CPSC 427: Object-Oriented Programming

Michael J. Fischer

Lecture 7
September 21, 2016

Brackets Example (continued)

Storage Management

Brackets Example (continued)

Brackets class

1. Data member `stk` is dynamically allocated in the constructor and deleted in the destructor. It is an object, not an array, and does *not* use the `[]`-forms of `new` and `delete`.
2. The type of `stk` has changed from `Stack*` to `Stack`. We can now print the stack by writing `cout << stk`. Formerly, we wrote `stk->print(cout)`.
3. `in.get(ch)` reads the next character without skipping whitespace. There are other ways to do this as well.
4. If read is `!in.good()`, we `break` from the loop and do further tests to find the cause.
5. Old functions `analyze()` and `mismatch()` have been replaced by `checkFile()` and `checkChar()`. This largely separates the file I/O from the bracket-checking logic.

Brackets design questions

- What are the pros and cons of `stk` having type `Stack&` rather than `Stack*`?
- The old `mismatch()` uses the `eofile` argument to distinguish two different cases.

```
void Brackets::
mismatch( const char* msg, Token tok, bool eofile ) {
  if (eofile) cout <<"\nMismatch at end of file: " <<msg <<endl;
  else        cout <<"\nMismatch on line " <<lineno <<" : " <<msg <<endl;

  stk->print( cout );   // print stack contents
  if (!eofile)          // print current token, if any
      cout <<"The current mismatching bracket is " << tok;
  fatal("\n");          // Call exit.
}
```

Is this a good design?

Main file

1. `main()` follows our usual pattern, except that it passes `argc` and `argv` on to the function `run()`, which handles the command line arguments.
2. `run()` opens the input file and passes the stream `in` to `analyze()`.
3. The istream `in` will not be closed if an error is thrown (except for the automatic cleanup that happens when a program exits). How might we fix the program?
4. Question: Which is better, to pass the file name or an open stream? Why?

Storage Management

Objects and storage

Objects have several properties:

- A **name**. This is one way to access the object.
- A **type**. This determines the size and encoding of the allowable **data values**.
- A **storage block**. This is a block of memory big enough to hold any legal value of the specified type.
- A **lifetime**. This is the time span between an object's creation and its demise. Data left behind in an object's storage block after it has died is unpredictable and shouldn't be used.
- A **storage class**. This determines the lifetime of the object, where the storage block is located in memory, and how it is managed.

Name

An object may have one or more names, or none at all!

Not all names are created equal. A name may exist but not be visible in all contexts.

- It is not visible from outside of the block in which it is defined.
- For a class data member, the name's visibility may be restricted, e.g., by the `private` keyword.
- An object may have more than one name. This is called **aliasing**.
- An object may have no name at all. Such an object is called **anonymous**. It can only be accessed via a pointer or subscript.

Type of a storage object

Declaration: `int n = 123;`

This declares an object of type `int`, name `n`, and an `int`-sized storage block, which will be initialized to 123. It's lifetime begins when the declaration is executed and ends on exit from the enclosing block. The storage class is `auto` (stack).

The unary operator `sizeof` returns the storage size (in bytes).

`sizeof` can take either an expression or a parentheses-enclosed type name, e.g., `sizeof n` or `sizeof(int)`.

In case of an expression, the size of the result type is returned, e.g., `sizeof (n+2.5)` returns 8, which is the size of a `double` on my machine.

Storage block

Every object is represented by a block of storage in memory.

This memory has an internal **machine address**, which is not normally visible to the programmer.

The size of the storage block is determined by the type of the object.

Connecting names to objects

A name can be given to an anonymous object at a later time by using a **reference** type.

```
#include <iostream>
using namespace std;
int main() {
  int* p;
  p = new int;  // Creates an anonymous int object
  *p = 3;       // Store 3 into the anonymous object
  cout << *p << endl;
  int& x = *p;  // Give object *p the name x
  x = 4;
  cout << *p << " " << x << endl;
}
/* Output
3
4 4
*/
```

Lifetime

Each object has a **lifetime**.

