CS31

Smallberg

Fall 2019

Contents

CS31

11.5.18

C++ Basics

```
#include <iostream>   // defines a special library of commands to be inserted here by
    compiler
using namespace std;  // different families / 3rd party libraries use
                      // diff. namespaces to help distinguish themselves
                      // 'using' now defaults to this namespace
int main()
{
  cout << "Hello!" << endl; // cout prints to standard output
                            // endl moves cursor to next line
}
```

- not `#include <iostream.h>` as in C
- **include** statement:
  - compiler has to learn standard library beyond built-in language
  - ie. fundamental parts of English vs. parts that change over time
- fully qualified **cout** statement would be `std::cout`
- whitespace in code doesn't matter (format for readability), but spaces in literal text does matter ("Hello!")
- **cin** reads input into a variable

```
int main()
{
  cout << "How many hours? ";
```

```
  // variable declaration form: type identifier
  double hoursWorked;   // uninitialized, value is unspecified / garbage
  cin >> hoursWorked;
  cout << "Pay rate? "; // enter for input already skips the next line
  double payRate;
  cin >> payRate;

  cout << "The hours worked are " << hoursWorked << endl; // testing print statements
  cout << "The pay rate is " << payRate << endl;

  cout << "You made $" << hoursWorked * payRate << endl;  // spaces matter in literal text!
  cout << "Money withheld $" << .1 * (hoursWorked * payRate) << endl;
}
```

- C++ naming conventions:
  - eg. hours_worked, HoursWorked, hoursWorked (can't be reserved words)
  - case sensitive!
- `cout <<`, output, read out
- `cin >>`, input, read in

Strings

```
#include <string>           // string header library

string personsName;
getline(cin, personsName);  // only used for strings

cin >> personsName;         // ignores whitespaces, only grabs one word

cin >> age;                 // must be used for integer variables
                            // skips input if getline() is called after because of the trailing '/n'

```

```
cin.ignore(10000, '/n');    // throws away buffer up to and including newline character
                            // this issue only occurs when reading a number then call getline()

string s = "Hello";
for (int k = 0; k != s.size(); K++) // s.size() returns 5, char c = s[1];
  cout << s[k] << endl;             // use subscript operator, behaves like an array of characters, from 0 to 4

const double PAYRATE_THRESHHOLD; // constants naming conventions and syntax

if (citizen == "US")
  if (age >= 18)
    cout << "You can work." << endl;
else cout << "Not U.S. citizen." << endl; // automatically pairs to second if (closesnt unpaired if)
                                          // must add brackets for intended operation
```

- calling **getline()** after a **cin** execution leaves a '/n' in the buffer
  - **cin** does not consume '/n'
  - so **getline()** sets string to empty string
  - program continues executing
  - must use `cin.ignore()`
- modifying **cout** with flags:
  - `cout.setf(ios::fixed);`, different double modes, scientific, exponential
  - `cout.precision(2);`, number of digits after decimal point
  - showpoint command always shows point (even with no floating digits)

Operations with Strings

- **size()** function (historically, **length()**)
- s[0] accesses individual characters
- use `i != s.size()` when iterating through string
  - technically, returns type string::size_type, or an unsigned int
  - if unsigned, can't iterate size_type backwards, will give an error
- '+=' operator can be used to append to strings
- **substr()** function

- `s.substr(5, 3)` create a substring starting at position 5 going for 3 characters
- `t = t.substr(6, t.size() - 6)`, clips off the first six characters

More Misc. String/Char Rules

- can't access index of a string out of bounds
- when using toupper() and other conversion functions, make sure to save the char
  - eg. `s[0] = toupper(s[0]);`
- `if (t[k] == 'E' || t[k] == 'e')` is equivalent to `if (toupper(t[k] == 'E'))`
- assignment returns a value:
  - `n = 2 * (k = 3 + 5);`

Char and Int

- chars share a lot of properties with int (automatic conversion)
- `char ch = 76;` ch is now 'L', depending on character set
- `int k = 'L';` k is now 76 (integer encoding of the character)
- standard dictates:
  - 'a' < 'b, 'y' < 'z', etc.
  - 'A' < 'B', 'Y' < 'Z', etc.
  - but no special relationship between the two
  - doesn't guarantee contiguous encoding! (but ASCII does)
  - '0' '1' '2' etc. are contiguous
  - `cout << tallySeats(...,...,s) << " " << s;`
    - where s is a reference to a declared int
    - standard doesn't dictate which operand is processed first (s or tallyseats())
    - solve by splitting into two statements
    - or use assert to test (will short-circuit / evaluate from left to right)

Character Classification

```cpp
#include <cctype> // do operations / checks on characters

isalpha();
isupper();
islower();
isdigit();
tolower();
```

```
toupper();
```

Functions

- useful for self-contained sequences, reusing specific codes / functions
- void functions don't return value, can use return within function block to break out
  - variables obey strict scope guidelines within function blocks
- functions with return type must return a value
  - every possible path must result in a return statement
- boolean type: holds true or false (keywords)
- naming convetion: use predicate forms, eg. isdigit(), isalpha(), livesin()
- it is not possible to return more than one value
  - instead have function save values into variables using pass by reference
    - passing by value -> copy
    - passing by reference -> another name for original
  - otherwise values will not save
  - double means a memory location that can hold a double
  - double& means another name for an already existing double, "reference to double"
- functions need prototypes so the compiler knows functions when they are called before implemented

