Arrays, Strings, Pointers

Reasons for learning about pointers

- All C programmers use them heavily, so if you want to use a C library you'll end up using pointers.
- It is assumed that you are familiar with pointers so lots of C++ classes are designed so that using them feels just like working with pointers.
- They allow you to store objects of different types inside a data structure. This is essential for polymorphism.

Arrays, the C alternative to vectors

```
// Create an unintialized of length 5
int myArray[5];
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- Create an array of 5 integers, without initialising it.
- Run through the entries and print them out.
- Just as with a vector, the entries start at 0.
- Just as with a vector we use [] to access entries.
- There is no size function.

Initialising an array

```
// Create an initialised array
int myArray[] = {1, 1, 2, 3, 5};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We can initialise an array by specifying the values.
- Simply place the values in a comma separated list between curly brackets.
- Notice that we no longer have to specify the length of the array when we create it.

Initialising an array to zero

```
// Create an initialised array
int myArray[5] = {};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- We specify the size of the array.
- We assign it the value {}.
- This gives an array of the desired length full of zeros.

General initialisation

```
int myArray[5] = {1,2,3};
for (int i=0; i<5; i++) {
    cout<<"Entry "<<i<<"=";
    cout<<myArray[i];
    cout<<"\n";
}
```

- This prints out the values 1, 2, 3, 0, 0.
- The length of the array is specified.
- Some of the values are specified.
- The rest is padded with zero.

Passing arrays to functions

```
int sumArray( int toSum[], int length ) {
    int sum = 0;
    for (int i=0; i<length; i++) {
        sum+=toSum[i];
    }
    return sum;
}
```

- As well as passing the array, pass the length. Having the array without knowing its length is useless.

Don't return arrays

- Do NOT return arrays from functions.
- When a function returns, all the variables it has created are removed from memory. This includes arrays.
- If you attempt to return an array, the behaviour is undefined.
- The caller just receives a pointer to *where the array used to be*. The computer may have reused that memory for almost anything.

You can't vary the length of an array

- You cannot change the length of an array.
- You cannot insert a new item or add some at the end.
- In fact the size is fixed AT COMPILE TIME!

Multi-dimensional arrays

```
// Create an initialised 3x5 array
int myArray[][5] = {{1, 2, 3, 4, 5},
                    {2, 0, 0, 0, 0},
                    {3, 0, 0, 0, 0}};
for (int i=0; i<3; i++) {
    for (int j=0; j<5; j++) {
        cout<<"Entry ("<<i<<","<<j<<")=";
        cout<<myArray[i][j];
        cout<<"\n";
    }
}
```

- The one strange thing here is that you have to specify all the dimensions but the first by hand.

Summary

- Arrays are a bit like vectors
- You can't return them from functions.
- You always need to pass their length as well as the array.
- You can't change their size. You have to choose it when you write the program (*at compile time*) rather than in response to the user (*at run time*).
- Because of the last feature, they are almost completely useless except for defining constants and data for tests.

Tip: Avoid arrays

Use `vector` instead of an array.

new[] - Working with memory directly

```
int n = 5;
int* myArray = new int[n];
for (int i=0; i<n; i++) {
    cout<<"Entry "<<i<<"=";
    cout << myArray[i];
    cout << "\n";
}
delete[] myArray;
```

- We use the `new ...[]` operator to allocate a chunk of memory.
- The good thing is you can choose the size at runtime.
- The memory crated will NOT be automatically deleted when the function exits.
- You must use `delete []` operator to manually delete everything you create with the `new[]` operator.

Using new[] and delete[]

- The data returned by `new int[n]` is called a *pointer*.
- It has type `int*` which means "a pointer to an `int`".
- The default constructor for the data will be called. Since `int` data is randomly initialised, the memory in this example will be randomly initialised.
- In fact the array we used before was of type `int*`. It's just that the notation for arrays in C hides this fact.
- All that the `int*` myArray contains is the memory address where the array starts.
- We have to remember ourselves that the block of memory is of length `n`.
- You can use `[]` with a pointer to find the integer at a given offset.

New[] with other data types

- You can use `new[]` to create blocks of memory with whatever type of object you like.
- We'll use the following `Pair` class in examples

