Lab Class Scientific Computing 2021, WISM454

Adriaan Graas, Week 5

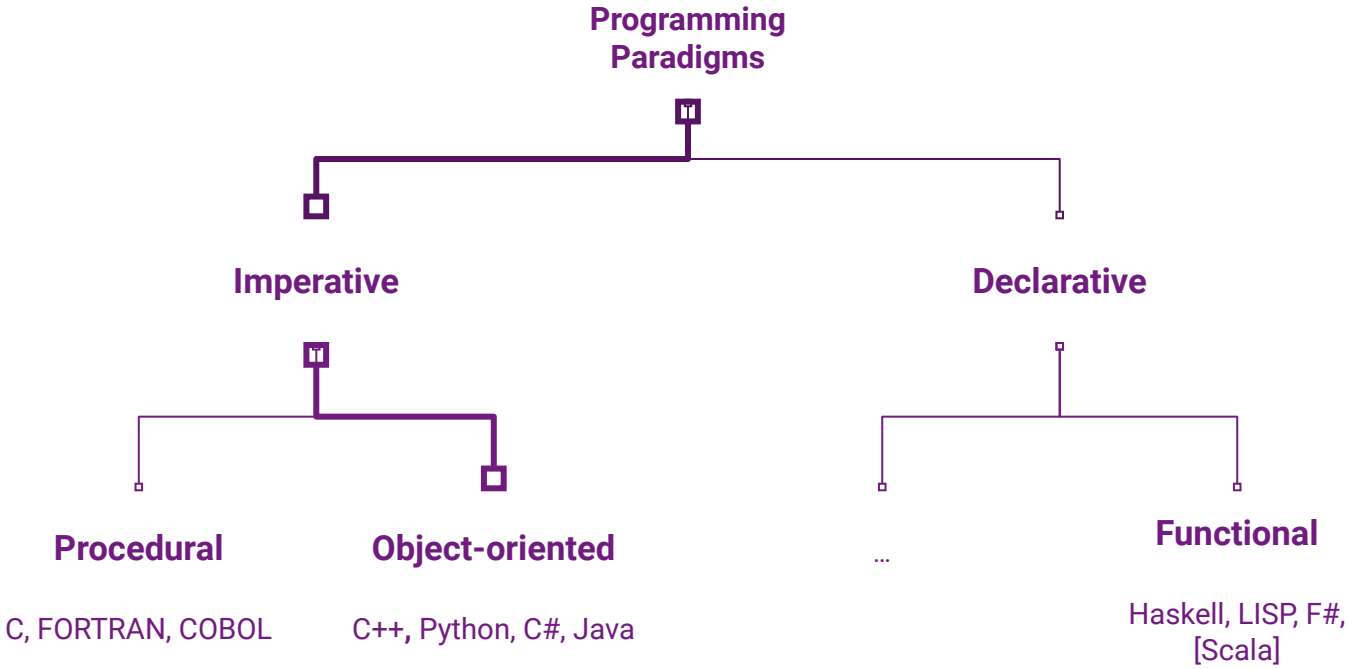
Mini-report 2

- Coding *nonuniform distributions* using
  - **Inversion method**
    - Requires the inverse cumulative density function
  - **Rejection method**
    - Requires another distribution with majorizing probability density function
- C++ and Object-Oriented Programming (OOP)
  - **Programming concepts**
    - References, classes
  - **Design concepts**
    - Composition, inheritance, polymorphism
  - **Compilation**
    - CMake, building from the editor

Object-oriented programming (OOP)

History

- First introduction at MIT in late ‘50s, early ‘60s.
- In ‘70s Xerox PARC developed a language called **Smalltalk**.
- Became more popular in early-mid ‘80s:
  - **Objective-C was developed by Brad Cox**
  - **C++, “C with classes” was developed by Bjarne Strousup**
- In ‘90s started seeing more integration into other languages:
  - **BASIC, Fortran, Pascal, COBOL**
- Nowadays, many popular languages are object-oriented,
  - **Python (developed at CWI)**
  - **Java (Sun Microsystems)**
  - **C# (Microsoft)**


Programming Paradigms
Imperative
Declarative
Procedural
Object-oriented
...
Functional
C, FORTRAN, COBOL
C++, Python, C#, Java
Haskell, LISP, F#, [Scala]

Procedural paradigm (C)

Procedures (functions) work on data objects.
Data and functions are independent.

```
struct LCRNG {
    unsigned a, c, m;
};

unsigned next(struct LCRNG * rng) {
   return rng->a...; }
void print(struct LCRNG * rng) {
    printf("a: %u, ...", rng->a, ...); }

...
struct LCRNG randu = { ... }; // object
print(randu); // `print` works on `randu`
```

