Life cycle of an object

- **construction: creating a new object**
  - implicitly, by entering the scope where it is declared
  - explicitly, by calling **`new`**
  - construction includes initialization
- **copying: using existing object to make a new one**
  - "copy constructor" makes a new object from existing one of the same kind
  - implicitly invoked in (some) declarations, function arguments, function return
- **assignment: changing an existing object**
  - occurs explicitly with =
  - meaning of explicit and implicit copying must be part of the representation
    default is member-wise assignment and initialization
- **destruction: destroying an existing object**
  - implicitly, by leaving the scope where it is declared
  - explicitly, by calling **`delete`** on an object created by **`new`**
  - includes cleanup and resource recovery

Strings: constructors & assignment

- **another type that C and C++ don't provide**
- **implementation of a String class combines**
  - constructors, destructors, copy constructor
  - assignment, operator =
  - constant references
  - handles, reference counts, garbage collection

- **Strings should behave like strings in Awk, Perl, Java**
  - can assign to a string, copy a string, etc.
  - can pass them to functions, return as results, …
- **storage managed automatically**
  - no explicit allocation or deletion
  - grow and shrink automatically
  - efficient

- **can create String from "..." C char* string**
- **can pass String to functions expecting char***

"Copy constructor"

- **when a class object is passed to a function, returned from a function, or used as an initializer in a declaration, a copy is made:**

```
String substr(String s, int start, int len)
```

- **a "copy constructor" creates an object of class X from an existing object of class X**
- **obvious way to write it causes an infinite loop:**

```
class String {
   String(String s) {...} // doesn't work
};
```

- **copy constructor parameter must be a reference so object can be accessed without copying**

```
class String {
     String(const String& s) {...}
     // ...
};
```

- **copy constructor is necessary for declarations, function arguments, function return values**

String class

```
class String {
  private:
    char    *sp;
  public:
    String() { sp=strdup(""); }  // String s;
    String(const char *t) { sp=strdup(t); } // String s("abc");
    String(const String &t) { sp=strdup(t.sp); } // String s(t);
    ~String() { delete [] sp; }

    String& operator =(const char *);// s="abc"
    String& operator =(const String &);// s1=s2

    const char *s() { return sp; } // as char*
};
```

- **assignment is not the same as initialization**
  - changes the state of an existing object
- **the meaning of assignment defined by a member function named operator=**

  x = y means **x.operator=(y)**

Assignment operators

```
String& String::operator =(const char *t) { // s = "abc"
    delete [] sp;
    sp = strdup(t);
    return *this;
}
String& String::operator=(const String& t) { // s1 = s2
    if (this != &t) {  // avoid s1 = s1
        delete [] sp;
        sp = strdup(t.sp);
    }
    return *this;
}
```

- **in a member function, `this` points to current object, so `*this` is a reference to the object**
- **assignment operators almost always end with**

  `return *this`

  which returns a reference to the LHS
  - permits multiple assignment `s1 = s2 = s3`

String class complete

```
class String {
  private:
    char    *sp;
  public:
    String()  { sp=strdup(""); }  // String s;
    String(const char *t) { sp=strdup(t); } // String s("abc");
    String(const String &t) { sp=strdup(t.sp); } // String s(t);
    ~String() { delete [] sp; }

    String& operator =(const char *);// s="abc"
    String& operator =(const String &);// s1=s2

    const char *s() { return sp; } // as char*
};
String& String::operator =(const char *s) {
        if (sp != s) {
                delete [] sp;
                strdup(s);
        }
        return *this;
}
String& String::operator =(const String &t) {
        if (this != &t) {
                delete [] sp;
                strdup(t.sp);
        }
        return *this;
}
```

continued

```
main()
{
    String s = "abc", t = "def", u = s, w;

    printf("%s %s %s [%s]\n",
       s.s(), t.s(), u.s(), w.s());
    s = "1234";
    s = s;
    printf("s=%s\n", s.s());
    s = s.s();
    printf("s2=%s\n", s.s());
    printf("u=%s\n", u.s());
    s = t = u = "asdf";
    printf("%s %s %s\n", s.s(), t.s(), u.s());
}
```