The lifetime begins when the object is **created** or **allocated**.

The lifetime ends when the object is **deleted** or **deallocated**.

Storage class

C++ supports three different storage classes.

1. `auto` objects are created by variable and parameter declarations. (This is the default.)
   Their visibility and lifetime is restricted to the block in which they are declared.
   The are deleted when control finally exits the block (as opposed to temporarily leaving via a function call).
2. `new` creates anonymous *dynamic* objects. They exist until explicitly destroyed by `delete` or the program terminates.
3. `static` objects are created and initialized at load time and exist until the program terminates.

Dynamic extensions

Recall that objects have a fixed size determined solely by the object type.

A variable-sized "object" is modeled in C++ by an object with a **dynamic extension**. This object has a pointer (or reference) to a dynamically allocated object (generally an array) of the desired size.

Example from stack.hpp.

```
class Stack {
private:
  int max = INIT_DEPTH; // Number of slots in stack.
  int top = 0;          // Stack cursor.
  T* s = new T[max];    // Pointer to stack base.
  string name;          // Print name of this stack.
  ...
```

Copying

A source object can be copied to a target object *of the same type*.

A **shallow copy** copies each source data member to the corresponding target data member. By default, this is done by performing a byte-wise copy of the source object's storage block to the target object's storage block, overwriting its previous contents.

For objects with dynamic extensions, the *pointer* to the extension gets copied, not the extension itself. This causes the target to end up sharing the extension with the source, and the target's previous extension becomes **inaccessible**. This results in **aliasing**—multiple pointers referring to the same object, which can cause a **memory leak**.

A **deep copy** recursively copying the extensions as well.

The double-delete problem

An object with dynamic extension typically uses `new` in the constructor and `delete` in the destructor to create and free the object.

When a shallow copy results in two objects sharing the same extension, then attempts will be made to delete the extension when each of the two copies of the object are deleted or go out of scope.

The first delete will succeed; the second will fail since the same object cannot be deleted twice.

This is called the **double delete** problem and is a major source of memory management errors in C++.

Takeaway: Don't copy objects with dynamic extensions.

When does copying occur?

C++ has two operators defined by default that make copies:

1. The assignment statement.
2. The copy constructor.

The symbol **=** means assignment when used in a **statement**, and it invokes the copy constructor when used in an **initializer**.
All-by-value argument passing also uses the copy constructor.

Assignment modifies an existing object;

The copy constructor initializes a newly-allocated object.

Assignment

The **assignment** operator `=` is implicitly defined for all types. The assignment `b=a` modifies an already-existing object `b` as follows:

- If `a` and `b` are primitive types, the storage object `a` is copied to the storage object `b` (after performing any implicit conversions such as converting a `short int` to an `int`). In the case of pointer types, this results in `a` and `b` pointing to the same block of memory.
- If `a` and `b` are objects, then each data member of `a` is recursively assigned to the corresponding data member of `b`, using the assignment operator defined for the data member's type.

Copy constructor

The **copy constructor** is implicitly defined for all types. Like any constructor, it can be used to initialize a newly-allocated object.

- Call-by-value uses the copy constructor to initialize a function parameter from the actual argument.
- The copy constructor can also be used to initialize a newly-created object.

Since the copy constructor uses shallow copy, any use of it on an object with dynamic extension leads to the double delete problem.

If you don't intend to use the copy constructor, you can disable it by writing `T(const T&) =delete;` in class `T`'s definition.

Redefining assignment and the copy constructor

You can redefine assignment for a class `T` by defining the function with signature `T& operator=(const T&);`.

You can redefine the copy constructor by defining the function with signature `T(const T&)`.

To get the implicit definitions (if they've been deleted and you want them), use `=default`. To cancel them, use `=delete`.