Object-oriented paradigm (C++)

Functions are attached to objects.
"Attached functions" can modify the data.

```
class LCRNG {
    unsigned a, c, m;

    unsigned next() {
        return a...; }
    void print() {
        printf("a: %u, ...", a, ...); }
};

...
LCRNG randu { ... }; // object
randu->print(); // `print` is a member of
`randu`
```

C++
Introduction

“View C++ as a federation of languages” (Scott Meyers, Effective C++)

- The four “sublanguages” of C++:
  1. **C language**
     - C++ is in a informal sense an extension of C: C with classes
     - There is often a C way, and a C++ way
  2. **Object-Oriented Programming**
     - Design that revolves around the design of objects, using classes
  3. **Standard library (STL)**
     - C++ code that has been prewritten (e.g. algorithms, data structures)
  4. **Template metaprogramming**
     - Pieces of code that can be reused to generate types.

C++ versions

- C++ versions
  - **C++98**
    - C++ is in a informal sense an extension of C: C with classes.
    - Already: templates, STL, strings.
    - A ton of functionality in external libraries (Boost).
    - Avoid C++ suggestions from 2011 or before.
  - **C++11, C++14, C++17, C++20**
    - Often termed as **Modern C++**.
    - **Resource safety** (no memory leaks, buffer overflows).
    - **Type safety** (data belonging to the right types).
    - Much more functionality in standard library.

Hello, world!

```cpp
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

C++

Value categories

Value categories

- *Expressions* in C++ are categorized
  - **Either** lvalue**, or** rvalue.
    - Actually: it's more complicated, but this is good enough for now.
  - **Confusing: it’s a category for expressions, not so much for values.**
- An *lvalue*:
  - **the expression has a name;**
  - **can be assigned to;**
  - **has a memory address.**
- An *rvalue*:
  - **is a “temporary” and disposable;**
  - **cleaned up after the statement (;).**

lvalue or rvalue?

```
// lvalue or rvalue?
x
```

lvalue or rvalue?

```
// lvalue or rvalue?
x;     // lvalue!
x = 4; // of course you can assign to it
```

lvalue or rvalue?

```
// lvalue or rvalue?
4
```

lvalue or rvalue?

```
// lvalue or rvalue?
4;     // rvalue!
4 = x; // error: lvalue required
```

lvalue or rvalue?

```
// lvalue or rvalue?
x + y
```

lvalue or rvalue?

```
// lvalue or rvalue?
x + y;     // rvalue!
x + y = 6; // error: lvalue required
```

lvalue or rvalue?

```
// lvalue or rvalue?
x = y
```

lvalue or rvalue?

```
// lvalue or rvalue?
x = y;       // rvalue!
(x = y) = z; // error: lvalue required
```

lvalue or rvalue?

```
// lvalue or rvalue?
x = y;       // rvalue!
(x = y) = z; // error: lvalue required
x = (y = z); // fine, `x` is lvalue, `y = z` an rvalue
```

lvalue or rvalue?

```
// lvalue or rvalue?
(*rng_pointer).m
```

lvalue or rvalue?

```
// lvalue or rvalue?
(*rng_pointer).m;       // lvalue!
(*rng_pointer).m = 2;   // no problem
```

lvalue or rvalue?

```
// lvalue or rvalue?
&(*pointer)
```

lvalue or rvalue?

```
// lvalue or rvalue?
&(*pointer);        // operators return rvalues
&(*pointer) = 2;    // error
```

C++
lvalue references

You might know references from Python

```
import numpy as np

a = np.zeros((3, 5))
b = [1, 2, 3, 4]
c = 4

a2 = a
b2 = b
c2 = c

a2[1, 2] = 1  # changed `a2`, but did `a` also change?
b2[0] = 3     # changed `b2`, but did `b` also change?
c2 = 7        # changed `c2`, but did `c` also change?
```

You might know references from Python

```
import numpy as np

a = np.zeros((3, 5))
b = [1, 2, 3, 4]
c = 4

a2 = a        # a reference
b2 = b        # a reference
c2 = c        # a copy

a2[1, 2] = 1  # changed `a2`, but did `a` also change? Yes!
b2[0] = 3     # changed `b2`, but did `b` also change? Yes!
c2 = 7        # changed `c2`, but did `c` also change? No!
```

	Pointer	Const pointer	lvalue reference	Const lvalue reference
Type	`T *`	`[const] T *` `[const]`	`T &`	`const T &`
Usage	`T * ptr = &a;` `T b = *ptr;`	Same as pointer.	`T & r = a;`	
Mutable?	Yes: change pointer or its value.	Pointer and/or value can be const.	Reference cannot change, but value can.	Neither reference nor value.

Lvalue references in C++ are denoted by T &