```
class Pair {
public:
    double x;
    double y;
    Pair();
    Pair( double x, double y );
};
```

New[] a set of Pairs

```
int n = 5;
Pair* myPairs = new Pair[n];
for (int i=0; i<n; i++) {
    double xValue = myPairs[i].x;
    double yValue = myPairs[i].y;

    cout<<"Pair (";
    cout<< xValue;
    cout<<",";
    cout<< yValue;
    cout<<")\n";
}
delete[] myPairs;
```

- The type is a `Pair*`, a pointer to a `Pair`.
- The default constructor is called. So, in this case the points are all at $(0, 0)$

Working with pointers

- A pointer is just the address in memory of some data.
- On a 32-bit computer a pointer will be 4 bytes long. On a 64-bit computer it will be 8 bytes long.
- This means a 32-bit computer can have up to 2^{32}bytes $\approx 2Gb$ of memory.
- 64-bit computers could in principle have vastly more memory.
- It is normal to write memory addresses as a hexadecimal number. e.g. `9ABF0132` is a typical memory address.

Pointer basics

```
int myVariable = 10;
int* pointerToMyVariable = &myVariable;

cout << "Memory location of myVariable ";
cout << pointerToMyVariable;
cout << "\n";

cout << "Value of myVariable ";
cout << (*pointerToMyVariable);
cout << "\n";
```

Running this on my 32 bit computer, I got the following output:

```
Memory location of myVariable 0013FE78
Value of myVariable 10
```

The operators & and *

- Use `&` to find the memory address of a variable, i.e. to obtain a pointer.
- Use `*` to find the value at a memory address, i.e. find out what is being pointed to.
- So `*p` means the same as `p[0]`.
- This is a completely different use of `&` from the use in pass by reference!
- When used as part of a type definition `&` means "reference".
- When used as an operator `&` means "memory location of".

The operator ->

```
Pair p;
Pair* pointerToP = &p;

// Use -> to access fields via a pointer
pointerToP->x = 123.0;
pointerToP->y = 456.0;

// We check that c has changed
ASSERT( p.x==123.0 );
ASSERT( p.y==456.0 );

// You could use * and .
ASSERT( (*pointerToP).x==123.0 );
ASSERT( (*pointerToP).y==456.0 );
```

- Use -> to access fields of an object via a pointer.
- This is equivalent to using * and . in combination, but is easier to read.

Working with pointers

```
int sumUsingPointer( int* toSum, int length ) {
    int sum = 0;
    for (int i=0; i<length; i++) {
        sum+=toSum[i];
    }
    return sum;
}
```

- We specify the type of the parameter as `int *`.
- The code here is identical to that with arrays except that we declare the type using `*` rather than `[]`.
- Note that you have to pass the number of elements as well as the pointer.

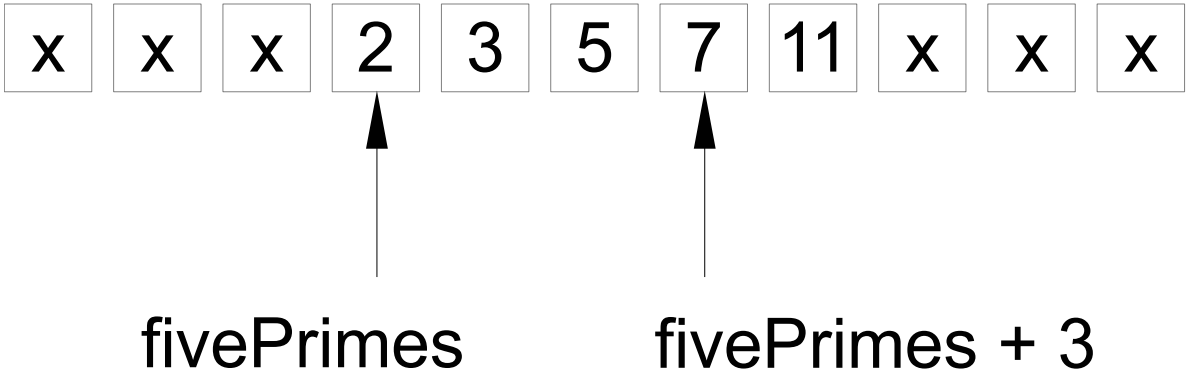
Using the ++ operator with pointers

