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 `=, +=, etc.
  - 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, Python, 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:
  ```c++
  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:
  ```c++
  class String {
      String(String s) {...} // doesn't work
  };
  ```
- copy constructor parameter must be a reference so object can be accessed without copying
  ```c++
  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

```cpp
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 the object (returned as a reference)
- assignment operators almost always end with
  ```cpp
  return *this
  ```
  which returns a reference to the LHS
  - permits multiple assignment `s1 = s2 = s3`
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 counts

\[
s = "abc"
\]

\[
t = s
\]

\[
t = "def"
\]
Reference/Use counts

class Srep {   // string representation
    char *sp;   // data
    int n;      // ref count
    Srep(const char *s = "") : n(1), sp(strdup(s)) {} 
    ~Srep() { delete [] sp; } 
    friend class String;
};

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; }
};
Reference counts, part 2

// constructors, destructor

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;
    }
}
Reference counts, part 3

```cpp
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 me now
        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
  - you 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 modify 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 superclass
  - D is the derived class or subclass
    - C++, Perl, Python, ... use base/derived; Java, Ruby, ... use 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 Shapes
  - lines, polylines, rectangles, squares, circles, ellipses, ...

• base class Shape handles methods and properties common to all
  - color, text, ...

• derived classes contain aspects that are different for different kinds
  - line: start, end, ...
  - rectangle: origin, corner, ...
  - circle: center, radius

• sometimes you care about the difference

• sometimes you don't
Derived classes

class Shape {  
    int color;  
    Shape& draw();  
    // other items common to all Shapes  
};
class Rect: public Shape {  
    Point origin; double ht, wid;  
    // other items specific to Lines  
};
class Circle: public Shape {  
    Point center; double rad;  
    // other items specific to Bonds  
};

• a Rect is a derived class of (a kind of) Shape
  - a Rect "is a" Shape
  - inherits all members of Shape
  - adds its own members

• a Circle is also a derived class of Shape
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 the same name and argument types

```cpp
class Rect: public Shape {
    Point origin; double ht, wid;

    public:
    bool is_square() {...} 
    Shape& draw() {...} // overrides Shape::draw()
};
class Circle: public Shape {
    Point center; double rad;
    public:
    Shape& draw() {...} // overrides Shape::draw()
};
```

Rect r;
Circle c;

r.draw(); // calls Rect::draw()
c.draw(); // calls Circle::draw()
Virtual Functions

• a function in a base class that can be overridden by a function in a derived class (with same name and arguments)

```cpp
class Shape {
    public:
        virtual Shape& draw();
    ...
};
```

• "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
• the base class can provide a default implementation
• if the base class is "pure", it must be derived from
  - pure base class can't exist on its own; no default implementation
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., drawing Shapes on a linked list:

```c
draw_all(Shape *sp) {
    for ( ; sp != NULL; sp = sp->next)
        sp->draw();
}
```

- virtual function mechanism automatically calls the right draw() function for each object
- the loop does not change if more kinds of Shapes are added
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
drawing different Shapes 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

```cpp
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];
    }
};
```

```cpp```
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 types are used
Template functions

• can define ordinary functions as templates
  - e.g., \( \text{max}(T, T) \)

  \[
  \text{template <typename T> T max(T x, T y) \{ }
  \[
  \hspace{1em} \text{return } x > y \ ? \ x : y; \\
  \hspace{1em} \}
  \]

• 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
  \[
  \text{max(double, double)} \\
  \text{max(int, int)} \\
  \text{max(int, double) doesn't compile: no coercion}
  \]

• compiler instantiates code for each different use in a program
Standard Template Library (STL)

Alex Stepanov

(GE > Bell Labs > HP > SGI > Compaq > Adobe -> A9)

- 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 in C++11, as "unordered_set" and "unordered_map"

• each container 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

```cpp
multiset<double> v;
multiset<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 (e.g., 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

```cpp
#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 \texttt{operator()} is overloaded

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

• to sort strings in decreasing order,
  \begin{verbatim}
  vector<string> vs;
  sort(vs.begin(), vs.end(), bigger<string>());
  \end{verbatim}

• to sort numbers in decreasing order,
  \begin{verbatim}
  vector<double> vd;
  sort(vd.begin(), vd.end(), bigger<double>());
  \end{verbatim}
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";
}
Some C++11 additions

- **nullptr**
  - type-safe and unambiguous replacement for NULL and 0 pointer values
- **auto**
  
  ```
  auto x = val;
  ```
  replaces
  ```
  VeryLongTypeNameLikeWhatYouOftenSeeInJava x = val;
  ```
  infers the type of `x` from the type of the initializing value
- **range for**
  
  ```
  for (v : whatever) ...
  ```
  replaces
  ```
  for (v = whatever.begin(); v != whatever.end(); ++v) ...
  ```

  ```
  for (std::vector<int>::const_iterator it = myvector.begin();
      it != myvector.end(); ++it)
  ```

  becomes
  ```
  for (auto it = myvector.begin(); it != myvector.end(); ++it)
  ```

  becomes
  ```
  for (auto it : myvector)
  ```
Word frequency count: C++ STL

```c++
#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";
}
```
Further reading

• [http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml](http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml)

• cpplint.py

• [http://isocpp.org/](http://isocpp.org/)

What to use, what not to use?

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

- **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)