COMP1008
Other OO Languages
C++ and Ruby

Agenda

- Categories of Object-Oriented Languages
- Type Checking
- C++
- Ruby

Other Object-Oriented Languages

- Many OO languages exist.
- Only a minority are in widespread use.
  - See http://en.wikipedia.org/wiki/Object-oriented_programming#History
  - and similar websites.
- Java is one of the most popular and actively used.
- All share the same underlying ideas and concepts.
- Try some other languages yourself!

Categories of OO Languages

- Fully compiled to machine code run directly by processor:
  - C++, Eiffel, (Ada95)
- Compiled to bytecode run by virtual machine:
  - Smalltalk, Java, (C#)
- Interpreted and scripting languages:
  - Ruby, Python, JavaScript

Type Checking

- Check that code follows language syntax and grammar specification.
- Check that variables and values have correct type.
- Check that values of different types are not mixed up.
  - e.g. void d = true + 10;
- Check that methods are called with correct kind of parameters.
- And so on.

Static v. Dynamic Type Checking

- Static type checking is done by the compiler.
  - Compiler checks and enforces the type rules.
- Dynamic type checking is done while a program is run.
  - Runtime code checks types of values/objects are correct.
- Compiled languages typically use static type checking but need some dynamic checking as well.
  - C++, Java have extensive static checking.
  - But, Java does make significant use of dynamic checking.

Static v. Dynamic Type Checking (2)

- Static
  - Advantages:
    - An entire category of errors can be detected before a program is ever run.
    - Errors are reported early during compilation.
    - Type safety.
  - Disadvantages
    - Can be complex.
    - Program code longer (due to type declarations, etc.) and takes more time to write.
    - Limits, or makes more complicated, what can be expressed with the language.
    - More complicated and slower compilers.

Static v. Dynamic Type Checking (3)

- Dynamic
  - Advantages:
    - Flexibility, greater ease of expression.
    - Allows more dynamic code (e.g., reflection, self-modifying code).
    - Don't need to declare variables, types, etc. before use.
    - Coding speed.
  - Disadvantages:
    - Type checking delayed until code is run.
    - Type errors may not be found for some time.
    - Less information in source code for understanding what it does.

Static v. Dynamic Type Checking (4)

- Which is best?
- Long running and contentious debate!
  - Static checking seen as good for larger, more complex programs.
  - Dynamic checking seen as good for rapid development, prototyping and agile development.
  - Static checking has been fairly dominant for several decades but dynamic checking on rise again.

Strong v. Weak Typing

- With a strongly typed language all type errors will be detected.
  - Either by the compiler or at runtime.
- With a weakly typed language type errors are ignored.
  - Binary representation just used without checking
    - e.g., Use an binary may represent an int but used as float.
  - Defaults used, e.g., treat everything as a string.
  - Or errors just happen.
- C++, Java, Ruby, Python, Smalltalk are all strongly typed.
  - Weakly typed languages are typically used for scripting, e.g., bash shell script.

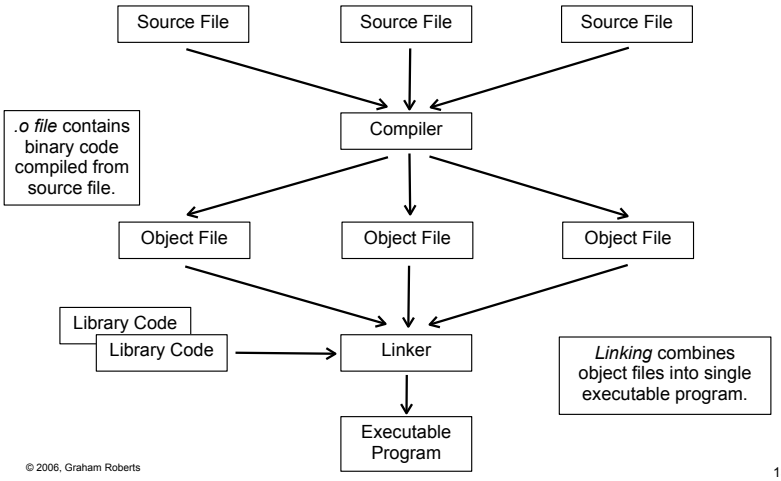
C++

- Well established, heavy-weight OO language.
  - Has sunk a bit under its own weight, though.
- Compiles direct to machine code, so seen as good for performance.
- Derived from C, which was developed with Unix.
- Syntax and many features inherited by Java.
- Widely used for systems programming and application development.

