ValueTypes

Jim Fawcett
CSE687 – Object Oriented Design
Summer 2018

Definition

- A Value Type is a user-defined type that supports:
  - Making copies of instances
  - Assigning the state of one instance to another
  - Managing all allocation of resources needed by instances in a way that is transparent to using code
- Value Types may also support:
  - Move copy
  - Move assignment

  Move operations transfer ownership of instance state instead of copying or assigning state.

Data Abstraction

- C++ provides strong support for data abstraction:

- designers create new types using classes
  - classes have both data members and member functions
  - these are divided into a public interface and private or protected implementation

- objects (instances of a class) are essentially active data. Public members provide safe and simple access to data which may have complex internal structure and private management

- objects are declared and destroyed in exactly the same way that primitive C++ variables are.
  - user defined constructors build class objects when they are declared
  - user defined destructors remove the objects when they go out of scope

- Operators can be overloaded to have meanings unique to each class
  - overloading, which applies to all functions, not just operators, is accomplished by using the function's signature (name and types of formal parameters) as its identifier. Thus two functions with the same name but different argument types represent unique functions.

Classes

- A class establishes the operations and "look and feel" for the objects it creates.
- We normally expect a class to provide the following operations for its objects:

  - **construction:**
    allocate any required resources for the object and provide a syntax for the client to invoke

  - **destruction:**
    deallocate resources and perform any needed cleanup

  - **arrays:**
    provide for the construction of arrays of valid initialized objects

  - **passing to functions:**
    support passing objects to functions and the return of objects from functions by value, pointer, or reference

  - **observing and modifying object state:**
    provide accessor and mutator functions which disclose and make valid modifications of an object's internal state

  - **assignment of objects:**
    assign the value (state) of one object to another existing object

  - **coercion of objects:**
    provide for promotion of some foreign objects to objects of this class, provide cast operators (only) to the built in types

  - **operator symbolism:**
    often we want the vocabulary provided by the class's public interface to include operator symbols like '+', '-', ...

- While providing these operations we expect the class to protect and hide its internal implementation.

Class and Object Syntax

class declarations

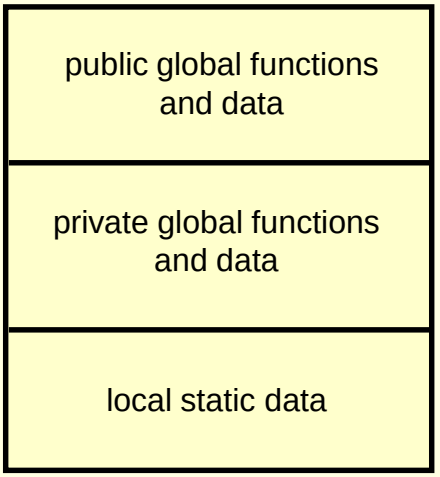
```
class X {
  public:
  // promotion constructor
    X(T t);

  // void ctor for arrays
    X();

  // destructor
    ~X();

  // copy ctor
    X(const X& x);

  // accessor
    T showState();

  // mutator
    void changeState(T t);

  // assignment
    X& operator=(const X&)

  // cast operator
    operator T ()

  private:  ...
};
```

code using class objects

```
// promote type T to type X
  X xobj = tobj;

// declare array of n elems
  X xobj[n];

// destruction calls are
// usually implicit

// pass object by value
  funct(X xobj);

// access state
  T t = xobj.showstate();

// change state
  xobj.changeState(t);

// assign
  xobj2 = xobj1;

// explicit cast
  T t = T(xobj) or (T)xobj;
    or static_cast<T>(xobj);
// implicit cast
  T t = xobj;
```

C/C++ Memory Model

Static memory: - available for the lifetime of the program

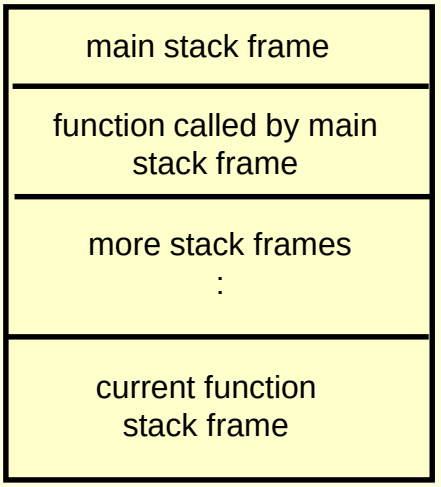


- defined outside any function (globals) and initialized before main is entered.

