CS 31 Review Sheet

Tau Beta Pi - Boelter 6266

Contents

1 Basics

```
#include <iostream> // no semicolons! provides cout, cin, endl, etc
#include <string>   // provides string methods such as size()
#include <cctype>   // provides islower, isupper, isalpha, isdigit, isalnum, toupper, tolower
#include <cmath>    // provides sin, cos, exp, log, etc

using namespace std;
int main() {
                              // this is a comment
int var;                      // this is a variable
cout << \something" << endl; // print to standard output, terminated by newline
cin >> var;                   // take from standard input, store result in variable var
}

‘\n’ is a newline, ‘\t’ is a tab

Assigning a value
int a = 3; // a is 3

By default, primitives are uninitialized and class objects are initialized via constructor if
we do not give a value.
int a;     // uninitialized
double d;  // uninitialized
char c;    // uninitialized
string s;  // initialized

Identifier Convention
- Letter/number/underscore
- Cannot start with a digit
- Underscore (e.g., sum_lists) or camelCasing (e.g., sumLists)

Keywords in Problems
"At least" is >=, no less than, no fewer than
"At most" is <=, no more than
```

2 Working with Decimals

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2); // 2 digits to the right

If a program fails, return 1 or a nonzero value; 0 if success
Pay attention to = (ASSIGNMENT) vs == (EQUAL)
```

3 Handling Strings and Numbers with stdin/stdout

```
getline(cin, variable);  // discards ‘\n’
cin.ignore(10000, ‘\n’); // use this if read number then next thing will be a string read
                            with getline. cin does not discard ‘\n’
```

4 Operators

```
* / % higher precedence (left to right if same precedence)
+ - lower precedence (left to right if same precedence)

Note: (a / b) * b + (a % b) == b
Careful: 17 / -5 can be -3 or -4, and 17 % -5 can be either 2 or -3, depends on compiler

double x = 3.1 + 14 / 5;
// x is 5.1, since 14/5 is integer division which truncates to 2

In general:
int,int is an int, int,double OR double,int OR double,double is a double
Comparison operators: >, >=, <, <=, !=, ==

|| is OR, && is AND, ! is NOT
```

5 Common Mistakes/Misconceptions

```
Runtime errors
0./0 returns NaN
2.3/0 returns inf or -inf

int a = 10;
int b = a * a;
int c = 25 / (b - 100); // can compile but runtime error

double d;
double e = 2 * d; // uninitialized, weird results in runtime

int f = 1000;
int g = f * f * f;
int h = f * g; // overflow int, since 2s complement, gives negative result
```

6 Constants

```
const int AGE = 30; // by convention, all capitals for constant variables
const double PAY_RATE_THRESHOLD = 12.00;
Cannot change value; if assign another value to the const, you get compile error
```

7 Shorthands

```
n = n + 7; is equivalent to n += 7;
n = n * 2 is equivalent to n *= 2;
n = n / 2 is equivalent to n /= 2;
n += 1 is equivalent to n++ (post increment) or ++n (pre increment)
n -= 1 is equivalent to n-- (post decrement) or --n (pre decrement)
```

8 If and Switch Statements

```
if(someCondition) {
...
}
else if(anotherCondition) {
...
}
else {
...
}

Without curly braces, else goes with the closest preceding if

switch(choice) {
case 1: ... break; // if no break, trickles down to next case!
case 2:
case 4: ... break; // equivalent to if(choice == 2 || choice == 4)
case 3:
case 5: ... break;
default: __________ // break is optional in default case
}

Switch statements only allow short, long, int, bool, or char ONLY. Can only check equality. No
comparisons (<, >, etc). If string/double passed in, compile error
```

9 Short-circuit Evaluation

```
- If || and 1st statement true, skips the rest
- If && and 1st statement false, skips the rest
- Executes left to right
- Evaluate ! before &&, && before ||, but can override precedence with parentheses
```

10 DeMorgan's Laws

```
!(a && b) == !a || !b
!(a || b) == !a && !b
Likewise,
!(a <= b) == a > b
!(a < b) == a >= b
!(a >= b) == a < b
!(a > b) == a <= b
```

11 While, do-while, and For loops

```
// while loop keep executing instructions inside while the condition is true
while(condition) {...}

// do-while loops always execute at least once
do
{Statements}
while(condition); // make sure semicolon!

