Life cycle of an object

• construction and initialization
  - happens by declaration
    stack s;
  - or explicit call of new
    stack *sp = new stack();
  - includes initialization
    different constructors specify different ways to initialize
    default constructor called for arrays
  - copy constructor is an important special case
    specifies how to make a new object from an existing one
    implicitly invoked in declarations, function arguments, and
    function return

• assignment: changing value
  - by explicitly assigning another object
    obj1 = obj2;
  - assignment is not the same as initialization

• destruction
  - reclaim resources
  - must call delete explicitly if allocated by new
    delete sp; // or delete [] sp for an array
  - happens implicitly by going out of scope otherwise
    return from function or exit from block

Strings

• another type that C (and C++) don’t provide
• implementation of a string class brings together
  all of these...
  - constructors, destructors
  - copy constructor
  - assignment versus construction
  - operator =
  - constant references
  - handles
  - reference counts, garbage collection

• an example of a non-trivial data type
Desirable properties for a string class

• behave like strings in Awk, Perl, Java
  - like first-class citizens

• can easily assign to a string, copy a string, etc.
• can pass them to functions, return as results, ...

• create from "..." C char* strings
• can pass them to functions expecting char* 's

• storage managed automatically
  - no explicit allocation or deletion
  - grow and shrink automatically
  - efficient

• would be nice to have other operations like
  - substring, search, tokenization, etc.

Copy constructor

• a constructor that creates an object of class X
  from an existing object of class X
• first try:
  class X {
    X(X); // copy constructor?
    // ...
  };

• notice a potential problem???

• parameter to copy constructor has to be a reference
  - so it can access the object without copying it

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

String::String(const String& s) ( ... )

• copy constructor is necessary for declarations,
  function arguments, function return values
Initial version of string class

class String {
    private:
    char  *sp;
    char  *dup(const char *);

    public:
    String() { dup("" ); }  // String s;
    String(const char *) { dup(t); } // String s="abc";
    String(const String &t) { dup(t.sp); } // String s=t;
    ~String() { delete [] sp; }
    
    void operator =(const String &);    // s1 = s2
    void operator =(const char *);      // s = "abc"

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

char *String::dup(const char *s) {
    sp = new char[strlen(s) + 1];   // bug: unchecked
    return strcpy(sp, s);
}

Potential problems...

- whole lot of copying going on
  - each constructor allocates a new string
  - may affect efficiency
- string assignment is not yet specified
  - what does this mean?
    String s1, s2;
    s1 = s2;
- want to permit multiple assignment  s1 = s2 = s3
- assignment is not the same as initialization
- the meaning of assignment is defined by a
  member function named operator=

```
class String {
    Strings& operator=(const String&);
    Strings& operator=(const char *);
    // ...
};
```

- " x = y " means " x.operator=(y) "
- returning a reference permits multiple assignment
Easy case: string = "]...

- implementation of operator=(char *)

String& String::operator =(const char *t)
    // s = "abc"
    {
        delete [] sp;
        dup(t);
        return *this;
    }

- within a member function, this points to the 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, for consistency with built-in assignment (a = b = c)

Harder case: str = str

- implementation of operator=(const String&)

- check if left and right operands are same object
  - to be sure we don't delete something before using it!
- do the assignment
  - often like destruction + copy constructor
- return the left-hand side

String& String::operator=(const String&t)
{
    if (this != &t) {
        delete [] sp;
        dup(t.sp);
    }
    return *this;
}
String class complete

class String {
    private:
    char    *sp;
    char    *dup(const char *); 

    public:
    String() { dup("\0"); }  // String s;
    String(const char *t) { dup(t); } // String s="abc";
    String(const String &t) { dup(t.sp); } // String s=t;
    ~String() { delete [] sp; }

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

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

class String:
char *String::dup(const char *s) {
    sp = new char[strlen(s) + 1];
    return strcpy(sp, s);
}

continued

String& String::operator =(const char *s) {
    if (sp != s) {
        delete sp;
        dup(s);
    }
    return *this;
}