More Loop Conventions

```
cout << "Phone #";
string phone;
getline(cin, phone);
while (!isValid(phone))
{
  cout << "Must have 10 digits" << endl;
  cout << "Phone #";
  getline(cin, phone);  // repetition of code twice! can we replace with do-while loop?
}
```

```
// VS.
for (;;)                // n-and-a-half-times loop
{
  cout << "Phone #";
  getline(cin, phone);
  if (isValid(phone))   // condition is tested in the middle, not top (while loop),
   or bottom (do-while loop)
    break;
  cout << "Must have 10 digits" << endl;
}

// Another example:
int nScores = 0;
int total = 0;
for (;;)                // n-and-a-half-times loop
{
  int s;
  cin << s;
  if (s < 0)
    break;
  total += s;
  nScores++;
}
// cout << "Average " << total / nScores << endl; error, integer division, also
    should check nScores == 0
cout << "Average " << static_cast<double>(total) / nScores << endl; // cast creates
    temporary, unnamed object

for (...)
{
  if (...) // if and else close, easy to see
    ...
  else
  {
    ...    // nested, indented code can be hard to read, can we improve?
    ...
```

```
  }
}
// VS.
for (...)
{
  if (...)
  {
    ...
    continue; // abandons current iteration of the for loop, jumps to end of the
   brackets
  }
  ...
  ...
}

int k;
for (k = 0; k < 10; k++)
{
  ...
  if (...)
    continue; // k++ still happens at the end of the loop after continue
}
// VS.
while (k < 10)
{
  if (...)
    continue; // k++ is skipped after continue
  k++;
}
// continue will act differently in these formats!
```

Arrays

- how to set up a table for irregular patterns such as month/day?
- use arrays:

```
const int daysInMonth[12] = {
  31, 28, 31, 30, 31, 30      // easier to see array if split up/organized
  31, 31, 31, 31, 31, 31
};
```

- arrays start with 0
- undefined out-of-bounds behavior
- can utilize paired/parallel arrays (eg. month name and days in month)
- good practice to hold similar digits in const variable with symbolic name
- there is NO size/length function for arrays!!!
- arrays size must be known at compile time!!!

```
int n;
cin >> n;
double d[n]; // error! not a const variable

int main()
{
  const int MAX_NUM_SCORES = 10000;
  int scores[MAX_NUM_SCORES];
  int nScores = 0;
  ... fill up array partially
  computeMean(scores, nScores);
  int stuff[100];
  computeMean(stuff, 100);
}
double computeMean(int a[], int n) // cannot check number of elements, must be passed
    as a parameter
{
  int total = 0;
  for (int k = 0; k < n; k++)
    total += a[k];                  // passes directly by reference, not a copy
  return static_cast<double>(total) / n;
}
```

- const arrays cannot be modified
  - cannot be passed to functions modifying it (without const)
  - compiler catches the error

C-Strings

```
#define _CRT_SECURE_NO_WARNINGS

#include <cstring>

char t[10] = { 'G', 'h', 'o', 's', 't'}; // allowed to have initializer list < total length
char t[10] = "Ghost"; // uses null character in order to denote end of a string, '\0' (zero byte)
char s[100] = "";

for (int k =0; t[k] != '\0'; k++)
  cout << t[k] << endl;
cout << t; // cout << is overloaded

cin.getline(s, 100);

// s = t; Error! Can't assign arrays!
strcpy(s, t); // strcpy(destination, source), up to and including zero byte

strcat(s, "!!!"); // now s is "Ghost!!!"

// if (t < s) Compiles, but compares addresses, not the actual strings.
if (strcmp(a, b) < 0)

//if (strcmp(a, b)) //Yields OPPOSITE result
if (strcmp(a, b) == 0)
```

- c-strings are character arrays terminated by zero byte

- null pointer != null character (zero byte)
- technically, string literals are always c-strings
- declaring with double quotes automatically appends a zero byte
- don't use `k != strlen[t]` when iterating through c-string, instead use `t[k] != '\0'`
- function library with c-strings:
  - cstring library has **strlen()**, unlike c++ strings
  - **getline()** has different parameters for c-strings, string and buffer size
  - when using **strcpy()**, make sure t is a valid c-string and destination has a large enough size.
  - **strcat()** finds zero byte and then appends. Make sure destination string is big enough and has a zero byte.
  - **strcmp(a, b)** returns:
    * c++ strings: a OP b
    * c-strings: strcmp(a, b) OP 0
    * negative if a < b
    * 0 if a == b
    * positive if a > b
- use `#define _CRT_SECURE_NO_WARNINGS` to stop compiler warnings

Comparing C++ Strings and C-Strings

C++ Strings	C-Strings
string s; // default constructor, guaranteed the empty string, unlike other built-in types	char s[100]; // unintialized, not empty string
size()	strlen(t)
[ ] operator for characters	[ ] subscript operator
getline(cin, s); // read in input	cin.getline(s, 10); // read in input for 10 characters
s = t;	s = t; // error, arrays can't be assigned
s += "!!!";	s += "!!!"; // error, not supported
t < s	use strcmp(a, b) for comparison

Converting C++ Strings and C-Strings

```
void f(const char cs[])
```

```
{
  ...
}

int main()
{
  string s = "Hello";
  f(s); // Won't compile
  f(s.c_str()); // OK

  char t[10] == "Ghost";
  s = t; // Assigning c-string to c++
  t = s; // Won't compile, can't use assignment OP with c-strings
  t = s.c_str(); // Won't compile
  strcpy(t, s.c_str()); // Works
}
```