Handles and reference counts

- **how to avoid unnecessary copying for classes like strings, arrays, other containers**
- **copy constructor may allocate new memory even if unnecessary**
  - e.g., in f(const String& s) string value would be copied even if it won't be changed by f
- **a handle class manages a pointer to the real data**
- **implementation class manages the real data**
  - string data itself
  - counter of how many Strings refer to that data
  - when String is copied, increment the ref count
  - when String is destroyed, decrement the ref count
  - when last reference is gone, free all allocated memory
- **with a handle class, copying only increments reference count**
  - "shallow" copy instead of "deep" copy

Reference/Use counts

```
class Srep {   // string representation
   char *sp;   // data
   int   n;    // ref count
   Srep(const char *);
   friend class String;
};
Srep::Srep(const char *s) {
   if (s == NULL)
      s = "";
   sp = strdup(s);
   n = 1;
}
class String {
   Srep *r;
 public:
   String(const char *);
   String(const String &);
   ~String();

   String& operator =(const String &);  // s1 = s2;
   String& operator =(const char *); // s = "abc";
   const char *s() { return r->sp; }
};
```

use counts, part 2

```
String::String(const char *s = "") {
   r = new Srep(s); // String s="abc"; String s1;
}
String::String(const String &t) { // String s=t;
   t.r->n++;   // ref count
   r = t.r;
}
String::~String() {
   if (--r->n <= 0) {
      delete [] r->sp;
      delete r;
   }
}
String& String::operator =(const char *s) {
   if (r->n > 1) {      // disconnect self
      r->n--;
      r = new Srep(s);
   } else {
      delete [] r->sp;   // free old String
      r->sp = strdup(s);
   }
   return *this;
}
String& String::operator =(const String &t) {
   t.r->n++;   // protect against s = s
   if (--r->n <= 0) {   // nobody else using it
      delete [] r->sp;
      delete r;
   }
   r = t.r;
   return *this;
}
```

Rules for constructors and assignment operators

- **all objects have to have a constructor**
  - if you don't specify a constructor the default constructor copies members by their constructors
  - need a no-argument constructor for arrays
  - constructors should initialize all members
- **if constructor calls new, destructor must call delete**
  - use **delete [ ]** for an array allocated with **new T[n]**
- **copy constructor X(const X&) makes an object**
  - from another one without making an extra copy
- **if there's a complicated constructor**
  - there will have to be an assignment operator
  - make sure that x = x works
- **assignment is NOT the same as construction**
  - constructors called in declarations, function arguments and function returns, to make a new object
  - assignments called only in assignment statements

    to clobber an existing object

Inheritance

- **a way to create or describe one class in terms of another**
  - "a D is like a B, with these extra properties..."
  - "a D is a B, plus…"
  - B is the **base** class or **super**class
  - D is the **derived** class or **sub**class

    Perl & C++ use base/derived; Java uses super/sub

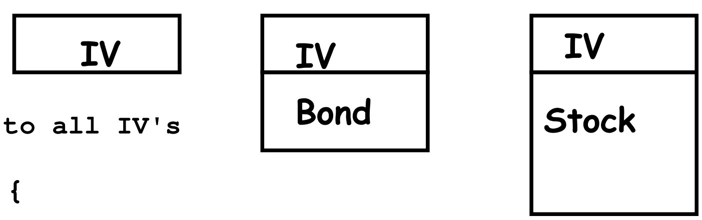
- **inheritance is used for classes that model strongly related concepts**
  - objects share some common properties, behaviors, ...
  - and have some properties and behaviors that are different

- **base class contains aspects common to all**
- **derived classes contain aspects different for different kinds**

Inheritance and derived classes

- **consider different kinds of Investment Vehicles**
  - stocks, bonds, commodities, currencies, ...
- **base class IV contains aspects common to all**
  - name
  - description
- **derived classes contain aspects that are different for different kinds**
  - stock: ticker symbol, exchange, common/preferred, ...
  - bond: coupon, maturity, callable, call date...
  - fund: vector of IVs
- **sometimes you care about the difference**
  - dividend rate vs. interest rate
- **sometimes you don't**
  - closing price

Derived classes




```
class IV {
     string name;
     void price();
     // other items common to all IV's
};
class Stock : public IV {
    String ticker;
    // other items specific to Stocks
};
class Bond : public IV {
    double coupon;
    bool callable;
    // other items specific to Bonds
};
```

