CS 101
Computer Programming and Utilization

Structures, Classes

Suyash P. Awate

Structures

- To represent many real-world entities, we need to store information of multiple kinds
  - e.g., a book has a title, authors, price, publisher, publication year, etc.
- We'd like a variable to store such information of multiple data types
- Such a variable, or a composite/mixed data type, is called a "structure"
- To store information about books in a library, we can use an array of structures
- In C++, keyword "struct" denotes a collection of variables
  - Variables are members of structure
  - Members may have different types

```
struct structure-type {
    member1-type member1-name;
    member2-type member2-name;
    ...
};
```

Structures

- Defining a struct data-type (not variable) for a book
- Creating instances/variables of struct to represent specific books
  - "Book textbookCS101, textbookCS102, xyz;"
- A member of a struct can be referred to as structInstanceName•structMemberName

```
struct Book{
 char title[50];
  char author[50];
  double price;
  int accessionNo;
  bool borrowed;
  int borrowerNo;
};
```

```
xyz.accessionNo = 1234;
cout << xyz.accessionNo + 10 << endl;
cin.getline(pqr.title,50);
```

- Initializing a struct instance

```
Book b = {"On Education", "Bertrand Russell", 350.00, 1235, true, 5798};
```

Structures

- Struct can contain another struct
  - Nested structures
- Accessing members

```
struct Point{
  double x;
  double y;
};
```

```
struct Circle{
  Point center;
  double radius;
};
```

```
Circle c1;
c1.center.x = 0.5;
c1.center.y = 0.9;
c1.radius   = 3.2;
```

- Definition of data-type + instantiation/definition of variable simultaneously

```
struct Circle{
  Point center;
  double radius;
} c1;
```

- Visibility of a structure type
  - If a struct is going to be used in multiple functions, then struct data-type must be defined outside all those functions

Structures

- Arrays of structures

```
Circle c[10];
Book library[100];
```

- Pointers to structures

```
Circle c1={{1,2},3}, *cptr;
cptr = &c1;
(*cptr).radius = 5;
```

  - C++ provides operator "–>" where "x–>y" means "(*x).y"

- Pointers as structure members
  - Creating 2 circles with same center-point instance

```
struct Circle2{
  double radius;
  Point *cptr;
}
```

```
Point p1 = {10.0,20.0};
Circle2 cc1, cc2;
cc1.radius = 5;
cc2.radius = 6;
cc1.cptr = &p1;
cc2.cptr = &p1;
cc2.cptr->y = 25;
```

Structures

- Linked structures
  - Suppose we want to build a network of students and their best friends
  - We design a structure to store the student's information and a pointer to the structure of their best friend

```
Student s1, s2, s3;
s1.rollno = 1;
s2.rollno = 2;
s3.rollno = 3;
s1.bestFriend = &s2;
s2.bestFriend = &s3;
s3.bestFriend = &s2;

cout << s1.bestFriend->rollno << endl;
cout << s1.bestFriend->bestFriend->rollno << endl;
```

```
struct Student{
    int rollno;
    Student* bestFriend;
};
```

Structures

- Just as functions can return local variables of type int, char, etc., they can also return a local variable of type struct
  - After call to midpoint() in 2nd line of main(), "return mp" causes an internal temporary variable to be created in main()
  - Value of mp is copied into that temporary variable
  - Then temporary variable is copied into p3, as part of assignment
  - Similar handling happens for other 2 calls

```
// Point as defined earlier
Point midpoint(Point a, Point b){
  Point mp;
  mp.x = (a.x+b.x)/2;
  mp.y = (a.y+b.y)/2;
  return mp;
}

int main(){
  Point p1 = {0.0, 0.0}, p2 = {100.0, 200.0}, p3;
  p3 = midpoint(p1, p2);
  cout << midpoint(midpoint(p1,p2), p2).x << endl;
}
```

```
struct Point{
  double x;
  double y;
};
```

Structures

- Passing structures by reference
  - As with other variables, passing by reference is convenient when we want:
  1. argument variable to be modified;
  2. to reduce copying during argument passing, especially when struct variable is large
  - e.g., consider the move() function for a point

```
struct Point{
   double x;
   double y;
};
```

```
void move(Point &p, double dx, double dy){
   p.x += dx;
   p.y += dy;
}
```

Structures

Sometimes C++ compiler may complain when:
we pass structures by reference and don't modify the passed structure

```
1  #include <iostream>
2  using namespace std;
3  struct Point { float x; float y; };
4  Point midpoint (Point & a, Point & b)
5  { Point c; c.x = (a.x + b.x)/2; c.y = (a.y + b.y)/2; return c; }
6  int main()
7  {
8      Point a; a.x = 0; a.y = 0;
9      Point b; b.x = 8; b.y = 8;
10     Point c = midpoint (a, midpoint (a,b));
11     cout << c.x << "\n" << c.y << endl;
12 }
```

Compilation error because compiler assumes that
the temporary/internal variable created
as a result of return from midpoint(a,b)
needs to be modifiable by calling/main function
(because it is being passed by reference
in subsequent midpoint(a,…) call)
but which isn't possible
(because it is an internal variable)

```
main.cpp:10:15: error: no matching function for call to 'midpoint'
    Point c = midpoint (a, midpoint (a,b));
              ^~~~~~~~
main.cpp:4:7: note: candidate function not viable: expects an lvalue for 2nd argument
Point midpoint (Point & a, Point & b)
      ^
1 error generated.
```

Structures

In such cases, helpful to declare argument as struct-type const

```
1  #include <iostream>
2  using namespace std;
3  struct Point { float x; float y; };
4  Point midpoint (Point & a, const Point & b)
5  { Point c; c.x = (a.x + b.x)/2; c.y = (a.y + b.y)/2; return c; }
6  int main()
7  {
8      Point a; a.x = 0; a.y = 0;
9      Point b; b.x = 8; b.y = 8;
10     Point c = midpoint (a, midpoint (a,b));
11     cout << c.x << "\n" << c.y << endl;
12 }
```

Link to this code: [copy]

options | compilation | execution

```
2
2
```

Structures

- Vectors in 3D Euclidean space

```
struct V3{
  double x,y,z;
};
```

```
double length(V3 const &a){
  return sqrt(a.x*a.x + a.y*a.y + a.z*a.z);
}
```

```
V3 scale(V3 const &a, double factor){
  V3 v;
  v.x = a.x*factor;
  v.y = a.y*factor;
  v.z = a.z*factor;
  return v;
}
```

```
V3 sum(V3 const &a, V3 const &b){
  V3 v;
  v.x = a.x + b.x;
  v.y = a.y + b.y;
  v.z = a.z + b.z;
  return v;
}
```

