LECTURE 11 · CSE 333 · Summer 2026

C++: objects (part 2)

Discussion: What's the purpose of an exam? What makes an exam bad? Or good?

Reminders

Assessments:

- Exercise 4 is due today at 11:59pm
- HW2 is due next Thursday at 11:59pm
- Exercise 5 will be released today

General:

- 3 lectures before the midterm
  - Personalized exams?

SECTION

Congratulations!

Team Can

Team Mango

Team Teamdef

Review

REVIEW

Pointers vs references

```
1  void IncPtr(int* p) { *p += 1; }   // pointer version
2  void IncRef(int& r) {  r += 1; }   // reference version
3
4  int main(int argc, char** argv) {
5    int a = 5;
6    IncPtr(&a);   // caller: &a,  body: *p
7    IncRef(a);    // caller:  a,  body:  r
8    // a is now 7
9    return EXIT_SUCCESS;
10 }
```

❖ **Both functions do the same thing**: add 1 to the caller's a.

❖ **Pointer:** the caller writes **&a**, the body writes ***p**.

❖ **Reference:** the caller writes **a**, the body writes **r**. No stars, no ampersands.

new and delete

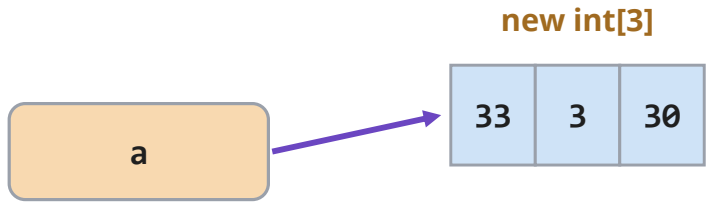
```
1  // one value on the heap
2  int* n = new int(333);   // allocate an int, set to 333
3  int* m = new int;        // allocate an int, UNINITIALIZED
4
5  delete n;                // give the memory back
6  delete m;
```

- **new T** allocates space for a T on the heap and returns a **T***.
- **delete p** gives that memory back to the heap.
- **new / delete** are typed: no cast, no sizeof, unlike malloc.

Takeaway: `new` allocates a typed block on the heap. `delete` gives it back to the system. Do NOT mix with malloc/free

new[] and delete[]

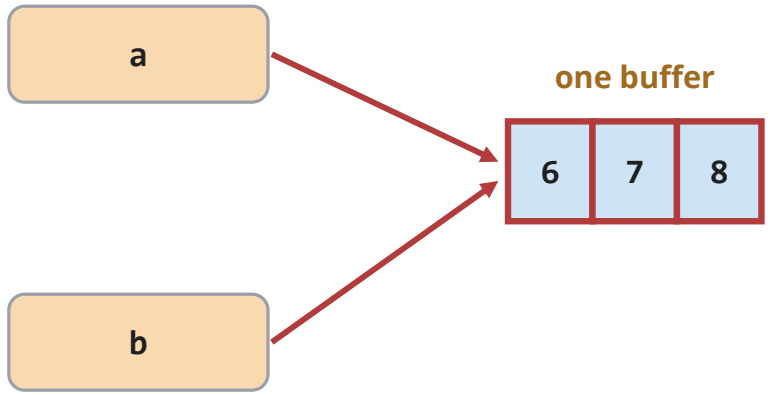
```
1  // dynamically allocated array of 3 ints
2  int* a = new int[3];
3
4  a[0] = 33;  a[1] = 3;  a[2] = 30;
5
6  delete[] a;   // note the [] !
```



- **type* a = new type[n]** allocates n contiguous elements on the heap.
- **Array new pairs with** delete[], not plain delete.

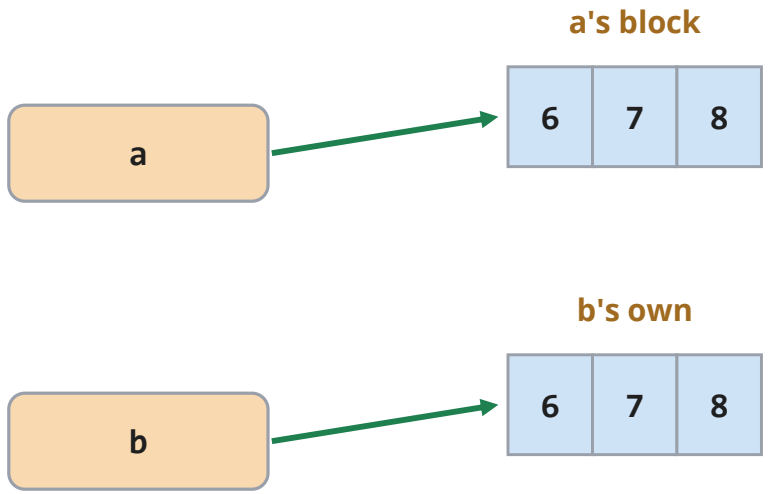
Common Error #1: double delete

```
1  int* a = new int[3]{6, 7, 8};
2
3  int* b = a;     // copies the POINTER, not the values
4  b[0] = 0;       // now a[0] is 0 too  (aliasing!)
5
6  delete[] a;
7  delete[] b;     // SAME array -> double free
```



Solution: deep copy

```
1  int* a = new int[3]{6, 7, 8};
2
3  int* b = new int[3];      // b gets its OWN arrays
4  for (int i = 0; i < 3; i++)
5    b[i] = a[i];            // copy the values across
6
7  delete[] a;
8  delete[] b;               // two arrays, two frees
```



❖ **Allocate a second block,** then copy the values across. Two owners, two clean deletes.

❖ A deep copy allocates its own buffer and copies the values, so each pointer owns and frees its own block.

Common Error #2: use-after-delete

```
1  int* a = new int[3]{6, 7, 8};
2  delete[] a;          // a is now a DANGLING pointer
3
4  // should do a = nullptr here
5
6  a[0] = 5;            // use-after-free: undefined behavior
7  delete[] a;          // double free: undefined behavior
8
9
```

After delete, the pointer is dead

- **delete[] a** frees the block, but a still holds the old address.
- **Using a now** (read, write, or delete again) is undefined behavior.
- **Habit:** set a = nullptr after delete, so misuse crashes loudly.

