POINTERS AND CLASSES

EQUIVALENCE: AUTOMATIC VARIABLES

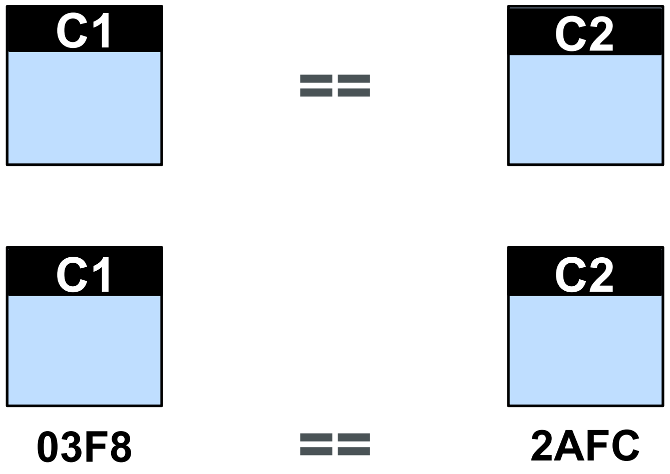
```
Course c1{575};
Course c2{575};
```

test if equivalent **objects** (requires == overload)

```
cout << (c1 == c2);
```

test if the same **object** (check address)

```
cout << (&c1 == &c2);
```



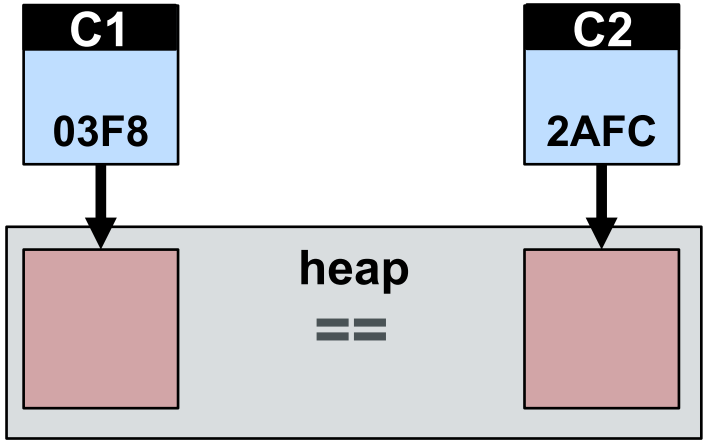
ex1_object_equivalence.cpp

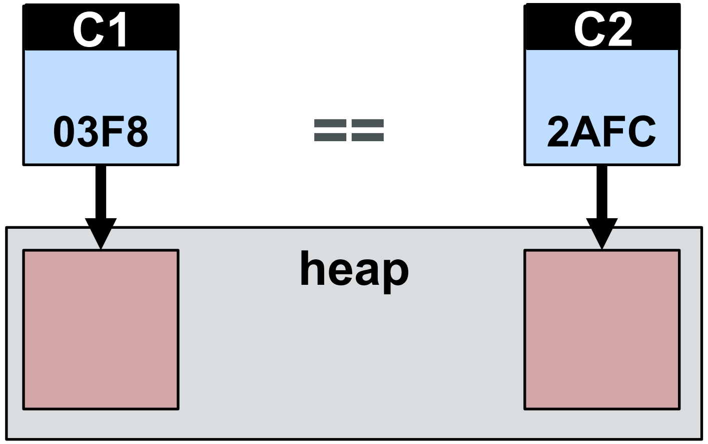
EQUIVALENCE: DYNAMIC VARIABLES

```
Course *c1 = new Course{575};
Course *c2 = new Course{575};

// test if equivalent objects (requires == overload)
cout << (*c1 == *c2);

// test if the same object (check address)
cout << (c1 == c2);
```





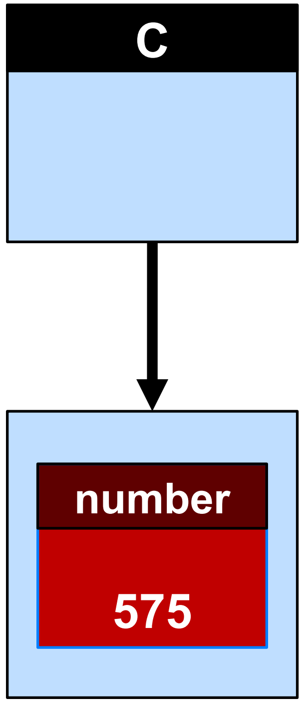
ARROW OPERATOR

```
class Course {
public:
    int number;
    Course(int n): number(n) {}
}

Course *c = new Course{575};

//deference c to access number
cout << (*c).number;

// alternative syntax using arrow operator
cout << c->number;
```



