An Introduction to C++

Part 2

Recursion

- is elegant and allowed

```
unsigned long fac(unsigned short k) {
  return k ? k*fac(k-1) : 1;
}
```

- however non-recursive version often faster

```
unsigned long fac(unsigned short k) {
  unsigned long r=1;
  if(k) do { r *=k;} while(--k);
  return r;
}
```

- exception: template codes, as they are evaluated at compile time. We will come back to that later.

Inlining

- A function call takes several hundred CPU cycles
- For simple functions that are called often this is a big waste of time:
  - ```
    float square(float);

    int main() {
      float sq[10000];
      for (int k=0;k<10000;++k)
        sq[k] = square(k);
    }
    ```
- It is better to inline the function
  - `inline float square(float x) {return x*x;}`
- This leads to the same optimized code as:
  - `sq[k] = float(k)*float(k);`

Default function arguments

- are sometimes useful

```
float root(float x, unsigned int n=2); // n-th root of x

int main()
{
  root(5,3); // cubic root of 5
  root(3,2); // square root of 3
  root(3); // also square root of 3
}
```

- the default value must be a constant!

```
unsigned int d=2;
float root(float x, unsigned int n=d); // not allowed!
```
```

Function overloading

- Consider a function max(A,B):
  - For int:
```
int max(int a, int b)
{ return (a>b ? a : b);}
```
  - For float:
```
float max(float a, float b)
{ return (a>b ? a : b);}
```
- This is a problem in C, FORTRAN, etc.
- Compiler chooses which one to use
- However be careful:
  - `max(1,3.1415927);` // Problem! which one?
  - `max(1.,3.1415927);` // OK
  - `max(1,int(3.1415927f));` // OK
  - or define new function `double max(int,float);`

Function templates

- We have now learned function overloading:
```
double max(double a,
           double b) {
  return (a<b ? b : a);
}

int max(int a, int b) {
  return (a<b ? b : a);
}
```
- All these max functions are identical except for the type!
- Tedious copying for each used type!
- Solution: templates
- tell the compiler how to generate a function for any type:
```
template <class T>
T max(T a, T b) {
  return (a<b ? b : a);
}
```
- works for any type T for which `<` has been defined
- if `<` is not defined a compile time error will result
- very convenient technique

Classes

- Are a method to create new data types
- Object oriented programming:
  - Instead of asking: “What are the subroutines?”
  - We ask:
    - What are the abstract concepts?
    - What are the properties of these concepts?
    - How can they be manipulated?
  - Then we implement these concepts as classes
- Advantages:
  - High level of abstraction possible
  - Hiding of representation dependent details
  - Presentation of an abstract and simple interface
  - Encapsulation of all operations on a type within a class
    - allows easier debugging

What are classes?

- Classes are collections of
  - functions
  - data
  - types
- representing one concept

- These “members” can be split into
  - `public`, accessible interface to the outside
    - should not be modified later!
  - `private`, hidden representation of the concept
    - can be changed without breaking any program using the class
  - this is called “data hiding”
- Objects of this type can be modified only by these member functions -> easier debugging

Class members

- A first example

```
class Trafficlight  {
  public: // access declaration
    enum light { green, orange, red}; // type member
    Trafficlight(); // default constructor
    Trafficlight(light); // constructor
    ~Trafficlight(); // destructor

    light state() const; //function member
    void set_state(light);
  private:
    light state_; // data member
};
```

Data hiding and access

- The concept expressed through the class is representation - independent
- Programs using a class should thus also not depend on representation
- Access declarators
  - public: only representation-independent interface, accessible to all
  - private: representation-dependent functions and data members
  - friend declarators allow related classes access to representation
- Note: Since all data members are representations of concepts (numbers, etc.) they should be hidden (private)!
- By default all members are private
  In a `struct` by default all are public

Member access

```
class Trafficlight  {
public:
  enum light
  { green, orange, red};

  Trafficlight();
  Trafficlight(light);
  ~Trafficlight();

  light state() const;
  void set_state(light);

private:
  light _state;
};
```

- Usage:
```
Trafficlight x;
Trafficlight::light l;

l = x.state();
l = Trafficlight::green;
```
- Members accessed with

  variable_name.member_name

- Type members accessed with

  class_name::member_name

  as they are not bound to specific object but common to all.