- global data and functions are made private by qualifying as static, otherwise they are public

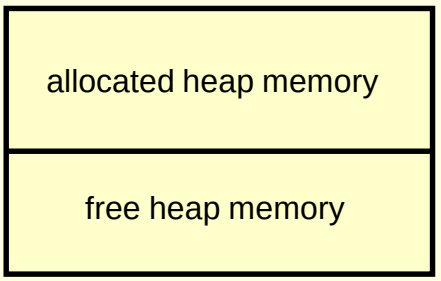
- memory allocations local to a function, but qualified as static

Stack memory: - temporary scratch pad



- defined only while thread of execution passes through a function, control, or anonymous scope.
- holds input parameters, local data, and return values, used as scratch-pad memory
- guaranteed to be valid during the evaluation of a containing expression, won't be valid after
- expression evaluation starts with function evaluation first, then expression evaluation as algebraic combination of terms
- stack frame is destroyed when expression evaluation is complete

heap memory: - valid from the time of allocation to deallocation



- allocated/deallocated at run time by invoking operators new /delete (or functions malloc/free)
- memory is available to anyone with a pointer to the allocated memory from the time of allocation until deallocated.

Str Class

- In the next few pages we examine an implementation of a value type representing strings.

- Each of the most important member functions are dissected. We discuss their:
  - declaration: how you declare member functions in the class declaration (part of STR module's header file).
  - Definition: how you define the function's behavior in its function body.
  - Invocation: how you invoke this member of the STR class.

- While this class makes a good vehicle for instruction, you should prefer the string class provided by the standard C++ library and documented in class texts.

Str Manual Page

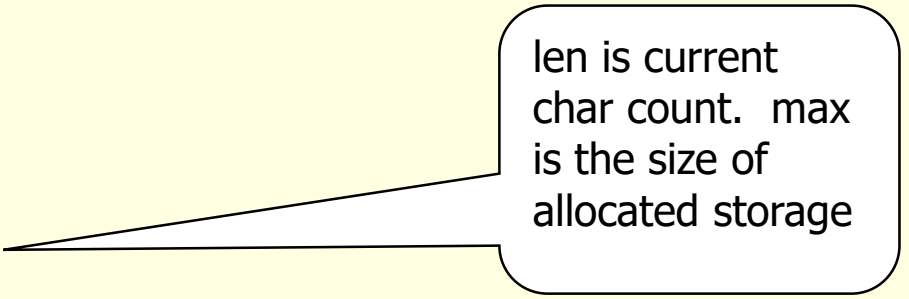
```
#ifndef STR_H
#define STR_H
///////////////////////////////////////////////////////////////
//  Str.h      -  header file for Str string class                 //
//  ver 2.1                                                      //
//                                                               //
//  Language:     Visual C++, ver 12.0                           //
//  Platform:     Dell XPS 2720, Win 8.0                         //
//  Application:  ADT example, CSE687 - Object Oriented Design   //
//  Author:       Jim Fawcett                                    //
//                Syracuse University, CST 4-187                 //
//                fawcett@ecs.syr.edu, (315) 443-3948            //
///////////////////////////////////////////////////////////////
/*
    Class Operations:
    =================
    This class defines a string data type.  It is a simple, but
    effective user defined type.  You should prefer the standard
    C++ string class.  The purpose of this class is to demonstrate
    basic class construction techniques.

    Instances of Str class perform bounds checking on all indexed
    operations and throw invalid_argument exceptions if the index
    is out of bounds, e.g., does not refer to a valid character.

    Public Interface:
    =================
    Str s;                  construct an empty string;
    Str s(15);              construct empty string that holds 15 chars
    Str s1 = s;             construct s1 as a copy of s
    Str s2 = "a string";    construct s2 holding a literal string
    s1 = s2;                assign the value of s2 to the string s1
    s1[2] = 'a';            modify the 3rd character of s1
```

