Memory Management

Lecture Objectives

- Understand how storage is allocated and reclaimed in C++
- C++ operators for storage management
  - new and using it in constructors
  - delete and using it in destructors
- Learn how to manage aliases

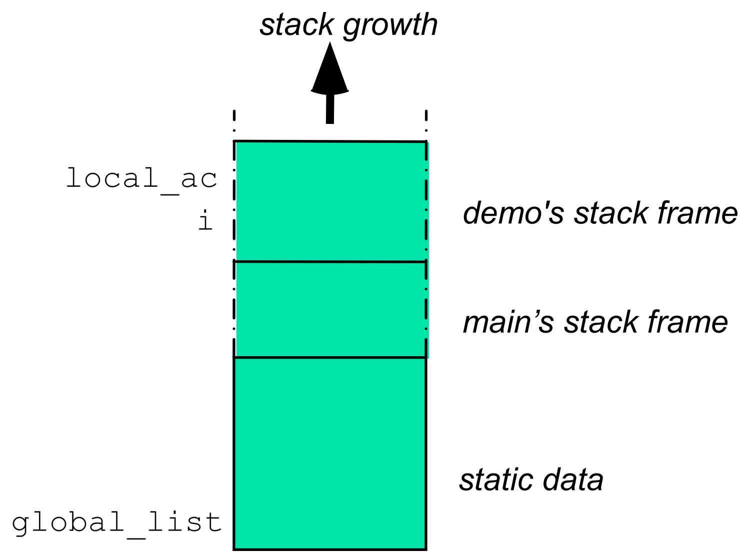
Runtime Storage Allocation

- Static data is allocated at program load time and deallocated on program termination
- Local data is allocated each time the function is called and deallocated when the function call returns

```
const int SIZE = 2;
int global_list[SIZE];

void demo() {
  int i;
  Account local_ac(10);
      ...
}

int main() {
  ...
  demo();
  ...
}
```

SIZE must be a constant expression

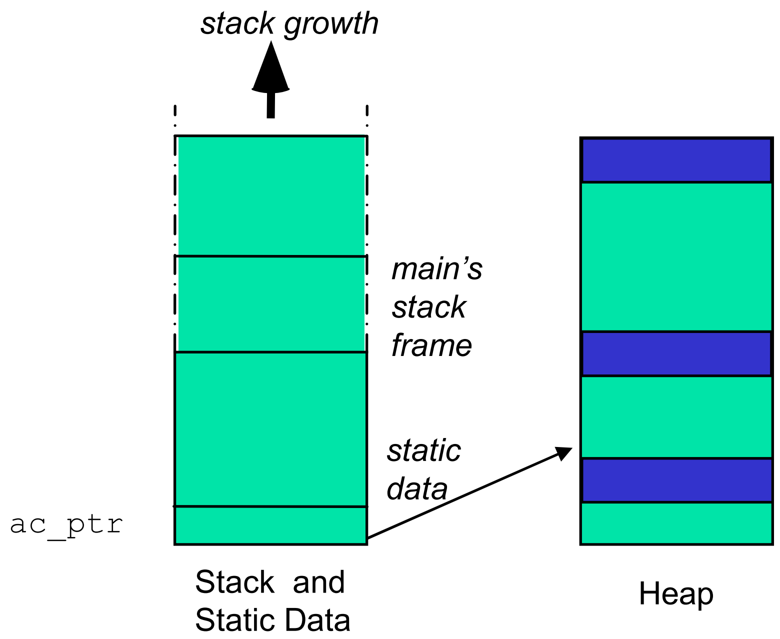
Stack view here



Runtime Storage Allocation (cont.)

- C provides the library functions `malloc()` and `free()` for allocating and deallocating blocks of storage on the heap
  - Unlike stack storage, heap storage is explicitly managed by the application

```
Account* ac_ptr;
ac_ptr =
  (Account*)
  malloc(sizeof(Account));
cout << ac_ptr->get_balance();
```



- What will be output by the `get_balance()` call? Why?

Runtime Storage Allocation (cont.)