Special members

- Constructors
  - initialize an object
  - same name as class
- Destructors
  - do any necessary cleanup work when object is destroyed
  - have the class name prefixed by ~
- Conversion of object A to B
  - two options:
  - constructor of B taking A as argument
  - conversion operator to type B in A:
    - operator B();
- Operators
- Default versions exist for some of these

Illustration: a point in two dimensions

- Internal state:
  - x- and y- coordinates
  - is one possible representation
- Construction
  - default: (0,0)
  - from x- and y- values
  - same as another point
- Properties:
  - distance to another point
  - x- and y- coordinates
  - polar coordinates
- Operations
  - Inverting a point
  - assignment

```
class Point {
private:
  float x_,y_;
public:
  Point(); // (0,0)
  Point(float, float);
  Point(const point&);
  float dist(const Point& ) const;
  float x() const;
  float y() const;
  float abs() const;
  float angle() const;
  void invert() ;
  Point& operator=(const Point&);
};
```

Constructors and Destructors

- Let us look at the point example:
  - ```
    public:
      Point(); // default constructor
      Point(float, float); // constructor from two numbers
      Point(const Point&); // copy constructor
    ```
- Most classes should provide a default constructor
- Copy constructor automatically generated as memberwise copy, unless otherwise specified
- Destructor normally empty and automatically generated
- Nontrivial destructor only if resources (memory) allocated by the object. This usually also requires nontrivial copy constructor and assignment operator. (example: array class)
    ```

Default members

- Copy constructor
  `A::A(const A&);`
  defaults to memberwise copy if not specified

- Assignment operator
  `A::operator=(const A&);`
  also defaults to memberwise copy

- Destructor
  `A::~A()`
  defaults to empty function

Declaration, Definition and Implementation

- Declaration

```
class Point;
```

- Definition

```
class Point {
private:
  float x_,y_;
public:
  Point(); // (0,0)
  Point(float, float);
  ...
};
```

- Implementation

```
float Point::abs() const {
  return std::sqrt(x_*x_+y_*y_)
}
```

- Constructors

```
Point::Point(float x, float y)
 : x_(x), y_(y)
{} // preferred method
```

- or

```
Point::Point(float x, float y)
{ x_ = x; y_ = y;}
```

`const` and `volatile`

- const
  - Variables or data members declared as `const` cannot be modified
  - Member functions declared as `const` do not modify the object
  - Only `const` member functions can be called for `const` objects
- volatile
  - Volatile variables
    `volatile char hardware_register;`
    can be modified from outside the program! (example: I/O ports)
  - No optimization or caching allowed!
  - Only member functions declared `volatile` can be called for `volatile` objects

`mutable`

- Problem:
  - want to count number of calls to `age()` function of animal
- Original source:

```
class Animal() {
public:
  age_t age() const;
private:
  long     cnt_;
  age_type age_;
};

age_t Animal::age() const {
  cnt_++; // error: const!
  return age_;
}
```

- Solution:
  - mutable qualifier allows modification of member even in const object!
- Modified source:

```
class Animal() {
  …
private:
  mutable long cnt_;
  …
};

age_t Animal::age() const {
  cnt_++;  // now OK!
  return age_;
}
```

Operators as functions

- Most operators can be redefined for new classes
- Same as functions, with function name:
  operator *symbol*(...)
- Example:

```
Matrix A,B,C;
C=A+B;
```

- is converted to
  - either `C.operator=( A.operator+(B));`
  - or `C.operator= (operator+(A,B));`

Comments about operators

- Symmetric operators, e.g. +, -, … best implemented as functions:
  `Vector operator+(const Vector&, const Vector&);`
- Asymmetric operators, e.g. +=, -=, <<, >>,... best implemented as member functions:

```
class Vector {
    ...
    const Vector& operator +=(const Vector&);
  };
```

- Extensions to existing classes implemented as functions.
  Example: stream I/O:

```
std::ostream& operator <<(std::ostream&,
                           const Vector&);
```

- Assignment operators to class A have return type const A&.
  The reason is to allow:

```
Vector a, b, c;
a = b = c;
```

More comments about operators

- `A a; ++a;` uses
  - `const A& A::operator++();`
  - or `const A& operator++(A&);`
- `A a; a++;` uses
  - `const A& A::operator++(int);`
  - or `const A& operator++(A&,int);`
  - The additional int argument is just to distinguish the two
- `A b; b=a;` uses the assignment
  - `const A& A::operator=(const A&);`
  - or `const A& operator=(A&, const A&);`
- `A b=a;` and `A b(a);` both use the copy constructor
  - `A::A(const A&);`

Conversion operators

- conversion of `A -> B` as in:
  - `A a; B b=B(a);`
- can be implemented in two ways
  - constructor `B::B(const A&);`
  - conversion operator `A::operator B();`
- Automatic conversions:
  - `char -> int`
  - `unsigned -> signed`
  - `short -> int -> long`
  - `float -> double -> long double`
  - integer -> floating point
  - as in: `double x=4;`

friends

- Consider geometrical objects: points, vectors, lines,...
  - ```
    class Point {
    …
    private:
      float x,y,z;
    };

    class Vector {
    …
    private:
      float x,y,z;
    };
    ```
- For an efficient implementation these classes should have access to each others internal representation
- Using friend declaration this is possible:
  - ```
    class Vector;
    class Point {
    …
    private:
      float x,y,z;
      friend class Vector;
    };
    class Vector {
    …
    private:
      float x,y,z;
      friend class Point;
    };
    ```
- also functions possible:
  - `friend operator<<(…);`
  - `friend Point::invert(…);`

this

- Sometimes the object needs to be accessed from a member function
- `this` is a pointer to the object itself:
- ```
  const Array& Array::operator=(const Array& o) {
    if(this!=&o) {// do nothing if x=x;
      // do assignment
      ...
    }
    return *this;
  }
  ```
  ```

Inlining of member functions

- For speed issues member functions can be inlined
- Avoid excessive inlining as it leads to code-bloat
- Either in-class definition:

```
class complex {
    float re_, im_;

  public:
      float real() const
        {return re_;}

    float imag() const
        {return im_;}
      …
  };
```

- or out-of-class:

```
class complex {
   float re_, im_;

  public:
       inline float real() const;
       inline float imag() const;
     …
   };

   float complex::real() const
   {
     return re_;
   }

   float complex::imag() const
   {
     return im_;
   }
```

Static members

- are shared by all objects of a type
- exist even without an object
- thus :: notation used:
  `Genome::gene_number`
  `Genome::set_mutation_rate(2);`
- Static member functions can only access static member variables! Reason: which object's members to use???
- Useful for properties shared by all objects
- must be declared and defined!

```
class Genome  {
public:
 Genome(); // constructor

  static const unsigned short
                gene_number=64;
             // static data member
 Genome clone() const;

  static void set_mutation_rate
             (unsigned short);
private:
   unsigned long gene_;
   static unsigned short
               mutation_rate_;
   };

unsigned short
   Gene::mutation_rate_=2;  // definition
```

Inheritance

- is another very important feature
- it models the concept:
  objects of type B are the same as A, but in addition have…
- Examples
  - A shape is a 2D figure which has an area and can be drawn, although I know neither generally
  - A triangle is a shape, but its area is … and it looks like …
  - A square is a shape, but its area is … and it looks like …
  - A complex figure is a shape and consists of an array of shapes
  - A monoid is a semigroup, but in addition contains a unit element
  - A group is a monoid, but in addition has an inverse

Abstract base classes

- are good for expressing common ideas
- We want to have a function that for any shape draws it and prints its area:

```
void show(Shape& s) {
  std::cout << "The area of this shape is "
            << s.area() << "\n";
  s.draw(); // show it
}
```

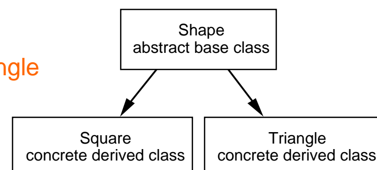
- This class must have an `area()` and a `draw()` member function

```
class Shape {
public:
  Shape() {};
  virtual double area() const =0;
  virtual void draw() const =0;
};
```

  - `virtual` means that this function depends on concrete shape
  - `=0` means that this function **must** be provided for any concrete shape

Concrete derived classes

- Triangles and Squares are both shapes:
  - ```
    class Triangle : public Shape {
    public:
      double area() const; // area of triangle
      void draw() const; // draws triangle
    };
    ```
  - ```
    class Square : public Shape {
    public:
      double area() const; // area of square
      void draw() const; // draws square
    };
    ```



- Examples
  - `Shape x;`
    // Error since it is abstract! `draw()` and `area()` not defined
  - `Square s;` // OK!
  - `Triangle t;` // OK!
  - `Shape& shape=s;` // also OK, since it is a reference!

