DM841

DISCRETE OPTIMIZATION

Elements of C++

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

C++ Features

- Declaration of a function or class that does not include a body only types (function prototype)
- Definition: declaration of a function that does include a body (implementation)

- Pointer variable: a variable that stores the address where another object resides: `int *m = nullptr;`
- Dynamic allocation via **new** operator. No garbage collector, we must free the memory via **delete**. Otherwise the memory is lost: Memory leak.
- Address-of operator: (&) (declares an lvalue reference, && declares rvalue reference, eg, x+y, "foo", 2; if an object has a name then it is an lvalue).

```
for( auto x: arr )
   ++x; \\ broken: assumes copy
```

```
for( auto & x: arr )
   ++x; // increment by one, ok!
```

C++ Features

Parameter passing:

- call-by-value:

```
double average( double a, double b );
double z = average( x, y )
```

- call-by-(lvalue)-reference:

```
void swap( double & a, double & b );
swap( x, y )
```

- call-by-reference-to-a-constant (or call-by-constant reference):

```
string randomItem( const vector<string> & arr );
```

1. Call-by-value is appropriate for small objects that should not be altered by the function
2. Call-by-constant-reference is appropriate for large objects that should not be altered by the function and are expensive to copy
3. Call-by-reference is appropriate for all objects that may be altered by the function

Temporary elements that are about to be destroyed can be passed by a call-by-rvalue-reference:

```
string randomItem( const vector<string> & arr );
string randomItem( vector<string> && arr );

vector<string> v { "hello", "world" };
cout << randomItem( v) << endl;
cout << randomItem( { "hello", "world" } ) << endl;
```

Often used in overloading of = operator, that can be implemented by a copy or a move

Return passing

```
LargeType randomItem1( const vector<LargeType > & arr )
{
   return arr[ randomInt( 0, arr.size() - 1 ) ];
}

const LargeType & randomItem2( const vector<LargeType > & arr )
{
   return arr[ randomInt( 0, arr.size() - 1 ) ];
}

vector<LargeType> vec;

LargeType item1 = randomItem1( vec ); // copy, return-by-value
LargeType item2 = randomItem2( vec ); // copy
const LargeType & item3 = randomItem2( vec ); // no copy, return-by-(lvalue)-constant
    -reference
```

Return passing

```
vector<int> partialSum( const vector<int> & arr )
{
  vector<int> result( arr.size() );
  result[ 0 ] = arr[ 0 ];
  for ( int i = 1; i < arr.size(); ++i)
    result[ i ] = result[ i-1 ] + arr[ i ];
  return result;
}

vector<int> vec;

vector<int> sums = partialSum( vec ); // copy in old C++, move in C++11
```

C++ Features

- Encapsulation (functions in the class as opposed to C)
- Constructors
- Rule of three: destructor, copy constructor, copy assignment operator (move constructor, move assignment)
- Public and private parts (and protected)
- Templates
- STL: `vector`
- (Some functions C-like: `string.c_str()` needed to transform a string in char array as in C)

Passing Objects

- In C++ objects are passed:
  - by value `F(A x)`
  - by reference `F(A& x)`
- In java objects are passed by reference, `F(A& x)`

In C++: `F(const A& x)` pass the object but do not change it.

If `F(A& x) const` the function does not change anything

Passing Objects

- In C++ objects are passed:
  - by value `F(A x)`
  - by reference `F(A& x)`
- In java objects are passed by reference, `F(A& x)`

In C++: `F(const A& x)` pass the object but do not change it.

If `F(A& x) const` the function does not change anything

Compare:

```
% vector<string> int2crs;
string Input::operator[](unsigned i) const { return int2crs[i]; }
string& Input::operator[](unsigned i) { return int2crs[i]; }
```

Inheritance

- General idea: extension of a class
- Example with `A` and `B` (next slide)
- Access level `protected`: only derived classes can see
- Hidden spaces: syntax with `::` (double colon), eg `std::cout`
- Hidden fields: syntax with `::` (double colon), eg `A::a1`
- Hidden methods (rewritten)
- Types of inheritance: public, private, and protected
- Invocation of constructors with inheritance: use of `:`
- Compatibility between base class and derived class (asymmetric)

```cpp
#include <iostream>

class A
{
public:
  A(int p1, double p2) { a1 = p1; a2 = p2; }
  int M1() const { return a1; }
  double a2;
protected: //not private
  int a1;
};

class B : public A
{
public:
  B(int p1, double p2, unsigned p3) : A(p1,p2) { b1 = p3; }
  unsigned B1() const { return b1; }
  void SetA1(int f)  { a1 = f; }
private:
  unsigned b1;
};

int main() // or (int argc, char* argv[])
{
  A x(1, 3.4);
  B y(-4, 2.3, 10);

  y.SetA1(-23);
  std::cout << y.a2 << " " << y.M1() << std::endl; // 2.3 -23
  return 0;
}
```

Polymorphism

One of the key features of class inheritance is that a pointer to a derived class is type-compatible with a pointer to its base class.