Objects

Objects

The Six Special Members

Point.c (what you see)

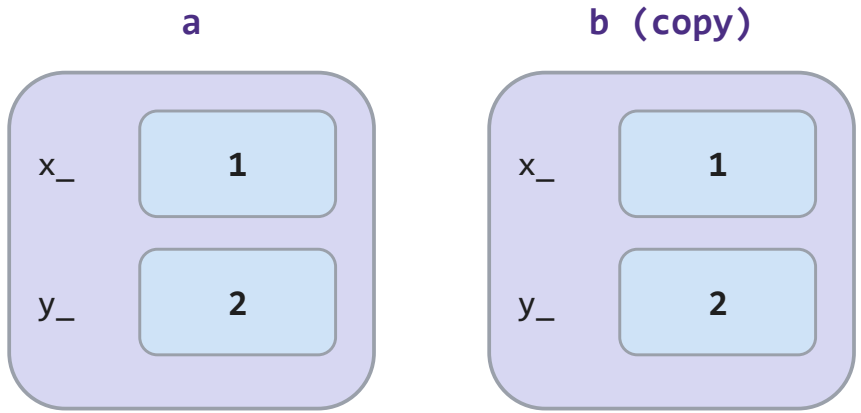
```
1  class Point {
2   private:
3    int x_;   // data member
4    int y_;
5  };
6
7
8
9
10
11
12
13
14
15
16
17
```

Point.c (what you get)

```
1  class Point {
2    Point() = default;                                  // constructor
3    Point(const Point& other) = default;                // copy constructor
4    Point& operator=(const Point& other) = default;     // copy assignment
5    Point(Poin&t&& other) = default;                    // move constructor
6    Point& operator=(Point&& other) = default;          // move assignment
7    ~Point() = default;                                 // destructor
8
9   private:
10   int x_;   // data member
11   int y_;
12 };
13
14
15
16
17
```

The Rule of Zero

```
1  Point a(1, 2);
2  Point b = a;   // synthesized copy: copies x_ and y_
3
4  Point c;
5  c = a;         // synthesized operator=: copies x_, y_
6
7  // destructor: nothing to free, x_ and y_ just vanish
```



no pointers, no heap:
two fully independent copies

❖ **Point holds only plain int members.** Copying it byte for byte is already correct, and destroying it frees nothing.

RULE:
If your class doesn't touch the heap, the compiler's synthesized copy, assignment, and destructor are all correct. Use defaults

IntArray

```
1  class IntArray {          // goal: OWN a heap int buffer
2   public:
3    size_t Size() const       { return size_; }
4    int    At(size_t i) const { return data_[i]; }
5    void   Set(size_t i, int v) { data_[i] = v; }
6   private:
7    size_t size_;
8    int* data_;              // points at the buffer we own
9  };
```

- **Goal:** IntArray owns an int[] on the heap.
- C++ offers to write all six special members for us, for free.
- **The question for the whole hour:** are those free defaults correct for a class that owns memory?
- We will hit one problem at a time, and write the member that fixes it.

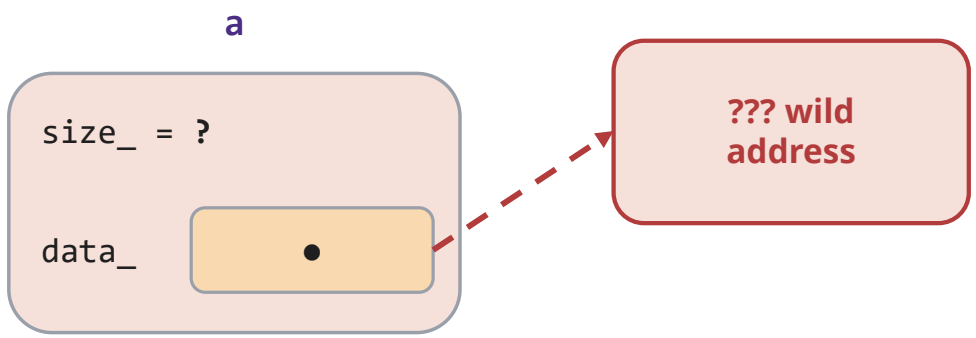
PROBLEM

Problem #1

```
1  class IntArray {
2   public:
3    void Set(size_t i, int v) { data_[i] = v; }
4   private:
5    size_t size_;
6    int* data_;
7  };
8
9  IntArray a;
10 a.Set(0, 5);
```

Problem #1

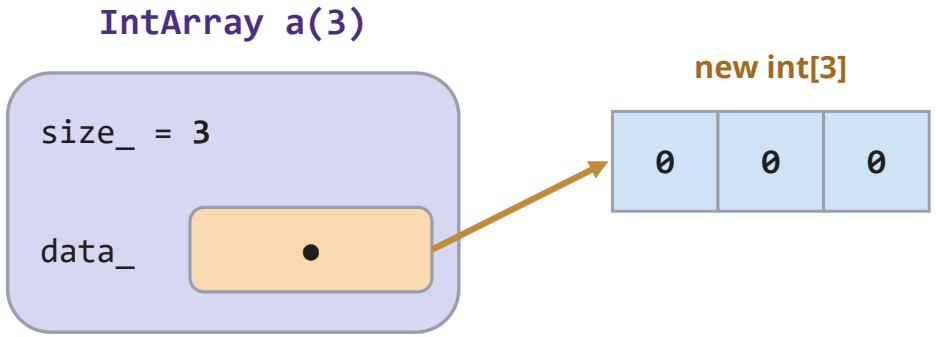
```
1  class IntArray {
2   public:
3    void Set(size_t i, int v) { data_[i] = v; }
4   private:
5    size_t size_;
6    int* data_;
7  };
8
9  IntArray a;
10 a.Set(0, 5);
```



- **No constructor ->** members are default-initialized; a raw int* gets an indeterminate value.
- **a.Set(0, 5)** runs data_[0] = 5, writing through a garbage pointer: undefined behavior.

Fix #1: Constructor

```
1  class IntArray {
2   public:
3    explicit IntArray(size_t size);
4    size_t Size() const { return size_; }
5    int At(size_t i) const { return data_[i]; }
6    void Set(size_t i, int v) { data_[i] = v; }
7   private:
8    size_t size_;    // declared FIRST
9    int* data_;      // heap buffer we OWN
10 };
11
12 IntArray::IntArray(size_t size) : size_(size) {
13   data_ = new int[size];
14
```



- The object holds a **pointer**, not the array. The array lives on the heap.
- **The ctor acquires the buffer** with new int[size] -- but nothing frees it yet.
- So: who calls **delete[]** on that buffer, and when?

PROBLEM

Problem #2

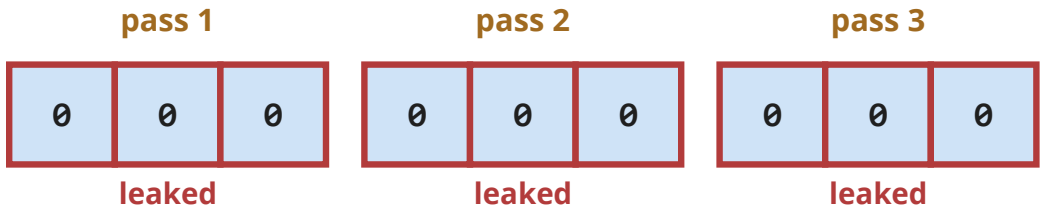
```
1  void ProcessBatches() {
2    for (int i = 0; i < 1000000; i++) {
3      IntArray scratch(1024);
4      // ... use scratch ...
5    }
6  }
```

PROBLEM

Problem #2

```
1  void ProcessBatches() {
2    for (int i = 0; i < 1000000; i++) {
3      IntArray scratch(1024);
4      // ... use scratch ...
5    }
6  }
```

- Each pass **new int[1024]** allocates a buffer.
- **When scratch dies,** its data_ pointer vanishes but the buffer is never freed: a leak that grows without bound.



