c'tor & d'tor, basics

Won't work: `int f(int x, int y = 0, int z);`

Every parameter after y must receive DV.

- allocated array? delete[] on d'tor!

Compiler won't distinguish between:

- `int g();`
- `double g();`

must have diff' amount/order of parameters!

`Complex arr[BIG];` -> calls default c'tor

```
A a1; -> default c'tor
A a2(2); -> c'tor A(int x)
A a3 = a1; -> copy c'tor
a2 = a3; -> operator=
```

const & static

const <- const : OK

const <- non-const : OK

non-const <- const : NOT OK

non-const <- non-const : OK

```
const T f(const T& t) const;
```

- const - return value is const
- *const* - t is const inside f's scope
- const (members only) - *this is const

static member/function can be access from both instance & class:

```
class A { public: static int x; };
```

- A::x
- A a; a.x;

friend

```
class A;
class B {
      friend A;
      // A can access B's private fields
};
```

Unless A has declared B as her friend, B can't access A's private fields!!!

```
class C {
      friend ostream& operator<<(ostream&
out, const C& c);
};
```

Here operator<< can access c's privates.
Friendship isn't transitive or symmetric.

תזכורת חשובה: אתה נמצא במבחן סוף בקורס תכנות מונחה עצמים עם המרצה ארז גרליץ. (נראלי)

operator overloading

```
A& operator++() { // prefix
      ++x;
      return *this;
}
A operator++(int) { // postfix
      A tmp = *this;
      ++(*this);
      return tmp;
}

A operator-() const { return A(-x); }
// member, may cause problems with 2 + a
A operator+(const A& rhs) const {
      return A(this->x + rhs.x);}

// non-member, A must declare friend/use get
A operator+(const A& lhs, const A& rhs) {
      return A(lhs.x + rhs.x);
}
// non-const and const versions,
int& operator[](int i) { return arr[i]; }
const int& operator[](int i) const {
      return arr[i];
}
```

Big 3

d'tor, copy c'tor, operator=

if you need one of them - implement all 3.

```
~A() { delete[] arr; }
A(const A& rhs) {
   if (this != &rhs) {
      if (arr) delete[] arr;
         arr = new int[rhs.n];
         for (int i = 0; i < n; i++)
             arr[i] = rhs.arr[i];
   }
}
A& operator=(const A& rhs) {
   if (this != &rhs) {
      if (arr) delete[] arr;
         arr = new int[rhs.n];
         for (int i = 0; i < n; i++)
             arr[i] = rhs.arr[i];
   }
       return *this;
}
```

templates

```
template<class T, int N> class Array;

template<class T, int N> ostream& operator<<
(ostream& out, const Array<T, N>& arr);
```

- Array<int, 1> and Array<int, 2> are different types!
- No implicit conversions
- C'tor declaration has no <>:
  `Array(int x=0) {}`
- Specialization - specify implementation for private cases:

```
template<int n> class A {
   friend class A<n - 2>;
};
template<> class A<5> {
   friend class A<10>;
};
```

Inheritance & Polymorphism

```
class Base {};
class Derived : public Base {};
```

Derived Is-A Base : Banana Is-A Fruit

	public	protected	private
Members	V	V	V
Derived			X
Unrelated		X	

- virtual function will call the most derived implemented function
- construction from Base to Derived, destruction from Derived to Base.
- base d'tor must be virtual!
- pure virtual function:
  `virtual void f() = 0;`

class with 1+ pure virtuals is abstract: can't create instances of same type.

subclasses of abstract class must implement all pure virtuals in order to be non-abstract.

```
class B {
      protected: int x;
      B(int x=0) : x(x) {} };
class D : public B  {
      D(int x=0) : B(x) {} };
```

STL

a.begin() – iterator, points to first item
a.end() – iterator, after last item

[first, last) – like range in python

common STL algorithms:

- reverse(BiDiIt first, BiDiIt last);
- for (auto x : a)
- sort(RandomIt first, RandomIt last, Compare comp);
-

Multiple Inheritance

```
class Base {};
class D1 : public Base {};
class D2 : public Base {};
class ZZ : public D1, public D2 {};
```

- order of calls:
- Base(D1), D1, Base(D2), D2, ZZ
- ~ZZ, ~D2, ~Base(D2), ~D1, ~Base(D1)
- use scope operator :: to differ same members names from different bases
- virtual inheritance:

```
class D1 : virtual public Base {};
class D2 : virtual public Base {};
```

- Base, D1, D2, ZZ
- ~ZZ, ~D2, ~D1, ~Base
- D1 and D2 have the same Base piece when inheriting virtually

Exceptions

```
#include <exception>

class myEx : public exception {
public:
      virtual const char* what() const
throw() { return "Exception!!!"; }
};

void f(int& x) {
      if (x == 0) throw myEx();
      x++;
}
```

Handle exception (in main):

```
try {
      int x;
      cin >> x;
      f(x);
}
catch (myEx& e) {
      cout << e.what();
}
```

Typing & Casting

`typename(b) == typename(d)`

- Cast via c'tor:
  `B(int i) { this->x = i; };`

Compiler now can implicitly cast int->B

- Prevent by adding explicit:
  `explicit B(int i);`
- Cast via operator:
  `operator int() const { return x; };`

- d = static_cast<D*>(b);
  - Run Time Error if cast failed
- d = dynamic_cast<D*>(b)
  - d == NULL if cast failed
- for pointers – use dynamic_cast only!