- C's `malloc()` bypasses C++'s constructors so should not be used
  - Thus balance is uninitialized in the example above, so `cout << ac_ptr->get_balance()` prints an undefined value
  - C++ provides an operator for dynamic storage allocation: **`new`**

```
Account* ac_ptr;
ac_ptr = new Account;   // ac_ptr->balance == 0
```

Runtime Storage Allocation (cont.)

- In Java, all arrays and class instances are allocated on the heap using a `new` operator
- The C++ compiler generates a call to the default constructor to initialize the space allocated by `new`
  - Constructor arguments are placed after the type name

```
Clip* c_ptr1 = new Clip;  // default constructor
Clip* c_ptr2 = new Clip("holiday.jpg", "My Holiday");
```

- What happens if heap storage is exhausted?

Runtime Storage Allocation (cont.)

- When storage is exhausted, C++ requires `new` to throw a standard exception `bad_alloc`
  - Applications must catch the exception or the program will terminate
- The `new[]` operator is used to allocate an array of objects

```
Clip *c_ptr1 = new Clip[50];

int *i_ptr1 = new int[50];
int *i_ptr2 = new int(50);
```

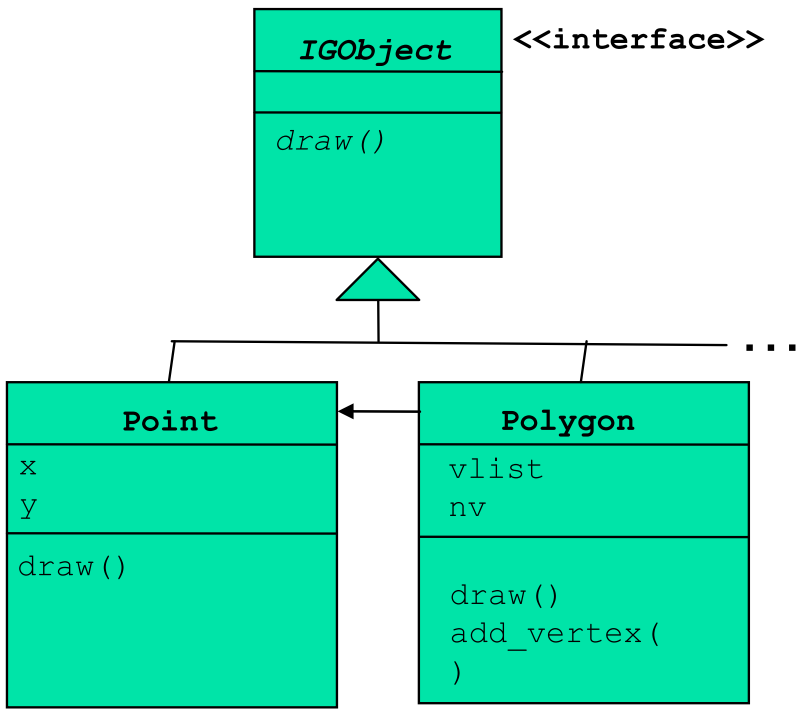
Calls the default constructor `Clip::Clip()` 50 times. A syntax error if no default constructor

Array of 50 `int`s, undefined initial values

WARNING: `new int(50)` not the same as `new int[50]`!!

Dynamically Sized Objects

- Dynamic storage allocation is used to define objects whose size varies
- Assume we have a `GraphicObjects` hierarchy, for a graphics application
  - The base class `IGObject` is an Interface, with a `draw()` function
  - Concrete classes include: `Line`, `Point`, `Polygon`, `Ellipse`, …



Dynamically Sized Objects (cont.)

- We will focus on the implementation of the `Polygon` class
  - `Polygon`'s data members include:
    - `vlist` - the vertex Points
    - `np` - the number of vertices
  - `vlist` can either be an array or a standard container
  - By default, we would use a standard container for simplicity and reliability
  - But we allocate many `Polygon`s with fixed vertices, so we will use an array

Polygon
`vlist` `nv`
`draw()` `add_vertex()`

Dynamically Sized Objects (cont.)

```
// File polygon.h
#include "point.h"
class Polygon : public IGObject {
  Point* vlist; // vertices
  const int nv; // Number of vertices
public:
  Polygon(int init_nv, Plist* init_v = 0);
  void draw();
};
```

```
// File igcobject.h
class IGObject {
  ...
  virtual void draw() = 0;
};
```

For initialization of vlist

```
// File point.h
#include "igobject.h"
#include <vector>
class Point: public IGObject {
  float x,y;
public:
  void draw();
  Point(float x = 0, float y = 0)
    : x(x), y(y) { }
  ...
};
typedef std::vector<Point> PList;
```

Default constructor needed for objects allocated in arrays on the heap

Dynamically Sized Objects (cont.)