```
int sumUsingForAndPlusPlus( int* begin, int n) {
    int sum = 0;
    int* end = begin + n;
    for (int* ptr=begin; ptr!=end; ptr++) {
        sum += *ptr;
    }
    return sum;
}
```

- You can use ++ to move a pointer on to the next item.
- You can use == to compare pointers.
- This code is equivalent to the last one, we just use ++ instead of arithmetic.
- Maybe this will be a tiny bit faster, since adding 1 to a number is easier than adding an arbitrary number?

Mental picture

Mental picture



Pointer arithmetic

```
int sumUsingPointerArithmetic( int* toSum,
                               int length ) {
    int sum = 0;
    for (int i=0; i<length; i++) {
        int* ithElement = toSum + i;
        int valueOfIthElement = *ithElement;
        sum+= valueOfIthElement;
    }
    return sum;
}
```

- You can perform arithmetic on pointers.
- `p[23]` is the value pointed to by `p+23`.
- So pointer arithmetic is an alternative to using `[]`.

Coping with pointers sensibly

- Since we always need the number of elements as well as the pointer it seems wise to introduce a class such as:

```
class IntArray {
public:
    int* firstElement;
    int  length;
};
```

- We could give this class a helpful function `size`.
- Of course this class *already exists*! It is a `vector<int>`.

`new[]` summary

- Pointers and `new []` allow you to recreate the behaviour of a `vector` but with more conceptual overhead.
- You must `delete[]` anything you create with `new []`.
- You cannot simply insert new values into data created with `new[]`.
- You must be careful only to reference valid memory when using `[]` otherwise you may get very unpleasant errors.

Tip: Avoid new []

You should avoid using `new []` and simply work with `vectors` instead. The only possible exception might be if you believed you could squeeze a bit more performance out of accessing raw memory. Unlikely.

Pointers to text

```
const char* charArray2 = "Hello";
for (int i=0; i<6; i++) {
    cout << "ASCII VALUE ";
    char c = charArray2[i];
    cout << ((int)c);
    cout << "\n";
}
```

Text in C

- In the C language, the standard was to represent text using a block of memory containing characters terminated by the ASCII code 0.
- ASCII is the coding used for characters. The code for 'A' is 65, the code for '1' is 49, the code for '0' is 48.
- The code 0 doesn't represent any character, so it can be used to mark the end.
- The C language provides a short-cut for creating an array of characters ending with the code zero. Just place the desired characters in double quotes.
- These are called "C-style strings" or "null-terminated strings"

Working with null-terminated strings

```
int computeLengthOfString( const char* s ) {
    int length = 0;
    while ((*s)!=0) {
        s++;
        length++;
    }
    return length;
}

void testComputeLengthOfString() {
    const char* quotation="To be or not to be";
    int l1 = computeLengthOfString( quotation );
    int l2 = strlen( quotation ); // built in
    ASSERT( l1==l2 );
}
```

- C contains various functions to help work with null-terminated strings.
- `strlen` computes the length. `strcpy` copies one string into another.

The difficulties of working with memory

Be careful not to access data outside the array

```
char* shortText = new char[20];
for (int i=0; i<1000; i++) {
    shortText[i] = 'x';
}
delete[] shortText;
```

- This code will behave unpredictably. It will probably crash horribly.
- When you use a `string` in debug mode, various checks are made so you at least get a somewhat helpful error message.

Be careful not to return an array

```
char* thisFunctionReturnsAnArray() {
    /* This produces a compiler warning */
    char text[] = "Don't do this";
    return text;
}

void someOtherFunction() {
    char text[] = "Alternative text\n";
    cout << text;
    cout << "\n";
}

void testDontReturnArrays() {
    char* text = thisFunctionReturnsAnArray();
    someOtherFunction();
    cout << text;
    cout << "\n";
}
```

Returning pointers

- Note that the text is an `char` array, so it will be deleted the moment the function exits.
- The code on the previous slide will behave unpredictably. It probably will print some junk if you run it.
- You can return a `string` without a problem.
- You are allowed to return a pointer created with `new []`, but then you'll have to make sure the caller knows whether or not they will be expected to call `delete[]` at some point.
- By convention in C and C++, if a function returns a pointer, the caller is NOT expected to call `delete[]`. For example, if you call `c_str()` on a `string` you shouldn't `delete[]` what it returns.