2-D Arrays

Structured tables are easier to visualize, eg. calendar:

```
const int NWEEKS = 5;
const int NDAYS = 7;

int attendance[NWEEKS][NDAYS];
cout << attendance[2][5];

for (int w = 0; w < NWEEKS; w++) // Iterate through 2-D arrays with nested loops
{
  int t = 0;
  for (d = 0; d < NDAYS; d++)
    t += attendance[w][d];
  cout << "The total for week " << w << " is " << t << endl;
}
```

```
const string dayNames[NDAYS] = {
  "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
};

int grandTotal = 0; // Put assignment in the right area
for (int d = 4  /*Friday*/ ; d < NDAYS: d++)
{
  int t = 0;
  for (int w = 0; w < NWEEKS; w++)
    t += attendance[w][d];
  cout << "The total for " << dayNames[d] << "is " << t << endl;
  gradTotal += t;
}
cout << grandTotal;
```

More 2-D Array Functions

```
double meanForADay(const int a[][NDAYS], int nRows, int dayNumber)
// MUST specify bounds for any other dimensions (first dimension usually pased as a
    parameter)
{
  if (nRows <= 0)
    return 0;
  for (int r = 0; r < nRows; r++)
    total += [r][dayNumber];
  return static_cast<double>(total) / nRows;
}

int main()
{
  int attendance[NWEEKS][NDAYS];

  double meanFri = meanForADay(attendance, NWEEKS, 4 /*Friday*/);
}
```

```
int multiplexChainAttendance[5][7][10][16];
// Valid, 10 multiplexes with 16 screenings, should use symbolic constants

void f(int b[][7][10][16], ...);
```

11.7.18

Array of C-Strings

Array of c-strings is an array of array of chars - another 2-D array.

```
const int MAX_WORD_LENGTH = 6;

int countLength(const char a[][MAX_WORD_LENGTH + 1], int n, int targetLength);

int main()
{
  const int MAX_PETS = 5;

  char pets[MAX_PETS][MAX_WORD_LENGTH + 1] = {
  // Longest word length in this case is 6, +1 to account for zero byte.
    "cat", "mouse", "eel", "ferret", "horse"
  };

  cout << countLength(pets, MAX_PETS, 5); // How many 5-character strings?
}
// Only really 1-D arrays : (2-D arrays are simply arrays of elements,
// where each element is another array).
int countLength(const char a[][MAX_WORD_LENGTH + 1], int n, int targetLength)
{
  int total = 0;
  for (int k = 0; k < n; k++)
  {
    if (strlen(a[k]) == targetLength) // a[1], or shorthand for an element of the 2-D
      array, is the second row,
```

```
        // is array of chars, can treat as c-string and use strlen()
        // However, arrays and columns are not treated similarly
      total++;
    }
    return total;
}
```

- Restriction of having to predetermine size of arrays / c-strings: use const variables.
- If using countLength() with strings, similar structure:
  - pass array as `const string a[]`
  - `if(a[k].size() == targetLength)`

Pointers

- Another way to implement passing by reference
  - `void f(int& n)` in C++, reference-to-int or another-name-for-some-int
  - in C, pointers are only way to pass by reference
- Traverse arrays
- Manipulate dynamic storage
- Represent relationships in data structures

1. Pointers as alternative way to pass as reference:
   - pointers are variables that point to the memory location of another variable
     - eg. pointer 'xx' passes "an indication of where x is", not a copy of 'x'
     - stores values in variable where 'x' is, not in 'xx'

```
#include <cmath>
int main()
{
  polarToCartesian(r, angle, &x, &y);
}
void polarToCartesian(double rho, double theta, double* xx, double* yy)
{ // 'xx' is not a double!
  // cannot pass double to 'yy' either, will not compile
  // pointers (has actual value) and references (another name) are not
     same type either
```

```
  // xx = rho * cos(theta); Will not compile!
  *xx = rho * cos(theta);
  *yy = rho * cos(theta);
}
```

- double& means reference-to-int or another-name-for-some-int
- double* means pointer-to-double or the-address-of-some-double
- &x means "generate a pointer to x" or "address of x" (operator)
- *p means "follow the pointer p" or "the object that p points to"

More pointer practice

```
double a = 3.2;
double b = 5.1;
double* p = &a;
                // double* q = 7.6; Won't compile, wrong types.
double c = a;
                // double d = p; Another type incompatibility.
double d = *p;
double& dd = d; // Usually references are only used when passing to functions.
                // p = b; Another type incompatibility.
p = &b; // Assigning one pointer to another
        //or
        //*p = b; Assigns b to a
*p += 4; // *p = *p + 4, b is 9.1
int k = 7;
                // p = &k; Won't compile since &k is pointer to int and p is pointer to double
                // bit patterns wouldn't match up
                // cannot convert pointers of one type to another
int* z = &k; // New pointer type
cout << (k * b);
                // cout << (k * p); Won't compile, can't multiply an int and a pointer.
cout << (k * *p); // Ignores whitespace, so equivalent to (k**p).
cout << (*z * *p);
```

11.14.18

Pointers cont.

```
double* q;
            // *q = 4.7; Undefined, trying to follow a pointer that hasn't been
   initialized.
            // Run-time error, similar to index-out-of-bounds.
            // could be outside accessible memory, or stored in random program
   memory location.
q = p; // points to b
double* r = &a;
*r = b; // assigns b to a;

if (p == r) // false. comparing two pointers
  cout << "Hello";

if (p == q) // true

if (*p == *r) // true
```

