenced directly from within a member function using the keyword `this`.

Within the definition of `advance()`, `count` and `this->count` are synonymous.

Simple variables

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. The 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 L-value from the left hand side (`x`).
2. Get an R-value from the right hand side (`3`).
3. Put the R value from step 2 into the storage location whose address was obtained from step 1.

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 `y`?

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 fetching its contents (`4`).

Then the assignment takes place as before.

Pointers

Pointer values

- A **pointer** is a primitive object with an associated L-value.
- The pointer itself is an R-value.
- The **type of a pointer** is the type of the value that can be stored at the associated L-value, 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`.

Pointer creation

- Pointers are created by applying the unary operator `&` to 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`.

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 `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".

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: If `q = &x` as before, then after `p = q`, `p` contains a copy of the pointer stored in `q`. Both of these pointers reference the address of `x`.
- Note that the L-value `q` is dereferenced to an R-value, which is then placed in `p`.
- Dereferencing does not follow the pointer.

Following a pointer

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

- The basic operator for following a pointer is unary `*`.
- `*` is the inverse of `&`. It takes a pointer and returns its corresponding L-value.
- If E is a pointer 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`.
- We say that E *points to* `*`E.

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 type name.

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

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 storage locations.

```
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 variable, 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 variables 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 variables can be changed to point to another object at any time using assignment (unless declared to be readonly).
- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.