Structures

- Vectors in 3D Euclidean space
  - Distance covered by particle having initial velocity (vector) u, undergoing constant acceleration (vector) a, after time (scalar) t

$$s = ut + \frac{1}{2}at^2$$

```
int main(){
  V3 u,a,s;
  double t;
  cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z >> t;
  s = sum(scale(u,t), scale(a,t*t/2));
  cout << length(s) << endl;
}
```

Structures

- Taxi queueing system
  - How it works
    - Taxis come and join queue (insert functionality)
    - Taxis wait in queue
    - When customer arrives, the taxi at front of queue is dispatched (remove functionality)
  - Design of data structures
    - We want a queue to be a struct that holds the following variables:

1. An array to store the taxi IDs waiting in queue (elements of queue)
2. Number of taxis currently waiting in queue (nWaiting)
3. Front of queue in the array (front) stored as an index

```
struct Queue{
    int elements[QUEUESIZE], nWaiting, front;
};
```

- QUEUESIZE is a compile-time constant (= maximum number of taxis ever waiting)

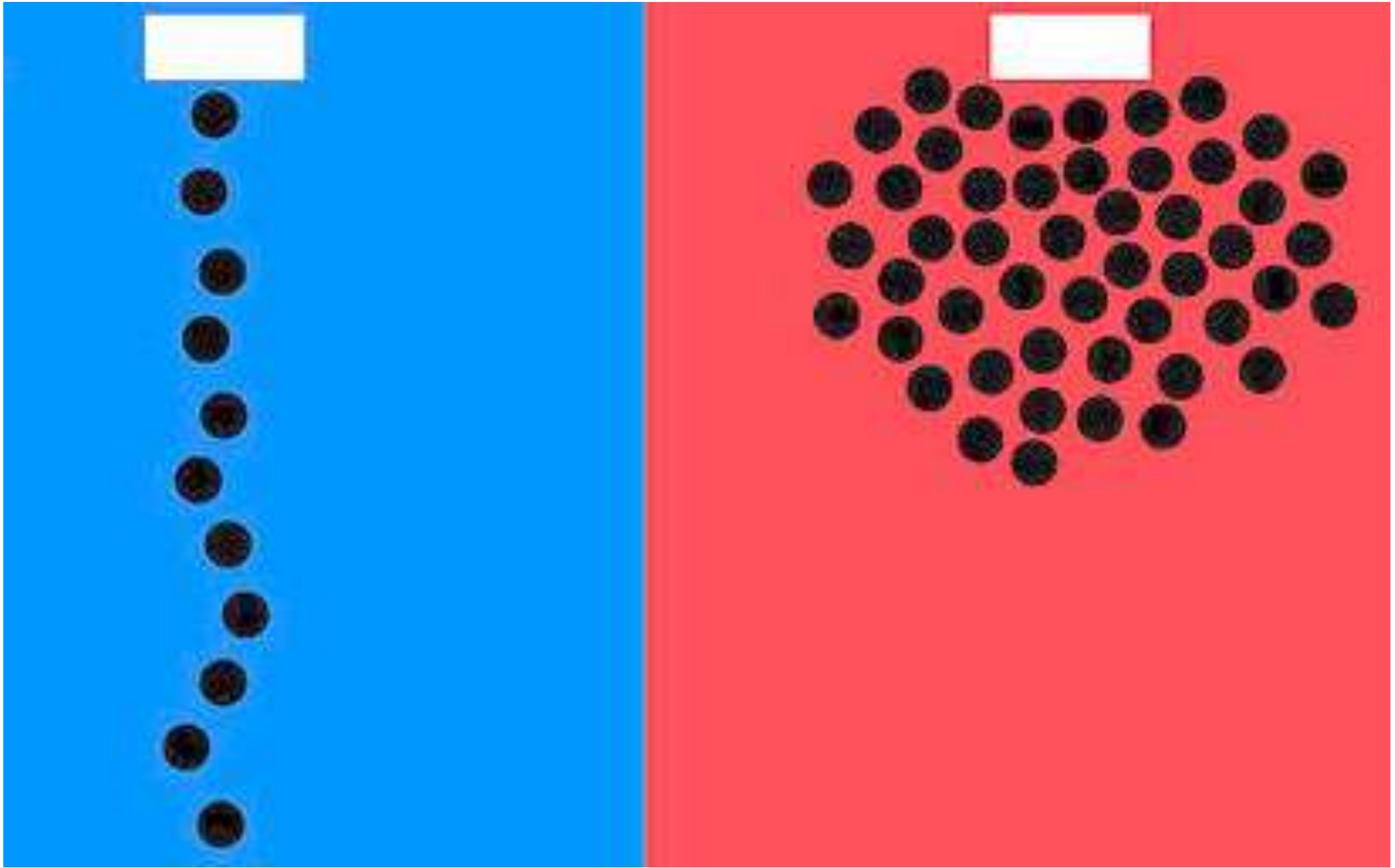
Structure

- Queue ?

Structures

Structures

- Queue
  - Which one ?



Structures

- Queue


gettyimages
Johnny Johnson
200558109-001

Structures

- https://www.bbc.com/news/world-us-canada-66704749
  - “Thousands queue for hours to leave Burning Man festival”
  - “Tens of thousands of people who spent days stranded at the Burning Man festival have started to leave, creating a huge traffic jam in the Nevada desert.”
  - “Heavy rain created a mud bath at the remote festival site and flooded roads, trapping some 72,000 people there.”