2. Pointers as an alternative to parsing through arrays:

- another way to visit each element

```
const int MAXSIZE = 5;
double da[MAXSIZE];
int k;
double* dp;

for (k = 0; k < MAXSIZE; k++)
  da[k] = 3.6;

for (dp = &da[0]; dp < &da[MAXSIZE]; dp++) // dp++ ==> dp +=1
                              // dp = &da[0] + 1;
```

```
                              // dp = &da[0 + 1];
                              // dp = &da[1];
                      // &da[t];
                      // &da[0 + 5];
                      // &da[0] + 5;
                      // da + 5;
  *dp = 3.6;
             // *dp = 3.6;
             // *(&da[0]) = 3.6;
             // da[o] = 3.6;
// syntax of loop above is equivalent to:
for (dp = da; dp < da + MAXSIZE; dp++)

int lookup(const string* a, int n, string target)
{
  ... a[k] ...
}
int main()
{
  string sa[5] = {"cat", "mouse", "eel", "ferret", "horse"};
  lookup(sa , 5, "eel");
  lookup(&sa[0], 5, "eel");
  lookup(sa + 1, 3, "ferret"); // passing &sa[1], checks elements 2 through 4
}
```

- `*&x ==> x`
- `&a[i] + j ==> &a[i + j]`
- `&a[i] <&a[j] ==> i < j`, also other logical operators such as `<=`, !=, etc as long as referencing the same array
- allowed to generate pointer just past end of the array, but cannot follow that pointer
  - cannot generate pointer of negative index
- pointer arithmetic always in terms of the pointer type
- `dp++` in machine language is translated to adding 8 bytes to the pointer-to-double
- in most expressions, array name by itself is treated as pointer to element 0 of array
  - ie. when passing arrays to functions

    * lookup(const string a[]), string sa[5], function call lookup(sa)
        · string a[] is really string*
    * generates a pointer to sa[0]
    * could have passed &sa[0]
- p[i] ==> *(p + i)
- thus, can work on any contiguous portion of an array by passing a pointer

```
string* fp;
sstring fish[5];
fp = &fish[4]; // fp = fish + 4;
*fp = "yellowtail"; // fp[0] = "yellowtail";
*(fish + 3) = "salmon"; // fish[3] = "salmon";
fp -= 3;
fp[1] = "loach"; // (fp + 1)[0] = "loach"; fp isn't changed
fp[0] = "trout";
bool d = (fp == fish);
bool b = (*fp == *(fp + 1));

/* fp[1] == *(fp + 1)
   *fish, *fp
   fish[0], fp[0] */
```

11.19.18

Even more pointers...

```
int findFirstNegative(double a[], int n) // Returning an index with array notation.
{
  for (int k = 0; k < n; k++)
  {
    if (a[k] < 0)
      return k;
  }
  return -1;
```

```
}
int findFirstNegative(double a[], int n) // Returning an index with pointer notation.
{
  for (double* p = a; p < a + n; p++)
  {
    if (*p < 0)
      return p - a; // How far *ahead* is one element from the other.
                    // Remember, in terms of memory, compilor is still
                    // working in terms of the type pointed to.
  }
  return -1;
}
double* findFirstNegative(double a[], int n) // Returning a pointer.
{
  for (double* p = a; p < a + n; p++)
  {
    if (*p < 0)
      return p;
  }
  return a + n; // Returns pointer just past end of the array.
  // return nullptr; Alternative return value.
}
int main() // Returning an index.
{
  double da[5];
  int fnpos = findFirstNegative(da, 5);
  if (fnpos == -1)
    cout << "No negatives." << endl;
  else
  {
    cout << "First negative value is " << da[fnpos] << endl;
    cout << "At element " << fnpos << endl;
  }

}
int main() // Where findFirstNegative() returns a pointer.
```

```
{
  double da[5];
  double*pfn = findFirstNegative(da, 5);
  if (pfn == da + 5)
  // if (pfn == nullptr) Alternative.
    cout << "No negatives." << endl;
  else
  {
    cout << "First negative value is " << *pfn << endl;
    cout << "At element " << pfn - da << endl; // pfn - &da[0];
  }
}
```

- There is another way to indicate a pointer function has failed
- null pointer value
- c++11: nullptr
- earlier: NULL
- double* p = nullptr;
  - `if (p == nullptr)`
  - `if (p != nullptr)`
  - `*p` is undefined if p has null pointer value

```
int* p1;
int* p2 = nullptr;
... *p1 ... // undefined behaior : p is not initialized
... *p2 ... // undefined behavior: p has the null pointer value
          // Reality is that program usually crashes
          // 0x0000000 indicates null pointer
```

Structures

- Remember, arrays must be of the same type.
- How to deal with keeping track of strings and other types all at once? (ie. data of employees)
  - Use multiple arrays with corresponding index.
  - But this is a little clunky to access corresponding values.

- Want a collection of employees!
- Can introduce *new* types into the language

```
struct Employee // Defines what it means to be an employer.
              // Usually means we will use a lot of Employees.
{
string name; // Called data members (fields, instance variables, attributes).
double salary;
int age;
}; // This type *NEEDS* a semicolon!
 // Without semicolon, compilor will throw an error regarding the next line.
int main()
{
Employee e1; // Has three data members.
             // Empty string, uninitialized double and int.
Employee e2;
e1.name = "Fred";
e1.salary = 60000;
e1.age = 50;

e2.name = "Ethel";

e1.age++; // Can do anything you would do to an int.

cout << "Enter name: ";
getline(cin, e2.name); // Can do anything you would do to a c++ string.

Employee company[100];
company[3].name = "Ricky";

// To print name vertically:
for (int k = 0; k != company[3].name.size(); k++) // Same '.' operator
{
  cout << company[3].name[k] << endl; // company[3]name is a string
}
```

```
}
```

