CSE 1325: Object-Oriented Programming

Lecture 25

Review for Exam #3

Mr. George F. Rice

george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

Santa's elves are subordinate clauses.

Reminder
Class Survey

SmartEvals.com

- The class survey is now in progress
  - I see *no* feedback until *after* your final grades are posted
  - I read and consider *every* comment!
  - **Completely anonymous**
- WARNING: Survey closes on April 29!

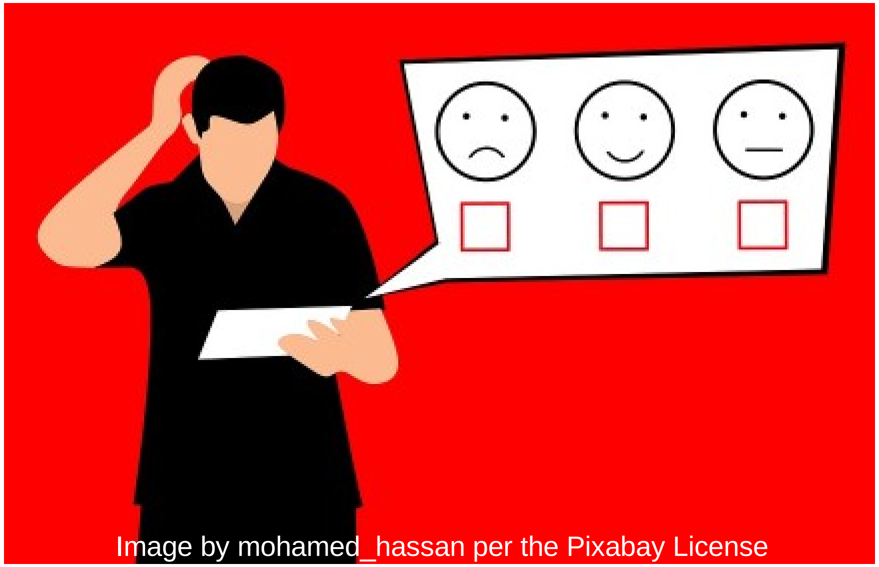


Image by mohamed_hassan per the Pixabay License
https://pixabay.com/en/experience-feedback-survey-customer-3239623/

Benefit for You: The nag screens and reminder emails *may* cease once you take the survey plus it's the right thing to do!

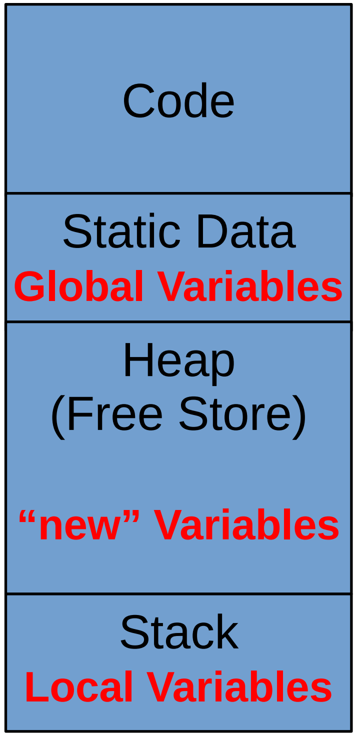
Benefit for Me: Invaluable insight into what worked and what needs to change.

C++ Memory Layout

- **Stack** is “scratch memory” for a thread
  - LIFO (Last-In, First-Out) allocation and deallocation
    - Allocation when a scope is entered
    - Automatically deallocated when that scope exits
  - Simple to track and so rarely leaks memory

- **Heap** is memory shared by all threads for dynamic allocation
  - Allocation and deallocation can happen at any time – but only when *specifically* invoked!
  - Many algorithms trade speed vs fragmentation

Memory Layout


Code

Static Data
Global Variables

Heap
(Free Store)

“new” Variables

Stack
Local Variables

Key Facts about C++

- Creating primitives, objects, and arrays on the stack and the heap: You decide!
  - **Stack variable:** int x;
  - **Stack array:** int x[3];
  - **Heap variable:** int* x = new int{}; and later delete x;
  - **Heap array:** int* y[ ] = new int[3]; and later delete[] y;
- 4 kinds of parameter mutability
  - **Pass by value:** void m(Coord c);
  - **Pass by reference:** void m(Coord& c);
  - **Pass by const reference:** void m(const Coord& c);
  - **Pass by pointer:** void m(Coord *c);
- Accessing the object or primitive pointed to by p: *p (called dereferencing)
- Accessing that object’s members: p->calc(); (same as (*p).calc();)