- **a Stock is a derived class of (a kind of) IV**
  - a Stock "is a" IV
  - inherits all members of IV
  - adds its own members
- **a Bond is also a derived class of IV**

More on derived classes

- **derived classes can add their own data members**
- **can add their own member functions**
- **can override base class functions with functions of same name and argument types**

```
class Stock : public IV {
    String ticker;
  public:
    void price() {...} // overrides IV::price()
};
class Bond : public IV {
    bool callable;
  public:
    bool is_callable() {...}
    void price() {...}  // overridesIV::price()
};

   Stock gm;
   Bond  ibm;

   gm.price();   // calls Stock::price()
   ibm.price();  // calls Bond::price()
```

Virtual Functions

- **what if we have bunch of different IVs and want to price them all in a loop?**
- **virtual function mechanism lets each object carry information about what functions to apply**

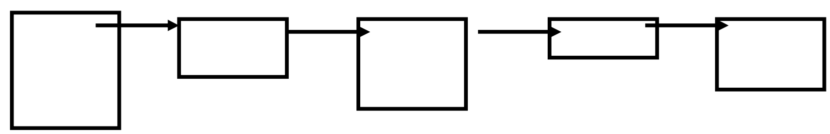
```
class IV {
  public:
    virtual void price();
     ...
};
```

- **"virtual" means that a derived class may provide its own version of this function, which will be called automatically for instances of that derived class**
- **base class can provide a default implementation**
- **a "pure" base class must be derived from**
  - can't exist on its own

Polymorphism

- when a pointer or reference to a base-class type points to a derived-class object
- and you use that pointer or reference to call a virtual function
- this calls the derived-class function
- "polymorphism": proper function to call is determined at run-time
- e.g., pricing IVs on a linked list:

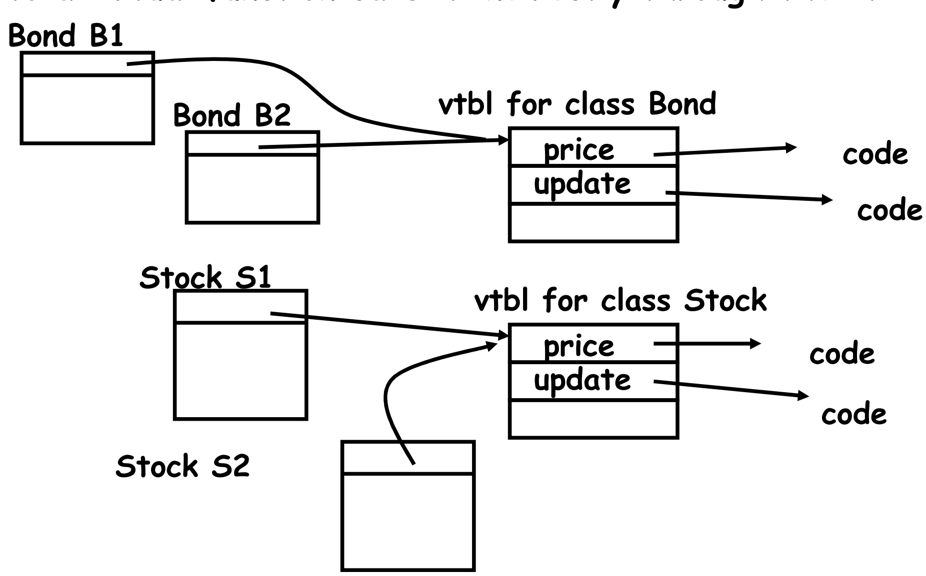
```
price_all(IV *ip) {
   for ( ; ip != NULL; ip = ip->next)
      ip->price();
}
```



- virtual function mechanism automatically calls the right price() function for each object
- the loop does not change if more kinds of IVs are added

Implementation of virtual functions

- each class object has one extra word that holds a pointer to a table of virtual function pointers ("vtbl") (only if class has virtual functions)
- each class with virtual functions has one vtbl
- a call to a virtual function calls it indirectly through the vtbl



Summary of inheritance

- **a way to describe a family of types**
- **by collecting similarities (base class)**
- **and separating differences (derived classes)**

- **polymorphism: proper member functions determined at run time**
  - virtual functions are the C++ mechanism

- **not every class needs inheritance**
  - may complicate without compensating benefit