1,000,000 buffers allocated, 0 freed. The pointer dies with scratch; the heap block does not.

Fix #2: Destructor

```
1  class IntArray {
2   public:
3    explicit IntArray(size_t size);  // acquire
4    ~IntArray();                     // release
5    // ...
6   private:
7    int* data_;
8    size_t size_;
9  };
10
11 IntArray::~IntArray() {
12   delete[] data_;   // give the heap back
13 }
```

❖ **~IntArray()** runs automatically when the object dies.

- a stack object: at the end of its scope
- a heap object: when you delete it

❖ **Its job:** release what the object owns, here **delete[] data_**. No return type, no params, one per class.

❖ **The private buffer can't leak:** cleanup is automatic, no delete[] at the call site.

PROBLEM

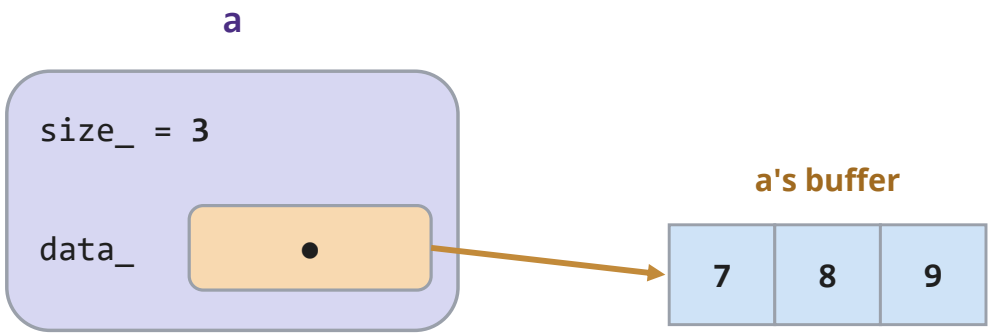
Problem #3

```
1  IntArray a(3);
2  a.Set(0, 7);
3  a.Set(1, 8);
4  a.Set(2, 9);
5
6  IntArray& b = a;
7  IntArray  c = a;
8
```

PROBLEM

Problem #3

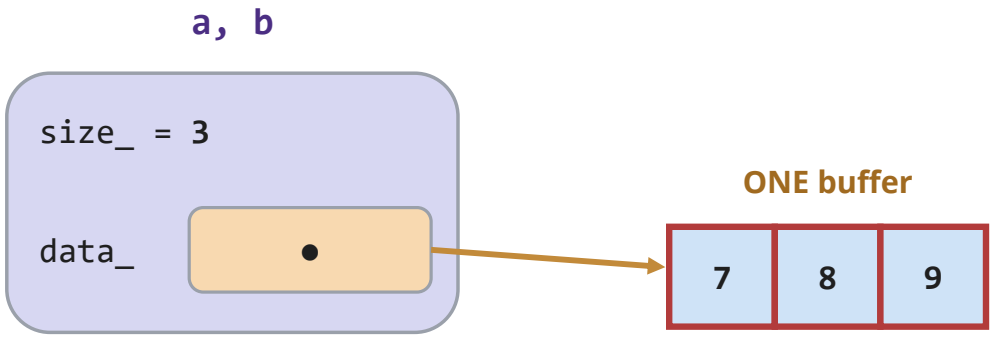
```
1  IntArray a(3);
2  a.Set(0, 7);
3  a.Set(1, 8);
4  a.Set(2, 9);
5
6  IntArray& b = a;   // reference: same object
7  IntArray  c = a;   // copy: shallow by default
8
```



- **a owns one heap buffer.** Its destructor frees that buffer exactly once when a dies.
- Next we make two more names for it: **a reference, then a copy.**

Problem #3

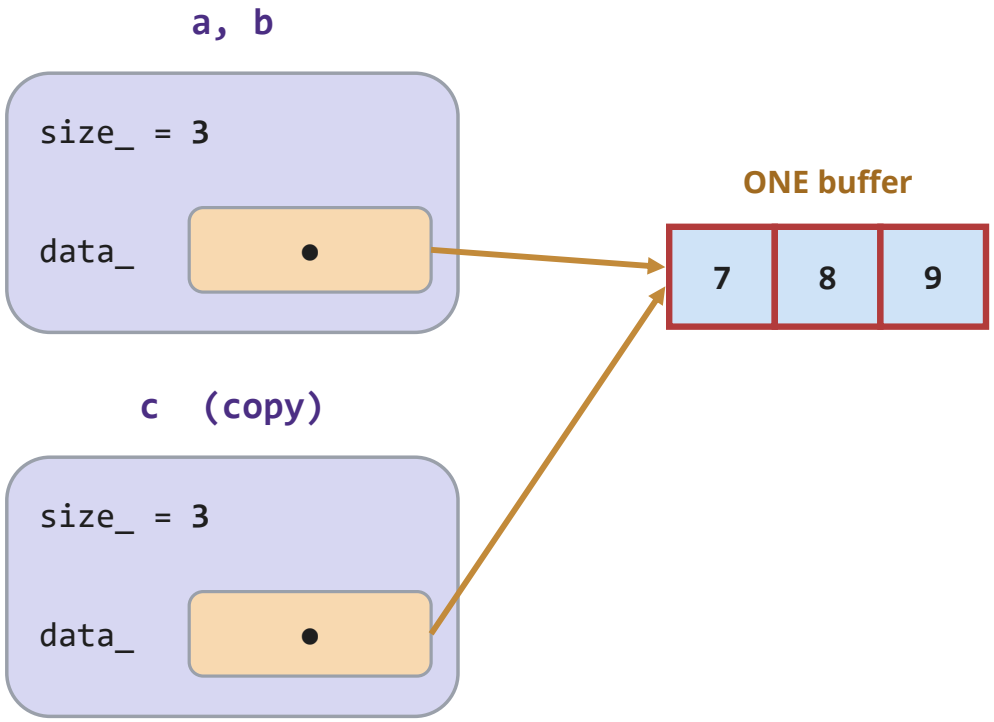
```
1  IntArray a(3);
2  a.Set(0, 7);
3  a.Set(1, 8);
4  a.Set(2, 9);
5
6  IntArray& b = a;   // reference: same object
7  IntArray  c = a;   // copy: shallow by default
8
```



- **b is a reference** so no new buffer is created
- b doesn't get a separate constructor/destructor call since it is an alias