Structures

- https://www.bbc.com/news/world-us-canada-66704749
  - “Thousands queue for hours to leave Burning Man festival”

Structures

- Queue and culture
  - www.jstor.org/stable/2775696

Queue Culture: The Waiting Line as a Social System[1]

Leon Mann
Harvard University

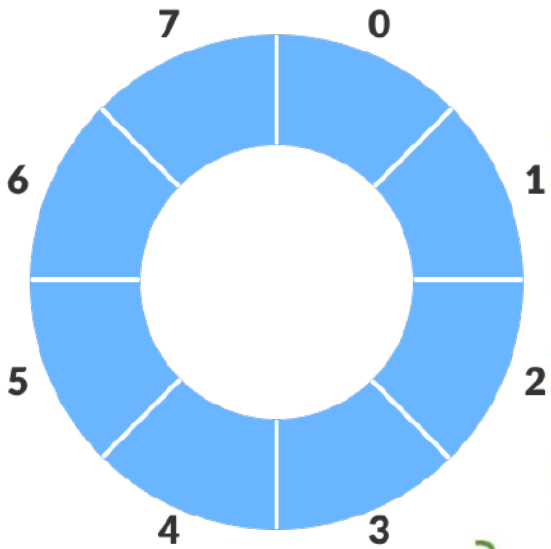
American Journal of Sociology
Vol. 75, No. 3 (Nov., 1969), pp. 340-354 (15 pages)
Published By: The University of Chicago Press

The long, overnight queue is seen as a miniature social system faced with the problems of every social system, formulating its own set of informal rules to govern acts of pushing in and place keeping, leaves of absence, and the application of sanctions. Cultural values of egalitarianism and orderliness are related to respect for the principle of service according to order of arrival which is embodied in the idea of a queue. The importance of time in Western culture is reflected in rules relating to "serving time" to earn one's position in line, and to the regulation of "time-outs." The value of business enterprise is expressed in the activities of professional speculators and queue "counters." Queue jumping is discouraged by a number of constraints, but, if social pressure fails, physical force is seldom used to eject the intruder. Principles of queue etiquette are illustrated with empirical and anecdotal evidence from the study of Australian football queues.

Structures

- Taxi queueing system

```
struct Queue{
  int elements[QUEUESIZE], nWaiting, front;
};
bool insert(Queue &q, int value){
  if(q.nWaiting == QUEUESIZE) return false;  // queue is full
  q.elements[(q.front + q.nWaiting) % QUEUESIZE] = value;
  q.nWaiting++;
  return true;
}
int remove(Queue &q){
  if(q.nWaiting == 0) return -1; // queue is empty
  int item = q.elements[q.front];
  q.front = (q.front + 1) % QUEUESIZE;
  q.nWaiting--;
  return item;
}
```



Structures

- Taxi queueing system
  - This doesn't implement a customer-waiting queue

```
struct Queue{
  int elements[QUEUESIZE], nWaiting, front;
};
```

```
int main(){
  Queue q;
  q.front = 0;
  q.nWaiting = 0;
  while(true){
    char command; cin >> command;
    if(command == 'd'){
      int driver; cin >> driver;
      if(!insert(q, driver)) cout << "Cannot register.\n";
    }
    else if(command == 'c'){
      int driver = remove(q);
      if (driver == -1) cout << "No taxi available.\n";
      else cout << "Assigning: " << driver << endl;
    }
  }
}
```

Structures

- Member functions
  - Structure can be used
    not only to group variables,
    but also
    to group functionality related/specific to those variables
    - Designs a more cohesive module
  - Member function is expected to be called
    on a variable (termed receiver) of that struct type

```
struct structure-type {
  member-description1
  member-description2
  ...
}
```

```
receiver.function-name(argument1, argument2, ...)
```

  - Inside body of function (called on a receiver),
    names of data members of that receiver can be directly referred to

```
struct V3{
  double x,y,z;
  double length(){                  // member function length
      return sqrt(x*x + y*y + z*z);
  }
  V3 sum(V3 b){                     // member function sum
    V3 v;
    v.x = x + b.x;  v.y = y + b.y;  v.z = z + b.z;
    return v;
  }
  V3 scale(double t){               // member function scale
    V3 v;
    v.x = x*t;  v.y = y*t;  v.z = z*t:
    return v;
  }
};
                                    int main(){
                                      V3 u, a, s;
                                      double t;
                                      cin >> u.x >> u.y >> u.z >> a.x >> a.y >> a.z >> t;
                                      s = u.scale(t).sum(a.scale(t*t/2));
                                      cout << s.length() << endl;
                                    }    (u.scale(t))  .sum(  (a.scale(t*t/2))  )
```

Structures

- Member functions
  1. If argument is a large structure being passed, efficient to pass by reference
  2. If argument isn't modified by member function, use const keyword for argument (as usual)
  3. If receiver isn't modified by member function, use const keyword in member function as follows:

```
V3 sum (V3 const &b) const{          // notice the two const keywords
  V3 v;
  v.x = x + b.x;  v.y = y + b.y;  v.z = z + b.z;
  return v;
}
```

Practice Examples for Lab: Set 14

- 1

Define a `struct` for storing complex numbers. Define functions for arithmetic on complex numbers.

  - Write test cases in the main program to test out addition, subtraction, multiplication, and division of two complex numbers input from the user.

- 2

Define a structure for representing axis parallel rectangles, i.e. rectangles whose sides are parallel to the axes. An axis parallel rectangle can be represented by the coordinates of the diagonally opposite points. Write a function that takes a rectangle (axis parallel) as the first argument and a point as the second argument, and determines whether the point lies inside the rectangle. Write a function which takes a rectangle and `double` values `dx,dy` and returns a rectangle shifted by `dx,dy` in the x and y directions respectively.

Practice Examples for Lab: Set 14

- 3

Define a structure class for storing information about a book for use in a program dealing with a library. The class should store the name, author, price, a library accession number for the book, and the identification number of a library patron (if any) who has borrowed the book. This field, patron identification number could be 0 to indicate that the book is not borrowed.

Read information about books from a file into an array of `book` objects. Then you should enable patrons to issue and return books. When a patron issues/returns a book, the patron identification number of the book should be changed. Write functions for doing this. The functions should check that the operations are valid, e.g. a book that is already recorded as borrowed is not being borrowed without first being returned.

Practice Examples for Lab: Set 14

- 4

Define a `struct` for storing dates. Define a function which checks whether a given date is valid, i.e. the month is in the range 1 to 12, and the day is a valid number depending upon the month and the year.

- 5

Write a program to answer queries about ancestry. Your program should read in a file that contains lines giving the name of a person (single word) followed by the name of the father (single word). Assume that there are at most 100 lines, i.e. 200 names. After that, your program should receive a name from the keyboard, and print all ancestors of the person, in the order father, grandfather, great grandfather and so on as known.