- The `Polygon` constructor takes two arguments:
  - The number of vertices, `init_nv`
  - An optional vector of `Point`s, `init_v`, for initializing `vlist`

```
// File polygon.cpp
#include "polygon.h"
Polygon::Polygon(int init_nv, Plist* init_v) :
    nv(init_nv), vlist(new Point[init_nv])
{
  if (!init_v) return; // use default vertex initialization
  Point* vi = vlist;  // vlist iterator
  Plist::iterator pi = init_v->begin(); // init_v iterator
  while (pi != init_v->end()) {
    if (vi >= &vlist[nv]) return; // vlist fully initialized
    *(vi++) = *(pi++); // copy Point
  }
};
```

Defaults on function declarations in `.h` files <u>not</u> on definitions in `.cpp` files or in both `.h` and `.cpp`

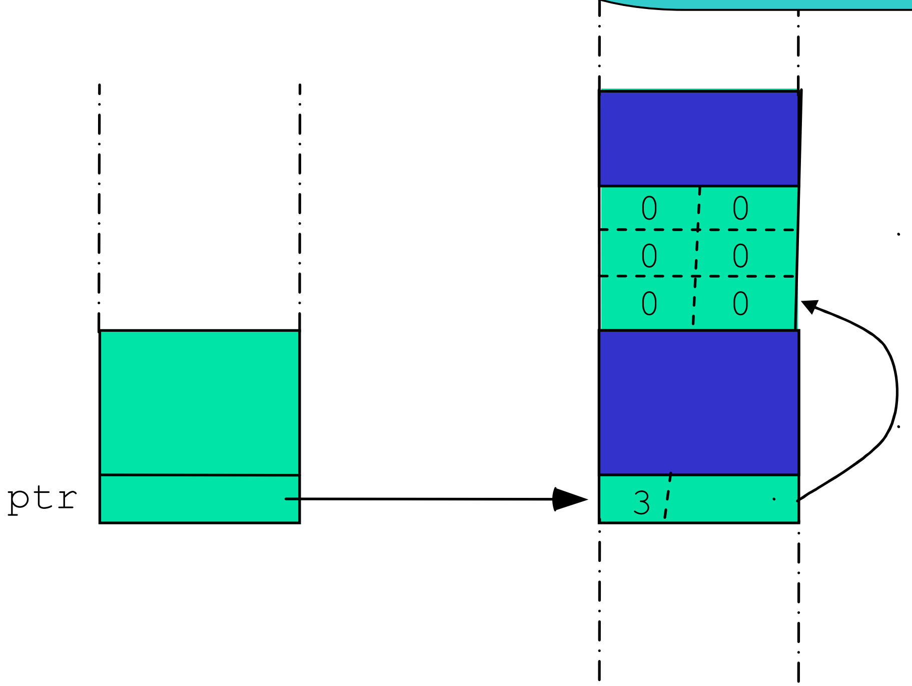
NOT `Point[nv]`! nv not initialized before `vlist`

Dynamically Sized Objects (cont.)

- Heap storage allocation can be recursive

```
Polygon* ptr = new Polygon(3);
```

`new` here allocates a `Polygon` on the heap and calls its constructor, which then allocates an array of 3 Points on the heap which then calls the Point default constructor 3 times



C++ Storage Management

- By default, storage allocated by new persists for the lifetime of the program because C++ has no automatic garbage collection
  - Reclaiming all storage when it is no longer accessible
  - Automatic garbage collection is not provided in C++, because it has a potentially high runtime overhead

- C++ provides a `delete` operator for explicitly reclaiming storage
  - **`delete`** `p` reclaims the object that `p` points to (not `p` itself)

- Java has no `delete`; instead it relies on built-in garbage collection

C++ Storage Management

- `new` and `delete` operators can be overloaded and used to implement garbage collection libraries.

```
Type* p;
p = new Type;
...
delete p;
...
```

`p` now has an undefined value

C++ Storage Management (cont.)