C++ Language

- Big and complex.
- Fully supports OO programming.
  - Methods called functions (& member functions).
- Also supports procedural programming (no classes)
  - Upwardly compatible with the C programming language.
- Standard libraries, including STL (standard template library).
- No virtual machine so gives direct access to memory.
- Efficient compilers can generate high-performance code.

C++ Compilation



C++ Compiler

- Many available
- Lots of programmers use GCC, the GNU Compiler Collection
  - Major Free Software Foundation project.
    - See http://gcc.gnu.org/
  - Supports C, C++, Objective-C, Objective-C++, Java, Fortran and Ada.
  - Operating Systems like GNU/Linux and OS X are written in C/C++ and compiled by GCC.
  - g++ command used for C++ compilation (gcc for C).

Declaration v. Definition

- A declaration introduces a name and its type.
  - Declarations are put into a *header file*, a .h file.
- A definition defines what a name is.
  - e.g., provides a function body.
  - Definitions are put in a .cpp (or .cc) file.
- Compiling a .cpp file typically requires one or more hearder files to be included.
- Linking a complete program requires all the declarations and definitions to be consistent.

C++ Class Person Header File

```
#include <string>
#include <vector>

using std::string;
using std::vector;

class Person
{
  private:
    string firstName;
    string familyName;
    vector<string> emailAddresses;

  public:
    Person(string firstName, string familyName);
    string getFullName();
    void addEmailAddress(string email);
    vector<string> getEmailAddresses();
};
```

Include header files for string and vector library classes. Vector is like ArrayList.

std::string means string in standard namespace. Using allows string to be used instead of std::string.

C++ Class Person Member Functions

```
#include "Person.h"

Person::Person(string firstName, string familyName)
  : firstName(firstName), familyName(familyName)
{}

string Person::getFullName()
{
  return firstName + " " + familyName;
}

void Person::addEmailAddress(string email)
{
  emailAddresses.push_back(email);
}

vector<string> Person::getEmailAddresses()
{
  return emailAddresses;
}
```

Include Person header file in order to declare class.

Declare member functions using Person:: to identify which class function belongs to.

Class can be compiled using:
g++ -c Person.cpp
Creates Person.o, unlinked binary code representation.

Using Class Person

```
#include <iostream>
#include "Person.h"
using std::cout;
using std::endl;

void show(vector<string> v) {
  cout << "---" << endl;
  vector<string>::iterator iter;
  for (iter = v.begin(); iter != v.end(); iter++) {
    cout << *iter << endl;
  }
  cout << "---" << endl;
}

void usePerson() {
  Person p("Arthur","Dent");
  cout << "Person fullName: " << p.getFullName() << endl;
  p.addEmailAddress("dent@earth.com");
  p.addEmailAddress("dent@earth.co.uk");
  show(p.getEmailAddresses());
}


int main (int argc, char* const argv[]) {
  usePerson();
  return 0;
}
```

cout << “message” is like System.out.println
<< is the inserter operator.

main function where execution starts.

Running Program

From the command line:

```
> g++ -c Person.cpp
> g++ -c main.cpp
> g++ -o Person Person.o main.o
> Person
```

Creates an executable program called Person.

For simple programs can do this in one step:

```
> g++ *.cpp
> a.out
```

This creates an executable program called a.out.

Objects and Memory

- The C++ Person class looks like it works the same way as the Java version.
  - Object references pointing to objects.
- But it is not...
- C++ has a more complex memory model.
  - Gives programmer control but programmer must understand how memory is used.

