C++ lab 5

Recap of few concepts you used during the fifth lab:

1. 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. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature.
- A virtual member is a member function that can be redefined in a derived class, **while preserving its calling properties through pointers and references**. The syntax for a function to become virtual is to precede its declaration with the `virtual` keyword.

```
struct A {
            void  print() const { std::cout << "A::print()  "; }
    virtual void vprint() const { std::cout << "A::vprint()  "; }
};
struct B: A {
            void  print() const { std::cout << "B::print()  "; }
    virtual void vprint() const { std::cout << "B::vprint()  "; }
};
int main() {
    B b;
    const A& rb = b;
    const A* pb = &b;

    b.print();  b.B::print();  b.A::print();  /* B::print()  B::print()  A::print() */
    b.vprint(); b.B::vprint(); b.A::vprint(); /* B::vprint() B::vprint() A::vprint() */

    rb.print();  rb.vprint();  /* A::print() B::vprint() */
    pb->print(); pb->vprint(); /* A::print() B::vprint() */
}
```

- Sometimes implementation of all function cannot be provided in a base class because we do not know their implementation. A **pure virtual method** or **abstract method** is a virtual function that is required to be implemented by a derived class if the derived class is not abstract. Classes containing (defined or inherited) pure virtual methods are termed **abstract** and they cannot be instantiated directly.
  A pure virtual function is declared by assigning 0 in its declaration (but may also be implemented).

```
struct A     {  virtual void print() = 0;               };
struct B : A {  void print() { std::cout << "B "; }  };
struct C : A {  void print() { std::cout << "C "; }  };
struct D : B {  void print() { std::cout << "D";
                               this->B::print(); }   };

int main() {
    //  A a;  /* illegal because A is an abstract class */
    B b;  b.print();  /* B  */
    C c;  c.print();  /* C  */
    D d;  d.print();  /* DB */

    A* ptrs[3] = { &b, &c, &d };
    for(int i=0; i<3; i++) {
        ptrs[i]->print();
    }
    /* B C DB */
}
```

- An **interface** (or pure abstract class) is a class that only pure virtual member functions and no data or concrete member functions. In general, a pure abstract class is used to define an interface and is intended to be inherited by concrete classes. In the example above, the class `A` is an interface.

- Once you are using virtual members (i.e. once your type is polymorphic) you should use the `dynamic_cast` operator to downcast your types instead of the classic `static_cast` operator (which is less safe because it performs no runtime checks):

```
struct A    { virtual void f() {}; };
struct B: A {};
int main() {
B b;
A& rb0 = b;                       /* ok, implicit upcast from B& to A&   */
A* pb0 = &b;                      /* ok, implicit upcast from B* to A*   */
B& rb1 = dynamic_cast<B&>(rb0);   /* ok, explicit downcast from A& to B& */
B* pb1 = dynamic_cast<B*>(pb0);   /* ok, explicit downcast from A* to B* */

A a;
A& ra0 = a;                       /* ok, no cast required here                 */
A* pa0 = &a;                      /* ok, no cast required here                 */
B& ra1 = dynamic_cast<B&>(ra0);   /* ko, runtime exception, throws std::bad_cast */
B* pa1 = dynamic_cast<B*>(pa0);   /* ko, dynamic_cast returns a null pointer     */
}
```

- How to safely check dynamic pointer downcasts ?

```
B* pa1 = dynamic_cast<B*>(pa0);
if (pa1 != nullptr) {
    /* ok pa0 pointed to an object type-compatible with B */
}
else {
    /* ko pa0 pointed to a type that is not a B */
}
```

- How to safely check dynamic reference downcasts ?

```
#include <typeinfo> /* std::bad_cast */
try {
    B& ra1 = dynamic_cast<B&>(ra0);
    /* ok ra0 was a reference to an object type-compatible with B */
} catch (std::bad_cast& bc) {
    /* ko ra0 was a reference to a type that is not a B */
}
```

- Since `C++11`, you can use the `override` specifier to specify that a virtual function overrides another virtual function (this just enforces a compile time check):

```
class A {
    virtual void foo();
    void bar();
};
class B : A {
    void foo()       override; /* ok, foo() overrides virtual A::foo()         */
    void foo() const override; /* ko, no override because of signature mismatch */
    void fooo()      override; /* ko, no override because of name mismatch      */
    void bar()       override; /* ko, A::bar() is not virtual                   */
}
```

- Since `C++11`, you can also use the `final` specifier to specify that a virtual function override is the final one (like for inheritance):

```
class A {
    virtual void foo();
    void bar();
};
class B : A {
    void foo() final; /* ok, foo() is virtual */
    void bar() final; /* ko, non-virtual function cannot be overriden */
}
class C: B {
    void foo() override; /* ko, foo() cannot be overriden because it was declared final */
}
```