```
char* thisFunctionReturnsAPointer() {
    char text[] = "This works";
    int n = strlen(text);
    char* ret = new char[n+1];
    /* We now get a compiler warning here */
    strcpy( ret, text );
    return ret;
}

void testReturnPointerJustAboutOK() {
    char* text = thisFunctionReturnsAPointer();
    someOtherFunction();
    cout << text;
    cout << "\n";
    // don't forget to free the memory
    delete[] text;
}
```

This violates the convention on NOT deleting the return value of a function, so it is considered to be confusing code.

Text in C++

- Initialising text using double quotes is too tempting to resist, so we allow that in C++ too.
- We instantly convert the `char*` data into a `string` instance.
- The `string` instance stores the actual length as a member variable.
- This is *much* more efficient than having to look at every character of a string every time you want to know its length!

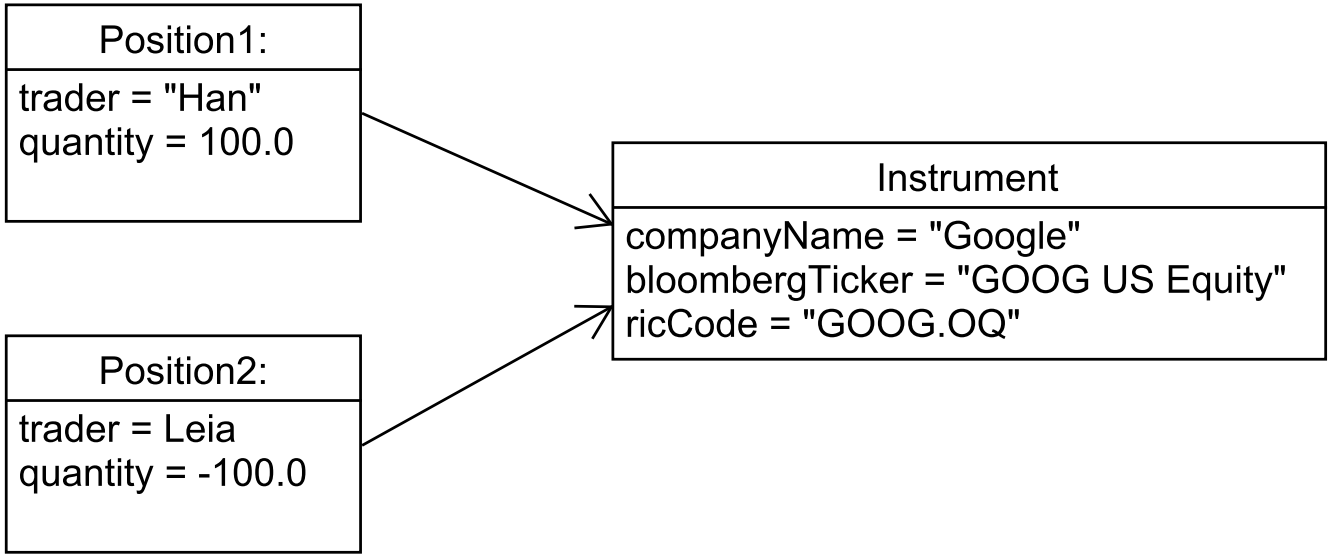
Tip: Avoid `char*`

Use text in quotation marks, but use `string` everywhere else. If you need to create a `char*` to call a legacy function, use the `c_str()` function of `string`.

Sharing data

Example

We have a class `Instrument`. It contains lots of data about a traded instrument, for example, the type of the instrument, the Bloomberg code, the Reuter's code, etc.
We have another class `Position`, which consists simply of an instrument, the quantity held in that instrument, and the name of the trader who has taken the position. We save memory by reusing the same `Instrument` instances.



new and delete

- To create a single object use `new`.
- When you no longer need it call `delete`.
- Be very careful to use `delete[]` when you have used `new []` and `delete` when you have used `new`.