- Suppose we wished to have an `add_vertex()` member in `Polygon`
  - We will keep `Polygon::vlist` as an array for efficiency
  - We assume that `add_vertex()` is rarely called, otherwise it is simpler to just make `vlist` a `std::vector`

```
// File polygon-d.cpp
#include  "polygon-d.h"
... // other member function definitions
void Polygon::add_vertex(const Point& new_vertex) {
  Point* new_vlist = new Point[nv+1];
  for (int i = 0; i<nv; ++i)
    new_vlist[i] = vlist[i];  // copy existing vertices
  new_vlist[nv] = new_vertex; // add new vertex
  ++nv;
  delete [] vlist;
  vlist = new_vlist;
}
```

`nv` can no longer be a `const` member

Array version of `delete`, arrays must be reclaimed with `delete[]`, NOT `delete`

Destructors

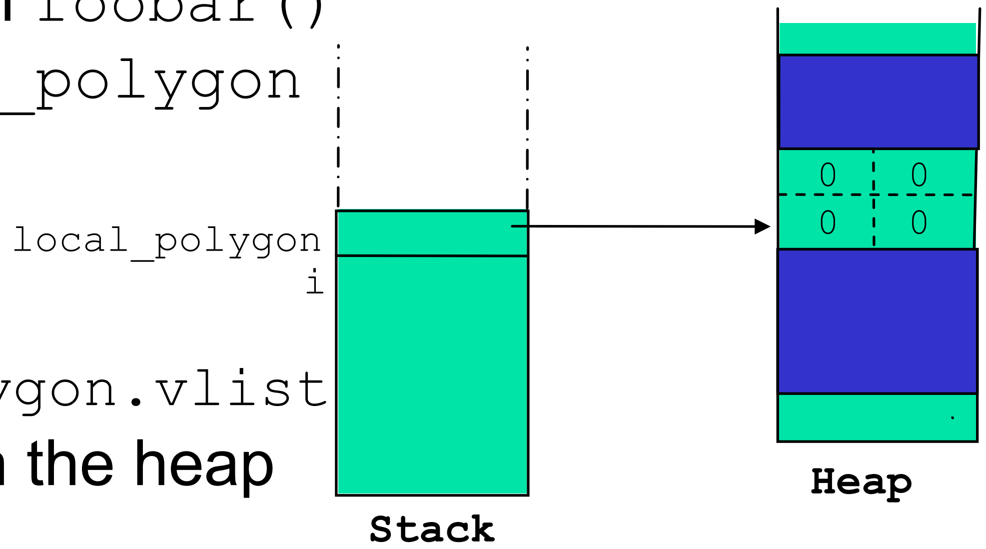
```
void foobar() {
    int i;
    Polygon local_polygon(2);
    ...
}
```

Allocates an array of two Points on the heap

Storage diagram below when here

Stack space for `i` and `local_polygon` reclaimed here

- On return from the function `foobar()` the stack space for `local_polygon` and `i` is reclaimed



- But then what `local_polygon.vlist` points to is left dangling on the heap
- How can we get the heap space back?

Destructors (cont.)

- Member data should be private, so clients of the class have no permission to explicitly delete it
  - In `foobar()` we cannot call

    ```
    delete []local_polygon.vlist
    ```

  - Users of `Polygon` should not know, or care, whether it dynamically allocates storage, `Polygon` should manage what it allocates
- Hence, C++ provides a mechanism, called *destructors*
  - A member function that is implicitly called whenever an object is deallocated
- The declaration of a destructor is just `~ClassName();`

Destructors (cont.)

```
// File polygon.h
#include "point.h"
class Polygon : public IGObject {
  Point* vlist; // vertices
  int nv; // Number of vertices
public:
  Polygon(int init_nv, Plist* init_v = 0);
  ~Polygon() {
    delete [] vlist;
  }
  void draw();
};
```

Destructor: no return type or arguments allowed in the specification

```
void foobar() {
  int i;
  Polygon local_polygon(2);
  ...
}
```

- Now, the memory pointed to by `local_polygon.vlist` is reclaimed when `local_polygon` is deallocated, just before `foobar()` returns
- delete of a `null` pointer is allowed and has no effect
  - There is no need to test for `(vlist != 0)` before `delete [] vlist` in the destructor

Destructors (cont.)

