C++:
THE GOOD, BAD, AND UGLY

6.172 Fall 2010 Reid Kleckner

The Language

- Originally “C with classes”, now much more.
- Intended to be a superset of C
- Has many new features, all add complexity
- Useful for writing fast, generic code
- Can become extremely verbose
- Only covering the most widely used features

Printing, C++ Style

- C++ supports operator overloading
- Best example: standard i/o streaming library
- Overloads << operator to mean "write this object to the stream"
- std::cout and std::cerr are streams for stdout/stderr
- Say goodbye to printf format specifiers =D

```
#include <iostream>
int main(int argc, char **argv) {
  std::cout << "Hello, World!\n";
  std::cout << "argc: " << argc << '\n';
  std::cout << "argv[0]: " << argv[0] << '\n';
}
```

Classes and Structs

- Can have private members, like Foo::a and Bar::d
- Classes default to private, structs to public, otherwise equivalent
- No need for typedef as in C
- See trailing semi-colon

```
class Foo {
  int a;
 public:
  int b;
};
struct Bar {
  Foo f;
  int c;
 private:
  int d;
};
int main(void) {
  Bar bar;
  bar.c = 1;
  bar.f.b = 2;
  // invalid:
  bar.d = 3;
  bar.f.a = 3;
}
```

New and Delete

- new operator allocates memory and calls ctor
- delete operator calls dtor and frees memory
- Always use new instead of malloc, or object will be uninitialized

```
struct Foo {
  int a_;
  Foo(int a);
  ~Foo();
};
Foo::Foo(int a) {
  printf("a: %d\n", a);
  this->a_ = a;
}
Foo::~Foo() {
  printf("destructor\n");
}
int main(void) {
  Foo *f = new Foo(5);
  delete f;
}
```

Constructors and Destructors

- Foo::Foo(int a) is a constructor (ctor) for Foo
- Foo::~Foo() is destructor (dtor)
- “this” is a pointer to the object (like Java)
- Prints:
  - a: 5
  - destructor

```
struct Foo {
  int a_;
  Foo(int a);
  ~Foo();
};
Foo::Foo(int a) {
  printf("a: %d\n", a);
  this->a_ = a;
}
Foo::~Foo() {
  printf("destructor\n");
}
int main(void) {
  Foo *f = new Foo(5);
  delete f;
}
```

Constructors and Destructors

- Ctors should initialize all member variables
- Dtors should clean up any resources owned by the object
- In this case, Str "owns" buf, so it deletes it
- If no ctor is declared, compiler generates implicit default ctor with no initialization!

```
struct Str {
  int len;
  char *buf;
  Str(int l, char *b);
  ~Str();
};
Str::Str(int l, char *b) {
  len = l;
  buf = b;
}
Str::~Str() {
  delete buf;
}
```

Methods

- Methods are defined similarly to constructors
- Methods are called using -> and .
- "member function" is another name for method

```
struct Foo {
  int thing_;
  void setThing(int thing);
  int getThing();
};
void Foo::setThing(int thing) {
  thing_ = thing;
}
int Foo::getThing() {
  return thing_;
}
int main(void) {
  Foo *f = new Foo();
  f->setThing(20);
  printf("thing: %d\n",
         f->getThing());
  delete f;
}
```

Header Files

- Class definitions live in header (.h) files
- Method definitions live in source files (.cpp, .cc)
- If class Foo is in Foo.h and Foo.cpp, #include “Foo.h” to call Foo’s methods
- C++ headers are large, so use header guards!

```
// Foo.h
#ifndef FOO_H
#define FOO_H
struct Foo {
  int thing_;
  void setThing(int thing);
  int getThing();
};
#endif // FOO_H
// Foo.cpp
void Foo::setThing(int thing)
{ thing_ = thing; }
int Foo::getThing()
{ return thing_; }
// Bar.cpp
#include "Foo.h”
int main(void) {
  Foo *f = new Foo();
  f->setThing(20);
  printf("thing: %d\n",
         f->getThing());
  delete f;
}
```

Inline Methods

- Function calls are too expensive for just get/set
- Compiler cannot inline across modules
- Solution: move definitions into header file
- Use for short routines, especially ctors/dtors

```
// Foo.h
struct Foo {
  int thing_;
  void setThing(int thing)
   { thing_ = thing; }
  int getThing() {
    return thing_; }
};
// Bar.cpp
#include "Foo.h”
int main(void) {
  Foo *f = new Foo();
  f->setThing(20);
  printf("thing: %d\n",
         f->getThing());
  delete f;
}
```

Virtual Methods

- Uses dynamic dispatch and indirect function call
- Subclasses can override virtual methods
- Java: default is virtual
- C++: default is final
- Virtual methods are slower and cannot be inlined
- Perf numbers:
  - inline: 8ms
  - direct: 68ms
  - virtual: 160ms
- Use when writing base classes