Classes

- In many scenarios, when we define a struct that will be used by others:

1. We want to maintain integrity of data within each struct var
   - Control how data variables (inside a struct) will be created, initialized, destroyed, etc.
2. We want to provide a limited view of interior of struct to users
   - Allow users to access struct variables only through a specific set of carefully-designed functions (called "member functions") that form the user-interface for that struct
   - e.g., hide some data members, hide some member functions, etc.

- A general framework for doing all this is the concept of "classes"
- C++ allows 3 kinds of classes: Struct, Class, Union

Classes

- Constructors
  - A special member function to help initialize data members within an instance
  - e.g.,

```
struct Queue{
    int nWaiting, front, elements[QUEUESIZE];
    Queue(){                                    // constructor begins
      front = nWaiting = 0;
    }                                           // constructor ends
}
```

  - General form: Constructor name is same as struct name

```
structure-type (parameter1-type parameter1, parameter2-type parameter2,
                ...){ body }
```

  - Doesn't have a return type
  - Automatically called every time a new struct variable is created
    - It is called on the variable just created

Classes

- Constructors
  - Can be of multiple kinds
    - Function overloading
  - Which constructor gets called depends on how struct variable was defined, e.g., without or with arguments
    - Note: "V3 vec2();" is a function declaration; doesn't call constructor
      - Yes, function declaration can appear inside a function →

```
struct V3{
  double x,y,z;
  V3(double p, double q, double r){     // constructor 1
    x = p;
    y = q;
    z = r;
  }
  V3(){                                 // constructor 2
    x = y = z = 0;
  }
// description of other member functions omitted
};

int main(){
  V3 vec1(1.0,2.0,3.0);
  V3 vec2;
}
```

```
#include <iostream>
using namespace std;
int main()
{
    int f();
    cout << f();
}
int f ()
{
    return 2;
}
```

2

Classes

- Constructors
  - On rare occasions, we can make an explicit call to a constructor
    - Example:

```
V3 vec3, vec4;
vec3 = V3(1.0, 2.0, 3,0);
vec4 = V3();
```

  - The right-hand-side expressions create an object (as a temporary variable), call appropriate constructor, then copy value to object on left-hand-side
  - Another example

```
struct V3{
  ...members and constructors 1 and 2...
  V3 sum (V3 b){
    return V3(x+b.x, y+b.y, z+b.z);
  }
}
```

Classes

```
V3(double p=0, double q=0, double r=0){
    x = p;
    y = q;
    z = r;
}
```

- Constructors
  - Allowing multiple options for number of arguments (via multiple constructors) can also be achieved via default parameters for a single copy constructor
  - Default constructor www.ibm.com/docs/en/zos/2.2.0?topic=only-default-constructors-c
    - A “default” constructor is a constructor that either has no parameters, or if it has parameters, all parameters have default values
      - e.g., above constructor implicitly defines a default constructor
    - When defining an array of class variables, C++ calls default constructor on each variable (syntax doesn’t easily allow calling any other constructor)
    - If code doesn’t define any constructor, then C++ creates one
      - e.g., for class A, C++ will create a default constructor as “A::A() {};”
    - If code defines a constructor that must take arguments, then C++ won’t allow definition of an array of such class variables

Classes

- Constructors of nesting and nested structures
  - After constructor of nesting/parent struct called, before its body executes, constructors of nested structures are called
  - Code shown on right will lead to compilation error
    - Why ?
    - Default constructor of Point is neither defined in code nor will be created by C++

```
struct Point{
  double x,y;
  Point(double p, double q){x=p; y=q;}
};
struct Circle{
  Point center;
  double radius;
};
Circle c;
```

Classes

- Constructors of nested structures
  - "Initialization list" is text following the ":" in definition of nesting/parent struct (Circle) constructor

```
struct Point{
  double x,y;
  Point(double p, double q){x=p; y=q;}
};
```

```
struct Circle{
  Point center;
  double radius;
  Circle(double x, double y, double r) : center(Point(x,y)), radius(r)
  {
  // empty body
  }
};
```

  - It specifies how data members (e.g., center) in nesting struct should be created, before execution of nesting-struct-constructor's body

Structures

- Static data members
  - Suppose we want to keep track of how many struct instances were created
  - C++ allows us to do this by placing the declaration of such a variable (for doing counting) within the definition of struct-type itself
    - e.g., "static int counter;"
    - Nicely organized; this is a good place to store such a variable
  - There is only a single copy of a static variable meant to be "shared" by all instances
  - We need to explicitly define this variable later (to allocate memory)
  - We refer to such a static variable (say, varName) by:
    - varName inside the struct definition, or
    - structName::varName anywhere
  - "static" implies that life of data member is entire run of program
    - Permanent, lifelong

Structures

- Static data members

```
struct Point{
  double x,y;
  static int counter;                    // only declares
  Point(){
    counter++;
  }
  Point(double x1, double y1) : x(x1), y(y1){
    counter++;
  }
};
int Point::counter = 0;                  // actually defines

int main(){
  Point a,b, c(1,2);
  cout << Point::counter << endl;
}
```

Structures

- Static member functions

```
static void resetCounter(){ counter = 0; } // note keyword ‘‘static’’
```

  - Referred to by function name inside struct, and using structName::functionName anywhere
  - Not invoked on an instance (not using dot operator)
    - e.g., called as Point::resetCounter();
  - Because static member functions aren’t invoked on any instance/receiver, it will be an error to refer to any non-static members

Structures

- The “this” pointer
  - Keyword “this” = pointer to receiver,
    inside definition of any (non-static) member function
  - Trivial example

```
double length(){
  return sqrt(this->x*this->x + this->y*this->y + this->z*this->z);
}
```

  - Interesting and common example
    - Member function needs to return an object chosen from (i) receiver and (ii) argument

```
Circle bigger(Circle c){
  return (radius > c.radius) ? *this : c;
}
```

  - How does this work ? When non-static member function is called,
    compiler passes object's address to function as a hidden argument = “this”

Classes

- "Constant" data members
  - Initialized to some values that remain unchanged throughout
  - Note: these values aren't compile-time constants

```
struct Point{
  const double x,y;
  Point(double x1, double y1) : x(x1), y(y1)
  {} // empty body
}
```

  - Semantics of the "const" keyword is that the value of a variable is to be fixed when the variable is created/instantiated (not declared)
    - In the above code,
      "const double x,y;" is a declaration (not definition)
      because it is within the class type definition