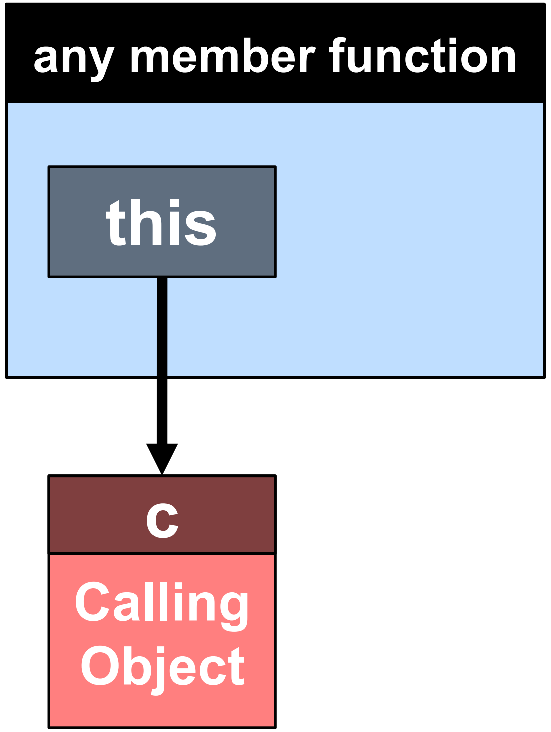
THIS POINTER

```
void Course::thisPrint() const {     // member function
    cout << this;                    // pointer to the calling object
    cout << *this;                   // the calling object
}

void Course::thisCompare(Course &c) {         // member function
    if( this == &c ) cout << “Same”;          // compare address
    if( *this == c ) cout << “Equivalent”;    // compare objects
}

bool Course::operator==(const Course &c) {
    return this->number == c.number;          // access number
}

ostream& operator<<(ostream& const Course &c);
```



ex3_this_pointer.cpp

DYNAMIC DATA MEMBERS

Dynamic data
- accessible by pointer
- constructor initialization requires new to allocate memory
- delete used to deallocate memory when object is destroyed

```
class Course {
	private:
		int *number;                                                   // pointer to heap memory
		string prof;
	public:
		Course(int n, string p): number( new int(n) ), prof(p) { }     // allocate heap memory
}
```

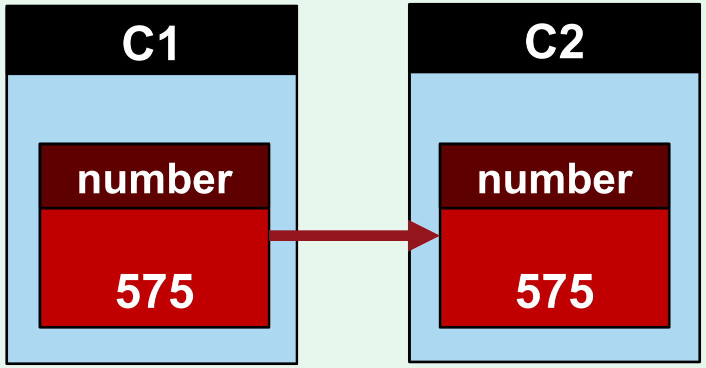
SHALLOW COPY

Concept

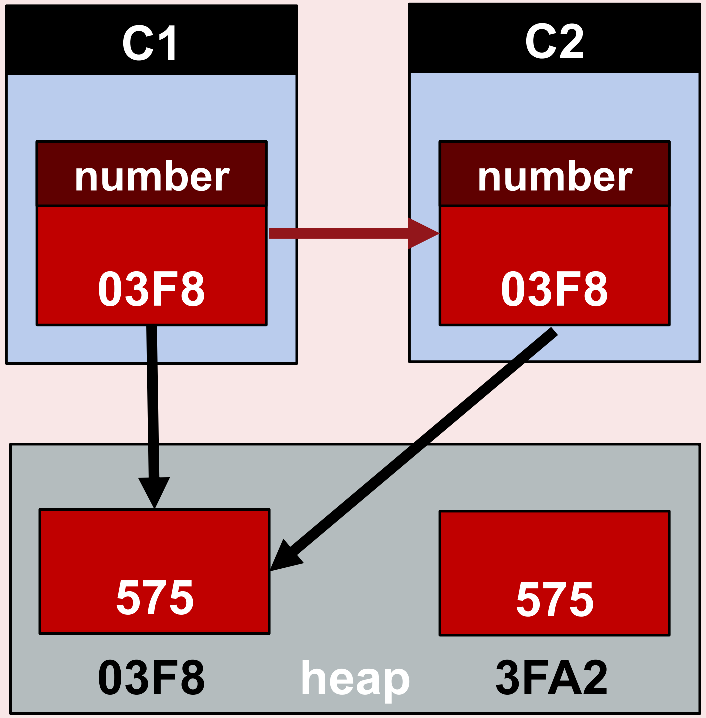
object copy where data member values are copied
ok if data is automatic, not ok if data is dynamic

c2.number = c1.number;