Key Facts about C++

- “Newton’s Third Law of C++”: For every *new* must be an equal and opposite *delete* (or *delete[ ]* if an array)
- Rule of 3: If you need a destructor, copy constructor, or copy assignment operator, you probably need all 3 (see previous bullet)
  - Copy constructor is invoked on Foo b{a}; or Foo b = a;
    and also on a pass-by-value and a return-by-value
  - Copy assignment operator is invoked on b = a;

```
class Foo {
    int* _val;
  public:
    Foo(int val) : _val{new int{val}} {}               // Non-default constructor
    Foo() : Foo(0) {}                                   // Chained constructor
    Foo(const Foo &rhs) : _val{new int{*rhs.get()}} {} // Copy constructor
    Foo& operator=(const Foo &rhs) {                    // Copy assignment operator
      if (this != &rhs) _val = new int{*rhs.get()};
      return *this;
    }
    ~Foo() {delete _val;}                               // Destructor
    int* get() const {return _val;}                     // Getter
    void set(int* v) {*_val = *v;}                      // Setter
};
```

Key Facts about C++

- Exceptions are similar to Java
  - throw std::runtime_error{“Bad date”}; // in eat() - no new!
  - try {date.eat();} catch(std::runtime_error& e) {std::cerr << e.what();
    // or catch(std::exception& e)
- Downcast with
  - std::static_cast<type>(var) for compile-time checks
  - std::dynamic_cast<type>(var) for runtime checks on polymorphic types
- Namespaces manage scope
  - Multiple declarations aggregate

```
namespace Jack {
  class Glob { /*…*/ };      // in Jack’s header file jack.h
  class Widget { /*…*/ };
}
```

```
#include "jack.h"         // this is in your code
#include "jill.h"

void my_func(Jack::Widget p) {   // No collision!
    // …
}
```

I/O

- For console I/O
  - std::cout << x; // data and std::cerr << x; // errors
  - std::cin >> x; // parse on whitespace and std::getline(std::cin, x); // parse on \n
- For files, replace std::cout with std::ofstream ofs{filename};
  and std::cin with std::ifstream ifs{filename}
  - NO try-with-resources – check the stream state instead
  - // Copy text file line by line
    ifstream ifs{source}; if (!ifs) throw std::ifstream::failure{“Unable to open “ + source};
    ofstream ofs{dest}; if (!ofs) throw std::ofstream::failure{“Unable to open “ + dest};
    while(std::getline(ifs, s)) ofs << s << std::endl; // copy text file
- For strings, replace std::cout with std::ostringstream oss and later oss.str(),
  and std::cin with std::istringstream iss{s}
  - double d; std::string s; ostringstream oss; oss < d; s = oss.str(); // convert double to string
- 4 stream states: **good**, **fail** (recoverable error, often including eof),
  bad (unrecoverable error), and **eof** (end of file)
  - while(std::getline(std::cin, s)) std::cout << s << std::endl; // copy input to output
  - while(iss >> s) std::cout << s << std::endl; // parse string into whitespace-separated words

I/O Manipulators

- Integer base
  - std::dec, std::hex, std::oct for output base
  - std::showbase prepends 0x (for hex, for example) to output integers
- Floating point
  - std::setprecision(5) shows 5 digits past decimal
  - std::defaultfloat, std::hexfloat, std::fixed, and std::scientific set display format
- Field width
  - std::setw(10) sets the width of the next value output to 10 characters (or more if necessary not to lose information) - **NOT “sticky”**
- Mix in streams to control output

  -
    ```
    int i=42; double d=3.1415;
    std::cout << std::hex << std::showbase << i << " ";
    std::cout << std::setprecision(2) << std::fixed << d << std::endl;
    ```
  - Output is **0x2a 3.14**

Key Facts about C++

- 3 ways to create a type
  - enum class (like enum but better compiler checks)
  - class (like struct but members are private by default)
  - typedef double Altitude; // Altitude is alias for double
- Functions are first-class language members
  - May be passed as parameters and returned

Enum Classes

- Are NOT classes
  - No fields, methods, or constructors
- Do empower g++ to find more bugs!