Problem #3

```
1  IntArray a(3);
2  a.Set(0, 7);
3  a.Set(1, 8);
4  a.Set(2, 9);
5
6  IntArray& b = a;   // reference: same object
7  IntArray  c = a;   // copy: shallow by default
8
```



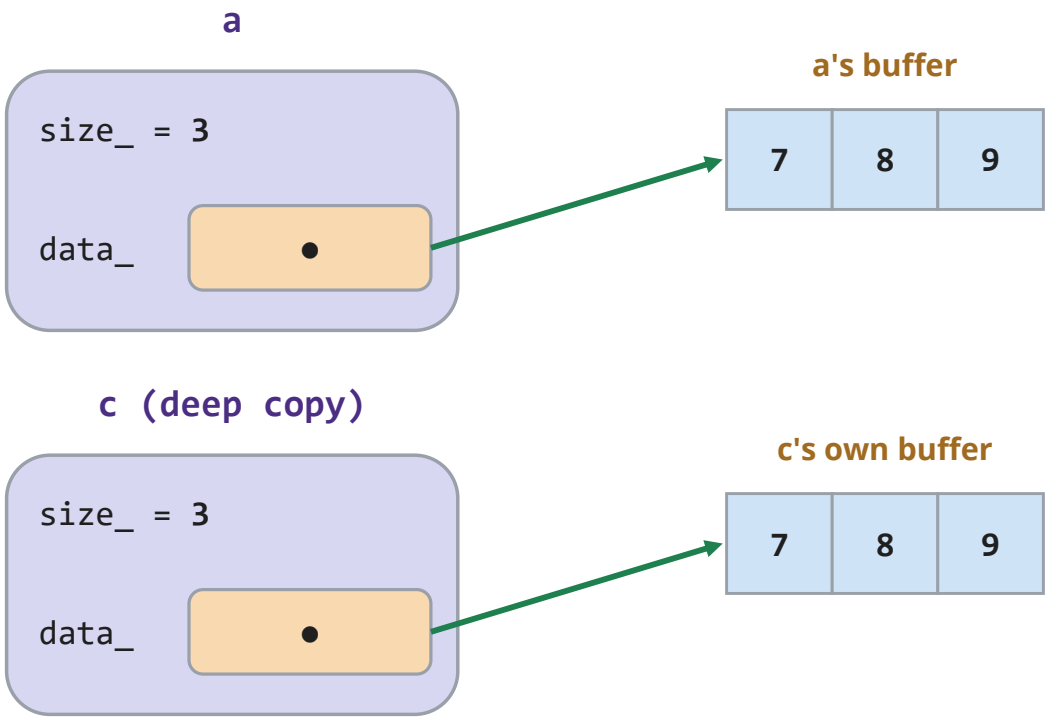
❖ The default copy copies each member. data_ is a pointer, so it copies the ADDRESS, not the buffer.

❖ **Now a.data_ == c.data_**: two separate objects, one shared buffer.

Fix #3: Copy Constructor

```
1  IntArray::IntArray(const IntArray& other)
2      : size_(other.size_) {
3    data_ = new int[size_];         // our own buffer
4    for (size_t i = 0; i < size_; i++) {
5      data_[i] = other.data_[i];    // copy the values
6    }
7  }
```

❖ **Allocate a fresh buffer,** then copy the elements over. Each object owns its own array.



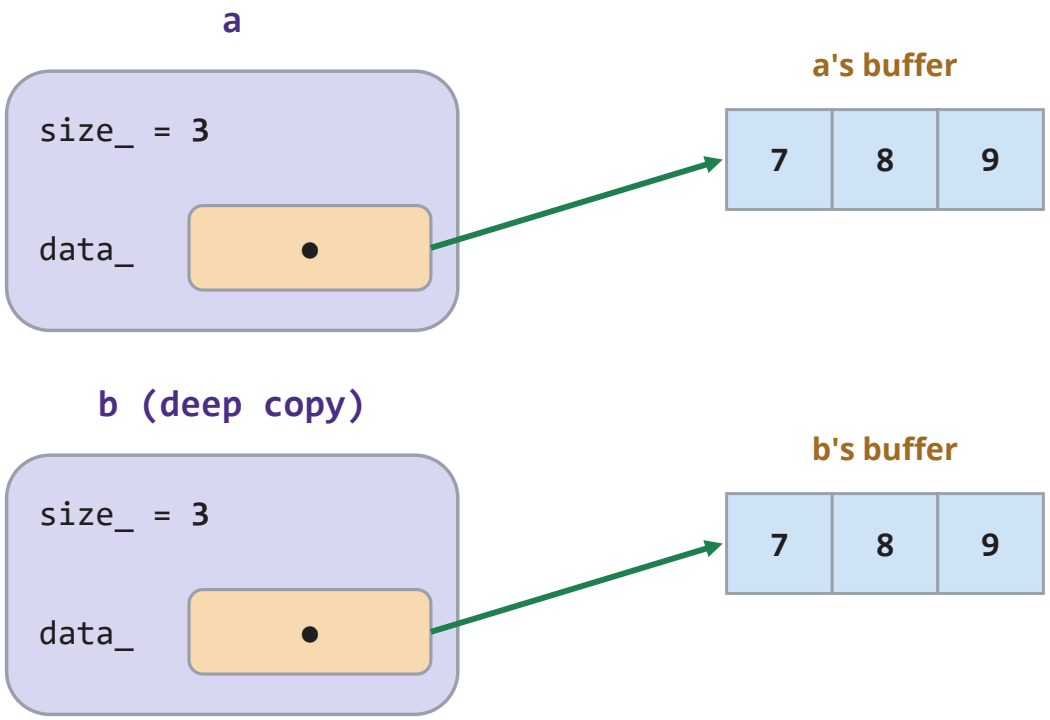
PROBLEM

Problem #4

```
1  IntArray a(3);
2  IntArray b = a;  // uses copy constructor
3  b = a;
4
```