Variables and Objects

```
Person p("Arthur","Dent");
Person q("Ford","Prefect");
...
q = p;
```

- Variable p actually holds the Person object, *not* a reference to it.
- The assignment *copies* the object.
- Hence in the Person example:
  - objects passed as parameters
  - objects returned from methods
  - and objects used in assignment expressions

  are *all copied*.
- No references, changing copy doesn't change original.
- Memory allocation is handled automatically.
- C++ allows a *copy constructor* to control copying.

Using Pointers

- We have to rewrite C++ Person to use *pointers*, to make it behave like the Java version.
- A pointer is a memory address.
- A pointer variable stores the address in memory where the object is located.
  - Like a reference but a pointer is a real memory address.

C++ Person Header with Pointers

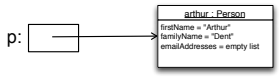
```
#include <string>
#include <vector>

using std::string;
using std::vector;

class Person2
{
  private:
    string* firstName;
    string* familyName;
    vector<string*>* emailAddresses;

  public:
    Person2(string* firstName, string* familyName);
    ~Person2();
    string* getFullName();
    void addEmailAddress(string* email);
    vector<string*>* getEmailAddresses();
};
```

* is used to denote a pointer type.
Person* is pointer to Person (object).



Objects are allocated on the Heap (heap object).

Also have to introduce a destructor.

Note - this is an example, not necessarily normal practice.

C++ Person Member Functions with Pointers

```
#include "Person2.h"
Person2::Person2(string* firstName, string* familyName)
: firstName(firstName), familyName(familyName) {
  emailAddresses = new vector<string*>();
}
Person2::~Person2() {
  delete firstName;
  delete familyName;
  vector<string*>::iterator iter;
  for (iter = emailAddresses->begin(); iter != emailAddresses->end(); iter++) {
    delete *iter;
  }
  delete emailAddresses;
}
string* Person2::getFullName() {
  return new string(*firstName + " " + *familyName);
}
void Person2::addEmailAddress(string* email) {
  emailAddresses->push_back(email);
}
vector<string*>* Person2::getEmailAddresses() {
  return emailAddresses;
}
```

Destructor must be declared to delete object and anything it points at. No garbage collection.

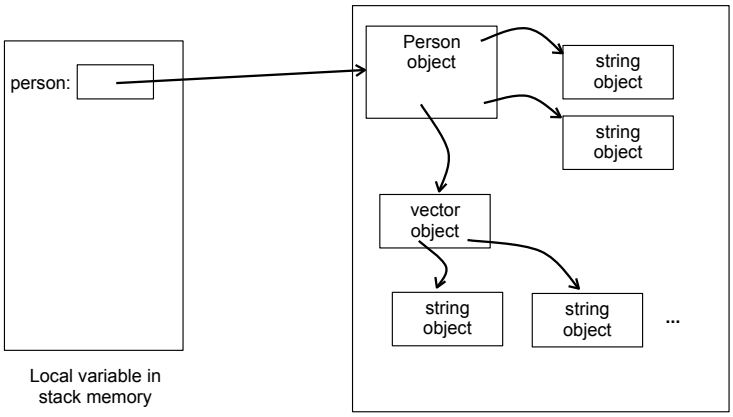
* also used as dereference operator to follow pointer and access object.
-> is also a dereference operator to call function.

Using Person with Pointers

```
#include <iostream>
#include "Person2.h"
using std::cout;
using std::endl;
void show2(vector<string*>* v) {
  cout << "---" << endl;
  vector<string*>::iterator iter;
  for (iter = v->begin(); iter != v->end(); iter++) {
    cout << **iter << endl;
  }
  cout << "---" << endl;
}
void usePerson2() {
  Person2* p2 = new Person2(new string("Ford"), new string("Prefect"));
  std::cout << "Person fullName: " << *(p2->getFullName()) << endl;
  p2->addEmailAddress(new string("dent@earth.com"));
  p2->addEmailAddress(new string("dent@earth.co.uk"));
  show2(p2->getEmailAddresses());
}
int main (int argc, char * const argv[]) {
  usePerson2();
}
```

