CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 5
September 16, 2010

IO Demos

Handling data errors and end of file

Section 3.7 of the textbook contains a demo program that illustrates how to handle data errors and end of file using C++ I/O. It has three parts that illustrate

- How to use `get()` to read text lines from a file.
- How to use `getline()` to do the same thing.
- How to read numbers from a file.

See 05-IOdemo.

How to write a test program

The 05-IOdemo was written in C-style using three global functions: `use_get()`, `use_getline()`, and `use_nums()`.

I rewrote the demo

- to eliminate the use of underscores in multipart names;
- to illustrate the use of C++ classes as lexical containers for gathering and isolating related code.

Here, each test is encapsulated within its own class.

The only responsibility of `main()` is to process the command line arguments and initiate the tests.

See 05-IOdemo-new.

Introduction to Classes

(Textbook, Chapter 4)

Classes, visibility, functions, inline

We covered much of the material from sections 4.1 and 4.2 in lectures 2 and 3.

The textbook covers it in greater depth, so be sure to also read the book.

Stack example

Section 4.3 presents a non-trivial object-oriented design for a bracket-matching program using a stack.

As with the insertion sort demo, it is written twice, once in C and once in C++.

Both versions have two modules: one for token and one for stack.

Note how `struct` in the C code becomes `class` in the C++ version.

Note also how the functions `analyze()` and `mismatch()` moved from `main.c` to a new class `Brackets` in the C++ version.

Functions and Methods

Call by value

Like C, C++ passes explicit parameters by value.

```
void f( int y ) { ... y=4; ... };
...
int x=3;
f(x);
```

- `x` and `y` are independent variables.
- `y` is created when `f` is called and destroyed when it returns.
- At the call, the *value* of `x` (3) is used to initialize `y`.
- The assignment `y=4;` inside of `f` has no effect on `x`.

Call by pointer

Like C, pointer values (which I call *reference values*) are the things that can be stored in *pointer variables*.
Also like C, pointers can be passed as arguments to functions with corresponding pointer parameters.

```
void g( int* p ) { ... (*p)=4; ... };
...
int x=3;
g(&x);
```

- `p` is created when `g` is called and destroyed when it returns.
- At the call, the *value* of `&x` (which is a reference value) is used to initialize `p`.
- The assignment `(*p)=4;` inside of `g` changes the value of `x`.

Call by reference

C++ has a new kind of parameter called a *reference* parameter.

```
void g( int& p ) { ... p=4; ... };
...
int x=3;
g(x);
```

- This does same thing as previous example; namely, `p=4;` changes the value of `x`.
- Within the body of `g`, `p` is a synonym for `x`.

I/O uses reference parameters

- The first argument to `<<` has type `ostream&`.
- `cout << x << y;` is same as `(cout << x) << y;`.
- `<<` returns a reference to its first argument, so this is also the same as

```
cout << x;
cout << y;
```

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.)