Classes

- Copy constructor
  - Constructor that takes a single argument
    whose type is a reference to the class type
  - If code doesn't provide one,
    then C++ creates a default copy constructor
    - C++ invokes this default copy constructor for
      every class instance passed by value to a function
    - Default copy constructor simply
      copies each data member from source instance to destination instance
  - Copy constructor takes argument by reference because … ?
    - Calling copy constructor shouldn't itself need a copy !
  - Sometimes we don't need this default behavior;
    so we can specify our own copy constructor
    - e.g., when copying our Queue, we only copy subset of array with valid information

Classes

- Copy constructor
  - When copying a Queue, we only copy subset of array with valid information

```
struct Queue{
  int front, nWaiting, elements[QUEUESIZE];
  Queue(){                                          // ordinary constructor;
    front = nWaiting = 0;
  }
  Queue(const Queue &other):
    front(other.front), nWaiting(other.nWaiting){   // copy constructor
    for(int i=front, j = 0; j<nWaiting; j++){
      elements[i] = other.elements[i];
      i = (i + 1) % QUEUESIZE;
    }
  }
  ... members insert and remove...
};
```

Classes

- When is a copy constructer called ?
- https://en.cppreference.com/w/cpp/language/copy_constructor
- “

  The copy constructor is called whenever an object is initialized (…)
  from another object of the same type (…),
  which includes:
  1) initialization: T a = b; or T a(b); , where b is of type T;
  2) function argument passing: f(a),
  where a is of type T and f is return-type f(T t);
  3) function return: return a; inside a function such as T f(arguments),
  where a is of type T, …
  ”

Classes

- Destructor
  - Member function (for class T, this will be ~T(); no arguments; no return)
  - Automatically called (cannot be explicitly called) on instance when it is about to go out of scope
  - C++ will provide a default destructor
  - We can provide one to override default destructor
    - To change functionality (relevant in case of dynamically-allocated data members; we'll see later)
    - To inform user

```
struct Queue{
  ... other member definitions as before ..
  ~Queue(){
    if(nWaiting > 0)
      cout <<"Warning: Non-empty Queue being destroyed.\n";
  }
};

int main(){
  Queue q;
  {
    Queue q2;
    q2.insert(5);
  }
}
```

Classes

```
sum(scale(u,t), scale(a,t*t/2));
```

```
u*t + a*t*t*0.5
```

- Overloading operators for classes
  - Consider objects of a Vector class, say v and w
  - Instead of calling sum(v,w), wouldn't it be more reader-friendly to call "v+w"
  - Here, we'd need to re-define "+" operator for objects of type Vector, by creating member function named "operator+" (operator is a keyword)
  - In this example, "v+w" would call member function operator+() on instance v with argument w
  - Similarly, we can overload operator * to work as "v*2"
  - What if we need to make "2*v" also work ?
    - We'll see soon

```
struct V3{
  // members and constructors as defined earlier
  V3 operator+ (const V3 &b) const{
    return V3(x + b.x, y+b.y, z+b.z);
  }
  V3 operator* (double t) const{
    return V3(x*t, y*t, z*t);
  }
};
```

Classes

- Overloading operators for classes
  - C++ allows many operators to be overloaded

```
+   -   *   /   %   ^   &   |   <   >   ==  !=  <=  >=  <<  >>  &&  ||
=   +=  -=  *=  /=  %=  ^=  &=  |=  <<=  >>=  []
```

Classes

- Operator overloading using non-member functions
  - Allowing “2*v”

```
V3 operator* (double factor, const V3 & v){
  return v*factor;
}
```

    - Assuming member-function operator* already defined
  - Such things need non-member functions because first operand (i.e., 2) isn’t an instance of class V3
  - Allowing “cout << v”

```
ostream & operator<< (ostream & ost, V3 &v){
ost << v.x <<’ ’<< v.y <<’ ’<< v.z <<’ ’;
  return ost;
}
```

    - “cout” has type “ostream”
    - Non-member function needed (as in 2*v)
    - Function returning ostream allows us to chain outputs as “cout << v1 << v2”;
  - C++ allows some operators overloaded as ordinary/non-member functions, not all

Classes

- Overloading assignment operator
  - Underlying functionality somewhat similar to copy constructor
    - Copy only valid elements in array
  - Why should we do "return *this" ?
    - To allow chaining e.g., Q1 = Q2 = Q3;

```
struct Queue{
   .. other members as before ..
   Queue & operator=(const Queue& rhs){
     front = rhs.front;
     nWaiting = rhs.nWaiting;

     for(int i = front, j=0; j<nWaiting; j++){
       elements[i] = rhs.elements[i];
       i = (i + 1) % QUEUESIZE;
     }
     return *this;
   }
};
```

Classes

- Access Control through Access Specifiers
  - Limiting access to data/function members by other programmer users
  - Designate each data/function member as:
    - Private: accessible only inside function members of class; super-sensitive data and functions
    - Public: accessible by all
    - Protected: more nuanced, beyond CS-101
  - Examples
    - Making assignment operator private prohibits assignment of struct instance outside function members of class
    - Making copy constructor private prohibits passing struct instance by value, or returning struct instance; passing by reference still allowed

```
struct Queue{
private:
   int front;
   int nWaiting;
   int elements[QUEUESIZE];
public:
   Queue(){...}
   bool insert(int value){...}
   int remove(){...}
};
```

Classes

- Keywords "struct" versus "class"
  - Very similar except that:
    for a "class", all members are private by default;
    for a "struct", all members are public by default
  - Instances/variables of a "class" are termed "objects"

```
class Queue{
...
};
```

- Interface of a class/struct = set of public data/function members
- C language first had notion of "struct"
  - In C++, this is called as plain-old data (POD) structure
    or passive data structure (PDS)
    - Classic notion of a record
  - C-style struct has no member functions,
    no user-defined constructor / destructor / assignment operator,
  - en.wikipedia.org/wiki/Struct_(C_programming_language)