- member function syntax:
  - an object of some member type . the name of a member of that type
- Structures essentially add a new functional type through declaration

```
void printPaycheck(const Employee& e);
void celebrateBirthday(Employee* ep);
double totalPayroll(const Employee eps[], int n);

int main()
{
  Employee company[100];
  int nEmployees = 0;
  // fill some of arrays and set nEmployees

  printPaycheck(company[0]);
  celebrateBirthday(&company[2]);
  cout << totalPayroll(company, nEmployees);

  for (Employee* ep = company; ep < company + nEmployees; ep++) // ep goes forward
    one employer
    cout << ep->name << endl;
}
void printPaycheck(Employee e) // pass by value
{
  cout << "Pay to " << e.name << " the amount $" << e.salary/12 << endl;
  // Passing by value, copies company[0] to e!
  // Could be an issue when copying huge structures
}
void printPaycheck(const Employee& e) // pass by constant reference, e is another
    name for company[0]
                                      // use const keyword, reference to a constant
    employee
                                      // makes it clear we are not modifying employee
{
```

```
  cout << "Pay to " << e.name << " the amount $" << e.salary/12 << endl;
}
void celebrateBirthday(Employee& e) // using reference to change object
{
  e.age++; // Won't compile if paramater is const Employee& e
}
void celebrateBirthday(Employee* ep) // using pointers to change object
{
  (*ep).age++; // NEED parantheses to bypass c++ order of operations
               // dot operator has higher precedence than star or ++ operator
  ep->age++; // or use arrow operator
}
double totalPayroll(const Employee eps[], int n) // array is really a pointer to
   first element
{
  double total = 0;
  for (int k = 0; k < n; k++)
    total += emps[k].salary;
  return total;
}
```

- caller's object should not change:
  - pass by value (cheap to copy)
  - pass by constant reference (not cheap to copy, large structure)
- caller's object should change
  - pass by non-constant reference
  - pass by non-constant pointers
- pointers can be declared constant as well
  - eg. constant types can only be assigned to the corresponding constant pointer type
  - for constant pointers, cannot modify the object being pointed to
  - but, that pointer itself can be modified, eg. replaced with another object's address
- a pointer to an object of some struct type -> the name of a member of that type
  - `p->m` = `(*p).m`
  - can't use `ep.age` or `e->age`

11.21.18

More Structures, Classes, and Abstraction

- "abstraction" - generalizing operations
  - "abstracts" away the intricacies of the actual machine language process
  - just go through the interface, not the implementation (*how* processes are done)
  - eg. * operator for multiplication

Code example:

```
class Target
{
  public: // any part of the program can access these members
  // Member functions AKA operations, methods
    Target(); // constructor, automatically called when object is created
              // no return type, not even void, never const
    void init(); // no longer necessary, use constructors instead
    bool move(char dir);
    int position() const; // const has to go after close parentheses, function promises not to modify anything
    void replayHistory() const;

  // Invariants (constraints, must have valid state):
  //   History consistst only of Rs and Ls
  //   pos == number of Rs in history minus number of Ls in history

  private: // can only be mentioned in the implementations of the member functions
  // Data members AKA fields, attributes, instance variables
    int pos;
    string history; // instead of array of ints or array of characters
};
Target::Target()
{
  pos = 0; // in implementation, if modifying data members, can leave off this->
```

```
        // compiler assumes we are are talking about the object the function was
    called with
        // as long as local variables/parameters of the same name don't exist
  this->history = "";
}
void Target::init() // no longer necessary, use constructors instead
{
  this->pos = 0;
  this->history = "";
}
bool Target::move(char dir) // move() by itself has nothing to do with targets
                            // needs to relate back to the Target class
{
  switch (dir)
  {
    case 'R':
    case 'r': // member function can use the keyboard 'this'
              // is the pointer to target object that called member function
      this->pos++;
      break;
    case 'L':
    case 'l':
      this->pos--;
      break;
    default:
      return false;
  }
  this->history += toupper(dir);
  return true;
}
int Target::position() const
{
  return this->pos;
}
void Target::replayHistory() const
{
```

```
  for (int k = 0; k != this->history.size(); k++) // this->history is a string!
    cout << this->history[k] << endl;
}
// can leave off this-> for all the above implementations!

void repeatMove(Target& x, char dir, int nTimes) // Non-member function! Not part of any type.
{
  for (int k = 0; k < nTimes; k++)
    x.move(dir); // simply calls a public member function of Target
}
void f(const Target& x) // won't compile even though position() doesn't modify x
                        // compiler can't distinguish or check
                        // will compile after position() is declared as a const member function
{
  cout << x.position() << endl;
}
int main()
{
  Target t; // automatically calls constructor
  // t.init(); no longer necessary, use constructors instead
  // t.pos = 0;
  // t.history = "";

  t.move('R'); // move() member function with t object
               // member function is AUTOMATICALLY passed a pointer to t
               // calling move() should retain the target in a valid state
               // throwing away return value because 'R' is known to be valid
  t.replayHistor`y();

  Target t2;
  ...
  t2.move('L');

  char ch;
```

```
    ... read a character into ch
    if (!t2.move())
      ... problem! ...
    /*
    t.pos++; // Nothing stops user from moving position but not recording in history
             // Is there a mechanism for minimizing the possibility of this issue?
    t.history += 'R';
    */
    t.pos = 42; // now it won't compile, private!

    repeatMove(t, 'R', 3); // Non-member function; doesn't need to use structure syntax

    cout << t.pos; // won't compile, can't even "look" at the data member!
    cout << t.position();
}
```

- the name of some struct type :: the name of a member of that type
- steps of abstraction
- the bulk of the program should not be allowed to modify/access position/history
  - eg. can encourage this by having a built-in function that handles valid states for target (move())
  - user has to go through the provided interfaces
  - writer of program can specify permissions for variables, etc.
  - to enforce this, should set up a "wall" with "gates", set up private data members
    - interfaces are the gates (accessible to user), implementations can access data members (inaccessible to user)
  - ie. code cannot directly access data members, but there are certain functions that can be called that do modify data members
  - member functions can also be private (helper functions)
- if user can't access data, how can we first instantiate these objects?
  - Let's try using an init() function
  - how 'bout constructors instead?
- now Targets can never be put in a bad state!
  - except if init() is never called!