```
int a = 5;
int & b = a;
b += 2;
std::cout << a << std::endl;  // `a` also changed to 7!
```

Const lvalue references

```
int a = 5;
const int & b = a;
b += 2; // error: assignment of read-only reference
```

Rules to remember

lvalues **bind to** const and non-const lvalue references
rvalues **bind only to** const lvalue references

```
// lvalues        bind to          T &
int & b = x;

// lvalues        bind to          const T &
const int & b = x;

// rvalues        do not bind to   T &
int & c = 3; // error: cannot bind non-const lvalue ...

// rvalues        bind to          const T &
const int & d = 4; // an exception that you just need to memorize
                   // think: “extend the lifespan of the rvalue”
```

Pass-by-reference

```
/* Contract: “I might change your `a`”! */
void f(int & a) {
    a += 7;
}

...
int n = 4;
f(n);             // no copy, no pointer!
std::cout << n;   // prints 11
f(7);             // error: why?
return 0;
```

Pass-by-const-reference

```
/* Contract: “I promise, I won’t change your `a`.” */
double f(const int & a) {
    return a * 2.0;
}

...
double a = f(4); // rvalue (4) binds to const int & (function parameter)
std::cout << a; // 8.0!
return 0;
```

C++
Classes

What are classes

- Classes are *compound types* and have *members*
  - **Exactly like *struct*!**
  - **Instances of classes are, again, objects.**
  - **Members are either functions, or variables.**
- Classes enjoy *encapsulation*
  - **Members that are declared *protected* or *private* cannot be accessed from outside.**
- Classes can be constructed through *inheritance*
  - ***Inheritance:* the type of one class can be based off another class.**
  - **Making variations of one type leads to *polymorphic* behavior.**

Encapsulation

Data members are not accessible when they are *protected* or *private*.

```
class Coffee {
public: // access specifier
    Coffee() { // special member: constructor
        std::cout << "Your Coffee has been constructed." << std::endl;
    }
    unsigned sugar{0}; // public member (generally, bad style: not kept internal)
protected:
    unsigned milk_{0}; // protected member (good style: internal)
};
```

Encapsulation

Data members are not accessible when they are *protected* or *private*.

```
class Coffee {
public:
    Coffee() {
        std::cout << "Your Coffee has been constructed." << std::endl;
    }
    unsigned sugar{0};
protected:
    unsigned milk_{0};
};



...
Coffee cup {};     // “Your coffee has been constructed.”

Coffee another {}; // “Your coffee has been constructed.”
another.sugar = 2;
another.milk_ = 2; // error: unsigned int Coffee::milk_ is protected
another.milk_;     // error: unsigned int Coffee::milk_ is protected
```

Encapsulation

Access data through class using member functions.

```
class Coffee {
public:
    ...
    void add_milk() { // good style: allows protection object state
        if (milk_ > 2) { // too much milk, doesn't fit in cup, raise error }
        milk_ += 1;
    }
    unsigned get_milk() { return milk_; }
protected:
    ...
    unsigned milk_{0};
};
```

Encapsulation

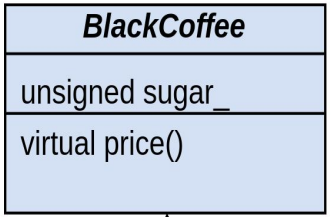
Access data through class using member functions.

```
class Coffee {
public:
    ...
    void add_milk() { // good style: allows protection object state
        if (milk_ > 2) { // too much milk, doesn't fit in cup, raise error }
        milk_ += 1;
    }
    unsigned get_milk() { return milk_; }
protected:
    ...
    unsigned milk_{0};
};

...
Coffee cup {};  // “Your coffee has been constructed.”
                // cup.get_milk() would give 0
cup.add_milk(); // now cup.get_milk() is 1
cup.add_milk(); // now cup.get_milk() is 2
cup.add_milk(); // error: too much milk
```

Inheritance

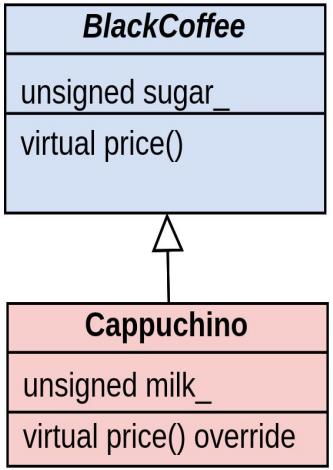
Derive one type from the other.