- An object's scope determines when its destructors is called

```
Polygon p_global_1(2);
Polygon p_global_2(3);

void farkle() {
  Polygon p_local_1(3);
  for (int i = 0; i<42; ++i) {
    Polygon p_local_2(4);
    ...
  }
  ...
}
int main() {
  farkle()
  ...
}
```

Global (static) object' s destructors called on exit from main(), in reverse order of declaration/construction: p_global_2 then p_global_1

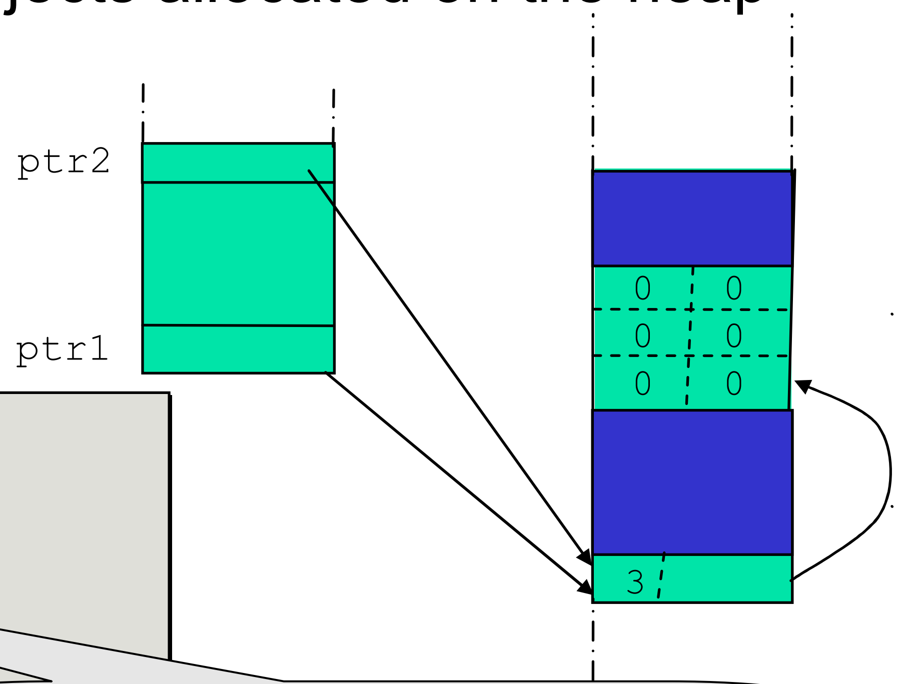
Local object' s destructors called on exit from their scope order of declaration/construction: p_local_1 called on exit from farkle; p_local_2 called at the end of <u>each</u> loop iteration

Destructors (cont.)

- When should destructors of objects allocated on the heap be called?

```
Polygon* ptr1;
```

```
void doit() {
Polygon* ptr2 = new Polygon(3);
  ptr1 = ptr2;
  // ...
}
```



Storage diagram when program is here.
ptr1 and ptr2 point to the same Polygon now

- When is it “safe” to call `*ptr2`’s (or equivalently `*ptr1`'s) destructor?

Destructors (cont.)

- No record is kept of how many pointers point to an object in C++
  - So in general it is never "safe" for a compiler to generate a call to a heap object's destructor!
  - So a heap object's destructor is <u>never</u> implicitly called
  - But deleting a heap object will cause that object's destructor to be called

```
delete ptr2;
```

Compiler notices that *ptr2 has a destructor, and generates a call to that destructor (deleting `ptr2->vlist`) before reclaiming `*ptr2`

Destructors (cont.)