Syntax with Structures

Left-Side	Operator	Right-Side
an object of some member type	.	the name of a member of that type
a pointer to some object of some struct type	->	the name of a member of that type
the name of some struct type	::	the name of a member of that type

Constructors

```
void f() // issues before Target had a constructor
{
  ...
  Target tg;
  ...
  tg.move('R'); // bug, window of opportunity between when target is created and then
    initialized
  ...
  tg.init(); // tg isn't in a valid state until here
  ...
  ...
}
```

- close this window of opportunity for bugs
- c++ has an initialization function that is immediately called when object is created called a constructor
  - same name as its type
  - no return type, not even void
  - automatically called when object is created
  - eg. strings have a constructor that creates an empty string
  - constructors cannot be called separately from object creation
  - constructors can be private, but there must be at least one public constructor
    - otherwise results in a compile error when attempting to create an object

Classes

- There is no difference between classes and structs in c++ except:

- struct without explicit public/private declarations assumes by default public
- class without explicit public/private declarations assumes by default private

• by convention:
  - for collection of data, eg. a point type, with no interesting behavior (functions), tend to use struct keyword
  - when adding behavior to types, eg. rotating or translating a point, tend to use class keyword

11.28.18

Classes and Pointers

- Data members should generally be private for two reasons:
  - prevent data from being sent into a bad state
  - gives more freedom to change the implementation
    * otherwise, if the program is modified, program may no longer compile, data members accessed in program may no longer exist/different functionality
  - EXCEPT, if a simple struct that is just a collection of data AND there is no way to set the data to a bad state, probably better to have public data
    * eg. point struct vs. date class

Code example:

```
int main()
{
  Target ta[3]; // calls the constructor three times, sets each of them to a valid state
                // constructor is called for each element of the array
                // what if we don't know how many targets we need? (eg. expensive objects)
  ta[0].move('L');
  ta[1].move('R');
  repeatMove(ta[2], 'L', 3);
}
```

```
void f()
{
  while (...)
    playGame();
}
void playGame() // needs a lot of targets, but not at the beginning
{
  Target* targets[1000]; // cheap declaration and provides uninitialized targets!
  int nTargets = 0;
  ...
  if (...)
    addTargets(targets, nTargets, 3);
  ...
  int i;
  ... something gives i a value, eg. 1
  targets[i]->move('R'); // have to use arrow operator
  ...
  delete targets[1]; // give a pointer to a dynamically allocated object
  // this following proccess is necessary to shift the dangling pointer away
  targets[1] = targets[2]; // have to get rid of dangling pointer
  nTargets--;
  targets[2] = nullptr; // not neccessary, but comforting to some

  // clearing all dynamically allocated memory
  for (int k = 0; k < nTargets; k++)
    delete targets[k];
  // now it's safe to leave the function
} // after playGame() ends, local variables (the array of pointers, ints) go away,
  // but not the storage allocated to targets by the new keyword
  // cannot even refer to these objects anymore; pointer variables have gone out of
   scope
  // on each iteration, we don't get rid of target objects, leads to crash due to
   lack of memory

void addTargets(Target* ta[], int& nt, int howManyMore) // want to update number of
```

```
    targets
{
  for (int k = 0; k < howManyMore; k++)
  {
    /*
     *  Target t;         // incorrect implementation: this creates a local target! (eg. local to main routine)
     *  ta[nt] = &t;      // at the end of the iteration of the loop, after the curly braces, target no longer exists
     *  nt++;             // don't want to point to local variables
     */

    ta[nt] = new Target; // allocates space for a target, call the constructor, and returns a pointer to that object
                         // target has no name, only way to access is through that pointer
                         // storage for the object does not go away, even if that pointer dissapears as a local variable
    nt++;
  }
}

class Person
{
  public:
    Person(string nm, int by); // constructors can take arguments
    string name() const;
  private:
    string m_name;
    int m_birthYear;
}
string Person::name() const // will not compile!!!
{
  return m_name; // does not realize if name is refering to data member or member function
}
```

```
Person::Person(string nm, int by)
{
  // there aren't really reasonable default initial values, so use a constructor with
     parameters
  m_name = mm; // or this->name = name if parameter was named 'name'
  m_birthYear = by;
}
Person p; // compiler writes a constructor that leaves built-in types uninitialized,
    but calls constructors for other classes (eg. string)
Person p("Fred", 1999); // with a different constructor with arguments
```