Problem #4

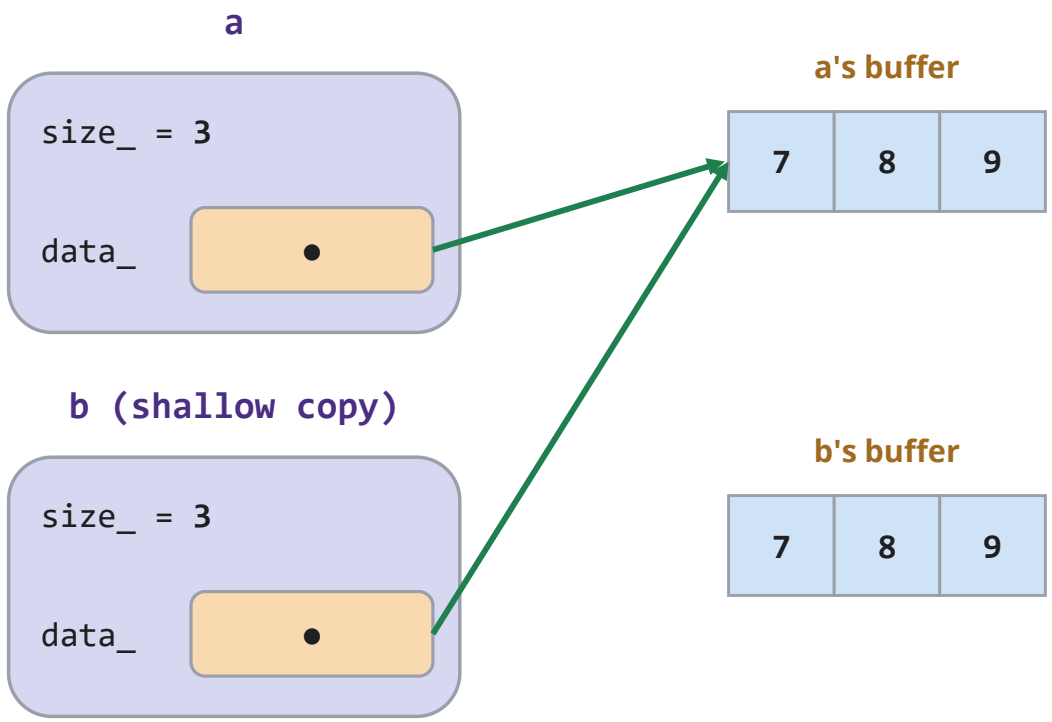
```
1  IntArray a(3);
2  IntArray b = a;  // uses copy constructor
3  b = a;
4
```



- At first, we use the new copy constructor to have two separate buffers

Problem #4

```
1  IntArray a(3);
2  IntArray b = a;  // uses copy constructor
3  b = a;
4
```



- We use a shallow copy on assignment (not construction)
- The copy of b's buffer is lost!

Fix #4: Copy Assignment

```
1  IntArray& IntArray::operator=(const IntArray& rhs) {
2    if (this != &rhs) {           // 1. self-assign?
3      delete[] data_;             // 2. free old buffer
4      size_ = rhs.size_;
5      data_ = new int[size_];     // 3. deep copy
6      for (size_t i = 0; i < size_; i++) {
7        data_[i] = rhs.data_[i];
8      }
9    }
10   return *this;                 // 4. enable chaining
11 }
```

- **Construction** builds a NEW object.
- **Assignment** overwrites one that ALREADY owns a buffer.
- So there is more to do. We walk the four steps next.

The Rule of Three

If your class needs any one of these, it almost certainly needs all three. They all manage the same owned resource.

1. Destructor

```
~IntArray()
```

release the resource

2. Copy constructor

```
IntArray(const IntArray&)
```

deep-copy on construction

3. Assignment operator

```
operator=(const IntArray&)
```

free old, then deep-copy

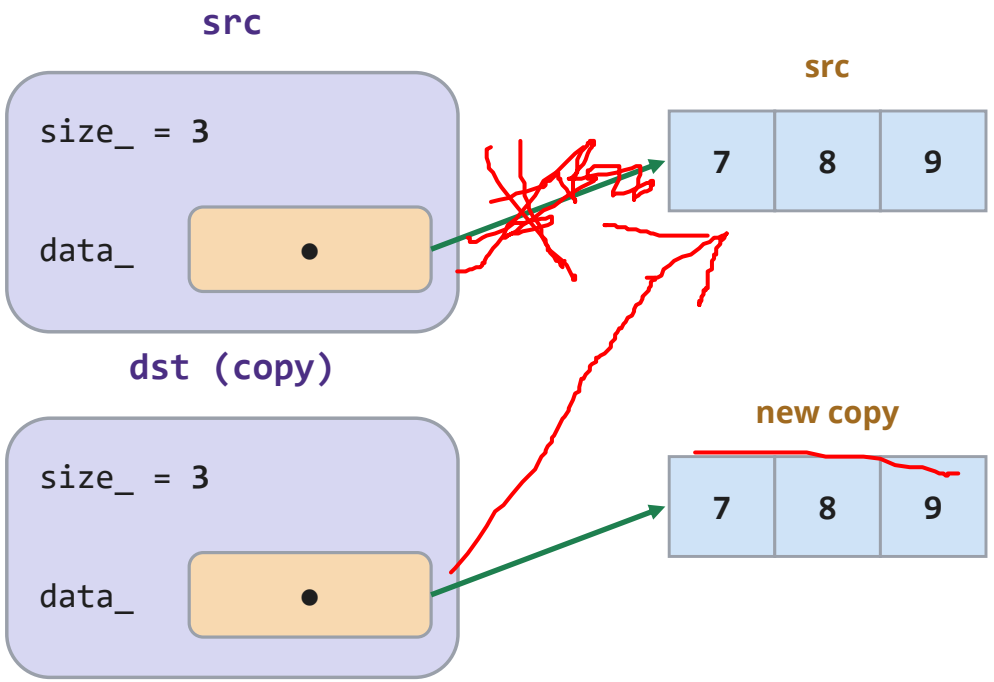
RULE:
If you write a destructor, copy constructor, or operator=, you almost certainly need all three. Ask first, who owns the resource?

Problem(atic) #5

```
1  IntArray MakeBig() {
2    IntArray big(1000000);
3    // ... fill big ...
4    return big;
5  }
6
7  IntArray a = MakeBig();
```

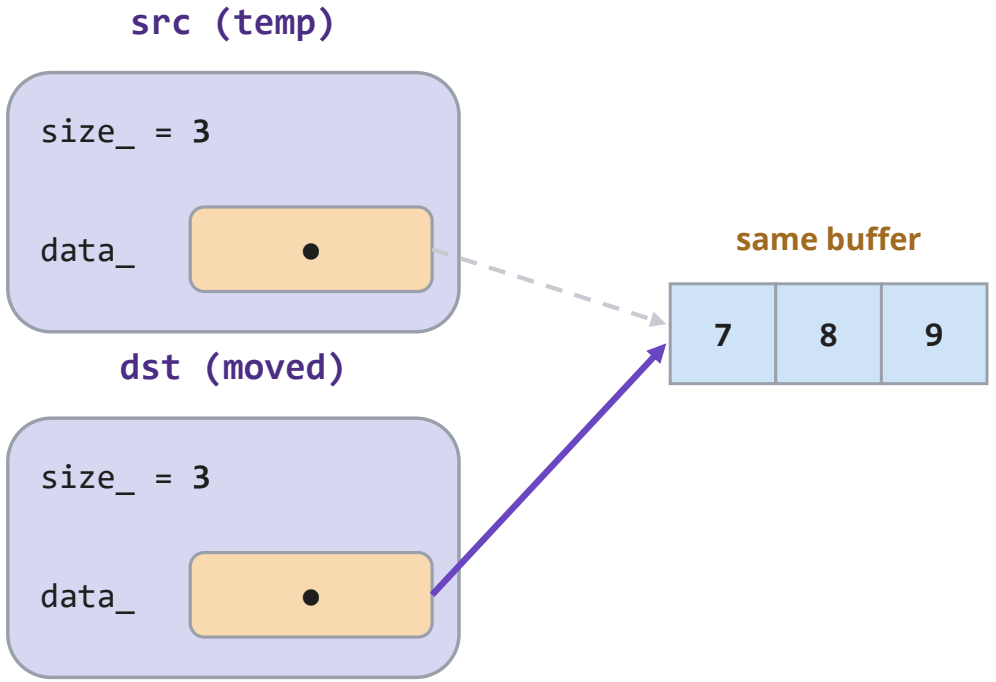
Problem #5

copy: duplicate the buffer



❖ **Move** steals a temporary's guts instead of deep-copying, then leaves it empty. Cheap.

move: steal the buffer



`src.data_ = nullptr`

Rule of Five = the Big Three + move ctor + move operator=.