new is used to create objects

Pointers and Memory



delete

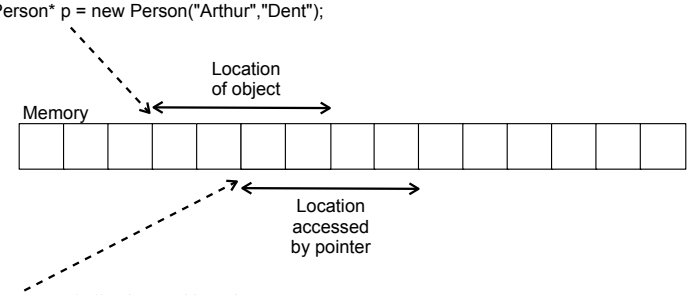
- C++ no garbage collection.
- Heap Objects must be deleted explicitly using the *delete* operator.
- If an object has pointers to other objects, then code must be written to delete all the objects.
  - Hence, the need to declare a *destructor* function.
- If a heap object is not deleted it remains in memory.
  - A *memory leak* occurs when heap objects are not deleted (a bug).
  - If the program is run long enough, memory can run out even though the objects cannot be used.

Pointer Arithmetic

- A reference in Java can only be used in a controlled way.
- A pointer in C++ can be changed by addition and substraction.
  - Any location in data memory can potentially be accessed.
  - Any piece of memory could be treated as an object, whether it holds an object or not.

Pointer Errors

```
Person* p = new Person("Arthur","Dent");
```




```
p = p + 2; // pointer arithmetic
p->getFullName(); // Runs and tries to use whatever is in memory as a Person object.
                  // May or may not fail, depending on memory content.
```


No Pointers in Java

- Pointers give a lot of power, but:
  - Programmer must get use correct.
  - Programmer must manage memory, new/delete.
  - Can easily be abused.
  - Allows encapsulation to be completely bypassed.
  - Cause of many bugs in C++ programs.
- Pointers can lead to very efficient code.
- Java deliberately replaced pointers with references and memory management.
  - To eliminate a large source of errors.

Questions?

Ruby

- An interpreted language.
  - No compiler.
  - An interpreter reads program text line by line and carries out each statement.
    - This can be optimised.
  - Trade-off performance for flexibility and rapid programming (no time compiling).
- Small, light-weight language.
- Very strongly typed, dynamically type checked.

Ruby Class Person — Person.rb

```
class Person
  def initialize(firstName,familyName)
    @firstName = firstName
    @familyName = familyName
    @emailAddresses = []
  end

  def getFullName
    @firstName + " " + @familyName
  end

  def addEmailAddress(address)
    @emailAddresses << address
  end

  def getEmailAddresses
    Array.new(@emailAddresses)
  end
```

Constructor method

@name used to denote an instance variable. Variable is added to object when used.

An empty array

Append to array

No type declarations needed

Using Ruby Class Person

```
person = Person.new("Arthur","Dent")
puts person.getFullName
person.addEmailAddress "arthur@earth.com"
emails = person.getEmailAddresses
puts emails.length
puts emails[0]
person.addEmailAddress "arthur@HoG.com"
emails = person.getEmailAddresses
puts emails.length
puts emails[0..-1]

Displays:
Arthur Dent
1
arthur@earth.com
2
arthur@earth.com
arthur@HoG.com
```

This code can simply be appended to the source file after the class declaration. Note that no class, method or main has to be declared.

Run using the command: ruby Person.rb
Ruby interpreter reads file and interprets code line by line.

puts (put string) is used to display output.

Ruby, objects and memory

- Like Java, Ruby uses object references.
- Provides garbage collection.
- Strong dynamic typing means that only methods declared by object's class can be called.
  - But Ruby language provides more ways of declaring methods than Java.
- *Everything* is an object, no primitive types.
  - e.g., 1.next => 2 (call method next on 1)

Summary

- Strong v. Weak Type Checking
- Static v. Dynamic Type Checking
- C++, compiled OO language
- Ruby, interpreted OO language