```
Pair* myPair = new Pair;
myPair->x = 1.3;
myPair->y = 2.5;

cout << "Pair (";
cout << (myPair->x);
cout << ", ";
cout << (myPair->y);
cout << ")\n";

delete myPair;
```

Sharing an instrument

```
class Instrument {
public:
    string bloombergTicker;
    string ricCode;
    string companyName;
    Instrument() {}
};

class Position {
public:
    string trader;
    double quantity;
    Instrument* instrument;
    explicit Position( Instrument * instrument );
};

Position::Position( Instrument* instrument ) :
    instrument( instrument ) {
}
```

Remarks

- Note that the Position contains a *pointer* to an instrument and not an actual instrument of its own.
- **WARNING:** If you don't initialise a pointer it will fail badly when you try to use it.

```
// Don't do this
Instrument* instrument;
cout << instrument->companyName << "\n";
```

- This is why we use an `Instrument*` to construct a `Position`

- Sometimes you wish to specify that a pointer doesn't yet point to anything, in which case you initialise it to `nullptr`.

```
string getCompanyName( Position& position ) {
    if (position.instrument==nullptr) {
        return "Name not set";
    } else {
        return position.instrument->companyName;
    }
}
```

However, if you use `nullptr` there is a danger that someone might forget the check, resulting in a nasty error.

Code to create a vector of positions

```
vector<Position> constructPositions() {
    // the caller of this function
    // should call delete on the instrument
    // when they are done with all the positions
    vector<Position> positions;

    Instrument* instrument = new Instrument;
    instrument->companyName = "Google";
    instrument->bloombergTicker = "GOOG US Equity";
    instrument->ricCode = "GOOG.OQ";

    Position p1(instrument);
    p1.trader = "Han";
    p1.quantity = 100.00;
    positions.push_back( p1 );

    Position p2(instrument);
    p2.trader = "Leia";
    p2.quantity = -100.00;
    p2.instrument = instrument;
    positions.push_back( p2 );

    return positions;
}
```

Problem

```
void testConstructPositions() {
    vector<Position> r = constructPositions();
    int n = r.size();
    for (int i=0; i<n; i++) {
        cout << "Position "<<i<<"\n";
        Position& p=r[i];
        cout << "Trader "<<p.trader<<"\n";
        cout << "Quantity "<<p.quantity<<"\n";
        cout << "Instrument ";
        cout << p.instrument->companyName<<"\n";
        cout << "\n";
    }
    delete r[0].instrument;
}
```

The caller of `constructPositions` has to know precisely how to call `delete`. This violates information hiding—you need to know how `constructPositions` actually works.

Solution - shared_ptr

- Whenever you use `new`, store the result using a `shared_ptr`.
- A smart pointer is a C++ class which behaves like a pointer but which handles working out when to call `delete` on your behalf. `shared_ptr` is the most useful smart-pointer class.
- `shared_ptr` keeps track of how often it has been copied. Once the number of copies of the smart pointer in existence drops to zero it calls `delete`
- (More precisely, just before the last remaining copy of the smart pointer is removed from memory, it calls `delete`.)

Position version 2

Here is a new version of the `Position` class which uses a `shared_ptr`:

```
class PositionV2 {
public:
    string trader;
    double quantity;
    shared_ptr<Instrument> instrument;
    explicit PositionV2( shared_ptr<Instrument> ins );
};

PositionV2::PositionV2( shared_ptr<Instrument> ins ) :
    instrument( ins ) {
}
```

Using shared_ptr

```
vector<PositionV2> constructPositionsV2() {
    vector<PositionV2> positions;

    shared_ptr<Instrument> ins
          = make_shared<Instrument>();
    ins->companyName = "Google";
    ins->bloombergTicker = "GOOG US Equity";
    ins->ricCode = "GOOG.OQ";

    PositionV2 p1(ins);
    p1.trader = "Han";
    p1.quantity = 100.00;
    positions.push_back( p1 );

    PositionV2 p2(ins);
    p2.trader = "Leia";
    p2.quantity = -100.00;
    p2.instrument = ins;
    positions.push_back( p2 );

    return positions;
}
```

Using a shared_ptr

- We have to initialise the `shared_ptr` by calling `make_shared`.
- After that, using a `shared_ptr` is just like using a pointer. In particular, `*` and `->` still work.

The payoff