```
enum class Month {Jan, Feb, Mar, Apr, May, Jun,
                  Jul, Aug, Sep, Oct, Nov, Dec};

const std::map<Month, std::string> m_to_s{
 {Month::Jan, "January"  },
 {Month::Feb, "February" },
 {Month::Mar, "March"    },
 {Month::Apr, "April"    },
 {Month::May, "May"      },
 {Month::Jun, "June"     },
 {Month::Jul, "July"     },
 {Month::Aug, "August"   },
 {Month::Sep, "September"},
 {Month::Oct, "October"  },
 {Month::Nov, "November" },
 {Month::Dec, "December" }
};
std::ostream& operator<<(std::ostream& ost,
                         const Month& m) {
    return ost << m_to_s[m];
}
…
Month month = Month::May;
std::cout << month;
```

Or you may use
a switch or
if / else if / else

Operator overloading example →
Also OK to write
    ost << m_to_s[m];
    return ost;

Classes

- Always public, must end declaration with ;
- Control visiblity by section (private:)
- Usually declare in .h with guard, define in .cpp

Declarations only!

date.h

```
#ifndef __DATE_H
#define __DATE_H
class Date {
  public:
    Date(int year=1970, int month=1, int day=1);
    void print_date();
  private:
    int _year, _month, _day;
    static Date today;
};
#endif
```

Default parameter values (in .h only).
This gives us a default constructor.
OR chain with a second constructor:

```
Date::Date() : Date(1970, 1, 1) { }
```

Always define static fields in the .cpp (to allocate memory)

Define field construction with “init list”!

date.cpp

Definitions only!

```
#include "date.h"
Date Date::today;
Date::Date(int year, int month, int day)
    : _year{year}, _month{month}, _day{day} {
    if (1 > month || month > 12) throw std::runtime_error{"Invalid month"};
    if (1 > day   ||   day > 31) throw std::runtime_error{"Invalid day"};
}
void Date::print_date() {
    std::cout << _month << '/' << _day << '/' << _year << std::endl;
}
```

Destructors

- Destructors run when the object is deleted
  - Free heap and other resources
- Exactly one ()
  - Default does nothing
- Usually declare as virtual
  - This creates a vtable for polymorphic subclasses

```
#include <iostream>
#include <vector>

class Rando {
  public:
    Rando() { // I’m the constructor
        std::cerr << "Constructing v" << std::endl;
        v = new std::vector<int>; // Allocate mem
        for(int i=0; i< 100; ++i)
            v->push_back(rand() % 100);
    }
    virtual ~Rando() { // I’m the destructor!
        std::cerr << "Destructing v" << std::endl;
        delete v;                 // Free mem
    }
    void printv() {
        for(int i : *v) std::cout << i << ' ';
        std::cout << std::endl;
    }
  private: std::vector<int>* v;
};

int main() {
    Rando r;     // Construct a Rando on the stack
    r.printv();  // Print out its vector from heap
}                // Rando’s destructor runs here!
```

Inheritance and Polymorphism

```
class Critter {
  public:
    Critter(int frequency) : _frequency{frequency}, _timer{0} { }
    virtual ~Critter() { }
    void count() {if (++_timer > _frequency) _timer = 0;}
    virtual void speak() = 0;
  protected:
    int _frequency;
    int _timer;
};
class Cow : public Critter {
  public:
    Cow(int frequency) : Critter(frequency) { }
    void speak() override { if (!_timer) cout << "Moo! Mooooo!" << endl; }
};
int main() {
  std::vector<Critter*> critters = {new Dog{11},new Dog{9},new Dog{3},
                                    new Cow{7}, new Cow{13},
                                    new Chicken{2}, new Chicken{5}};

  for (int i=0; i<120; ++i) {
    for (auto c: critters) {
        c->count();
        c->speak();
    }
  }
}
```

speak() is abstract (pure virtual)

: public same as Java’s extends (any number, comma separated)

Chain to superclass’ constructor by name, not *super*

Declare *override* as keyword just before ; or {

Polymorphism only works for virtual methods called via either pointer or (const) reference

Operator Overloading

- Most (not quite all) operators may be overloaded
  - Only existing operators (you can’t create your own)
  - At least one type must be non-primitive
  - Many may be defined as members, but sometimes a function is required
  - Just append the operator name to keyword “operator” (see examples below)
- Operators are NOT symmetric (int + Month is different from Month + int)
- Combo operators are unique
  (Month = Month + int is a different operator than Month+= int)

Function

```
Month& operator++(Month& m) {
  switch(m) { … } // removed
  return m;
}
Month operator++(Month& m, int) {
  Month result{m};
  ++m;
  return result;
}
```

Method

```
Month& Month::operator++() {
  switch(*this) { … } // removed
  return *this;
}
Month Month::operator++(int) {
  Month result{*this};
  ++(*this);
  return result;
}
```

Pre-increment (++m)

Post-increment (m++)

Friends

- Class may declare other classes and functions as “friend”
  - Friend code may access non-public members
  - This is most useful for operators such as << and >>

inch.h