Maintenance Page

```
///////////////////////////////////////////////////////////
//  Build Process                                        //
///////////////////////////////////////////////////////////
//  Required files:                                      //
//    Str.h, Str.cpp                                     //
//                                                       //
//  compiler command:                                    //
//    cl /GX /DTEST_STR str.cpp                          //
///////////////////////////////////////////////////////////
/*
    Maintenance History:
    ====================
    ver 2.1 : 12 Jan 2014
    - added move constructor and move assignment for C++11
    ver 2.0 : 25 Jan 2009
    - added initialization sequences.
    ver 1.9 : 29 Jan 2006
    - cosmetic changes
    ver 1.8 : 03 Feb 2005
    - added operator+, changed return type of operator+= from void
      to Str&, qualified promotion ctor with explicit - note impact
      on test stub.
    ver 1.7 : 01 Feb 2005
    - Str has an invariant that all string arrays held by the pointer
      array must be null terminated.  The default constructor, Str(),
      did not correctly satisfy that, but now has been fixed.
    ver 1.6 : 29 Jan 2004
    - removed all checks for memory allocation failures, as the
      standard language behavior is to throw exceptions when this
```

Class Declaration

```
class Str {

private:
  char *array;
  int len, max;

public:
  Str(int n = 10);                  // void and size ctor
  Str(const Str& s);                // copy ctor
  Str(Str&& s);                     // move ctor
  explicit Str(const char* s);      // promotion ctor
  ~Str();                           // dtor
  Str& operator=(const Str& s);     // copy assignment operator
  Str& operator=(Str&& s);          // move assignment operator
  char& operator[](int n);          // index operator
  char operator[](int n) const;     // index for const Str
  Str& operator+=(char ch);         // append char
  Str& operator+=(const Str& s);    // append Str s
  Str operator+(const Str& s);      // concatenate strs
  operator const char* ();          // cast operator
  int size() const;                 // return number of chars
  void flush();                     // clear string contents
};
```



Str Void (default) Constructor

- **Purpose:**
  - to build a default object (or array of default objects)
  - if, and only if, no constructors are defined by the class, the compiler will generate a void constructor which does void construction of class bases and members

- **Declaration** (part of class declaration in header file):

```
Str(int n=10);    // can be used for void con-
                  // struction with default arg
```

- **Definition** (part of implementation file):

```
//----< sized constructor >-----------------------------

Str::Str(int n) : array(new char[n]), max(n), len(0)
{
    array[0] = '\0';
}
```

Note:

new throws an exception if allocation fails.

- **Invocation** (part of test stub or application code):

```
Str s;                  // define default object
Str s[5];               // initialize array
Str* sptr = new Str;    // initialize object on heap
```

- Note that constructors and the destructor have no return values, not even void.

Str Copy Constructor

- **Purpose:**
  - to build object which is a logical copy of another
  - used when objects are passed or returned by value
  - if no copy constructor is defined by the class the compiler will generate one if needed which does member-wise copies.

- **Declaration** (in class declaration in header file):

```
Str(const Str& s);
```

- **Definition** (in implementation file):

```
//----< copy constructor >---------------------------

Str::Str(const Str& s)
    : array(new char[s.max]), max(s.max), len(s.len)
{
  for(int i=0; i<=len; i++)
    array[i] = s.array[i];
};
```

No assignment here. Just the single copy operation

- **Invocation** (in test stub or application code)

```
Str s2 = s1;              // copy construction!
Str s2(s1);               // same as above
Str s[2] = { s1, s2 };    // copy state into array
Str *sptr = new Str(s1);  // copy state onto heap
void myFun(Str s);        // pass by value

Str yourFun();            // return by value
```

Str Move Constructor

- **Purpose:**
  - to build object stealing the resources of a temporary
  - used when moveable objects are returned by value
  - if no move constructor is defined by the class will fallback to copy.
  - compiler will generate only if no potentially implicit operations are explicitly declared, i.e., copy ctor, …

- **Declaration** (in class declaration in header file):

```
Str(Str&& s);
```

- **Definition** (in implementation file):

```
//----< move constructor >---------------------------

str::Str(Str&& s)
    : array(s.array), max(s.max), len(s.len)
{
  s.array = nullptr;
};
```