Functors

```
class Functor {
public:
      bool operator()(const A& a) {
            return a.getName() == "idan";
      }
};
class Comparator {
public:
bool operator()(const A& a1, const A& a2) {
      return a1.getName() < a2.getName();
}
};

find_if(arr, arr + 5, Functor());
sort(vec.begin(), vec.end(), Comparator());
```

- can also recieve parameters
- create new (tmp) instance in every use or construct instance beforehand

Singleton

```
class Singleton{
public:
	static Singleton & Instance() {
		static Singleton* instance = new
Singleton;
		return *instance;
	}
	// more public methods
private:
	Singleton() {}
	// c'tor, operator= : only declaration!
	Singleton(Singleton & old);
	const Singleton& operator=(Singleton& old);
	~Singleton() {}
	Attribute a;
};
```

- private static attribute
- public static accessor + “lazy init’”
- define all c’tor to be non-public
- the accessor is the only way for clients to manipulate the Singleton

Strategy

```
class B {
public:
	virtual void f() = 0;
	virtual ~B();
};
class B1 : B {
public: void f() { cout << "b1"; }
};
class B2 : B {
public: void f() { cout << "b2"; }
};
class Client {
	B& b;
public:
	Client(B& b) : b(b) {}
	void applyB() { b.f(); }
	void setB(B& b) { this->b = b; }
};
```

- allows client to contain */& to base class, so requests are “anonymous” – we don’t know which derived class is there

Abstract Factory

```
class Base {
public:
	virtual void f() = 0;
};
class Derived1 : public Base {
public:
	virtual void f() {}
};
class Derived2 : public Base {
public:
	virtual void f() {}
};
class AbstractFactory {
public:
static Base* NewInstance(const string& desc)
{
	if (desc == "D1")
		return new Derived1;
	if (desc == "D2")
		return new Derived2;
	return NULL;
}};
```

- interface with derived to instantiate
- factory method lets client defer which subclass to instantiate

Adapter

```
class A1 {
	int a;
public:
	A1(int a) : a(a) {}
	void f1() {}
};
class A2 {
public:
	virtual void f2() = 0;
};
class Adapter : public A1, public A2 {
public:
	Adapter(int a) : A1(a) {}
	virtual void f2() { f1(); }
};
```

Allows two incompatible classes to work together by converting the interface of one class into the interface expected by clients

Builder

```
class Object {
public: void setAttribute(const Attribute& a);
private: Attribute a;
};
class Builder {
public:
Object* getObject() { return o; }
void createNewObject() { o = new Object; }
virtual void buildAttribute() = 0;
protected:
	Object* o;
};
class SpecBuilder : public Builder {
public: virtual void buildAttribute() {
	o->setAttribute(/*specific value*/);
	}
};
/* using Builder */
Builder* builder;
builder = new SpecBuilder;
builder->createNewObject();
builder->buildAttribute();
delete builder;
```

Observer

```
class A {};
class AObserver {
public:
virtual void notify(vector<A*>& As);
virtual ~AObserver()
};
class Containes {
	vector<AObserver*> AObservers;
	vector<A*> As;
public:
	void addA(A* c) {
		As.push_back(c);
		for (auto observer : AObservers)
			observer->notify(As);
	}
	void attach(AObserver* observer) {
		AObservers.push_back(observer);
	}
};
```

- A list of observers notified when a relevant state is changes

Decorator

```
class Base {
public:
	virtual int f() = 0;
	virtual ~Base() {}
};
class Derived : public Base {
public:
	int f() { return 2; }
};
class BaseDecor : public Base {
	Base* b;
public:
	BaseDecor(Base* b) : b(b) { }
	~BaseDecor() { delete b; }
	int f() { return b->f(); }
};
class Decor1 : public BaseDecor {
public:
	Decor1(Base* b) : BaseDecor(b) { }
	int f() { return BaseDecor::f() + 3; }
};
class Decor2 : public BaseDecor {
public:
	Decor2(Base* b) : BaseDecor(b) { }
	int f() { return BaseDecor::f() + 4; }
};
Base* b = new Decor2(new Decor1(new Derived));
```

- like a polymorphic linked list
- allows to add changes dynamically to instances of some type without affecting other instances of the same type

Good Luck!!!
Etgar 16, 2021-2022

Template Method

```
class AbstractSort {
virtual bool needSwap(int, int) = 0;
void doSwap(int& a, int& b) { /*swap a, b*/ }
public:
// shell code
void sort(int v[], int n) {
	for (int i = 0; i < n - 1; i++)
		for (int j = 0; j < n - i - 1; j++)
			if (needSwap(v[j], v[j + 1]))
				doSwap(v[j], v[j + 1]);}
};
class SortDescending : public AbstractSort {
	bool needSwap(int a, int b) {
		return a < b;
	}
};
```

- a method in a superclass (usually abstract) which defines the skeleton of an operation in terms of a few high level steps. These steps are themselves implemented by additional helper methods in the same class as the template method