Object Oriented Programming in C++

2501ICT/7421ICT Nathan

René Hexel and Joel Fenwick

School of Information and Communication Technology
Griffith University

Semester 1, 2012
Outline

1. Subclasses, Access Control, and Class Methods
   - Subclasses and Access Control
   - Class Methods

2. Advanced Topics
   - Introduction to C++ Memory Management
   - Strings
C++ Subclasses
Subclasses in C++

- Like in Java and Objective-C, classes can extend other classes
  - `class AClass: public SuperClass`
  - `no single root class like NSObject`
- `this`
  - references the current object
- There is no `super` keyword in C++
  - the parent class needs to be referenced by name for method invocations
  - constructors needs to specify an initialisation list if superclass constructors need to be invoked
Creating Subclasses: **Point3D**

**Objective-C Child:** Point3D.h

```objective-c
#import "Point.h"

@interface Point3D : Point
{
    int z; // add z dimension
}
- init; // constructor

- (void) setZ: (int) newz;
- (void) setX: (int) newx
    y: (int) newy
    z: (int) newz;
@end
```

**C++ Child:** Point3D.h

```cpp
#include "Point.h"

class Point3D : public Point
{
protected: // needed in C++
    int z; // add z dimension

public:
    Point3D(); // constructor

    void setZ(int newz);
    void setXYZ(int nx, int ny, int nz);
};
```
Subclass Implementation: Point3D

**Objective-C: Point3D.m**

```c
#import "Point3D.h"

@implementation Point3D

- init // initialiser
{
    if ([super init])
        z = 0;
    return self;
}

- (void) setZ: (int) newz
{
    z = newz;
}

- (void) setXYZ: (int) nx, (int) ny, (int) nz
{
    [super setX: nx y: ny];
    [self setZ: nz];
}
@end
```

**C++: Point3D.m**

```c
#include "Point3D.h"

// constructor with initialisation list
Point3D::Point3D(): Point()
{
    z = 0;
}

void Point3D::setZ(int newz)
{
    z = newz;
}

void Point3D::setXYZ(int nx, int ny, int nz)
{
    setXY(nx, ny); // in super class
    // the following does the same:
    // Point::setXY(nx, ny);

    setZ(nz); // in Point3D
}
```
Access Control in C++


**Access Control**

- **public:**
  - everyone has access

- **private:**
  - nobody has access, except the defining class
  - default for C++ classes
  - useful for variables that should not be accessed by subclasses

- **protected:**
  - only the defining class and subclasses have access
  - useful for most member variables

- **In C++, public, private, and protected apply to methods as well as member variables**
Subclasses, Access Control, and Class Methods

Advanced Topics

Access Control Example

Objective-C Access Control

```objective-c
#import <Foundation/Foundation.h>

@interface MyClass: MySuperClass
{
    @public // public vars
    int a;
    int b;

    @private // private vars
    int c;
    int d;

    @protected // protected vars
    int e;
    int f;
}

- init; // constructor

@end
```

C++ Access Control

```cpp
// MyClass with access control

class MyClass: public MySuperClass
{
    public:
        int a;
        int b;

    private:
        int c;
        int d;

    protected:
        int e;
        int f;

    public: // public methods
        MyClass(); // constructor

    // ... other class methods

};
```

René Hexel and Joel Fenwick

Object Oriented Programming in C++
Which `printf` is wrong?

