CPSC 427a: Object-Oriented Programming

Michael J. Fischer

Lecture 8
September 27, 2011

Storage management

Variables and storage

An ordinary variable consists of three parts:
A **type**, a **name** and a **storage register**.

- The type determines the size and encoding of the storage register.
- The name is used to access the storage register.
- The storage register is a machine register long enough to hold any of the legal values of the specified type.

Example of a variable

Declaration: `int n = 123;`

This declares a variable of type `int`, name `n`, and an `int`-sized storage register, which will be initialized to 123.

The `sizeof` operator returns the size of its operand (in bytes).

The operand can be an expression or a type name in parentheses, 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, the size of a `double` on my machine.

Properties of variables

Not all variables are created equal.

The name may not be visible in all contexts.

- It is not visible from outside of the block in which it is defined.
- If a data member in a class, the name's visibility may be restricted by the `private` keyword.

Each storage register has a **lifetime** – the interval of time between the **creation** or **allocation** of the variable, and the **deletion** or **deallocation** of the variable.

A variable can also be **anonymous**, in which case it has no name and can only be accessed via a pointer or subscript. The notion of lifetime still applies.

Storage classes

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. `static` objects are created and initialized at load time and exist until program termination.
3. `new` creates anonymous *dynamic* objects. They exist until explicitly destroyed by `delete` or the program terminates.

Assignment and copying

The assignment operator `=` is implicitly defined for all types.

- `b=a` does a shallow copy from `a` to `b`.
- Shallow copy on objects means to copy all data members from one object to the other.
- Call-by-value uses the copy constructor to copy the actual argument to the function parameter.
- If the argument object contains pointer data members, the pointers are copied but *not* the objects they point to. This results in **aliasing**—multiple pointers to the same object.

Static data members

A static class variable must be *declared* and *defined*.

- A static class member is declared by preceding the member declaration by the qualifier `static`.
- A static class member is defined by having it appear in global context *with* an initializer but *without* `static`.
- Must be defined *only once*.

Example

In `mypack.hpp` file, inside class definition:

```
class MyPack {
  static int instances; // count # instantiations
```

In `mypack.cpp` file:

```
int MyPack::instances = 0;
```

Static function members

Function members can also be declared `static`.

- As with static variables, the are declared inside class by prefixing `static`.
- They may be defined either inside the class (as inline functions) or outside the class.
- If defined outside the class, the `::` prefix must be used and the word `static` omitted.

Five common kinds of failures

1. **Memory leak**—Dynamic storage that is no longer accessible but has not been deallocated.
2. **Amnesia**—Storage values that mysteriously disapper.
3. **Bus error**—Program crashes because of an attempt to access non-existant memory.
4. **Segmentation fault**—Program crashes because of an attempt to access memory not allocated to your process.
5. **Waiting for eternity**—Program is in a permanent wait state or an infinite loop.

Read the textbook for examples of how these happen and what to do about them.

Bells and whistles

Optional parameters

The same name can be used to name several different member functions if the *signatures* (types and/or number of parameters) are diffent. This is called overloading.

Optional parameters are a shorthand way to declare overloading.

Example

```
int myfun( double x, int n=1 ) { ...  }
```

This in effect declares and defines two methods:

```
int myfun( double x ) {int n=1; ...}
int myfun( double x, int n ) {...}
```

The body of the definition of both is the same.
If called with one argument, the second parameter is set to 1.

const

`const` declares a variable (L-value) to be readonly.

```
const int x;
int y;
const int* p;
int* q;

p = &x;  // okay
p = &y;  // okay
q = &x;  // not okay -- discards const
q = &y;  // okay
```

const implicit argument

`const` should be used for member functions that do not change data members.

```
class MyPack {
private:
  int count;
public:
  // a get function
  int getCount() const { return count; }
...
};
```

Operator extensions

Operators are shorthand for functions.

Example: `<=` refers to the function `operator <=()`.

Operators can be overloaded just like functions.

```
class MyObj {
  int count;
  ...
  bool operator <=( MyObj& other ) const {
      return count <= other.count; }
};
```

Now can write `if (a <= b) ...` where `a` and `b` are of type `MyObj`.

Classes

What is a class?

- A collection of things that belong together.
- A struct with associated functions.
- A way to encapsulate behavior: public interface, private implementation.
- A way to protect data integrity, providing world with functions that provide a read-only view of the data.
- A data type from which objects (instances) can be formed. We say the instances belong to the class.
- A way to organize and automate allocation, initialization, and deallocation of storage.
- A way to break a complex problem into manageable, semi-independent pieces, each with a defined interface.
- A reusable module.