- **Invocation** (in test stub or application code)

```
Str testFunction()
{
    Str s("string created in testFunction");
    return s;
}

Str sTest = testFunction();
```

sTest gets temporary s's array

Promotion Constructor

- **Purpose:**
  - to coerce an object of another class to one of this class
  - in this case we coerce a "C string" to become a Str object
  - compiler will not generate promotion ctor

- **Declaration** (in class declaration):

```
explicit Str(const char* s);
```

- **Definition** (in implementation file)

```
//----< promotion constructor >----------------------

Str::Str(const char* s)
     : len(static_cast<int>(strlen(s)))
 {
   max = len+1;
   array = new char[len+1];
   for(int i=0; i<=len; i++)
     array[i] = s[i];
 }
```

Every constructor that takes a single argument of a type different than the class type is a promotion constructor. They're used for conversions and can be called implicitly if not qualified as explicit.

- **Invocations** (in test stub or application code):

```
Str s = Str("this is a string");
Str sa[2] =
      { Str("first string") , Str("second string") };
Str *sptr = new Str("defined on heap");
void myFun(const Str &s); myFun(Str("a string"));
```

Destructor

- **Purpose:**
  - to return system resources when object goes out of scope
  - if no destructor is defined by the class the compiler will generate one which calls each member's destructor if one is defined

- **Declaration** (in class declaration in header file):

```
~Str(void);
```

- **Definition** (in implementation file):

```
//----< destructor >---------------------------

Str::~Str() {
   delete [] array;
   max = len = 0;
   array = nullptr;
 }
```

You must delete with [] if you new with []!

- **Invocation** (in test stub or application code):
  - Destructors are called implicitly whenever an object goes out of scope.
  - When you allocate an object using the "new" operator a constructor of the object is called to initialize the object.

```
Str *sptr = new Str;
```

  - When you delete the pointer to an allocated object its destructor is called automatically.

```
delete sptr;
```

Copy Assignment Operator

- **Purpose:**
  - to assign the state values of one existing object to another
  - if no copy assignment operator is defined by the class the compiler will generate one which does member-wise copy assignments

- **Declarations** (in class declaration in header file):

```
Str& operator=(const Str& s);
```

- **Definitions** (in implementation file):

```
Str& Str::operator=(const Str& s) {
  if(this == &s) return *this;    // don't assign to self
  if(max >= s.len+1) {            // don't allocate new
    len = s.len;                  //   storage if enough
    int i;                           //   exists already
    for(i=0; i<=len; i++)
      array[i] = s.array[i];
    return *this;
  }
  delete [] array;                   // allocate new storage
  array = new char[max = s.max];
  len = s.len;
  for(int i=0; i<=len; i++)
    array[i] = s.array[i];
  return *this;
}
```

Note i<=len because we want to copy terminal '\0'

- **Invocation** (in test stub or application code):

```
s2 = s1;              // algebraic notation
s2.operator=(s1);     // equivalent operator notation
```

Move Assignment Operator

- **Purpose:**
  - to assign the state values of a temporary object to another by moving, e.g., by passing ownership of the state values.
  - if no other potentially implicit operation is defined, the compiler will generate a move assignment which does member-wise move assignments if defined

- **Declarations** (in class declaration in header file):

```
Str& operator=(Str&& s);
```

- **Definitions** (in implementation file):

```
Str& Str::operator=(Str&& s) {
    if(this == &s) return *this;   // don't assign to self
    max = s.max;
    len = s.len;
    delete [] array;
    array = s.array;
    s.array = nullptr;
    return *this;
}
```

- **Invocation** (in test stub or application code):

```
s1 = s2 + s3;         // s1 move assigned from temporary
s2 = std::move(s3);   // s3 no longer owns internal chars
                      // we normally would not do this
```

Index Operator

- **Purpose:**
  - read or write one character from the string
- **Declaration** (in class declaration in header file):

```
char& Str::operator[](int n);
```

- **Definition** (in implementation file):

Note

```
char& Str::operator[](int n) {

   if(n < 0 || len <= n)
     throw invalid_argument("index out of bounds");
   return array[n];
}
```

Standard exception type

- **Invocation** (in test stub or application code):

The function returns a reference to the nth character so client code can either read or write to the result, e.g.:

```
char ch = s[3] = 'z';
```

This statement is equivalent to:

```
s.operator[](3) = 'z';
```

Note: We are assigning to a function! How does that work?