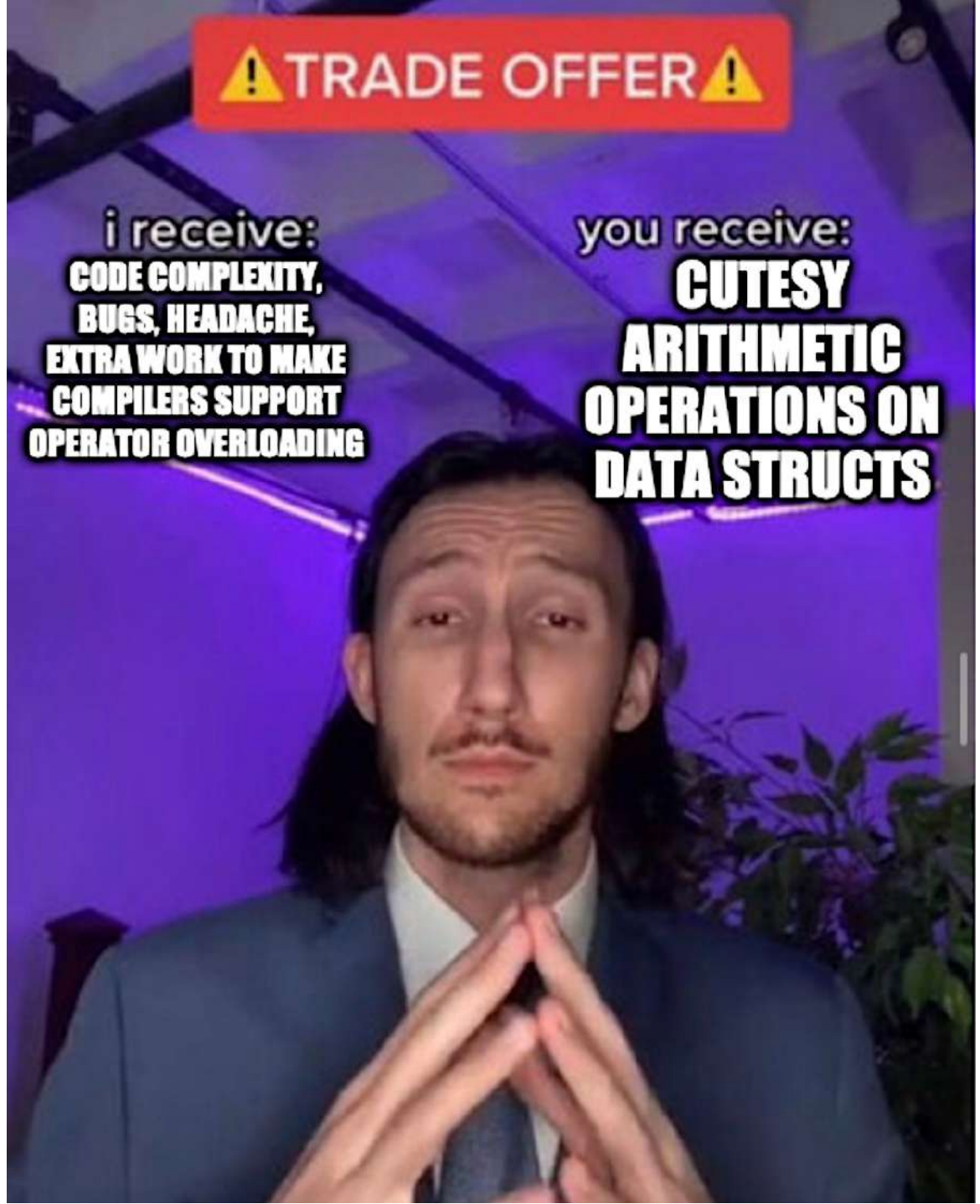
Classes

- Friends

```
ostream & operator<< (ostream & ost, V3 &v){
  ost << v.x <<' '<< v.y <<' '<< v.z <<' ';
  return ost;
}
```

- If you want to declare data members of V3 as private,
  but still want to allow, say, operator overloading for << used with cout
- Then you need to declare non-member function operator<<
  as a "friend" within struct definition

```
struct V3{
   ...
   friend ostream & operator<< (ostream &ost, const V3 &v);
   ...
}
```


⚠TRADE OFFER⚠
i receive:
CODE COMPLEXITY, BUGS, HEADACHE, EXTRA WORK TO MAKE COMPILERS SUPPORT OPERATOR OVERLOADING
you receive:
CUTESY ARITHMETIC OPERATIONS ON DATA STRUCTS


the 'h' in
'software development'
stands for
'happiness'

Classes

- Example: Ratio Class
- Constructor (public)
  - Default
  - With parameters
  - Initializer list
- Copy constructor (public)

```
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
COPY CONSTRUCTOR CALLED
```

```
class Ratio
{ public:
    Ratio(int n=0, int d=1) : num(n), den(d) { reduce(); }
    Ratio(const Ratio& r) : num(r.num), den(r.den)
    { cout << "COPY CONSTRUCTOR CALLED\n"; }
  private:
    int num, den;
    void reduce();
};

Ratio f(Ratio r)  // calls the copy constructor, copying ? to r
{ Ratio s = r;    // calls the copy constructor, copying r to s
  return s;       // calls the copy constructor, copying s to ?
}

int main()
{ Ratio x(22,7);
  Ratio y(x);  // calls the copy constructor, copying x to y
  f(y);
}
```

Classes

- Example: Ratio Class
- Destructor (public)

```
class Ratio
{ public:
    Ratio() { cout << "OBJECT IS BORN.\n"; }
    ~Ratio()  { cout << "OBJECT DIES.\n"; }
  private:
    int num, den;
};

int main()
{ { Ratio x;                          // beginning of scope for x
    cout << "Now x is alive.\n";
  }                                        // end of scope for x
  cout << "Now between blocks.\n";
  { Ratio y;
    cout << "Now y is alive.\n";
  }
}
```

```
OBJECT IS BORN.
Now x is alive.
OBJECT DIES.
Now between blocks.
OBJECT IS BORN.
Now y is alive.
OBJECT DIES.
```

Classes

- Example: Ratio Class
- Overloading >> operator (friend)

```
class Ratio
{   friend istream& operator>>(istream&, Ratio&);
    friend ostream& operator<<(ostream&, const Ratio&);
  public:
    Ratio(int n=0, int d=1) : num(n), den(d) { }
    // other declarations go here
  private:
    int num, den;
    int gcd(int, int);
    void reduce();
};
int main()
{ Ratio x, y;
  cin >> x >> y;
  cout << "x = " << x << ", y = " << y << endl;
}
istream& operator>>(istream& istr, Ratio& r)
{ cout << "\t  Numerator: ";  istr >> r.num;
  cout << "\tDenominator: ";  istr >> r.den;
  r.reduce();
  return istr;
}
```

```
          Numerator: -10
        Denominator: -24
          Numerator: 36
        Denominator: -20
x = 5/12, y = -9/5
```

Practice Examples for Lab: Set 15

- 1

Define a class for storing polynomials. Assume that all your polynomials will have degree at most 100. Write a member function `value` which takes a polynomial and a real number as arguments and evaluates the polynomial at the given real number. Overload the `+,*,-` operators so that they return the sum, product and difference of polynomials. Also define a member function `read` which reads in a polynomial from the keyboard. It should ask for the degree d of the polynomial, check that $d \leq 100$, and then proceed to read in the first $d+1$ coefficients from the keyboard. Define a `print` member function which causes the polynomial to be printed. Make sure that you only print $d+1$ coefficients if the actual degree is d. Carefully decide which members will be private and which will be public. Overload the `>>`, `<<` operators so that the polynomial can be read or printed using them.