Fix #5: Move

```
1  // move constructor: steal the buffer
2  IntArray::IntArray(IntArray&& other)
3      : size_(other.size_), data_(other.data_) {
4    other.data_ = nullptr;   // leave the source empty
5    other.size_ = 0;
6  }
7
8  // move assignment: free old, then steal
9  IntArray& IntArray::operator=(IntArray&& rhs) {
10   if (this != &rhs) {
11     delete[] data_;        // free our old buffer
12     size_ = rhs.size_;
13     data_ = rhs.data_;     // steal the pointer
14     rhs.data_ = nullptr;   // null the source
15     rhs.size_ = 0;
16   }
17   return *this;
18 }
```

❖ **Both steal** rhs's pointer, then null it so its destructor frees nothing.

❖ **No new, no element copy:** a move is a few pointer assignments.

❖ **Big Three suppressed these.** Declaring the dtor and copy ops turned off the compiler's move members; you opt back in by writing them.

❖ **= default** would be wrong: it copies data_ without nulling the source, so both free it -> double free.

Rule of Zero, Three, and Five

Rule of Zero

- **Own no resource** (like Point).
- All six defaults are already correct.
- Write **none** of the special members.

Rule of Three

- **Own a resource** (like IntArray).
- Write the three that manage it:
  - destructor
  - copy constructor
  - copy assignment

Rule of Five

- The Big Three, plus two for speed:
  - move constructor
  - move assignment
- **Declaring the Three** removed the free moves; add them back.

Operator Overloading

Operators are functions with special names

```
class IntArray {
 public:
  int& operator[](size_t i) { return data_[i]; }
  // ...
};

IntArray a(3);
a[0] = 7;           // write:  calls operator[]
int first = a[0];   // read:   calls operator[]
```

```
// operator== is just a function with a special name
bool operator==(const Point& a, const Point& b) {
  return a.GetX() == b.GetX()
      && a.GetY() == b.GetY();
}

Point p(1, 2), q(1, 2);
if (p == q) { ... }   // the compiler calls operator==
```

- **operator[]** makes **a[i]** read and write elements.
- It returns a reference, so a[0] = 7 assigns.
- **operator==** defines what **p == q** means.
- You already met one: **operator=** (assignment), coming up next.

=default and =delete

```
1  class IntArray {
2   public:
3    IntArray() = default;    // opt back into the
4                             // synthesized default ctor
5
6    // ban copying entirely:
7    IntArray(const IntArray&) = delete;
8    IntArray& operator=(const IntArray&) = delete;
9  };
```

❖ **= default** asks for the compiler's synthesized version explicitly, so you keep it even after declaring other ctors.

❖ **= delete** removes a function. Calling it is a compile error.

❖ **Deleting the copy ctor and operator=** makes a type non-copyable, which prevents the accidental shallow copies we just saw.

❖ This is how unique-ownership types are built.

Inheritance

One class per asset type is redundant

asset #1

Stock

```
symbol_
total_shares_
total_cost_
current_price_
---
GetMarketValue()
GetProfit()
GetCost()
```

asset #2

DividendStock

```
symbol_
total_shares_
total_cost_
current_price_
dividends_
---
GetMarketValue()
GetProfit()
GetCost()
```

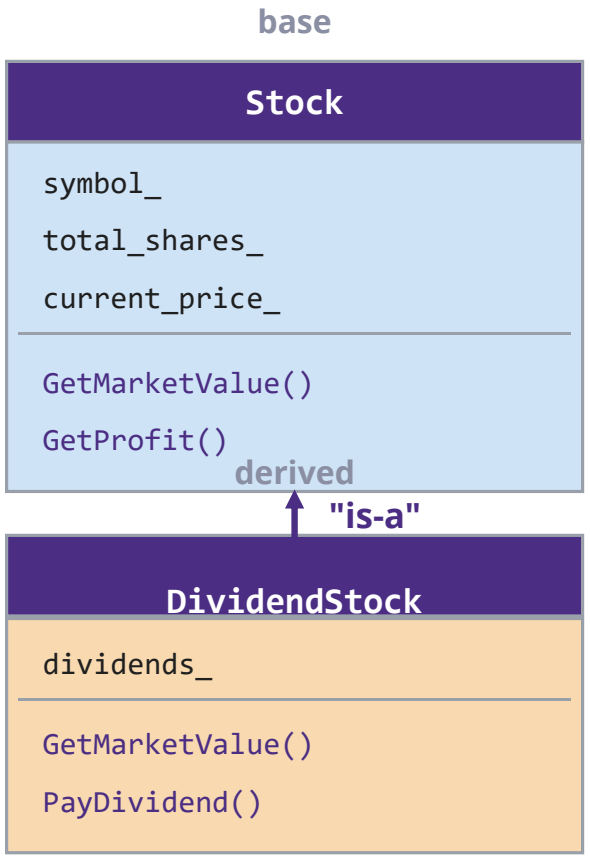
asset #3

Cash

```
amount_
---
GetMarketValue()
```

- **Duplicated** fields and methods across Stock and DividendStock.
- **No common type:** you cannot make one vector holding a mix of assets.
- Adding an asset means copy-pasting the shared parts again.

is-a: base and derived classes



Java	C++
Superclass	Base class
Subclass	Derived class
extends	: public Base

Why inherit?

- **Code reuse:** the derived class inherits the base's fields and methods.
- **Polymorphism:** redefine behavior while keeping the same interface.
- **Extensibility:** the derived class adds new state and behavior.

class Derived : public Base

```
1  #include "Stock.h"
2
3  class DividendStock : public Stock {   // is-a Stock
4   public:
5    DividendStock(...);                  // NOT inherited
6    double GetMarketValue() const override;
7    void PayDividend(double amount);     // new behavior
8   private:
9    double dividends_;                   // new state
10 };
```

What the colon buys you

- **public inheritance** is like Java's extends: every non-private member of Base is part of Derived too.
- **Access:** public visible to all, private only to the class, protected visible to derived classes.
- **NOT inherited:** constructors, destructor, copy ctor, operator=. The derived class writes its own.