Virtual Methods

- "= 0" means "pure virtual", aka abstract
- A and B inherit from Base
- Output is:
  - A
  - B

```
#include <stdio.h>
struct Base {
  void virtual printName() = 0;
};
struct A : public Base {
  void virtual printName() {
    printf("A\n"); }
};
struct B : public Base {
  void virtual printName() {
    printf("B\n"); }
};
int main(void) {
  Base *p = new A();
  p->printName();
  p = new B();
  p->printName();
}
```

References

- Reference vs. pointers:
  - int& a = b;
  - int* a = &b;
- References are like pointers, except:
  - Must always be initialized where declared
  - Cannot be reassigned
  - Use . instead of -> to access fields and methods
  - Never need to use * to dereference, compiler will "do the right thing"
  - Cannot take address of reference variable, you get the address of the referenced object

References: Simple Example

- p and r point to a
- Prints:
  - 0 0 0
  - 1 1 1
  - 2 2 2
- Can convert from pointer to reference with *
- Can convert from reference to pointer with &

```
#include <stdio.h>
int main(void) {
  int a = 0;
  int *p = &a;
  int &r = a;
  printf("%d %d %d\n”,
         a, *p, r);
  *p = 1;
  printf("%d %d %d\n",
         a, *p, r);
  r = 2;
  printf("%d %d %d\n",
         a, *p, r);
  // Conversion
  int *p2 = &r;
  int &r2 = *p;
}
```

References: Swap Example

- ref_swap automatically takes addresses of args
- In both cases, a and b are modified in place
- Assembly is *identical*
- Output:
  - 2 1
  - 1 2

```
#include <stdio.h>
void ptr_swap(int *a, int *b) {
  int c = *a;
  *a = *b;
  *b = c;
}
void ref_swap(int &a, int &b) {
  int c = a;
  a = b;
  b = c;
}
int main(void) {
  int a = 1, b = 2;
  ptr_swap(&a, &b);
  printf("%d %d\n", a, b);
  ref_swap(a, b);
  printf("%d %d\n", a, b);
}
```

Const

- Const does *not* mean immutable
- A const reference or pointer means “I promise not to modify this data through this pointer”
- However, someone else may change the data
- Can also have pointers whose value does not change, like cant_reseat

```
const char *str;
char * const cant_reseat = NULL;
bool isLowerCase() {
  for (int i = 0; i < 26; i++)
    if (str[i] < 'a' || str[i] > 'z')
      return false;
  return true;
}
int main(void) {
  char buf[26];
  str = buf; // Note buf is not const
  // cant_reseat = buf; // illegal
  for (int i = 0; i < 26; i++) {
    buf[i] = 'A' + i;
  }
  // Prints 0
  std::cout << isLowerCase() << '\n';
  for (int i = 0; i < 26; i++) {
    buf[i] += 'a' - 'A';
  }
  // Prints 1
  std::cout << isLowerCase() << '\n';
}
```

Stack vs. Heap Allocation

- new is used to allocate on the heap
- Simply declaring a stack variable calls the default constructor
- Can call other constructors by "calling" the variable

```
#include <iostream>
struct Foo {
  int a_;
  Foo() {
    a_ = 0;
    std::cout << "default ctor\n";
  }
  Foo(int a) {
    a_ = a;
    std::cout << "a: " << a << '\n';
  }
  ~Foo() {
    std::cout << "dtor a: " << a_ << '\n';
  }
};
int main(void) {
  Foo a; // default
  Foo b(3); // other
}
```

Stack vs. Heap Allocation

- Destructors are called in reverse order of construction
- Program prints:
  - default ctor
  - a: 1
  - dtor a: 1
  - dtor a: 0

```
#include <iostream>
struct Foo {
  int a_;
  Foo() {
    a_ = 0;
    std::cout << "default ctor\n";
  }
  Foo(int a) {
    a_ = a;
    std::cout << "a: " << a << '\n';
  }
  ~Foo() {
    std::cout << "dtor a: " << a_ << '\n';
  }
};
int main(void) {
  Foo f0; // default
  Foo f1(1); // other
}
```

Resource Allocation is Initialization

- Want to allocate a resource (lock, memory, file or socket) on entry, release on exit
- Accomplished in C with gotos and booleans
- In C++, exceptions make this harder
- Insight: destructors for stack allocated variables are *always* calledwhen exiting a scope
- Works when leaving via return, exceptions, break, continue, goto, or normal flow
- Idea: write lightweight class to manage the resource