Practice Examples for Lab: Set 15

- 2

Define a class for storing complex numbers. Provide 0, 1, 2 argument constructors which respectively construct the complex number 0, a complex number with imaginary part 0 and real part as specified by the argument, and a complex number with real and imaginary parts as specified by the arguments. Overload the arithmetic operators to implement complex arithmetic.

- 3

  Implement the following operators for the Ratio class:
  addition, subtraction, multiplication, division, <, >, ==, =

- 4

  Implement the following operators for the Point class:
  =, << (for output), ==, !=, +, -, * (inner product)

Practice Examples for Lab: Set 15

- 5

Define the operator `>>` for the class `V3`. This should enable you to write `cin >> v;` where `v` is of type `V3`. When this is executed, the user will type in 3 floating point numbers which will get placed in `v`.

- 6

Implement a `Matrix` class for 2-by-2 matrices:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

Include a default constructor, a copy constructor, an `inverse()` function that returns the inverse of the matrix, a `det()` function that returns the determinant of the matrix, a Boolean function `isSingular()` that returns 1 or 0 according to whether the determinant is zero, and a `print()` function.

Practice Examples for Lab: Set 15

- 7

Implement a `Circle` class. Each object of this class will represent a circle, storing its radius and the x and y coordinates of its center as `floats`. Include a default constructor, access functions, an `area()` function, and a `circumference()` function.

- 8

Implement a `Point` class for two-dimensional points (x, y). Include a default constructor, a copy constructor, a `negate()` function to transform the point into its negative, a `norm()` function to return the point's distance from the origin (0,0), and a `print()` function.

- 9

Implement a `Time` class. Each object of this class will represent a specific time of day, storing the hours, minutes, and seconds as integers. Include a constructor, access functions, a function `advance(int h, int m, int s)` to advance the current time of an existing object, a function `reset(int h, int m, int s)` to reset the current time of an existing object, and a `print()` function.

Classes

- Template classes
  - Generalize class type definition
  - Instantiate by giving a specific data type to T, e.g., “V3<float> a, b, c;”

```
#include<iostream>
using namespace std;

template<typename T> struct V3
{
    T x, y, z;
    V3 (T a=0, T b=0, T c=0) { x = a; y = b; z = c; }
    V3 operator+ (V3 w);
};

template<typename T> V3<T> V3<T>::operator+ (V3 <T> w)
{ return V3(x+w.x, y+w.y, z+w.z); }
```

Classes

- Sharing a class: header file and cpp file
  - Header V3.h file contains class type definition, data members, member-function declarations, friend-function declarations (twice; see below)

```
class V3{
private:
  double x, y, z;
 public:
  V3(double p=0, double q=0, double r=0);
  V3 operator+(V3 const &w) const;
  V3 operator*(double t) const;
  double length() const;
  friend ostream & operator<<(ostream & ost, V3 v);
};

ostream & operator<<(ostream & ost, V3 v);
```

Classes

- Sharing a class: header file and cpp file
  - V3.cpp file has implementations of all functions
  - Now, as earlier, compile V3.cpp to give V3.o
  - Share with user: V3.o , V3.h
  - User has her own user.cpp, with #include “V3.h”
  - User compiles: s++ user.cpp V3.o

```
#include <simplecpp>
#include "V3.h"

V3::V3(double p, double q, double r){  // constructor
  x = p;   y = q;   z = r;
}
                                       // member functions
V3 V3::operator+(V3 const &w) const { return V3(x+w.x, y+w.y, z+w.z); }

V3 V3::operator*(double t) const { return V3(x*t, y*t, z*t); }

double V3::length() const { return sqrt(x*x+y*y+z*z); }

                                       // other functions
ostream & operator<<(ostream & ost, V3 v){
  ost << "(" << v.x << ", "<< v.y << ", "<< v.z << ")";
  return ost;
}
```

File IO

- To read data from files (instead of terminal input)
- #include <fstream>
- Create file streams for input and output
- Basic usage is similar to what we did for cin, cout
- Check for infile/outfile being NULL to check if file operations succeded
- While reading, use infile.eof() to check if end of file reached

```
cat input.txt
1
2
3
4
5
6
```