A base pointer can point at a derived object

```
1  #include "Stock.h"
2  #include "DividendStock.h"
3
4  DividendStock dividend;
5  DividendStock* ds = &dividend;   // exact type
6  Stock* s = &dividend;            // base ptr to derived
7  // s exposes the Stock interface; the ACTUAL type of
8  // *s may decide which version actually runs.
```

PromisedType* p = new ActualType

- In Java: **Base var = new Derived();**
- In C++: **Base* p = new Derived();** (references work too)
- **ActualType** must be the same or a derived class of the promised type.
- **The pointer's type** sets what you can call; the actual object may decide which version runs.

virtual and override

```
1  class Stock {
2   public:
3    virtual double GetMarketValue() const;  // opt in
4    double GetProfit() const;               // inherited
5  };
6  class DividendStock : public Stock {
7   public:
8    double GetMarketValue() const override; // CSE 333 style
9  };
```

Two keywords

- **virtual** on the base method requests dynamic dispatch (this is Java's default, always on).
- **override** on the derived method says 'I mean to override', and the compiler checks the signature matches.
- **Almost always** you want virtual. Google style: use override, not virtual, in the derived class.

pure virtual

```
1  class Shape {
2   public:
3    virtual double Area() const = 0;   // pure virtual
4    virtual ~Shape() { }
5  };
6
7  class Circle : public Shape {
8   public:
9    explicit Circle(double r) : r_(r) { }
10   double Area() const override { return 3.14159*r_*r_; }
11  private:
12   double r_;
13 };
14
15 // Shape s;                  // ERROR: abstract
16 Shape* p = new Circle(2.0);  // OK: ptr to concrete type
```

= 0 makes it "pure"

- **virtual double Area() const = 0;** declares the method with no body.
- **Any pure virtual** makes the class abstract: you cannot instantiate it.
- **Derived classes** must override every pure virtual to become concrete.
- **Only pure virtuals?** That is exactly a Java interface.

Dynamic Dispatch

```
1  DividendStock dividend;
2  DividendStock* ds = &dividend;
3  Stock* s = &dividend;
4
5  ds->GetMarketValue();   // DividendStock::GetMarketValue
6  s->GetMarketValue();    // DividendStock::GetMarketValue
7  s->GetProfit();         // Stock::GetProfit (inherited),
8                          //   calls DividendStock's value
```

Reading the calls

- **ds->GetMarketValue():** derived version, obviously.
- **s->GetMarketValue():** also the DERIVED version, through a Stock pointer.
- **s->GetProfit():** runs the inherited Stock::GetProfit, which itself calls DividendStock::GetMarketValue.

When we specify virtual, dynamic dispatch picks the most-derived version from the actual object, even when you call through a base pointer or from inside inherited code.

Static Dispatch

```
1  class Base {
2   public:
3    void Foo();          // NOT virtual -> static dispatch
4  };
5  class Derived : public Base {
6   public:
7    void Foo();          // hides, does not override
8  };
9  int main(int argc, char** argv) {
10   Derived d;
11   Derived* dp = &d;
12   Base* bp = &d;
13   dp->Foo();   // Derived::Foo  (compile-time type)
14   bp->Foo();   // Base::Foo     (compile-time type!)
15   return EXIT_SUCCESS;
16 }
```

Two rules to remember

- **Sticky:** once a method is virtual in a base, it stays virtual in every derived class, override or not.
- **No virtual = static dispatch:** the call resolves by the COMPILE-TIME (pointer) type.
- **So bp->Foo()** runs Base::Foo even though bp points at a Derived!
- **Different from Java,** where every method is virtual.

Inheritance + Constructors

```
1  class Base {                    // no default ctor
2   public:
3    Base(int y) : y_(y) { }
4    int y_;
5  };
6  class Der1 : public Base {      // ERROR: needs Base()
7   public:
8    int z_;
9  };
10 class Der2 : public Base {      // OK
11  public:
12   Der2(int y, int z) : Base(y), z_(z) { }
13   int z_;
14 };
```

How construction chains

- ❖ **Base part first:** the base constructor runs before the derived body.
- ❖ **Pick the base ctor** in the derived initializer list: Der2(int y, int z) : Base(y), z_(z).
- ❖ **No default base ctor** and you do not call one? Compiler error (see Der1).
- ❖ Constructors are not inherited; Der writes its own.

Inheritance + Destructors

```
1  class Base {
2   public:
3    Base()  { x_ = new int; }
4    ~Base() { delete x_; }      // NON-virtual
5    int* x_;
6  };
7  class Der1 : public Base {
8   public:
9    Der1()  { y_ = new int; }
10   ~Der1() { delete y_; }
11   int* y_;
12 };
13 Base* b = new Der1;
14 delete b;   // only ~Base runs -> y_ LEAKS (UB)
```

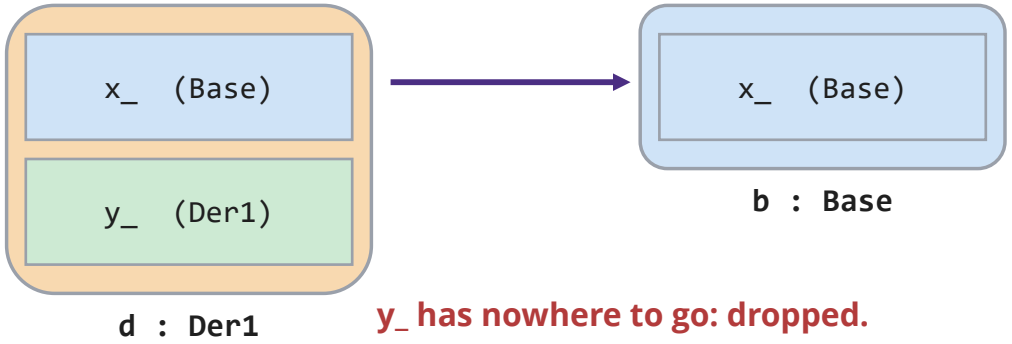
```
1  class Base {
2   public:
3    Base()          { x_ = new int; }
4    virtual ~Base() { delete x_; }   // virtual dtor
5    int* x_;
6  };
7  // ...
8  Base* b = new Der1;
9  delete b;   // ~Der1 THEN ~Base run -> no leak
```

Destruction runs top-down: ~Der1 then ~Base. A non-virtual base dtor called through a Base* skips ~Der1, so y_ leaks (formally undefined behavior). Marking ~Base virtual fixes it.