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
};

class Rectangle: public Polygon {
  public:
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
    int area()
      { return width*height/2; }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << '\n'; // 20
  cout << trgl.area() << '\n'; // 10
  return 0;
}
```

Polymorphism

```
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
   virtual int area() {return 0;}
};

class Rectangle: public Polygon {
  public:
    Rectangle(int a, int b) {width=a;
         height=b;}
    int area()
      { return width*height; }
};

class Triangle: public Polygon {
  public:
   Triangle(int a, int b) {width=a;
        height=b;}
   int area()
   { return width*height/2; }
};
```

```
int main () {
  Polygon * ppoly1 = new Rectangle (4,5)
        ;
  Polygon * ppoly2 = new Triangle (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  delete ppoly1;
  delete ppoly2;
  return 0;
}
```

Virtual functions

- Compatibility in case of redefined methods
- Late binding
- Pure virtual functions
- Abstract classes

Redefinition M1

```
class A {
int M1() {return a1;}
int a1
}

class B {
int M1() {return a1;}
int a1;
}


A a(,);
B b(,,);
x=b.M1();

cout<<x<<" "<<a.M1()<<endl;
```

Virtual functions

```
void F(A a) {
...
}

A x(,);
B y(,,);

F(y);
```

It calls method from class A. It copies an object of class B in A by removing what y had more. It doesn't even know that A exists

```
void F(A& a)
```

function for class A
It is not obvious which one of A or B it is going to use.
Eg. Persons (A) and student (B)

Methods are of two types:

- Final (in java) methods
- Virtual methods

If F is a virtual method it calls the last one defined.
Virtual $\rightsquigarrow$ Late binding makes binding between F and M late, ie, at execution time.

Pure virtual functions

We can have that the function is undefined in the parent class:

```
virtual int H() = 0;
```

pure virtual function, virtual function that is not defined but only redefined.

A becomes an abstract class hence we cannot define an object of class A. Like interfaces in java. There everything is virtual, here it is mixed.

Why? I might have different subclasses that implement the functions in different ways.

```
class A {
public:
  A(int p1, double p2) { a1 = p1; a2 = p2; }
  virtual int M1() const { cout << "A::M1"; return a1; }
  double a2;
  virtual int H() = 0;
protected:
  int a1;
};

class B : public A {
public:
  B(int p1, double p2, unsigned p3) : A(p1,p2) { b1 = p3; }
  unsigned B1() const { return b1; }
  void SetA1(int f)  { A::a1 = f; }
  int M1() const { cout << "B::M1"; return a2; }
protected:
  unsigned b1;
  vector<float> a1;
};

void F(A& a) {
  cout << a.M1() << endl;
}

int main() {
  A x(1,3.4);
  B y(-4,2.3,10);
  F(y);
  return 0;
}
```

Abstract Classes

They are classes that can only be used as base classes, and thus are allowed to have pure virtual member functions.

```
#include <iostream>
// abstract base class
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area (void) =0;
};

class Rectangle: public Polygon {
  public:
    int area (void)
      { return (width * height); }
};

class Triangle: public Polygon {
  public:
    int area (void)
      { return (width * height / 2); }
};
```

```
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon mypolygon;   // not working if
        Polygon is abstract base class
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

Casting

C++ is a strongly-typed language. Conversions, specially those that imply a different interpretation of the value, must be explicit, type-casting.
Note:

```
unsigned a,b;
unsigned x = abs((int) a - (int) b);
unsigned x = abs(static_cast<int> a - static_cast<int> b);
```

`static_cast<int>` instead of `(int)` (C-like syntax)
If used with pointers, no checks are performed during runtime to guarantee that the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe.

`dynamic_cast<int>`
can only be used with pointers and references to classes (or with void*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type.
Example: pointer upcast (converting from pointer-to-derived to pointer-to-base); pointer downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if -and only if- the pointed object is a valid complete object of the target type. Require run time checking and it is therefore more costly.

Templates

Generic programming

```
template <class T>
class mypair {
    T values [2];
  public:
    mypair (T first, T second)
    {
      values[0]=first; values[1]=second;
    }
};
```

(must be fully defined in the header files.)

```
mypair<int> myobject (115, 36);
mypair<double> myfloats (3.0, 2.18);
```