Index Operator for const Str

- **Purpose:**
  - read one character from const Str object

Note

- **Declaration** (in class declaration in header file):

Note

```
char Str::operator[](int n) const;
```

Note

- **Definition** (in implementation file):

```
char Str::operator[](int n) const {

   if(n < 0 || len <= n)
     throw invalid_argument("index out of bounds");
  return array[n];
}
```

- **Invocation** (in test stub or application code):

The function returns a copy of the nth character
So client code can only read the result, e.g.:

```
char ch = s[3];
```

Append a Character

- **Purpose:**
  - add one character to the end of string
- **Declaration** (in class declaration in header file):

```
void Str::operator+=(char ch);
```

- **Definition** (in implementation file):

```
void Str::operator+=(char ch) {
  if(len < max-1) {                // enough room
    array[len] = ch;               // so just append
    array[len+1] = '\0';
    len++;
  }
  else {          // not enough room so resize array
    max *= 2;     // multiply by 2
    char *temp = new char[max];
    for(int i=0; i<len; i++)
      temp[i] = array[i];
    temp[len] = ch;
    temp[len+1] = '\0';
    len++;
    delete [] array;
    array = temp;
  }
}
```

Increase size in binary steps, so fewer memory allocations if we guess wrong.

- **Invocation** (in test stub or application code):

```
s += ‘a’;
```

Append Another String

- **Purpose:**
  - add one string to the end of another string
- **Declaration** (in class declaration in header file):

```
void Str::operator+=(const Str& s);
```

- **Definition** (in implementation file):

```
void Str::operator+=(const Str& s) {
 if(len < max-s.size()) {
   for(int i=0; i<=s.len; i++)
     array[len+i] = s[i];
   len += s.size();
 }
 else {
   max += max + s.size();
   char *temp = new char[max];
   for(int i=0; i<len; i++)
     temp[i] = array[i];
   for(int i=0; i<s.size(); i++)
     temp[len+i] = s[i];
   temp[len+s.size()] = '\0';
   len += s.size();
   delete [] array;
   array = temp;
 }
}
```

- **Invocation** (in test stub or application code):

```
s += Str(" another string");
```

Addition Operator

- **Purpose:**
  - add two strings to create a new string result
- **Declaration** (in class declaration in header file):

```
Str Str::operator+(const Str& s);
```

- **Definition** (in implementation file):

```
Str Str::operator+(const Str& s) {
  Str temp = *this;
  temp += s;
  return temp;
}
```

- **Invocation** (in test stub or application code):

```
s = Str(“first, ”) + Str(“second”);
```

Calls operator+(const Str&) then operator=(Str&&)

Cast Operator

- **Purpose:**
  - to coerce object of class to an object of another class
  - here we cast a Str object to a pointer to a const char array
- **Declaration** (in class declaration in header file):

```
Str::operator const char*( );
```

- **Definition** (inline in header file):

```
inline Str::operator const char*() {
  return array;
}
```

- The const says that the character array values can't be changed.
- Note that the cast operator is the only operator which has, by definition, no return value (not even void).
- **Invocations** (in test stub or application code):

```
const char* ptr = s;           // implicit invocation
const char* ptr = static_cast<const char*>(s)
const char* ptr = char*(s);    // newer cast notation
const char* ptr = (char*)s;    // classic cast notation
```

Insertion Operator

- **Purpose:**
  - send string to output stream
- **Declaration** (in header file):

```
ostream& operator<<(std::ostream& out, const Str& s);
```

- **Definition** (in implementation file):

Note that this function is not a member of the Str class nor is it a friend.

```
ostream& operator<<(ostream& out, const Str& s) {
  for(int i=0; i<s.size(); i++)
    out << s[i];
  return out;
}
```

- **Invocation** (in test stub or application code):

```
std::cout << s;
```

Extraction Operator

- **Purpose:**
  - accept a string from input stream
- **Declaration** (in header file):

```
istream& operator>>(std::istream &in, Str &s);
```

- **Definition** (in implementation file):

Note that this function is not a member of the Str class nor is it a friend.

```
istream& operator>>(istream& in,  Str& s)
{
   char ch;
  s.flush();
  in >> ch;
  while((ch != '\n') && in.good()) {
    s += ch;
    in.get(ch);
  }
  return in;
}
```

Str memory management means this function is simple!