Inheritance + Copy

```
1  class Base {
2   public:
3    Base(int x) : x_(x) { }
4    int x_;
5  };
6  class Der1 : public Base {
7   public:
8    Der1(int y) : Base(16), y_(y) { }
9    int y_;
10 };
11 void Foo() {
12   Base b(1);
13   Der1 d(2);
14   b = d;   // SLICED: copies x_, drops y_
15 }
```

b = d; copies only the Base part



Takeaway: Assigning a derived object to a base VALUE slices off the derived parts. A Base variable has no room for Der1's y_.

Casting

Basic Casts (C-style)

```
1  double d = 3.9;
2  int n = (int) d;              // truncates to 3
3
4  Stock* s = (Stock*) &ds;      // pointer reinterpretation
5
6  // One blunt syntax for very different operations:
7  // numeric conversion vs pointer reinterpretation.
8  // Hard to search for, easy to do the wrong thing.
```

Why C++ moved on

- **(type) expr** does everything: truncate a double, reinterpret a pointer, strip const.
- **Intent is hidden:** the reader cannot tell a safe numeric cast from a dangerous pointer reinterpret.
- **Unsearchable:** you cannot grep for '(type)' meaningfully.
- **No safety check** beyond what the compiler must allow.

Takeaway: The C cast is legal in C++ but too blunt: one syntax hides wildly different operations. Prefer the named C++ casts.

Say what you mean

static_cast<T>(e)

- **Compile-time** conversion between related pointers or numeric types.
- Example: **static_cast<int>(3.9)**, or up a hierarchy. Down-casts are unchecked and risky.

dynamic_cast<T>(e)

- **Run-time checked** down-cast; needs a polymorphic base (a virtual method).
- Example: **dynamic_cast<Der1*>(bp)** returns nullptr if bp is not really a Der1.

const_cast<T>(e)

- **Adds or removes const** (or volatile), nothing else.
- Example: **const_cast<int*>(p)**. Writing through a stripped const can be UB. Use rarely.

reinterpret_cast<T>(e)

- **Raw bit reinterpretation** between unrelated types (pointer <-> int, incompatible pointers).
- Example: **reinterpret_cast<int*>(addr)**. Dangerous; used carefully in hw3.

One-arg constructors convert implicitly

```
1  class Foo {
2   public:
3    Foo(int x) : x_(x) { }   // one-arg = conversion path
4    int x_;
5  };
6  int Bar(Foo f) { return f.x_; }
7
8  int main(int argc, char** argv) {
9    return Bar(5);   // becomes Bar(Foo(5)) implicitly!
10 }
```

```
1  class Foo {
2   public:
3    explicit Foo(int x) : x_(x) { }   // no implicit path
4    int x_;
5  };
6  int Bar(Foo f) { return f.x_; }
7
8  int main(int argc, char** argv) {
9    return Bar(5);   // compiler error (good!)
10 }
```

A single-argument constructor doubles as a conversion path, so Bar(5) silently becomes Bar(Foo(5)). At most one user-defined conversion is applied. Mark such constructors **explicit** to turn that off.

Class Design

Case Study

DESIGN DISCUSSION

Design Challenge: An Online Gradebook

- You are designing the backend for an online gradebook, think Canvas or Gradescope.
- The system has instructors and students.
- Instructors create assignments. An assignment is made up of questions.
- There are different kinds of questions (multiple choice, free response, coding, ...). Each is graded differently. Some can be graded automatically; some need a human.
- Students take an assignment by submitting responses.
- Instructors grade submissions and record scores. Students must not be able to change scores.
- The system can report a student's total score on an assignment.

Things you might consider

- Where does inheritance fit? What are the "is-a" relationships?
- Which member functions should be virtual vs. non-virtual? Any pure virtual (abstract)?
- Which behavior belongs to the class as a whole rather than one object?
- How should functions take parameters: by value, reference, or pointer? const?
- How do you store a mixed collection of different question types together?
- How do you enforce "only instructors can assign grades"?
- Who owns heap-allocated objects, and what does that imply for destructors?

There is no single right answer. Be ready to defend your tradeoffs.

Class Design

Class Design

Solution: People (a simple hierarchy)

```
1  class Person {
2   public:
3    explicit Person(const std::string& name);
4    virtual ~Person();              // virtual: safe to delete via Person*
5    std::string Name() const;       // non-virtual: same for everyone
6   private:
7    std::string name_;
8  };
9
10 class Student    : public Person { /* ... */ };
11 class Instructor : public Person { /* ... */ };
```

Solution: Question is abstract

```
1  class Question {
2   public:
3    explicit Question(int points);
4    virtual ~Question();
5
6    int PointsWorth() const;                          // non-virtual
7
8    // Each kind grades itself differently:
9    virtual int  Grade(const Response& r) const = 0;  // pure virtual
10   virtual bool AutoGradable() const = 0;            // -> abstract
11  private:
12   int points_;
13 };
```

Solution: Concrete question types

```
class MultipleChoice : public Question {
 public:
  MultipleChoice(int points, int correct_choice);
  int  Grade(const Response& r) const override;   // compare to key
  bool AutoGradable() const override { return true; }
 private:
  int correct_choice_;
};

class FreeResponse : public Question {
 public:
  explicit FreeResponse(int points);
  int  Grade(const Response& r) const override;   // needs a human
  bool AutoGradable() const override { return false; }
};
```

Solution: Assignment

```
class Assignment {
 public:
  static int NextId();                    // class-wide id source

  explicit Assignment(Instructor* owner);
  ~Assignment();                          // deletes owned Question*

  void AddQuestion(Question* q);          // takes ownership (heap)
  void Take(const Student& s, const Response& r);  // student submits
  int  TotalScore(const Student& s) const; // non-virtual

 private:
  friend class Instructor;                // only instructors grade
  void RecordScore(const Student& s, int pts);

  static const int kMaxItems = 50;
  static int next_id_;

  Instructor* owner_;                     // borrowed, not owned
  Question* questions_[kMaxItems];        // base ptrs -> polymorphism
  int num_questions_;

  const Student* graded_[kMaxItems];      // parallel arrays:
  int scores_[kMaxItems];                 //   graded_[i] -> scores_[i]
  int num_scores_;
};
```

Solution: the interesting bits

```
int Assignment::next_id_ = 0;              // define static storage
int Assignment::NextId() { return next_id_++; }

// Non-virtual algorithm built on virtual steps:
int Assignment::TotalScore(const Student& s) const {
  int total = 0;
  for (int i = 0; i < num_questions_; ++i) {  // dynamic dispatch
    total += questions_[i]->Grade(/* s's response */);
  }
  return total;
}

// Instructor is a friend -> may call private RecordScore:
void Instructor::EnterGrade(Assignment* a,
                            const Student& s, int pts) {
  a->RecordScore(s, pts);                  // OK: friend access
}
```

Reminders

Assessments:

- Exercise 4 is due today at 11:59pm
- HW2 is due next Thursday at 11:59pm
- Exercise 5 will be released today

General:

- 3 lectures before the midterm
  - Personalized exams?

Questions?

Thanks for coming - see you next lecture!