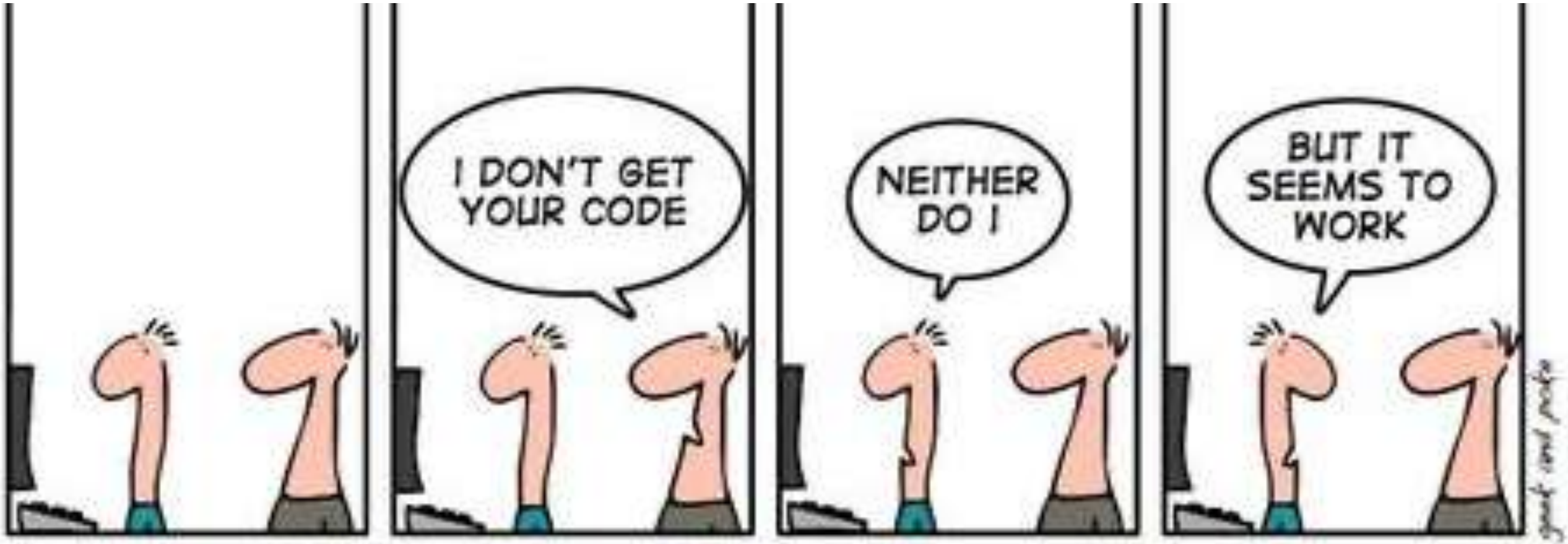
```
cat inputCopy.txt
1
2
3
4
5
6
```

```
#include<fstream>

int main ()
{
  std::ifstream inFile ("input.txt");
  if (inFile == NULL) return -1;

  std::ofstream outFile ("inputCopy.txt");
  if (outFile == NULL) return -2;

  int value;
  inFile >> value;
  while (! inFile.eof())
    {
      outFile << value << std::endl;
      inFile >> value;
    }
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    while (x --> 0) // x goes to 0
        cout << x << " ";
}
```

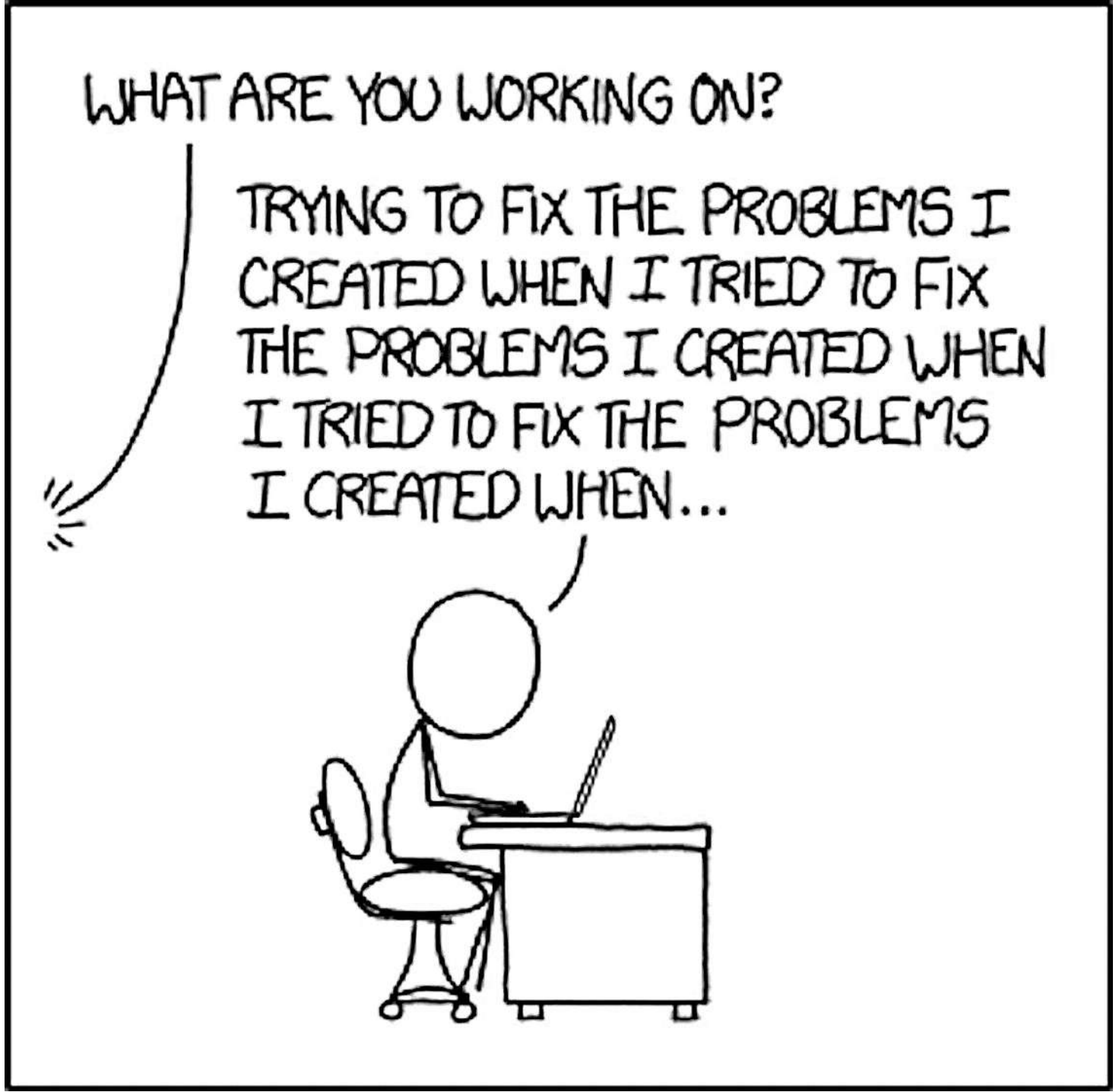
```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    while (0 <---- x)
    // x comes to 0, even faster
        cout << x << " ";
}
```

```cpp
#include <iostream>
using namespace std;
int main()
{
    int x = 10;
    while (x --\
               \
                \
                 \
                  > 0) // x slides to 0
        cout << x << " ";
}
```



THE ART OF PROGRAMING

- Recursion in debugging



In Unix there is a "man" command and a "cat" command...

so why no "dog" command???

Feeling left our of the fun....