Advanced C++

Ron Akkersdijk

Come along. Saxion.

Autumn 2014

Overview

- Internal and External Linkage
- Callbacks & Function/Method Pointers
- The new & delete actions
- Operator overloading
- Datatype convertions
- Template Classes
- Template Functions
- Generic Algoritms
- ...

new & delete

- The **new** action allocates <u>space</u> for a new object and then calls the <u>*constructor(s)*</u>
- The **delete** action calls the <u>*destructor(s)*</u> and then frees the allocate <u>space</u>
- The ordinary `new` may give a **std::bad_alloc** exception when you run out of space
- The same applies to **new[]** and **delete[]**

new & delete

- Getting a very big buffer:

```
// Get the largest possible buffer
char* getBigBuffer(unsigned max) {
   char* bp = 0; // result pointer
   for (unsigned n = max ; bp == 0 && n > 0 ; n /= 2)
   {
      try {
         bp = new char[n];                 // try this size
      } catch( std::bad_alloc& e ) { // it was to much
         ;    // this problem was expected, ignore it
      }
   }
   if (bp == 0)                   // still 0 ?
      throw std::bad_alloc();   // give up
   return bp;
}
```

new & delete

- If it is inconvenient to handle exceptions somewhere, you can use:

```
Object* op = new(std::nothrow) Object;
```

- Which returns a 0 pointer instead

```
// Get the largest possible buffer
char* getBigBuffer(unsigned max) {
   char* bp = 0; // result pointer
   for (unsigned n = max ; bp == 0 && n > 0 ; n /= 2)
      bp = new(std::nothrow) char[n];
   if (bp == 0)
      throw std::bad_alloc();  // give up
   return bp;
}
```

placement new

- The *placement new* variant only calls the *constructor*, with the given address as *this*

```
// get some space for an object ...
void* where = new char[sizeof(Object)];
// ... and turn it into an Object
Object* op = new(where) Object(…);
```

- In this case the program determines the location of the new object in memory
- Since we did not use new we may not use delete. Instead we simply call the destructor!

```
op->~Object(); // !! special case !!
```

- e.g. used for RamDisk & ObjectCaches

An object cache

```
Object* ospace = new Object[100]; // Get some space
vector<Object*> ocache;             // For bookkeeping
for (unsigned i=0;i<100;++i)        // Register the …
   ocache.push_back(&ospace[i]);    // … free objects

Object* oalloc() {                // Get a free Object:
   Object* xp = ocache.back();// Take the last one and
   ocache.pop_back();             // update the cache
   return xp;
}
// usage: Object* op = new(oalloc()) Object(...);


void ofree(Object* xp) {          // Delete an Object:
   xp->~Object();                 // Destroy it and put
   ocache.push_back(xp);          // it back into cache
}
```

Questions?

Operator overloading

- If an expresion like `a*b` has no standard meaning in the language C++ itself, then you can give that operation a meaning, provided it involves at least one *user defined* datatype
- Some operators can <u>not</u> be overloaded, e.g. `?:` `::` `.*` `.->` and `.`
- Some <u>must</u> be class members e.g. *assign* `=`, *dereference* `*`, *member* `->`, *index* `[]`, *call* `()`
- Advice: If the operator changes an object it should be a class member to prevent confusion
- Advice: The new meaning should match existing expressions to prevent confusion

Operator overloading

- Tricky operators:
  - The `,` operator as in: `int a;  a = 4+2 , 3*3 ;`
    Overloading discards *left→right* evaluation order
  - It is also bad practice to overload `&&` and `||`
    - Normally they have *left→right* *short-cut* behaviour, i.e. evaluation stops when the result becomes predictable
    - Overloading loses this behaviour!
- Unexpected operators:
  - `sizeof(…)` and `typeid(…)`   (done by the compiler!)
  - `new`, `new[]`, `delete` and `delete[]`   (next slide)

new & delete operators

- The new keyword uses the new operator to obtain space. The delete keyword uses the delete operator to release space. Same for new[] and delete[]
- These can be overloaded *per class* or *globally*

```
#include <cstdlib>    // malloc, free from C

class SomeClass { // AND derived classes!
public:
   // Note: the 'static's are implied!!
   static  void*  operator new(size_t n) {
      return malloc(n);
   }
   static  void   operator delete(void* xp) {
      free(xp);
   }
};
```

new & delete revisited

- The behaviour for new can now be described as ...

```
// Original source code: Fred* p = new Fred();
Fred* p;                          // For the result
// Get space for a new object
void* tmp = operator new(sizeof(Fred));
// A constructor could throw some exception!
try {
   new(tmp) Fred();               // Placement new
   p = (Fred*) tmp;               // Assigned only on succes
} catch (...) {                   // Construction aborted
   operator delete(tmp);          // Deallocate the memory
   throw;                         // Re-throw the exception
}
```

User defined helpers

- Many *infix* operators follow the same pattern:

```
class Pet {
   friend bool operator ==(const Pet&,const Pet&);
   string     name;
   int        age;
       …
};

bool operator ==(const Pet& p1, const Pet& p2) {
   return p1.name == p2.name; // disregard age
}
```

- Such non-member operators are called *helpers*

Compiler generated

- The C++ compiler will generate some operators on demand, unless already defined by you

```
class Pet {
      …
   Pet& operator =(const Pet& p);
   // copies all attributes
};

bool operator ==(const Pet& p1, const Pet& p2);
bool operator !=(const Pet& p1, const Pet& p2);
// compares all attributes using && or ||
```

- Note: Writing your own version ussually only makes sense if the class has pointers or special demands