Example (Which of the following lines will cause a compiler error?)

```cpp
class ClassX
{
    public    int x;
    private   int y;
    protected int z;
};

class ClassY: public ClassX
{
    void print();  // a print method
};

// implementation of ClassY:

void ClassY::print()
{
    printf("x = %d\n", x);  // print x
    printf("y = %d\n", y);  // print y
    printf("z = %d\n", z);  // print z
}
```
Class Methods in C++
C++ also supports Class Methods

- method that can be invoked without an instance
- like in Java, these methods are designated static

C++ also supports static member variables

- e.g. variables that are common between instances
Class Method Example

**Objective-C**

```objc
#import <Foundation/Foundation.h>

@interface Point : NSObject
{
    int x, y;
}

+ (int) numberOfInstances;
- init;
@end

@implementation Point
static int instanceCount = 0;

+ (int) numberOfInstances
{
    return instanceCount;
}

- init
{
    if (!(self = [super init]))
        return nil;
    instanceCount++;
    
    return self;
}
@end
```

**C++**

```
class Point
{
    protected:
    int x, y;

    public:
    static int numberOfInstances();
    Point();

    protected:
    static int instanceCount = 0;
};

int Point::numberOfInstances
{
    return instanceCount;
}

Point::Point()
{
    x = 0;
    y = 0;
    instanceCount++;
}
```
C++ Memory Management
Memory Management

- **C++ Memory management is completely manual**
  - no garbage collector
  - no reference counting
  ⇒ program needs to track how long an object is required

- **new operator**
  - allocates memory for an object
  - invokes the corresponding constructor

- **delete operator**
  - releases an object (frees memory)

- **Problem: how to track object usage?**
  - copies instead of references
    → often inefficient
    → stack objects
  - individual solutions for individual programs
  - implement reference counting
    - difficult because of lack of reflection capabilities
    - possible only through complex language features
### Objective-C

```objective-c
#import <Foundation/Foundation.h>

@interface Person : NSObject
{
    int yearOfBirth;
    NSString *name;
    Person *mother, *father;
}

- (void) setYearOfBirth: (int) born;
- (void) setName: (NSString *) newName;
- (void) setMotherFather: (Person *) m, (Person *) f;
- (int) yearOfBirth;
- (NSString *) name;
- (Person *) mother;
- (Person *) father;
- (void) dealloc;
@end
```

### C++

```cpp
#include <string>

class Person
{
    int yearOfBirth;
    std::string name;
    Person *mother, *father;

public:
    void setYearOfBirth(int born);
    void setName(std::string &newName);
    void setMotherFather(Person m, Person f);
    int getYearOfBirth();
    std::string &getName();
    Person *getMother();
    Person *getFather();
};
```
# Objective-C

```objective-c
#import "Person.h"

@implementation Person

-(int) yearOfBirth
{
    return yearOfBirth;
}

-(NSString *) name
{
    return name;
}

-(Person *) mother
{
    return mother;
}

-(Person *) father
{
    return father;
}

-(void) setYearOfBirth: (int) born
{
    yearOfBirth = born;
}
@end
```

---

# C++

```cpp
#include "Person.h"

// Person implementation

int Person::getYearOfBirth()
{
    return yearOfBirth;
}

std::string &Person::getName()
{
    return name;
}

Person *Person::getMother()
{
    return mother;
}

Person *Person::getFather()
{
    return father;
}

void Person::setYearOfBirth(int born)
{
    yearOfBirth = born;
}
```

---

René Hexel and Joel Fenwick

Object Oriented Programming in C++
Objective-C

- (void) setName: (NSString *) newName
{
    [name release];
    name = [newName copy];
}

- (void) setMother: (Person *) m
    father: (Person *) f
{
    [m retain]; [f retain];
    [mother release];
    [father release];
    mother = m; father = f;
}

- (void) dealloc
{
    [name release];
    [mother release];
    [father release];
    [super dealloc];
}
@end

C++

```cpp
void Person::setName(std::string &newName)
{
    name.assign(newName);
}

void Person::setMotherFather(Person *m, Person *f)
{
    /*
     * no reference counting,
     * program needs to track m/f
     */
    mother = m;
    father = f;
}
```

/*
* for this C++ program, dealloc
* or equivalent is difficult --
* the program needs to manually
* track object ownership and
* release objects accordingly
*/