- What would have happened if we had copied the pointer earlier?

```
ptr1 = ptr2;
delete ptr2;
...
```

ptr1 now points to “orphan storage” it may be reused for something else anytime soon or never, depending on run time needs

- It is a <u>programmer’s responsibility</u> to ensure that there are:
  - no aliases, or other pointers, to the object or its member data when an object is reclaimed

Destructors (cont.)

```
// File polygon.h
#include "point.h"
class Polygon : public Gobject {
  Point* vlist;
  const int nv;
public:
  ...
  ~Polygon() {
    delete [] nv;
  }
};
```

```
#include "Polygon.h"
...
GObject* p;
...
p = new Polygon(6);
...
delete p;
  ...
```

```
// File gcobject.h
class Gobject {
  ...
public:
  virtual ~GObject() {
    logfile
      << "Deleting Object: "
      << this << "\n";
  }
  virtual void draw() = 0;
};
```

`GObject` destructor logs deletions

- Destructors are useful for more than just reclaiming storage
  - Use include closing files, maintaining counts or logs of deleted objects

Destructors (cont.)

- Destructors should always be declared `virtual`
  - Otherwise delete p above will not call `Polygon`'s destructor!
- Destructors are called in the opposite order to constructors: “derived then base…”

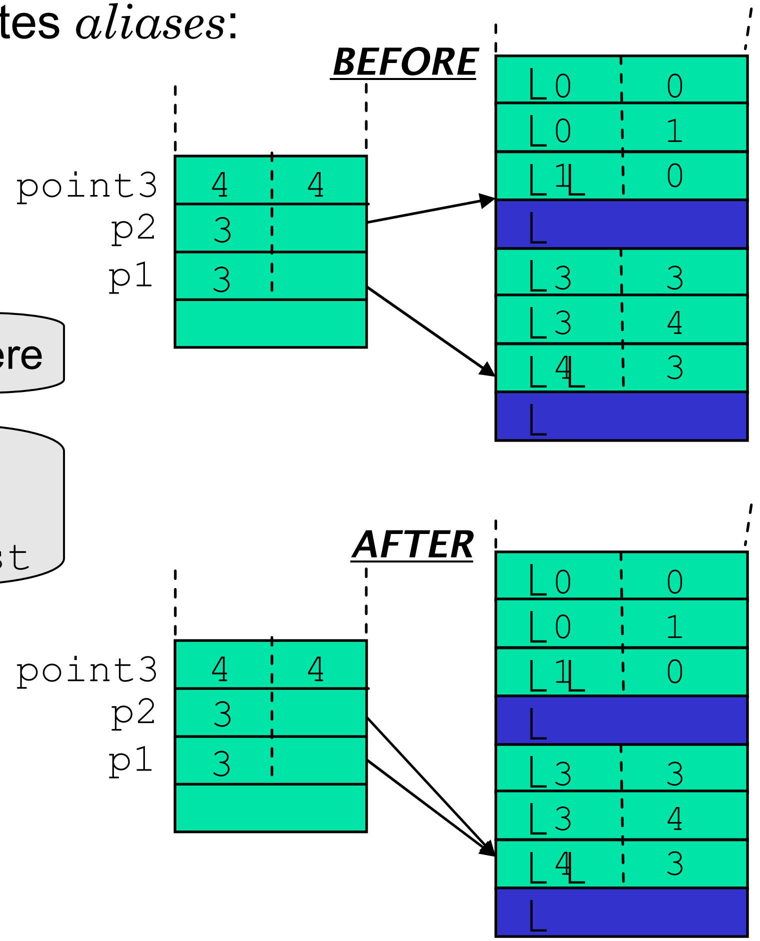
Aliases and Dynamic Allocation

- **For any class, the default assignment operator uses *memberwise* assignment of the data members and base class (if any)**
  - If data members are pointers, this creates *aliases*: two names for the same storage

```
Polygon p1(3, initList1);
Polygon p2(3, initList2);
Point point3(4, 4);

p2 = p1;

...
p1.add_vertex(point3);
```

BEFORE diagram here

AFTER diagram here
now `p2.nv == p1.nv`
and `p2.vlist == p1.vlist`



Aliases and Dynamic Allocation (cont.)