- **use composition instead of inheritance?**
  - an object contains an (has) an object
    rather than inheriting from it
- **"is-a" versus "has-a"**
  - inheritance describes "is-a" relationships
  - composition describes "has-a" relationships

Templates (parameterized types, generics)

- **another approach to polymorphism**
- **compile time, not run time**
- **a template specifies a class or a function that is *the same* for several types**
  - except for one or more **type parameters**

- **e.g., a vector template defines a class of vectors that can be instantiated for any particular type**

```
vector<int>
vector<String>
vector<vector<int> >
```

- **templates versus inheritance:**
  - **use inheritance when behaviors are different for different types**
    pricing different IVs is different
  - **use template when behaviors are the same, regardless of types**
    accessing the n-th element of a vector is the same,
    no matter what type the vector is

Vector template class

- **vector class defined as a template, to be instantiated with different types of elements**

```
template <typename T> class vector {
    T *v;      // pointer to array
    int size; // number of elements
  public:
    vector(int n=1) { v = new T[size = n]; }
    T& operator [](int n) {
        assert(n >= 0 && n < size);
        return v[n];
    }
};

vector<int> iv(100);            // vector of ints
vector<complex> cv(20);         // vector of complex
vector<vector<int> > vvi(10); // vector of vector of int
vector<double> d;               // default size
```

- **compiler instantiates whatever is used**

Template functions

- **can define ordinary functions as templates**
  - *e.g.*, `max(T, T)`

```
template <typename T> T max(T x, T y) {
    return x > y ? x : y;
}
```

- **requires operator> for type T**

  already there for C's arithmetic types

- **don't need a type name to use it**

  compiler infers types from arguments

  `max(double, double)`

  `max(int, int)`

  `max(int, double)` doesn't compile: no coercion

- **compiler instantiates code for each different use in a program**

Scoped pointer class

- **allocates space when used**
- **frees it automatically when pointer goes out of scope**

```
template <typename T> class SP {
   T *tptr;
  public:
   SP(T *p) { tptr = p; }
   ~SP() { printf("SP destructor %s\n", tptr->fs); delete tptr; }
   T* operator ->() { printf("op->%s\n", tptr->fs); return tptr;
};
class ptr {
  public:
   char *fs;
   ptr(char *s) { printf("construct ptr(%s)\n", fs=strdup(s)); }
   ~ptr() { printf("destruct ptr(%s)\n", fs); delete fs; }
};
int main() {
   printf("start\n");
   SP<ptr> ptr1p = new ptr("new ptr1");
   SP<ptr> ptr2p = new ptr("new ptr2");
   ptr1p->fs = "change ptr1 value";
   printf("end\n");
}
```

Standard Template Library (STL)

Alex Stepanov

(GE > Bell Labs > HP > SGI > Compaq > Adobe)

- **general-purpose library of**
  - **containers (vector, list, set, map, …)**
  - **generic algorithms (find, replace, sort, …)**
- **algorithms written in terms of iterators performing specified access patterns on containers**
  - rules for how iterators work, how containers have to support them
- **generic: every algorithm works on a variety of containers, including built-in types**
  - e.g., find elements in char array, vector<int>, list<…>
- **iterators: generalization of pointer for uniform access to items in a container**

Containers and algorithms

- **STL container classes contain objects of any type**
  - sequences: vector, list, slist, deque
  - sorted associative: set, map, multiset, multimap
    - hash_set and hash_map are non-standard
- **each class is a template that can be instantiated to contain any type of object**
- **generic algorithms**
  - find, find_if, find_first_of, search, ...
  - count, min, max, …
  - copy, replace, fill, remove, reverse, …
  - accumulate, inner_product, partial_sum, …
  - sort
  - binary_search, merge, set_union, …
- **performance guarantees**
  - each combination of algorithm and iterator type specifies worst-case (O(…)) performance bound
    - e.g., maps are O(log n) access, vectors are O(1) access

Iterators

- **a generalization of C pointers**

```
for (p = begin; p < end; ++p)
    do something with *p
```