René Hexel and Joel Fenwick

Object Oriented Programming in C++
The Call-By-Reference Type

- In a type definition, the ampersand character \& defines an implicit reference type.
  - like a pointer, it references the memory address of a variable
  - even though it is a pointer, it uses non-pointer notation (like call-by-reference in Java)
- References can be passed to methods and returned by methods
Stack Objects and Member Objects

- C++ allows Stack Objects
  - object lives on the stack instead of the heap
  → similar to primitive types (int, double, …)

- C++ allows Member Objects
  - whole objects inside of other objects
  → whole copies, not just references

- Accessed through type name without the * pointer symbol

- No new and delete operators needed
  - object lifetime is equivalent to its scope
  - object gets allocated when it comes into scope
    → constructor gets called
  - object gets deallocated when it loses scope
    → destructor gets called
Destructors

Like `dealloc` in Objective-C, the destructor is used to clean up an object before its memory is deallocated. They have the same name as the class with a `∼` in front, and no return type. If you do not declare one C++ makes an empty one for you.

Example (Destructor Example)

```cpp
class MyClass {
public:
    MyClass();
    ∼MyClass();
...
    // Meanwhile in .cc
    MyClass::∼MyClass() {
        ...
    }
```
Destructor Example

Header File

class Card { /* ... */};

class Test
{
protected:
    int i;
    Card* c;

public:
    Test();
    ~Test();
};

Implementation

Test::Test()
{
    c = new Card();
}

Test::~Test()
{
    delete c;
}
## Object Lifecycle

<table>
<thead>
<tr>
<th>Task</th>
<th>Objective-C</th>
<th>Java</th>
<th>C++ Heap</th>
<th>C++ Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate initialise</td>
<td>+ alloc</td>
<td>new</td>
<td>new constr.</td>
<td>entry constr.</td>
</tr>
<tr>
<td>hold object let go</td>
<td>- init</td>
<td>constr.</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>- retain</td>
<td>automatic constr.</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>- release</td>
<td>automatic</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>destroy clean up</td>
<td>final - release</td>
<td>G.C.</td>
<td>G.C.</td>
<td>fn exit destr.</td>
</tr>
<tr>
<td>deallocate</td>
<td>final - dealloc</td>
<td>finalise()</td>
<td>delete</td>
<td>destr.</td>
</tr>
<tr>
<td></td>
<td>[super dealloc]</td>
<td>G.C.</td>
<td>delete</td>
<td>return</td>
</tr>
</tbody>
</table>
Strings

String Objects in C++
C++ Strings

- Like Objective-C, there is a String class in C++
  - `std::string`
    - lower case!
    - much nicer than having to use `char *`

- Mutable Strings
  - `all std::string objects are mutable`

- String Constants
  - there are no string constants in C++
    - C++ strings can be created from C strings
    - like in Objective-C, this also works for C string constants embedded in " "

René Hexel and Joel Fenwick
Object Oriented Programming in C++
C++ String Examples

Example (Some `std::string` methods)

```cpp
#include <string>
#include <cstdio>

std::string s1; // empty string
std::string s3("Hello, void"); // from C string
std::string *s4 = new std::string("Hi, it’s "); // string pointer

s4->append("28 degrees celsius"); // appending char *
s1.append(s3); // appending string

int len4 = s4->length(); // get length of s4

if (s1.compare(s3) == 0) // same content?
    printf("s1 is equal to s3 -- how come?\n");
else if (s1.compare(s3) < 0) // which one comes first?
    printf("s1 comes before s3\n");
else
    printf("s3 comes before s1\n");

printf("s3 is: %s\n", s3.c_str()); // convert s3 to a C string for printf

delete s4; // don’t forget proper memory management!
```
Other Useful Methods

- getline()
  - reads a single line of a file (stream) into a string

- find()
  - searches for a string within another String

- substr()
  - returns a substring within a given range

→ See the Strings Section of the C/C++ API