RAII: Mutexes

- Consider a shared FIFO queue
- Both push and pop have error conditions
- lock_guard is an RAII-style class that calls lock when created, and unlock when destroyed
- Unlocks even if we return early

```
#include <vector>
#include "cilk_mutex.h"
#include "lock_guard.h"
struct Queue {
  std::vector<int> data_;
  cilk::mutex lock_;
  void push(int e);
  int pop();
};
void Queue::push(int e) {
  cilk::lock_guard<cilk::mutex> guard(lock_);
  if (data_.size() > 100)
    return;  // Still unlocks
  data_.push_back(e);
}
int Queue::pop() {
  cilk::lock_guard<cilk::mutex> guard(lock_);
  if (data_.size() == 0)
    return -1;  // Still unlocks
  int t = data_.front();
  data_.erase(data_.begin());
  return t;
}
```

Pass by Value

- Can pass objects “by value”
- Allocates new stack memory
- Calls copy constructor passing original
- Copy ctor for Foo would be:
  - Foo::Foo(const Foo &f) {…}
- See this frequently for std::string and std::vector, objects are < 24 bytes

```
#include <iostream>
#include <string>
std::string getstr() {
  std::string s("Hello, World!");
  return s;
}
void println(std::string s) {
  std::cout << s << '\n';
}
int main(void) {
  std::string s = getstr();
  println(s);
}
```

Templates

- Templates are like Java generics (sort of)
- Templates are "instantiated" at compile time
- Two versions of my_min generated, one for strings and one for ints
- Very efficient! No virtual calls
- Prints:
  - 4
  - book

```cpp
template <typename T>
T my_min(T l, T r) {
  return (l < r) ? l : r;
}
int main(void) {
  std::cout << my_min(10, 4) << '\n';
  std::string a("staple");
  std::string b("book");
  std::cout << my_min(a, b) << '\n';
}
```

STL

- The Standard Template Library (STL) provides many useful generic containers:
  - std::vector<T> : resizeable array
  - std::deque<T> : double-ended queue
  - std::map<T> : red-black tree map
  - std::set<T> : red-black tree set
- Similar to java.util.* data structures

Vectors

- Similar to ArrayList in Java
- Dynamically resizeable array
- Subscript operator overloaded to support array-style indexing

```
#include <string>
#include <vector>
#include <iostream>
int main(void) {
  std::vector<int> nums;
  for (int i = 0; i < 10; i++) {
    nums.push_back(i);
  }
  int sum = 0;
  for (int i = 0; i < nums.size(); i++) {
    sum += nums[i];
  }
  std::cout << "sum 0-9: " << sum << '\n';

  std::vector<std::string> strs;
  strs.push_back("Lorem");
  strs.push_back("Ipsum");
  for (int i = 0; i < strs.size(); i++) {
    std::cout << strs[i] << " ";
  }
  std::cout << '\n';
}
```

STL Iterators

- Similar to Java iterators
- Uses operator overloading to match pointer iteration
- No special foreach loop in C++ ☹
- Can become verbose
- At –O3, generates same assembly as pointer version
- Much more efficient than Java iterators, which involve 2 virtual calls on each iteration

```
int main(void) {
  std::vector<int> nums;
  for (int i = 0; i < 10; i++)
    nums.push_back(i);

  int sum = 0;
  for (std::vector<int>::iterator
       i = nums.begin(),
       e = nums.end();
       i != e;
       ++i) {
    sum += *i;
  }
  std::cout << sum << '\n';

  // equivalent (for vectors) to:
  int *i, *e;
  for (i = &nums[0],
       e = &nums[nums.size()];
       i != e;
       ++i) {
    sum += *i;
  }
  std::cout << sum << '\n';
}
```

Namespaces

- Avoids name collisions between libraries
- Example: use mm namespace instead of mm_ prefix
- Can access mm namespace with mm::
- Starting :: means root namespace
- Needed to call libc malloc instead of mm::malloc

```
// mm.h
namespace mm {
  void *malloc(size_t size);
  void free(void *ptr);
};
// mm.c
namespace mm {
  void *malloc(size_t size) {
    return ::malloc(size);
  }
}
void mm::free(void *ptr) {
  ::free(ptr);
}
// app.c
int main(void) {
  void *ptr = mm::malloc(10);
  mm::free(ptr);
  ptr = malloc(10);
  free(ptr);
}
```

Namespaces

- Can import names
- "using namespace std" makes all names in std available
- "using std::vector" redeclares vector in the global namespace of this file
- Alternatively, just use std:: always

```
// app1.c
using namespace std;
int main(void) {
  vector<int> nums;
}
// app2.c
using std::vector;
int main(void) {
  vector<int> nums;
}
// app3.c
int main(void) {
  std::vector<int> nums;
}
```

Conclusion

- That's it!
- C++ is a large and complex language
- These are the most widely used bits
- Skipped exceptions, RTTI, multiple inheritance, and template specialization
- Check out cplusplus.com/reference/ for C/C++ library reference

6.172 Performance Engineering of Software Systems
Fall 2010