- **range from `begin()` to just before `end()` [begin, end)**
- **`++iter` advances to the next if there is one**
- **`*iter` dereferences (points to value)**
- **uses operator `!=` to test for end of range**

```
for (iter i = v.begin(); i != v.end(); ++i)
    do something with *i
```

```
#include <vector>
#include <iterator>
using namespace ::std;
int main() {
   vector<double> v;
   for (int i = 1; i <= 10; i++)
      v.push_back(i);
   vector<double>::const_iterator it;
   double sum = 0;
   for (it = v.begin(); it != v.end(); ++it)
      sum += *it;
   printf("%g\n", sum);
}
```

Iterators (2)

- **no change to loop if type or representation changes**

```
set<double> v;
set<double>::const_iterator it;
for (it = v.begin(); it != v.end(); ++it)
    sum += *it;
```

- **not all containers support all iterator operations**

- **input iterator**
  - can only read items in order, can't store into them (input from file)
- **output iterator**
  - can only write items in order, can't read them (output to a file)
- **forward iterator**
  - can read/write items in order, can't go backwards (singly-linked list)
- **bidirectional iterator**
  - can read/write items in either order (doubly-linked list)
- **random access iterator**
  - can access items in any order (array)

Example: STL sort

```
#include <iostream>
#include <iterator>
#include <vector>
#include <string>
#include <algorithm>
using namespace ::std;

int main() {  // sort stdin by lines
    vector<string> vs;
    string tmp;
    while (getline(cin, tmp))
        vs.push_back(tmp);
    sort(vs.begin(), vs.end());
    copy(vs.begin(), vs.end(),
        ostream_iterator<string>(cout, "\n"));
}
```

- **vs.push_back(s) pushes s onto "back" (end) of vs**

- **3rd argument of copy is a "function object" that calls a function for each iteration**
  - uses overloaded operator()

Function objects

- **anything that can be applied to zero or more arguments to get a value and/or change the state of a computation**
- **can be an ordinary function pointer**
- **can be an object of a type defined by a class in which the function call operator `operator()` is overloaded**

```
template <typename T> class bigger {
  public:
    bool operator()(T const& x, T const& y) {
       return x > y;
    }
};
```

- **to sort strings in decreasing order,**

```
vector<string> vs;
sort(vs.begin(), vs.end(), bigger<string>());
```

- **to sort numbers in decreasing order,**

```
vector<double> vd;
sort(vd.begin(), vd.end(), bigger<double>());
```

Template metaprogramming

- **do computation at compile time to avoid computation at run time**
  - evaluating constants, unrolling loops, building data structures

```
// from effective c++ 3e, by scott meyers

#include <iostream>
using namespace ::std;

template<unsigned n> struct Factorial {
   enum { value = n * Factorial<n-1>::value };
};
template<> struct Factorial<0> {
   enum { value = 1 };
};

int main()
{
   std::cout << Factorial<5>::value << "\n";
   std::cout << Factorial<10>::value << "\n";
}
```

Word frequency count: C++ STL

```
#include <iostream>
#include <map>
#include <string>

int main() {
    string temp;
    map<string, int> v;
    map<string, int>::const_iterator i;

    while (cin >> temp)
        v[temp]++;
    for (i = v.begin(); i != v.end(); ++i)
        cout << i->first << " "
             << i->second << "\n";
}
```

Exception handling

- **necessary so libraries can propagate errors back to users**

```
class subscriptrange {
  public:
    int n;
    subscriptrange(int n) { this->n = n; }
};
int& ivec::operator [](int n) {
    if (n < 0 || n >= size)
        throw subscriptrange(n);
    else
        return v[n];
}
int g(ivec& v) { return v[1000]; }

int f() {
    ivec iv(100);
    try {
        printf("normal\n");
        return g(iv);   // normal return if no exceptions
    } catch (subscriptrange sr) {
        printf("subscriptrange %d\n", sr.n);
        return 0; // if subscriptrange raised in g() or anything it cal
    } catch (...) {       // get here if some other
        printf("other\n");
        return -1;      // exception was raised
    }
}
```

What to use, what not to use?

- **Use**
  - classes
  - const
  - const references
  - default constructors
  - C++ -style casts
  - bool
  - new / delete
  - C++ string type

- **Use sparingly / cautiously**
  - overloaded functions
  - inheritance
  - virtual functions
  - exceptions
  - STL

- **Don't use**
  - malloc / free
  - multiple inheritance
  - run time type identification
  - references if not const
  - overloaded operators (except for arithmetic types)
  - default arguments (overload functions instead)