String& String::operator =(const String &t) {
    if (this != &t) {
        delete sp;
        dup(t.sp);
    }
    return *this;
}

main() {
    String s = "abc", t = "def", u = s;
    printf("%s %s %s\n", s.s(), t.s(), u.s());
    s = "1234";
    s = s;
    printf("%s\n", s.s());
    s = t = u = "asdf";
    printf("%s %s %s\n", s.s(), t.s(), u.s());
}
Handles and use counts

- how to avoid unnecessary copying for classes like strings, arrays, other containers

- default copy and assignment allocate new memory even if unnecessary
  - e.g., in f(const String& s), argument is 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
  - data pointer
  - counter of how many Strings point to that data
  - when String is copied, increment the use count
  - when String is destroyed, decrement the use count
  - when last use is done, free the characters

- picture

- with a handle class, copying only increments use count
  - "shallow" copy instead of "deep" copy

Use counts

```cpp
#include <string>

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 = new char[strlen(s) + 1];
  strcpy(sp, s);
  n = 1;
}

class String {
  Srep *p;
public:
  String(const char *);
  String(const String &);
  ~String();
  String& operator ==(const String &);  // s1 = s2;
  String& operator ==(const char *);   // s = \"abc\";

  char *s() { return p->sp; }
};
```
part 2

String::String(const char *s ="")
{  // String s="abc"; String s1;
    p = new Srep(s);
}

String::String(const String &t)    // String s=t;
{    t.p->n++;      // ref count
    p = t.p;
}

String::~String()
{    if (--p->n <= 0) {
        delete [] p->sp;
        delete p;
    }
}

part 3

String& String::operator=(const char *s)
{    if (p->n > 1) {         // disconnect self
        p->n--;      // disconnect self
        p = new Srep(s);
    } else {  
        delete [] p->sp;    // free old String
        p->sp = new char[strlen(s) + 1];
        strcpy(p->sp, s);
    }
    return *this;
}

String& String::operator=(const String &t)
{    t.p->n++;      // protect against s = s
    if (--p->n <= 0) {  // nobody else using me now
        delete [] p->sp;
        delete p;
    }
    p = t.p;
    return *this;
}
Rules / heuristics

- all objects have to have a constructor
  - if you don’t specify a constructor the default constructor copies members by their constructors
- 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
- 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
- Checklist for class authors
  adapted from Ruminations on C++ (Andy Koenig & Barb Moo)
  - does your class need a constructor?
  - are your data members private?
  - does your class have a constructor without arguments?
  - does every constructor initialize every data member?
  - does the class need a destructor?
  - does your class need a copy constructor?
  - does your class need an assignment operator?
  - did you use delete[] for arrays?
  - did you use const reference parameters for functions?

Base & derived classes

class Widget {
    int bgcolor;
    // other vars common to all Widgets
};
class Scrollbar : public Widget {
    int min, max, current;
    // other vars specific to Scrollbars
};

a Scrollbar is a (kind of) Widget
- inherits all members of Widget
- adds its own members

" : public Widget " means public base class members are public in derived as well
- protected means derived class can see but not others

widget  widget  widget
  button  scrollbar
Derived classes

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

class Scrollbar : public Widget {
    private:
        int min, max, current;
    public:
        draw() { ... }
        setSlider(int) {};
};
class CheckButton : public Widget {
    private:
        bool checked;
    public:
        draw() { ... }
        setState(bool) { ... }
};

CheckButton b; Scrollbar s;
b.draw(); // call CheckButton::draw
s.draw(); // call Scrollbar::draw

Virtual Functions

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

class Widget {
    private:
        String caption;
    public:
        setCaption(String c) { caption = c; }
        virtual draw();
        virtual update();
};

- "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
  - indicated by "= 0" on a virtual function declaration
Dynamic binding and virtual functions

- when a pointer or reference to a base-class type points to a derived-class object
- and you use that pointer/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 Widgets on a linked list:

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

- virtual function mechanism automatically calls the right draw() function for each object
- the loop does not change if more kinds of widgets 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
    updating different Widgets 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) {
        if (n < 0 || n >= size)
            assert(n >= 0 && n < size);
        else
            return v[n];
    }
    T& elem(int n) { 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
```

Template functions

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

```cpp
template <class 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
  ```cpp
  max(double, double)
  max(complex, complex)
  ```
- compiler instantiates code for each different use in a program
Scoped pointer class

// scoped pointer class
// allocates space when used, frees it
// automatically when deleted

#include <stdio.h>

struct foo { int x; double y; };  

template<typename X> class SP {
X *xp;
public:
SP(X *p) : xp(p) {}  
~SP() { printf("deleting\n"); delete xp; }  
X* operator ->() { return xp; }  
};

int main() {
printf("top\n");  
{
SP<struct foo> foop(new struct foo);
foop->x = 1;
foop->y = 2.3;
}
printf("bot\n");  
}

Standard Template Library (STL)

Alex Stepanov

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

• general-purpose library of
  generic algorithms (find, replace, sort, ...)
  containers (vector, list, set, map, ...)
• 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<>.
• iterator: generalization of a pointer
  • performance guarantees
    - each combination of algorithm and iterator type
      specifies worst-case (O(...)) performance bound
      e.g., maps are O(log n) access
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, ...

Iterators

- a generalization of C pointers
- a 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
- basic loop:
  - for (iter_type i = v.begin(); i != v.end(); ++i)
    - do something with *i

- input iterator
  - can only read items in order, can't store into them
- output iterator
  - can only write items in order, can't read them
- forward iterator
  - can read/write items in order, can't go backwards
- bidirectional iterator
  - can read/write items in either order (doubly-linked list, array)
- random access iterator
  - can access items in any order (e.g., for sorting)
Example 1

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

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

- `v.push_back(s)` pushes `s` onto "back" (end) of `v`
- 3rd argument of `copy` is a "function object" that calls a function for each iteration
  - uses overloaded operator()
    - `sort(v.begin(), v.end(), greater<string>())`
      - would sort in decreasing order

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

```cpp
class add() {
    public:
        int operator()(int x, int y) {
            return x + y;
        }
}
```
Iterator example

- STL copy algorithm
- satisfies constraints on iterators

template <class InputIterator,
          class OutputIterator>
OutputIterator mycopy (InputIterator first,
                      InputIterator last, OutputIterator result)
{
    while (first != last)
        *result++ = *first++;
    return result;
}

main()
{
    vector<int> v;
    for (int i = 0; i < 10; i++)
        v.push_back(i);
    mycopy(v.begin(), v.end(),
            ostream_iterator<int>(cout, "
"));
}

Should I use the STL?

- code is often extremely clean and elegant
- usually easy to change underlying data structure
- often runs slow, sometimes extremely slow
- implementations are getting better

#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

```cpp
class ivec {
  int *v;        // pointer to array
  int size;      // number of elements
  public:
    int &operator [](int n);      //...
};

int& ivec::operator [](int n) {
  if (n < 0 || n >= size)
    throw(subscriptrange(n));
  else
    return v[n];
}

int f() {
  ivec iv(100);
  try {
    return g(iv);   // normal return if no exceptions
  }
  catch (subscriptrange) {
    return 0;  // get here if `subscriptrange' was raised in g() or anything it calls
  }
  catch (...) {   // get here if some other exception was raised
    return -1;
  }
}
```

C++ reprise: things to remember

- abstraction: separating what from how
  - creating internal firewalls and barriers in code
  - separating interface from implementation
- classes are user-defined types
  - they should model objects in the application
- object-oriented programming
  - public methods define interface to the world
  - private methods and members for implementation
  - overloading functions and operators
- constructors, assignment operators, destructors
  - complete control over creation, copying, deletion
  - references provide access without copying
- inheritance to describe family of related types
  - base and derived classes
  - polymorphism to call the right functions dynamically
- templates and parameterized types
  - generic algorithms, container classes, iterators
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)