```
 // the base class (parent) has only `sugar_`
class BlackCoffee {
public:
    virtual double price() { ... } // virtual allows overriding
protected:
    unsigned sugar_;
};
```

Inheritance

Derive one type from the other.

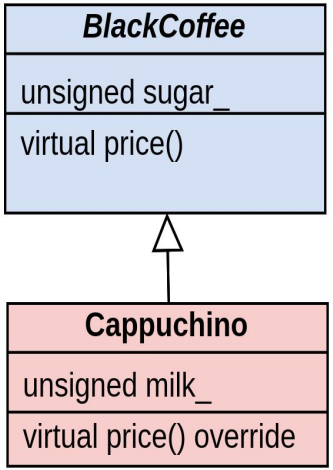



```cpp
/* The base class (parent) */
class BlackCoffee {
public:
    virtual double price() { ... } // virtual allows overriding
protected:
    unsigned sugar_;
};

/* The derived class (child)
 * “is-a” BlackCoffee but has also `milk_` */
class Cappuccino : public BlackCoffee {
public:
    double price() override { ... } // - replaces `price()` from base
                                    // - has access to `sugar_` from base
                                    // - can call `price()` from base
protected:
    unsigned milk_;
};
```

Inheritance

Derive one type from the other.




```cpp
/* The base class (parent) */
class BlackCoffee {
public:
    virtual double price() { ... } // virtual allows overriding
protected:
    unsigned sugar_;
};

/* The derived class (child)
 * “is-a” BlackCoffee but has also `milk_` */
class Cappuccino : public BlackCoffee {
public:
    double price() override { ... } // - replaces `price()` from base
                                    // - has access to `sugar_` from base
                                    // - can call `price()` from base
protected:
    unsigned milk_;
};
```

C++

Abstract & concrete classes

Abstract class

A class that has one
or multiple **pure virtual functions**.

```
/* An abstract class has at least one pure virtual method */
class Priceable {
public:
    virtual double price() = 0; // A function is called pure virtual when:
                                //  - virtual  allows override in subclass
                                //  - = 0      function is not implemented
}
```

Concrete class

A class that is not abstract.

```
/* An abstract class */
class Priceable {
public:
    virtual double price() = 0; // pure virtual
}

/* A concrete class, because `price` has been implemented. */
class BlackCoffee : public Priceable {
public:
    virtual double price() override {
        ...
    }
}
```

C++

Constructors and initialization

Constructors

- The **constructor** of a class *initializes* an object.

- Two steps:
  1. **Initialization list**
     - **Must** initialize all the members of the class!
  2. **Body**
     - Some code block that can be run once the initialization has completed.

Constructors

```
class Coffee {
public:
    Coffee()
    : sugar_(1), milk_(1) // 1. initializer list ": ..., ..., ..."
    {                     // 2. body "{ ... }"
        std::cout << "Your Coffee has been constructed." << std::endl;
        milk_ = 2; // assignment, not initialization!
    }
protected:
    unsigned sugar_;
    unsigned milk_{1}; // default initializer was ignored
};
```

Constructor with arguments & constructor overloading

```
class Coffee {
public:
    Coffee(unsigned sugar, unsigned milk = 1)
    : sugar_(sugar), milk_(milk)
    { }

    Coffee() // constructor overloading, having two or more constructors is totally fine
    : sugar_(0), milk_(0)
    { }
};

...
Coffee foo {2};    // first ctor: 2 sugar, 1 milk
Coffee bar {2, 2}; // first ctor: 2 sugar, 2 milk
Coffee baz;        // second ctor: 0 sugar, 0 milk
```

There are many ways of initialization

Thanks to *overload resolution*, they usually do the same thing.

```
class Coffee {
...
};

// https://en.cppreference.com/w/cpp/language/initialization
Coffee x;                 // “default initialization” for ctor without args
Coffee x {};              // brace initialization for ctor without args
Coffee x(2, 2);           // direct initialization
Coffee x {2, 2};          // brace initialization (preferred)
Coffee x = {2, 2};        // copy-list-initialization
Coffee x = Coffee(2, 2); // copy initialization (makes no copy here)

Coffee y(x);          // copy initialization
Coffee y = x;         // also copy initialization
```

This week

- Today / this week:
  1. **Start C++ project with CMake**
  2. **Read about *inversion method* and *rejection method***
  3. **Exercises 2.1 and 2.2**
     - Make classes for RNGs, *UniformDistribution*, and *UniformSampler*.
     - Look at Notes 2.3 for C++ source-header example of a class.