Using inheritance

- recall the function `void show(Shape&);`
- let us call it for two shapes

```
Triangle t;
Square s;
show(t); // will use Triangle::area() and Triangle::draw()
show(s); // will use Square::area() and Triangle::area()
```

- All virtual function can be redefined by derived class
- In addition a derived class can define additional members
- There exists a third access specifier: `protected`
  - means `public` for derived classes
  - means `private` for others

Class templates

- same idea as function templates, classes for any given type T
- Learn it by studying examples:
  - Array of objects of type T
  - Complex numbers based on real type T
  - Statistics class for observables of type T

- Take care with syntax, where `<T>` must be used!
  - ```
    template <class T> class Array {
      …
      Array(const Array<T>&); // constructor!
      …
    };
    ```
  - ```
    template <class T>
    Array<T>::Array(const Array<T>& cp)
    { … }
    ```

The complex template

- The standard complex class is defined as a template

- `template <class T> class complex;`

- It is specialized and optimized for
  - `complex<float>`
  - `complex<double>`
  - `complex<long double>`
- but in principle also works for `complex<int>`, …

- it is a good exercise in template class design to look at the `<complex>` header

Inheritance emulated by templates

- We used inheritance before. The same could be done with templates:
  - ```
    template <class SHAPE> void show(const SHAPE& s) {
      cout << s.area() << "\n";
      s.draw();
    }
    ```
  - ```
    class Triangle {
    public:
      double area();
      void draw();
    };
    ```
  - ```
    Triangle t;
    show(t); // instantiates the template for triangle
    ```
- But type of `SHAPE` must be known at compile time!

Templates vs. inheritance

- Object oriented programming
- uses inheritance
- decision at run-time
- works for objects derived from the common base
- one function created for the base class -> saves space
- virtual function call needs lookup in type table -> slow
- extension possible using only definition of base class
- useful for application frameworks, user interfaces, high level structures

- Generic programming
- uses templates
- decision at compile-time
- works for objects having the right members
- a new function created for each class used -> wastes space
- no virtual function call, can be inlined -> fast
- extension needs definitions and implementations of all functions
- useful for small, low level constructs and generic algorithms
    ```

How to design classes

- Design was practiced in the exercises:
  - what are the logical entities (nouns)?
    - -> classes
  - what are the internal state variables ?
    - -> private data members
  - how will it be created/initialized and destroyed?
    - -> constructor and destructor
  - what are its properties (adjectives)?
    - -> public constant member functions
  - how can it be manipulated (verbs)?
    - -> public operators and member functions
- Now you should know how to convert this into classes!
- We will show more examples next week in the exercises. Please wait with coding the Penna model!

Do not avoid `typedef`!

- Do not store the age of an animal in an int
- Instead define a new type age_type

```
class Animal {
  public:
    typedef unsigned short age_t;
    age_t age() const;
  private:
    age_t age_;
};
```

- Allows easy modifications. If we want to allow older ages, just change the typedef to:

```
typedef unsigned long age_t;
```

- The rest of the code can remain unchanged!

References as return types

- **Warning!** What is wrong?

```
typedef Array<int> IA;
IA& operator+(const IA& x, const IA& y) {
  IA result=x;
  result+=y;
  return result;
}
IA a,b,c;
c=a+b;
```

- Problem: we return reference to temporary object!
  - Very dangerous, will in most cases crash the program
- Correct version copies the result

```
IA operator+(const IA& x, const IA& y) {
  Array<int> result=x;
  x+=y;
  return result;
}
```

static keyword

- has several meanings: “global” or “local to this file”
- in classes
  - shared “global” member variable
  - “global” member function
- in files: defines symbol visible only in this source file!
  - `static double pi=3.1415927;`
  - `static void error(char*);`
- in functions: value persists across function calls
  - ```
    double square(double x) {
      static cnt=0;
       cout << "Called " << ++cnt << "th time\n";
       return x*x;
     }
    ```
    ```