- There exist a powerful technique for customizing the behavior of polymorphic classes: `cross delegation` (also called delegation to a sister class).

```
struct A { /* interface */
    virtual void f() const = 0;
    virtual void g() const = 0;
};
struct B : virtual A {
    void f() const final override { std::cout << "f() "; };
};
struct C : virtual A {
    void g() const final override { std::cout << "g()->"; f(); };
};
struct D final: B,C {};
int main() {
    D d;

    B& b = d;
    b.f(); /* f()      */
    b.g(); /* g()->f() */

    C& c = d;
    c.f(); /* f()      */
    c.g(); /* g()->f() */
}
```

- Virtual members may change the size of your types (most likely because the compiler will add a hidden member variable to the class called a vtable). For example you can compare the output of this program to the one without the `virtual` methods from the recap of the lab 4:

```
#include <iostream>
#include <algorithm> /* std::fill_n, fill an array with a constant */
struct A    {
    char i[1024];
    A(char a=0) { std::fill_n(i, 1024, a); }; // just fill array i with the value of a
    virtual void print() const { std::cout << "A" << int(i[1023]) << ' '; }
};
struct B: A {
    char j[1024];
    B(char a=1, char b=2) : A(a) { std::fill_n(j, 1024, b); };
    virtual void print() const { std::cout << "B" << int(j[1023]) << ' '; }
};
int main() {
    std::cout << "Size of type A is " << sizeof(A) << '.' << std::endl;  // 1032 = 1024 + 8 bytes
    std::cout << "Size of type B is " << sizeof(B) << '.' << std::endl;  // 2056 = 2048 + 8 bytes

    A   a0;        a0.print();    // A0
    A& ra0 =  a0;  ra0.print();   // A0
    A* pa0 = &a0;  pa0->print();  // A0

    B   b0;        b0.print();    // B2
    B& rb0 =  b0;  rb0.print();   // B2
    B* pb0 = &b0;  pb0->print();  // B2

    /* example of implicit upcasts from B to A, B& to A& and B* to A* */
    A  a1  =  b0;  a1.print();    // A1
    A& ra1 =  b0;  ra1.print();   // B2 (dynamic dispatch of print())
    A* pa1 = &b0;  pa1->print();  // B2 (dynamic dispatch of print())

    /* example of valid explicit downcasts A& to B& and A* to B* */
    B& rb1 = dynamic_cast<B&>(ra1);  rb1.print();  // B2, ok because ra1 references b0 of type B.
    B* pb1 = dynamic_cast<B*>(pa1);  pb1->print(); // B2, ok because pa1 points to b0 of type B.

    /* example of bad explicit downcasts from A& to B& and A* to B* */
    B& rb2 = dynamic_cast<B&>(ra0);  rb2.print();  // runtime error, std::bad_cast (C++ exception)
    B* pb2 = dynamic_cast<B*>(pa0);  pb2->print(); // runtime error, returns a nullptr (segfault)
}
```

2. Four ways to iterate over a `std::vector` or a `std::array`:

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v = { 0,1,2,3 }; /* C++11 style initialization */
    /* C++98 */
    for(unsigned int i=0; i<v.size(); ++i) {
        std::cout << v[i] << std::endl;
    }
    for(std::vector<int>::const_iterator it = v.cbegin(); it!=v.cend(); ++it) {
        std::cout << *it << std::endl;
    }
    /* C++11 */
    for(int k : v) {
        std::cout << k << std::endl;
    }
    for(auto k : v) {
        std::cout << k << std::endl;
    }
}
```

3. Four ways to iterate over a `std::map`:

```
#include <iostream>
#include <map>
int main() {
    std::map<int, char> m = { {0, 'a'}, {1, 'b'}, {42, 'c'} }; /* C++11 style initialization */

    /* C++98 */
    for(std::map<int, char>::const_iterator it = m.cbegin(); it!=m.cend(); ++it) {
        std::cout << it->first << ' ' << it->second << std::endl;
    }
    /* C++11 */
    for(const std::pair<int, char>& p : m) {
        std::cout << p.first << ' ' << p.second << std::endl;
    }
    for(const auto& p : m) {
        std::cout << p.first << ' ' << p.second << std::endl;
    }
    /* C++17 */
    for(const auto& [k, v] : m) {
        std::cout << k << ' ' << v << std::endl;
    }
}
```

4. Miscellaneous
   - As usual, you can enable a specific standard with the compiler flags `-std`. For example to enable `C++14` just add `-std=c++14` to your build options. To enable `nullptr` you will need at least `-std=c++11`. Old compilers may not support most recent standards (`C++11`, `C++14`, `C++17`, `C++20`).
   - Add the `-fmax-errors=N` compiler option to your `CXXFLAGS` variable to tell the compiler to give up after `N` errors. Use `-fmax-errors=1` if you do not want to scroll hundreds of error messages to find the first error ! If you are using `clang` you may need to use another compiler option: `-ferror-limit=N`.
   - You can use the `nm` utility to look at the symbols that have been compiled into your object files (or libraries). For example to see what symbol is contained in `main.o`, just use `nm -C main.o`
   - During those 5 labs, you just saw the tip of the iceberg of `C++` ! Here are the links to have an overview on all `C++` language features and the standard library.