```
class Inch {
  public:
    …
    friend std::ostream& operator<<(std::ostream& ost, const Inch& inch);
    friend std::istream& operator>>(std::istream& ist, Inch& inch);
  private:
    int _whole;
    int _numerator;
    int _denominator;
};
```

Don’t memorize operator overload declarations, they will be provided on the exam as needed

inch.cpp

```
std::ostream& operator<<(std::ostream& ost, const Inch& inch) {
    return ost << inch._whole << " " << inch._numerator << " / " << inch._denominator;
}
std::istream& operator>>(std::istream& ist, Inch& inch) {
    char c;
    return ist >> inch._whole >> inch._numerator >> c >> inch._denominator;
}
```

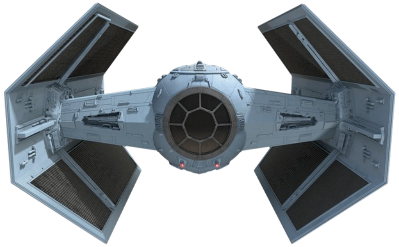
Captures and discards the “/”

Comparing Dates 6 Ways to Sundays

The “Spaceship” (<=>) Operator

in C++ 20 and Later

- The obvious default would be to compare each field in order of declaration – simple!
- C++ 20 can do this... *if you ask it nicely*
  - We just declare the “spaceship operator” (<=>)
  - Earlier versions of C++ required additional code – next slide!



date.h

```
class Date {
  public:
    Date(int year = 1970, Month month = Month::Jan, int day = 1);

    auto operator<=>(const Date&) const = default;
    // NOTHING is required in the .cpp file!
```

Star Wars Tie Fighter model by Jansen_G via the Pixabay License
https://www.freeimg.net/photo/1141024/spaceship-model-toys-starwars

Operator Overloading

The “Spaceship” (<=>) Approach

in C++ 17 and Earlier

In C++ 17 and earlier, we write our own “spaceship”!

Define all operators using the private compare() method

date.h

```
inline bool operator==(const Date& rhs) {return (compare(rhs) == 0);}
inline bool operator!=(const Date& rhs) {return (compare(rhs) != 0);}
inline bool operator< (const Date& rhs) {return (compare(rhs) <  0);}
inline bool operator<=(const Date& rhs) {return (compare(rhs) <= 0);}
inline bool operator> (const Date& rhs) {return (compare(rhs) >  0);}
inline bool operator>=(const Date& rhs) {return (compare(rhs) >= 0);}
```

The operators match!

Date::compare returns -1 if `this < rhs`, 0 if `this == rhs`, and 1 if `this > rhs`

```
int Date::compare(const Date& rhs) {
    if(year <rhs.year ) return -1;
    if(year >rhs.year ) return  1;
    if(month<rhs.month) return -1;
    if(month>rhs.month) return  1;
    if(day  <rhs.day  ) return -1;
    if(day  >rhs.day  ) return  1;
    return 0;
}
```

The compare method returns -1 if object is less, 0 if equal, or 1 if greater than its parameter.

date.cpp

Inline tells the compiler to replace any call to these methods with the literal code instead of a function call and return.

Standard Template Library (STL)

```
    // Vectors (using int as the index type)
    std::vector<std::string> s;
    s.push_back("Maps rock");
    for (int i=0; i < s.size(); ++i)
       std::cout << i << " = " << s[i] << std::endl;

    // Maps (using in this case a std::string as the key and double as the value)
    std::map<std::string, double> m;
    m["earth"] = 5.97;
    for (auto& [ planet, mass ] : m )
        std::cout << planet << " = " << mass << std::endl;
```

Iterating over a vector
with a for-each loop

Iterating over a map
with a for-each loop

```
std::vector<int> v = {1, 2, 3, 4, 5};
std::vector<int>::iterator it = v.begin();
do {
    std::cout << *it << std::endl;
} while(++it != v.end());
```

Iterating over any container
or constant container with iterators

An iterator behaves *very* much
like a pointer – pointer math,
dereference, etc.