Output operator

- The STL *ostream* classes use **<<** for output

```
cout << some_object;
```

- Because the object to be printed occurs on the right hand side, the **<<** can not be a class member
- To gain access to private/protected data the operator should be declared a **`friend`**
- To ensure we print the original and not some copy of it, we should use *call by reference*

```
ostream& operator <<(ostream& os, const Class& o);
```

Output operator

```
#include <ostream>    // for: std::ostream
#include <string>     // for: std::string
class Pet {
   std::string name;
   int         age;
   …
   friend std::ostream&
            operator <<(std::ostream&, const Pet&);
};

std::ostream&
     operator<<(std::ostream& os, const Pet& p)
{
   // output: name age
   return os << p.name << ' ' << p.age; // no endl!
}
```

Input operator

- The STL *ostream* classes use >> for input

```
cin >> some_object;
```

- Because the object to be printed occurs on the right hand side, **>>** can not be a class member
- To gain access to private/protected data the operator should be declared a **`friend`**
- To ensure we alter the given object and not a local copy of it, it should *call by reference* without const!

```
istream& operator >>(istream& is, Class& o);
```

Input operator

```
#include <istream>    // for: std::istream
#include <string>     // for: std::string
class Pet {
   std::string name;
   int         age;
   …
   friend std::istream&
            operator >>(std::istream&, Pet&);
};

std::istream&
     operator>>(std::istream& is, Pet& p)
{
   // input: name age (assume name without spaces!)
   return is >> p.name >> p.age;
}
```

Iterators

- An *iterator* is a class which has the * and -> operators defined, making it behave like a pointer

```
class vector<Pet>::iterator {
   Pet* where; // Where we are now in the array
   ...  // Note: The details depend on the container!
public:
   Pet&  operator* () const  { return *where; }
   Pet*  operator->() const  { return  where; }
   ...  // and others like ++, --, ==, !=, etc
};
```

- For a const_iterator add 'const' to the result
- The matching container class should provide suitable `begin()` and `end()` methods

Increment/Decrement

- The `++` and `--` operators change an object and therefore *should* be class members (but this is not mandatory!)
- They exist in two flavors:
  - The prefix `++x` returns the NEW value

```
class& class::operator++();
```

  - The postfix `x++` returns the OLD value:

```
class& class::operator++(int dummy); // always 0
```

Increment/Decrement

```
class vector<Pet>::iterator {
   Pet* where; // Where we are now in the array
public:
   iterator&  operator++() {          // prefix: ++x
      ++where;
      return *this;
   }
   iterator   operator++(int) {       // postfix: x++
      iterator  old(where); // a local to save old value
      ++where;
      return old;            // the copy of the old value …
   }
};
```

- Note: The postfix version is always more expensive!

Index operator

- To make an object behave like an *array*, define the [] operator(s) as *class members*
- The operator has 1 argument, the index (which can have any type)
- Common practice is to define two of them

```
const Type& class::operator[](… index) const;   // rhs
      Type& class::operator[](… index);         // lhs
```

- The compiler knows which to use when:

```
    object[index]   =  object[index];
    // lhs             // rhs (const)
```

- Used by containers like:

    std::string, std::vector and std::map

Call operator

- To make an object behave like a *function*, define the () operator(s) as *class members*

```
class Crazy {
public:
   int operator () (int a, int b, int c) {
      return a+b+c;
   }
};


Crazy   crazy;              // a crazy object
int x = crazy(1,2,3);    // calling crazy?
// which does: x = crazy.operator()(1,2,3);
```

- Note: You can have multiple *call* operators provided their signatures differ

Type operator

- To extract a value of a desired type from an object define a *TYPE* operator as class member

```
class Double {
   double dval;
public:
   operator int () const { return int(dval); }
   operator string () const;
};
// mimic java toString()
Double::operator string () const {
   stringstream  ss;
   ss << dval;
   return ss.str();
}
```

- Note: The return type is implied here!

Questions?

Datatype conversions

- If the arguments for an operation have different datatypes, the compiler tries to unify their types when needed:

```
short  s;
double d;
long   l = s * d; // (short × double) => long
```

  - The short value is promoted to double
  - The double result is converted to long
- When *user-defined types* are involved the compiler tries to do the same (unless there already exists an appropriate *operator*)
- Note: It must be possible to do the needed conversion in a single step!

Datatype conversions

- 1: A constructor which only needs one argument

```
class Pet {
   Pet(const char* name, int age=0);
};
Pet p = "felix";      // Pet p("felix",0);
```

  - Note: The datatype of “felix” is const char[]

- If this behaviour is unwanted, make the constructor explicit

```
class Pet {
   explicit Pet(const char* name, ...);
};
Pet c("felix");   // oke, explicit usage
Pet d = "fido";   // error, implied usage
```

Datatype conversions

- 2: Using a *type operator* for the needed type

```
class Pet {
   string  name;
public:
   // mimics java toString()
   operator string() const  { return name; }
};
Pet c("felix");
string  s = c; // does: s = c.operator string();
```

- Note: In C++11 these can also be made explicit

Datatype conversions

- Java autoboxing: int <=> Integer

```
class Integer {
   int    value;
public:
   // Conversion: int => Integer
   Integer(int x=0) : value(x) {}
   // Conversion: Integer => int
   operator int() const  { return value; }
};                                          tijdelijk object!
Integer i; i = 8; // i = Integer(8);
int j; j = i + 7; // j = i.operator int() + 7
```

- Note: typedef Integer int; is a lot cheaper

Questions?