Value copy with automatic data members is fine



OK



Value copy with pointer data members results in objects sharing memory

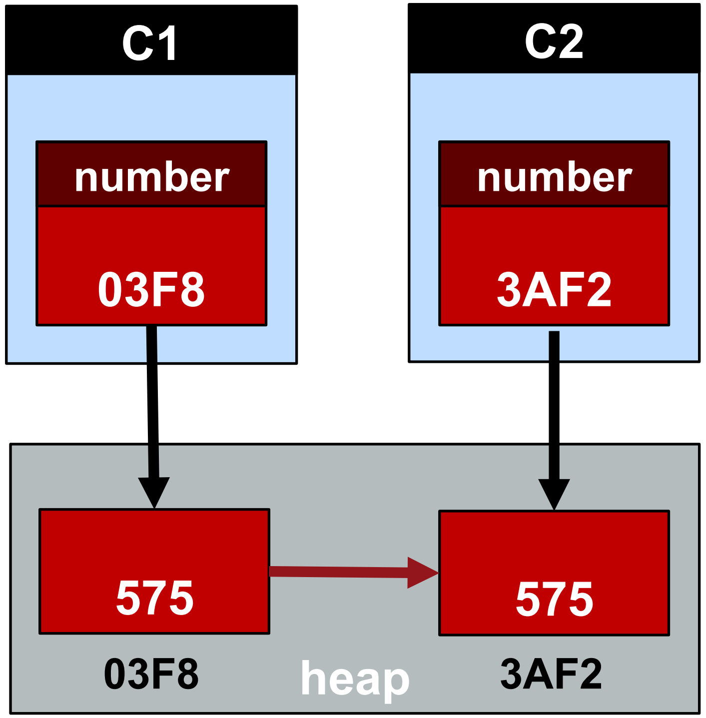
NOT OK

DEEP COPY

Concept

object copy where pointer data members are dereferenced then copied
required for copying dynamic data members

```
*c2.number = *(c1.number);
```



Dereference, then copy, object memory space remains unique

THE BIG THREE

The Big Three	member functions required for classes that use dynamic memory
	shallow copy versions are automatically generated by the compiler
	deep copy versions must be coded by the programmer
Copy constructor	a function that initializes a new object from an existing object
Assignment overload	a function that copies all data from one object to another object
Destructor	a function that manages memory when an object is destroyed

THE BIG THREE: COPY CONSTRUCTOR

```
class Course {
      public:
      int *number;
      Course(int n): number( new int (n) ) { }
      Course(const Course &c): number( new int( *(c.number) ) ) { }      // copy constructor
}

Course c1{575};     // call the one-parameter constructor
Course c2{c1};      // call the copy constructor
```

c.number is deep copied to number
number is initialized with dynamic memory

THE BIG THREE: ASSIGNMENT OPERATOR

```
class Course {
    public:
    int *number;
    Course(int n): number( new int (n) ) { }
    Course& operator=(const Course &c) {     // assignment operator overload
        if(this != &c) {                     // are the objects different or the same
            *number = *(c.number);           // deep copy
        }
        return *this;                        // return the modified calling object
    }
}

Course c1{575};     // call the one-parameter constructor
Course c2{580};     // call the one-parameter constructor
c1 = c2;            // call the assignment operator overload
```

c.number must be dereferenced to access number data

THE BIG THREE: DESTRUCTOR

```
class Course {
      public:
      int *number;
      Course(int n): number( new int (n) ) { }
      ~Course() { delete number; }                    // destructor
}

void f () {
      Course c{575};
}                       // c goes out of scope so the destructor called
```

number must be deallocated when c is destroyed to prevent a memory leak

DYNAMIC ARRAY DATA MEMBERS

Dynamic array
- accessible by pointer
- constructor initialization requires new to allocate memory
- [] delete used to deallocate memory when object is destroyed

```
class Course {
    private:
        int *studentIds;                                        // pointer to dynamic array
        int size;                                               // size of the array
    public:
        Course(int s): size(s), studentIds( new int[s] ) { }    // allocate a dynamic array
}
```