```
const std::vector<int> v = {1, 2, 3, 4, 5};

std::vector<int>::const_iterator it = v.cbegin();

do {
    std::cout << *it << std::endl;
} while(++it != v.cend());}
```

STL containers such as std::vector
cannot be extended (use composition)

Common Operations

(Know the methods but not the footnotes)

	std::vector	std::string	std::set	std::map
Is it empty?	**v.empty()**	**s.empty()**	**s.empty()**	**m.empty()**
Clear	**v.clear()**	**s.clear()**	**s.clear()**	**m.clear()**
How many?	**v.size()**	**s.size() or s.length()**	**s.size()**	**m.size()**
Random access	**val = v[index]**	**c = s[index]**	--	**val = m[key]**
Throw if bad index	**val = v.at(index)**	**c = s.at(index)**	--	**val = m.at(key)**
Value or key exists?	**std::count**	**std::count**	**s.count(val)**[1]	**m.count(key)**
Overwrite value	**v[index] = val**	**s[index] = c**	**s.insert(val)**	**m[key] = val**
Throw if bad index	**v.at(index) = val**	**s.at(index) = c**	--[2]	**m.at(key) = val**
Erase value	**v.erase(it)**	**s.erase(index, len)**	**s.erase(val)**	**m.erase(key)**
Insert value	**s.insert(val, it)**	**s.insert(index, str)**	**s.insert(val)**	**--** (available but omitted)
Iterate	**for(auto& val : v)**	**for(auto& c : s)**	**for(auto& val : s)**	**for(auto& [key,val] : m)**
Get iterator to first	**it = v.begin()**	**it = s.begin()**	**it = s.begin()**	**it = m.begin()**[3]
Get iterator to last+1	**it = v.end()**	**it = s.end()**	**it = s.end()**	**it = m.end()**[3]

v = vector   s = string or set   m = map
val = value   it = iterator   c = char

[1] Or (in C++ 20 or later) `s.contains(val)`

[2] **s.insert** succeeded if `result.second` is true

[3] `it->first` for key, `it->second` for value

Common Iterator Methods

The methods in green are the most used!

- All **iterators** must provide:
  - Destructor, copy constructor, and copy assignment operator (it1 = it2)
  - Increment (++it)
  - Deferenced access (x = *it)
- Input iterators add:
  - Comparisons (it1 == it2 and it1 != it2)
- Output iterators add:
  - Dereferenced assignment (e.g., *it = x)
- Forward iterators add:
  - Default constructor
- Bidirectional iterators add:
  - Decrement (--it)
- Random access iterators add:
  - Pointer math (it+3, it-2, it+=5, it[4]), comparison (it1 < it2 and so on)

Containers that support iterators usually provide (x is item, p is iterator):

- begin(), cbegin() - returns p to first item
- end(), cend() - returns p to 1 past last item
- size() - number of elements
- empty() - true if no elements
- push_back(x) / push_front(x) – insert x at end / beginning, respectively
- insert(p, x) – insert x immediately before p
- front(), back() - first / last item, respectively
- pop_back() / pop_front() – delete at end / beginning, respectively
- erase(p) – remove item at p

STL Algorithms

Algorithms work from the first iterator to one less than the second iterator

```
int number = std::count(v.begin(), v.end(), target);

std::random_shuffle(v.begin(), v.end());

std::sort(v.begin(), v.end());
std::sort(v.begin(), v.begin()+25); // sort the first 25 elements only

auto it = std::find(v.begin(), v.end(), target);
std::cout << "Found first " << target
          << " at v[" << std::distance(v.begin(), it) << "]";
```

Finding ALL matching elements, a possible bonus candidate

```
auto it_next = v.begin();
while(it_next != v.end()) {
    auto it = std::find(it_next, v.end(), target);
    if(it == v.end()) break;
    else std::cout << "Found at " << std::distance(v.begin(), it) << std::endl
    it_next = it+1;
}
```

STL for the Exam

- **Containers**
  - Be able to *code* with **`std::vector`**, **`std::map`**, **`std::set`**, **`std::string`**
  - Be able to *iterate* using a for-each, get with **`[]`** and **`at`**, and overwrite with **`[]`**
  - Be able to *code* using **`empty, clear, size, insert, erase, count, push_back / push_front, front / back,`** and **`pop_front / pop_back`**
- **Iterators**
  - Be able to *obtain* using **`begin / cbegin, end / cend`** and *code* using **`it1 = it2, ++it, x = *it, *it = x, it1 == it2, it1 != it2`**
- **Algorithms**
  - Be able to *code* using **`std::find`** (and map's **`find(key)`** *method*), **`std::distance, std::count, std::random_shuffle,`** and **`std::sort.`**
  - Understand the *concept* of iteratively finding all elements matching the search key with **`std::find`** ("start subsequent searches just after the previously found element") but you won't be asked to code this on the exam

For Next Class

- **Exam #3** (16⅔% of final grade)
- Study sheet and practice exams are on Canvas

Questions?

Owl by PeterKraayvanger per the Pixabay License
https://pixabay.com/photos/owl-bird-animal-uhu-european-uhu-62703/