- **Invocation** (in test stub or application code):

```
cin >> s;
```

C++ Binary Operator Model

- A C++ operator is really just a function. Assignment, for example, may be written either way shown below:

```
x = y;
or
x.operator=(y);
```

Here, the x object is invoking the assignment operator on itself, using y for the assigned values.

- The left hand operand is always the invoking object and the right hand operand is always passed to the function as an argument.

- General form of the binary operator:

x^y ⇔ x.operator^(y) – member function

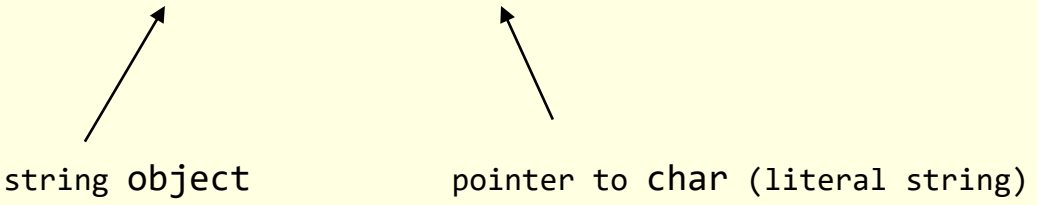
x^y ⇔ operator^(x,y) – global function

Coercions

- We often write code which contains type mismatches.

  For example:

  Str s1 = “this is a string”;



  The compiler scans this expression, notes the type mismatch, and looks for means to resolve it.

  It finds the promotion constructor which takes a pointer to char and builds an Str object. So the compiler generates code to build the s1 Str object, if the promotion is not explicit. Our Str promotion is intentionally explicit so we need to wrap the string, above, with Str(“…”), otherwise compilation fails.

- We write non-explicit promotion constructors and cast operators so this kind of “silent” coercion can happen. It makes programming easier when sensible conversions happen auto-matically.

Overloading

- Function overloading occurs when two functions have the same identifier but different calling sequences, e.g., the sequences of types passed as arguments.
  - `Str(int n=10);`
  - `Str(const Str& s);`

  Here Str is a common identifier used for both functions. The functions differ in their calling sequences, e.g., int n vs. const Str&.

- The compiler distinguishes overloaded functions on their sequences, but not on return values.
  - `char& operator[](int n);`
  - `char operator[](int n) const;`

  are distinguished by const, not the return type.

Constness

- What const implies is determined by where you find it:
  - `Str(const Str& s);`

  is a contract that the argument s will not be changed. The compiler attempts to enforce the contract.
  - `char operator[](int n) const;`

  implies the state of the object on which the operator is applied will not change. Again, the compiler attempts to enforce the contract. Thus:
  - `const Str cs = “a constant string”;`
  - `cs[3] = ‘a’;`

  will fail to compile because the compiler will call the const version of operator[] on the const string and will disallow changes to the string.

C++ Expression Model

- When a C++ compiler evaluates an expression it performs the following steps:
  - evaluates all function invocations, replacing the call with its return value
  - scans the expression checking for type mismatches.

    If any are found, the compiler looks for ways to resolve the mismatches by implicitly calling a promotion constructor or cast operator.

    If there is exactly one way to resolve the mismatch the compiler generates code to do so, and the expression evaluation succeeds.

    If there is more than one way to resolve the mismatch the compiler declares an ambiguity and the compilation fails.
  - All stack frames for any functions invoked by the expression are guaranteed to be valid until the evaluation is complete.

    This allows the return values of functions, which are deposited in an output area of the stack frame to participate in the expression just like the value of a cited variable. Any value residing in a stack frame during evaluation is called a ***temporary***.

Conclusions - ADTs

- User defined data types can be endowed (by you) with virtually all of the capabilities of built in types:
  - declaration of multiple objects at either compile or run time
  - declaration and initialization of arrays of objects
  - objects take care of themselves, e.g., acquire and release system resources.
  - objects can participate in mixed type expressions, implicitly calling promotion constructors or cast operators as needed
  - objects can be assigned and passed by value to functions
  - objects can use the same operator symbolism as built in types
- All of these things have syntax provided by the language, but semantics provided by you.
- You can choose to provide as much or as little capability as you deem appropriate for your class.

PRESENTATION END