- The default assignment operator causes
  - *Memory leaks*
    - `p1`'s original vertex list `vlist` is inaccessible
  - Side effects if one object is modified, such as:
    - Calling `p2`'s destructor means that `p1` points to deallocated storage
    - When we add a `vertex` to `p1`, `p2` does not know about it

The Assignment Operator

- The assignment operator can be defined as
  - Object Copy - a *deep copy*, instead of a default "member-wise copy"
  - Pointer Copy - or shallow copy

```
Polygon& Polygon::operator=(const Polygon& p) {
  if (this == &p)
    return *this;
  delete [] vlist;
  nv = p.nv
  vlist = new Point[p.nv];
  for (int i = 0; i<nv; ++i)
    vlist[i] = p.vlist[i];
  return *this;
}
```

Check for "self-assignment, e.g., `p1 = p1` if so, just return

Reclaim old vlist

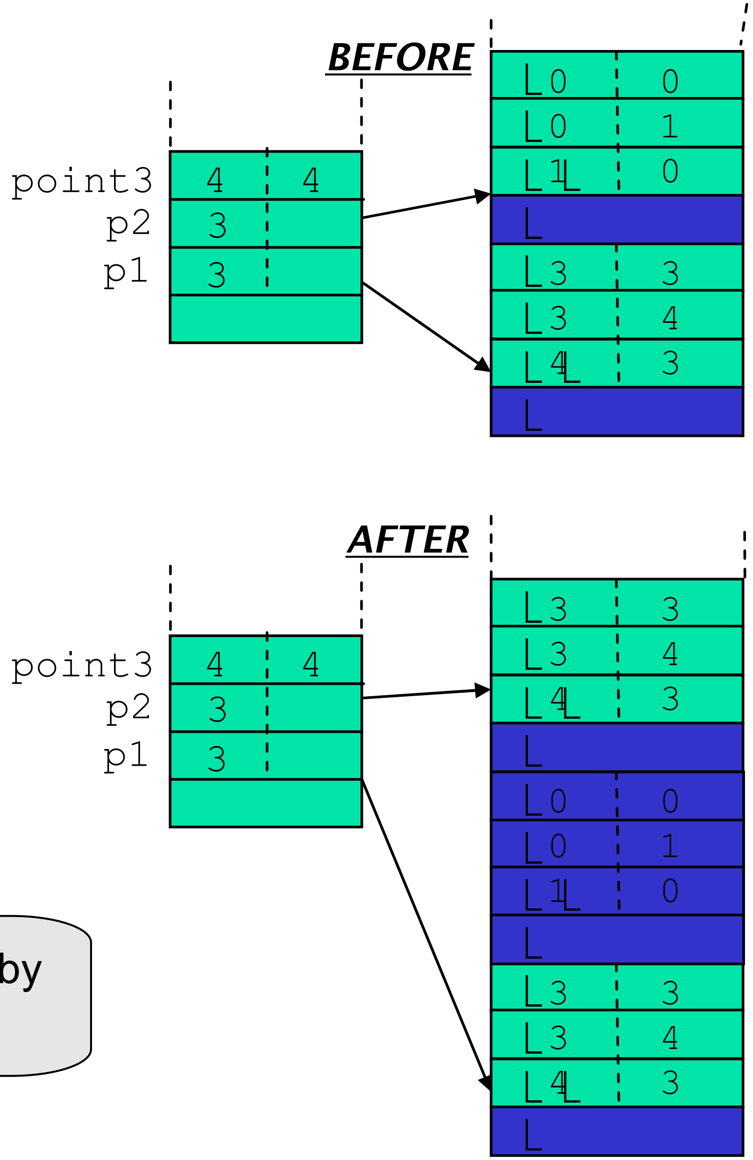
Allocate new vlist of the right size

Copy each of the Points in the Polygon, calling the Points copy constructor

return current object by reference for consistency with predefined operators

Assignment Operators (cont.)

- **Now assignment works correctly**
  - `p2` has its own copy of `p1`'s `Points` and storage has been reclaimed




```
Polygon p1(3, initList1);
Polygon p2(3, initList2);
Point point3(4, 4);

p2 = p1;



...
p1.add_vertex(point3);
```