```
void testConstructPositionsV2() {
    vector<PositionV2> r = constructPositionsV2();
    int n = r.size();
    for (int i=0; i<n; i++) {
        cout << "Position "<<i<<"\n";
        PositionV2& p=r[i];
        cout << "Trader "<<p.trader<<"\n";
        cout << "Quantity "<<p.quantity<<"\n";
        cout << "Instrument ";
        cout << p.instrument->companyName<<"\n";
        cout << "\n";
    }
}
```

We no longer call `delete` in this code. Information hiding has been saved!

Pointers summary

- Don't use pointers directly, use `shared_ptr` instead.
- Use a `shared_ptr` when you need to create long-lived objects that are shared by different objects. Hence the name `shared_ptr`, of course.
- Always initialise `shared_ptr` instances, otherwise you will see some very nasty errors.

References revisited

- A lot of what you can do with a pointer, you can do with a reference.
- You can have a member variable which is a reference.
- By storing data by reference you save memory, just as with pointers.
- You *must* initialise member variables which are references in the constructor.
- Owning a `shared_ptr` to an object means that you are guaranteed the object won't be deleted until you no longer need it.
- If you use a reference, there's a danger that someone else might delete your object.

How NOT to use references as a member variable

```
class PositionV3 {
public:
    string trader;
    double quantity;
    Instrument& instrument;
    explicit PositionV3( Instrument& instrument );
};

PositionV3::PositionV3( Instrument& instrument ) :
    instrument(instrument) {
}
```

- This code is technically correct. Note that you *must* initialise a member variable reference in the constructor.
- The problem is that the `Position` class has no control over when the `instrument` might be deleted.

How NOT to use references as a member variable

```
PositionV3 constructPositionV3() {
    // This function doesn't work, the instrument
    // is deleted, so all the returned positions
    // contain broken references
    vector<PositionV3> positions;

    Instrument instrument;
    instrument.companyName = "Google";
    instrument.bloombergTicker = "GOOG US Equity";
    instrument.ricCode = "GOOG.OQ";

    PositionV3 position(instrument);
    position.trader = "Han";
    position.quantity = 100.00;
    return position;
}
```

- This code is a disaster waiting to happen. We're creating a Position that appears to be initialised, but then immediately deleting the Instrument it refers to.
- We're not consciously deleting the instrument, it just happens as part of automatic cleanup when the method exits.

The disaster

```
void testConstructPositionV3() {
    // This will fail horribly
    PositionV3 p = constructPositionV3();
    cout << "Trader "<<p.trader<<"\n";
    cout << "Quantity "<<p.quantity<<"\n";
    cout << "Instrument ";
    cout << p.instrument.companyName<<"\n";
    cout << "\n";
}
```

- You can run this code by uncommenting the appropriate line in the `main` method.
- There is no guarantee what it will do.

Some acceptable code using a reference as a member

```
double integralToInfinity(RealFunction& f,
    double lowerLimit, int nPoints) {

    class DefiniteIntegrand : public RealFunction {
    public:
        RealFunction& g;
        double lowerLimit;

        DefiniteIntegrand(RealFunction& g,
                          double lowerLimit) :
            g(g), lowerLimit(lowerLimit) {
        }

        double evaluate(double x) {
            return (1/(x*x))
                * g.evaluate(lowerLimit - 1 + (1/x));
        }
    };

    DefiniteIntegrand integrand(f, lowerLimit);
    return integral(integrand, 0, 1, nPoints);
}
```

Why this is acceptable

- We know that our `DefiniteIntegrand` instance will only be kept in memory until the `integralToInfinity` method returns.
- The caller has given us a reference, so they are promising that it won't be deleted until the function returns.
- So we can safely use the reference.

Summary

- Pointers, references and `shared_ptr` can be used to achieve similar things.
- References are a bit safer than pointers. For example, you can't call `delete` on a reference, and you *must* initialise reference variables.
- `shared_ptr` is a bit slower than a reference or a pointer, but is really the only viable option for long-lived data.
- In summary:
  - Use references if you are sure that the automatic deletion of local variables won't be a problem.
  - Use `shared_ptr` if you want to create long-lived data.
  - Don't use raw pointers.
  - If your head is hurting and you don't want to think about it right now, use `shared_ptr`