THE BIG THREE: COPY CONSTRUCTOR II

```
class Course {
    public:
    int *studentIds;
    int size;
    Course(int s): size(s), studentIds( new int[s] ) { }
    Course(const Course &c):                                                  // copy constructor
        size( c.size ),                                                       // size copied
        studentIds( new int[c.size] )                                         // array created
    {
        for(int i=0; i<c.size; ++i) { studentIds[i] = c.studentIds[i]; }     // array copied
    }
}
```

c.studentIds values copied from studentIds

THE BIG THREE: ASSIGNMENT OPERATOR II

```
class Course {
    public:
    int *studentIds;
    int size;
    Course(int s): size(s), studentIds ( new int[s] ) { }
    Course& operator=(const Course &c) {                    // assignment operator overload
        if(this != &c) {                                    // are the objects different or the same
            if(size != c.size) {                            // are the arrays of same size
                delete [] student Ids;                      // delete original array
                size = c.size;                              // update size
                studentIds = new int[ c.size];              // create new array of correct size
            }
            for(int i=0; i<c.size; ++i) { studentIds[i] = c.studentIds[i]; }    // copy array
        }
        return *this;                                       // return the modified calling object
    }
}
```

ex5ab_the_big_three.cpp

THE BIG THREE: DESTRUCTOR II

```
class Course {
      public:
      int *studentIds;
      int size;
      Course(int s): size(s), studentIds ( new int[s] ) { }
      ~Course() { delete [ ] studentIds; }                    // destructor
}

void f () {
      Course c{575};
}                             // c goes out of scope so the destructor called
```

studentIds must be deallocated when c is destroyed to prevent a memory leak

COPY VS ASSIGN

Constructor	`Course c1{};`	// create object
Copy constructor	`Course c2{c1};` `Course c2 = c1;`	// create object from an object // create object from an object
Assignment Overload	`c2 = c1;`	// copy using existing objects

Note copy constructors and assignment operator overload functions are slower than regular constructors because they look up data to copy, especially with dynamic memory (pointer overhead)

ELISION

Elision compiler optimization for passing temporary objects by value

copy constructors are slower than regular constructors

temporary objects passed by value avoid calling the copy constructor,
instead they are constructed in the memory space of the new object

Example

```
void f(Course obj) { }        // function that accepts a course object by value

Course c1{575};               // construct a course object c1
f(c1);                        // passing a named object by value,
                              // calls copy constructor to construct obj

f(Course{});                  // passing a temporary object by value,
                              // constructs the temporary object in the memory space
                              // of obj so the copy constructor is not needed to copy
```

RETURN VALUE OPTIMIZATION

RVO compiler optimization for returning temporary objects by value

temporary objects returned by value avoid calling the copy constructor,
instead they are constructed in the memory space of the new object

Example

```
Course f1() { Course c; return c; }      // function returns temporary by value

Course c1 = Course{};      // temporary object is constructed in the memory space
                           // of c1 so the copy constructor is not needed to copy

Course c2 = f1();          // temporary object returned by value,
                           // constructs the temporary object in the memory space
                           // of c2 so the copy constructor is not needed to copy
```

PASS BY POINTER VS PASS BY REFERENCE

Pass by value	make a copy of data
Pass by pointer	make a copy of the pointer value, which is the memory address of data
Pass by reference	send the memory address of data
Guidelines	with exception to very small primitive data types, pass by pointer and pass reference are faster because the data that is sent is only a memory address pass by pointer and pass by reference are just as fast pass by reference is optimal because of reduced complexity pass by value using move semantics can be faster (not covered in ET580)

PASS BY POINTER VS PASS BY REFERENCE

Syntax

```
void output(Course *c) {}          // pass by pointer
void output(Course &c) {}          // pass by reference


Course *c1 = new Course{};         // dynamic variable
output(c1);                        // pass by pointer
output(*c1);                       // pass by reference

Course c2{};                       // automatic variable
output(&c2);                       // pass by pointer
output(c2);                        // pass by reference
```

STACK VS. HEAP

We now understand that programs can be written to use stack or heap memory. However, as programmers we do not concern ourselves with where information is stored. Instead, we focus upon the lifetime and size of our data and how the use of automatic and dynamic variables impact these concerns.

If manual control of lifetime or significant storage space are required, we use dynamic memory. If performance is our primary concern and the stack provides enough storage, we aim to use automatic memory.

These are very basic and general guidelines, there are always exceptions.