now `p2` unaffected by operations on `p1`

Copy Constructors

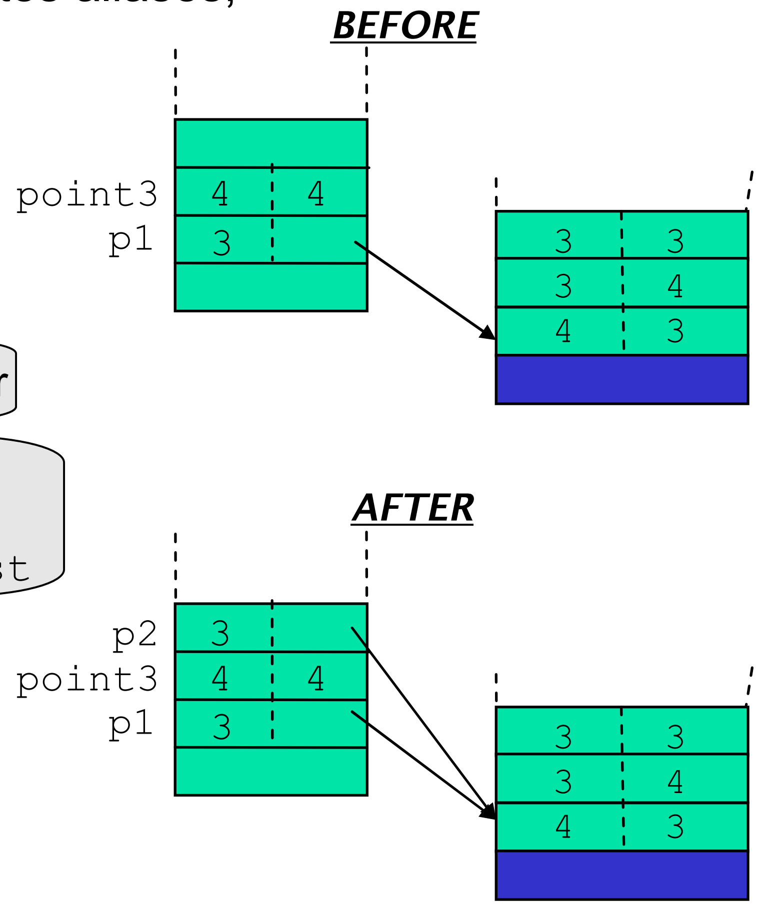
- **Like assignment, the default copy constructor creates a shallow, member-wise copy**
  - If data members are pointers, this creates aliases, just as assignment did

```
Polygon p1(3, initList1);
Point point3(4, 4);

Polygon p2(p1);

...
p1.add_vertex(point3);
```

BEFORE diagram here

call copy constructor

AFTER diagram here
now `p2.nv == p1.nv`
and `p2.vlist == p1.vlist`



Copy Constructors (cont.)

- A member copy constructor that creates a deep copy is needed
  - Similar to that for assignment
  - Except that no storage needs to be deleted, and a check for self-assignment is not needed, as `Polygon p1(p1)` is not syntactically legal

```
Polygon:: Polygon(const Polygon& p)
  : nv(p.nv),
    vlist(new Point[p.nv]) {
  for (int i = 0; i<nv; ++i)
    vlist[i] = p.vlist[i];
  return;
}
```

Override default: `vlist(p.vlist)`

Copy each of the `Point`s in the `Polygon`, calling the `Point`s copy constructor

Principles for Dynamic Allocation

- Dynamic storage allocation is a perennial source of bugs and problems in C++ applications
- Both allocation and deallocation need to be planned at design time, not added as an afterthought
- All storage needs to have “an owner”, who is responsible for allocating and deallocating it
  - Either the class that has the pointer members, such as `Polygon` above, or a manager class, such as `GOManager`, if sharing is objects is required
  - Sharing objects usually requires a “reference counting idiom” using a proxy design pattern
- There is a standard C++ coding idiom for implementing deep copies of objects that have dynamically allocated, unshared, data members

Principles for Dynamic Allocation (cont.)

```
class C {
  T* tp;
public:
  C() { tp = new T; ...}  // constructor
  C(const C& c)                 // copy constructor
      : tp(new T)   // or new T[...] for an array {
    // copy *(c.tp) into *tp
  }
  C& operator= (const C& c) { // assignment operator
    if (this == &c) return *this;
      delete tp;    // or delete [ ] if an array allocated
      tp = new T;   // or new T[...] for an array
      // copy *(c.tp) into *tp
      return *this;
  }
  virtual ~C() {                // destructor
    delete tp;  // or delete [ ] if an array allocated
  }
};
```