- Another use for pointers: manipulating dynamic storage
  - eg. only create targets when we need to use them
  - built-in types are not initialized
  - array of targets is expensive, but an array of pointers to targets is very cheap (pointers are a built-in type)
  - use `new` keyword; this is called dynamic allocation
- targets created by the new keyword *DO NOT* go away unless explicitly told to
  - program may crash because there is no more storage, called a memory leak
  - "garbage" are objects that have been allocated, but are no longer accessible
  - issue may not be detected unless program runs for a while
- have to delete objects with delete keyword
  - delete takes a pointer to a dynamically allocated object
  - leaves a dangling pointer not pointing to any valid object, cannot follow that pointer (although it may look like the object is still there)
- naming conventions
  - the same name will often repeat in the data members, member functions, parameters, etc.
  - generally use the most direct name for the public member functions (will be seen/used the most)
  - for data members then, should follow a pattern/convention
    * eg. `name_` or `m_name`
  - parameters are the least visible!
    * sp can have suggestive, but not necessarily 'pretty' name, eg. nm

11.30.18

Classes and Pointers Review

```
int x = 5;
int y = 10;
int z = 15;
int* arr[3];
*arr = &x;
arr[0] = &x;
arr[1] = &y;
arr[2] = &z;

// c++ will write a constructor if there isn't one, simply calls the constructors for
    data members
struct Chair
{
  int height; // default to public
  int legs[4];
  void destroy();
};
class Table
{
  int height; // default to private
  void destroy();
  public:
    Table();  // must be declared public
    ~Table(); // destructor, goes with delete()

    int getheight() const; // won't modify data members, won't call any const member
    functions
  private:
    bool ischanged;
};
Table::Table()
```

```
{
  height = 10;
}
Table::getheight() const
{
  return height;
}

int DontChangeTable(const Table& t)
{
  // can only call const functions
}
Chair c1;     // memory is allocated for data members contiguously, similar to an
   array
              // function is also stored in memory
c1.destroy(); // looks for that function in c1's "array" of memory
Table t1;
t1.destroy(); // won't compile, this function can only be called with other Tables
Table* pt = &t1;
pt->destroy();
(*pt).destroy();
```

Strange Code Interaction

```
#include <iostream>

class Table
{
  private:
    int h;
  public:
    int* hptr;
    Table()
    {
      h = 0;
      hptr = &h;
```

```
    }
};

int main()
{
  Table t;
  std::cout << *(t.hptr) << std::endl;
  (*(t.hptr))++;
  std::cout << *(t.hptr) << std::endl;
}
```

Pointer Warmup

- Take in 2 int pointers and swaps those two ints

```
void swap(int* p1, int* p2)
{
  int temp = *p1;
  *p1 = *p2;
  *p2 = temp;
}
// Swap two pointers
void ptrSwap(int*& p1, int*& p2)
{
  int* temp = p1;
  p1 = p2;
  p2 = temp;
}
```

Dynamic Allocation

- normal static objects have their attributes/sizes known at compile-time (static memory, the stack)
- vs. dynamically allocated objects are just a pointer to an object at compile-time (dynamic memory, the heap)
  - these must be deleted!

  - can't delete statically allocated memory!

12.3.18

Memory & More Classes

- Where different types of variables can lie in memory:
  - local variables (automatic-variables) live on "the stack"
    - automatically go away, scope
  - variables declared oytside of any storage live in the "global storage area" (static storage area)
    - don't automatically go away, live for the life of the program
    - end with main routine
  - dynamically allocated objects live on "the heap"
    - manage using `new` and `delete` keywords

Classes in Classes

```
class Toy
{...
};
class Pet
{
  public:
    Pet(string nm, int initialHealth);
    ~Pet(); // destructor, no return type not even void, no arguments
    void cleanup();
    void addToy();

  private:
    string m_name;
    int m_health;
    Toy* m_favoriteToy; // is an optional feature, can be a nullptr
    // Toy m_favoriteToy; not what is desired, Pet may or may not have a Toy
};
Pet::Pet(string nm, int initialHealth);
{
```

```
  m_name = nm;
  m_health = initialHealth;
  m_favoriteToy = nullptr; // all data members should be in a valid state
}
Pet::~Pet() // automatically called when object is about to go away
            // write destructors whenever cleanup is neccessary, eg. dynamic variables
            // eg. strings have a destructor, remove targets from the display
{
  delete m_favoriteToy; // harmless if delete is passed a nullptr
}
void Pet::cleanup()
{
  delete m_favoriteToy;
}
void Pet::addToy()
{
  delete m_favoriteToy;      // delete old toy, harmless if nullptr
  m_favoriteToy = new Toy(); // must make sure to delete this Toy
}

void f()
{
  Pet p("Frisky", 20);
  p.addToy();
  p.addToy();
  Pet* pp;
  // pp = new Pet; // doesn't work!!!
                   // only generates default constructor if there is no constructor
  pp = new Pet("Fido", 10);
  pp->addToy();
  // pp going away will not call the destructor automatically, it's a pointer
  delete pp;

  // p.cleanup(); // would work, but unpleasant to use
```

```
                  // would have to call cleanup() every possible way we leave the
    function
  // delete p.m_favoriteToy; // WRONG, private data member

  // destructor is automatically called
  // but what if addToy() is called twice?

  Pet p("Fido", 10);
  Pet* pp = new Pet("Fluffy", 20); // 1+ arguments, use parentheses when constructing

  Target t;
  Target* tp = new Target;         // 1 argument, no parentheses
  Target* tp = new Target();
  Target t2();                     // compiles, but doesn't actually create a target
                                   // technically, this is a function declaration
  t2.move('R');
}
```

- if you declare no constructor at all:
  - compiler writes a zero-argument constructor (default-constructor) for you
    - any built-in data types are uninitialized
    - class data types have their default-constructors called (eg. strings default to empty string)
- if you do write a constructor with arguments, there is no zero-argument constructor automatically created

```
Target ta[100];   // default constructor, pos at 0
string sa[100];   // 100 empty strings
Employee ea[100]; // unintialized ints, name is an empty string
// Pet pa[100];   // ??? no default constructor, compiler doesn't write one
                  // cannot declare an array where there is no default constructor
    whatsoever
                  // could define another default constructor, but what would the
    reasonable default values be?
                  // for some types, eg. string, there is a natural default value (
    empty string)
```

```
Pet* ppa[100];    // dynamically allocate new pets instead
```

12.05.18

Classes with Pointers

- eg. for a class representation of a class registrar:
  - better for each class to have array of pointers to students
  - rather than arrays of actual students (who will take multiple classes, expensive)
  - 'has-a' relationship
  - but then how do we find all the courses a single student is taking?
    - need additional pointers the other way, from student to the class
    - 'is-a' relationship
- so, when to have objects within the class, or reference external objects with pointers?
  - directly contain: always there, existence tied to the class
  - pointer: optional, existence independent from the class

Code example:

```
class Fan  // turn on fan of robot when carrying heavy rocks
{ public: void turnOn();
};
class Rock // various rocks in the arena
{ public: double weight() const;
};

class Robot
{
  Fan   m_cooler; // every robot has a fan, fans don't need to be in the game after
    robot is destroyed
  Rock* m_rock;   // rock is not always necessarily associated with the robot (
    optional)
};
void Robot::blah()
```

```
{
  if (m_rock != nullptr && m_rock->weight() >= 50) // arrow operator and check for null
    m_cooler.turnOn(); // dot operator
}
```

More Classes in Classes

```
class Employee
{
  public:
    Employee(string nm, double sal);
    void receiveBonus const;
    // void receiveBonus(double rate) const;
  private:
    string m_name;
    double m_salary;
    Company* m_company;
};
Employee::Employee(string nm, double sal, company* cp)
{
  m_name = nm;
  m_salary = sal;
  m_company = cp;
}
Employee::receiveBonus() const // previously had parameter 'double rate'
{
  // cout << "pay to " << m_name << " $" << rate * m_salary << endl;
  cout << "pay to " << m_name << " $" << m_company->bonusRate() * m_salary << endl;
}

class Company
{
  public:
    company();
    ~company();
```

```
    void hire(string nm, double sal);
    void setBonusRate(double rate);
    void giveBonuses() const;
    double bonusRate() const;
  private:
    Employee* m_employees[100];
    int m_nEmployees;
    double m_bonusRate;
};
Company::Company()
{
  m_employees = 0;
  m_bonusRate = 0;
}
Company::~Company(){
  for (int k = 0; k < m_nEmployees; k++)
    delete m_employees[k];
}
void Company::hire(string nm, double sal)
{
  if (m_nEmployees == 100)
    ERROR
  m_employees[m_nEmployees] = new Employee(nm, sal, this);
  m_nEmployees++;
}
void Company::setBonusRate(double rate)
{
  m_bonusRate = rate;
}
void Company::giveBonuses() const
{
  for (int k = 0; k < m_nEmployees; k++)
    m_employees[k]->receiveBonus(); // previously passed m_bonusRate

}
double Company::bonusRate() const
```

```
{
  return m_bonusRate;
}

int main()
{
  Company myCompany;
  myCompany.hire("Ricky", 80000);
  myCompany.hire("Lucy", 50000);
  myCompany.setBonusRate(.02);
  myCompany.giveBonuses();
  Company yourCompany;
  yourCompany.hire("Fred", 40000);
}
```

- what happens when a company goes away?
- in real world model, employees remain, find another job
- however, this implementation is not representative of the real world
  - thus, make sure to document which elements of reality are accounted for in programs
  - in this case, the employees go away when the company goes away
- when adding new data members, make sure to check constructors and destructors for additional behavior
- different possible implementations of a bonus function
  - company contains the bonus function, must ask for parts of the employee
  - employee contains the bonus function and is passed the bonus rate
  - employee contains a zero parameter bonus function
    - so now has to ask company for its bonus rate, but how does employee know which company to ask?
    - need pointers both ways

Overloading

```
class Complex
{
  public:
    Complex(double re, double im);
    Complex();
```

```
    double real() const;
    double imag() const;
  private:
    double m_rho; // polar, more efficient apparently
    double m_theta;
};
Complex::Complex(double re, double im)
{
  m_rho = sqrt(re*re + im*im);
  m_theta = atan(im, re);
}
Complex::Complex() // can have multple constructors with different number of arguments and/or types
{                   // function overloading works for any functions, not just constructors
  m_rho = 0;
  m_theta = 0;
}
double Complex::real() const
{
  return m_rho * cos(m_theta);
}
double Complex::imag() const
{
  return m_rho * sin(m_theta);

}

int main()
{
  Complex c1(4, -3); // 4-3i
  cout << c1.real(); // writes 4
  Complex ca[100];   // won't compile, no default constructor
}
```

- you can overload a function name if the functions differ in the number or types of parameters

- overloading is not possible in C
- would need functions with different names, eg. `drawRect()`, `drawCirc()`
- vs. in C++, would only need one function named `draw()`

Code example:

```
void draw(Rectangle r);
void draw(Circle c);

int main()
{
  Rectangle a;
  Circle c;

  draw(a);
  draw(b)
}

void f(int i);
void f(double d);
void g(int i, double d);
void g(double d, int i);

int main()
{
  f(3);
  f(3.0);

  // what if there's not an exact match?
  f('A');   // not int or double, but will be treated as an int
  g(1, 2);  // ambigiuous! no best function! compilation error!
}
```