// for(initialization; test; increment)
for(int x = 0; x < 3; x++) {...}
```

12 Strings

```
string s = \Hello";
s.size();                 // 5
s[0];                     // H
s[-1], s[7], s[5];        // undefined behavior
s.substr(index, length); // s.substr(2, 2) returns ll

To concatenate strings, do s += "some string";
To loop through a string:
for(size_t k = 0; k != s.size(); k++) { ... } // make sure k nonnegative
                                              // note size_t is unsigned int

int k = ‘a’;  // k is 97 since ‘a’ is 97 in ASCII
char c = 97;  // c is ‘a’
c++;          // c is now ‘b’ since 98
‘A’ < ‘B’ < ‘a’ < ‘b’ for ASCII (another format is EBCDIC but we won’t use this here)

Comparisons
string s1 = "hello";
string s2 = "help";
string s3 = "helping";
string s4 = "hElp";
s1 < s2 true since ‘l’ < ‘p’ in terms of ASCII
s2 < s3 true since s2 runs out of characters first
s2 < s4 false since ‘e’ > ‘E’ in ASCII but true in EBCDIC since ‘E’ > ‘e’
```

13 Functions

```
Must be defined in order.
void greet(); //required function prototype so compiler knows greet() exists. We can also
                place the greet() implementation up here as well.
int main() {
greet();
}

void greet() {
cout << \Hello" << endl;
return;                   // legal. However, you cannot return in a constructor
}
```

14 Arrays

```
int var[10];              // initialize a static array of size 10
const int SIZE = 5;
int var2[SIZE];           // initialize a static array of size 5
var2 = {0, 1, 2, 3, 4};   // compile error
int a[];                  // compile error
int k[3] = {1, 2, 3};     // initialize a static array of size 3, containing 1, 2, 3
int m[2] = {1, 2, 3, 4};  // illegal
int c[4] = {1};           // initialize a static array of size 4, containing 1, 0, 0, 0
int a[] = {1, 2, 3};      // legal

THERE IS NO SIZE FUNCTION IN ARRAYS

Passing in arrays in functions
int sum(const int a[], int n) {...} // allowed to pass in regular int[] and const int[] data
void setAll(int a[], int n) {...} // only int[] can be passed; const int[] is compile error

Array sizes must be known in compile time.
int num = 10;
int arr[num]; // not allowed

2D Arrays
int attendance[5][7]; // 5 rows, 7 columns
int table[2][2] = {1, 2}; prints out 1 2
                                     0 0
int table[2][2] = {1, 2, 3, 4} OR prints out 1 2
int table[2][2] = {{1, 2}, {3, 4}            3 4

Passing in 2D arrays in functions
You must define size for 2D+ arrays from 2nd dimension beyond in the parameters:
int calc(char a[][SIZE], int n) {...}
```

15 C-strings

```
#include <cstring>
‘\0’ zero byte to terminate C-string
char t[10] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’};
char t[10] = "hello";  // ‘\0’ is implied, tacked on at t[5]

You cannot assign or concatenate C-strings with regular = or +=. Use these methods:
strlen(t)         // length of C-string, excluding ‘\0’
strcpy(s, t)      // copt string t to string s (strcpy(dest, src)), adjust ‘\0’ accordingly
strcat(s, \!!!")  // concatenate \!!!" to the C-string s and add ‘\0’ to the end
strcmp(s, t)      // if s < t return negative, s == t return 0, s > t return positive
```

16 Data type sizes

```
char, bool: 1 byte
short: 2 bytes
float, unsigned, int: 4 bytes
double, long, long long: 8 bytes
long double = 10 bytes
```

17 Pointers

```
int&  "reference-to-double" or "another-name-for-some-double" (alias)
int*  "pointer-to-double" or "address-of-some-variable"
&x    "generate a pointer to x" or "address of x"
*p    "the object that p points to" or "follow the pointer p"
double a = 3.2;
double* p = &a;
double d = *p;
double& dd = d;
p = &b;          // remove pointer to a and move to b
double d = p;    // bad usage
double* q = 7.6; // bad usage
int k = 10;
p = &k;          // bad usage: no conversion from pointer-to-int to pointer-to-double
nullptr: null pointer (can also be called NULL or set pointer to address 0)

Make sure you initialize all pointers! Uninitialized pointers are disastrous! (runtime error)

*&x == x
&a[i] + j == &a[i + j]
&a[i] < &a[j] == i < j
a == &a[0]
p[i] == *(p + i)
&a[i] - &a[j] == i - j

Traverse an array through pointers
const int MAXSIZE = 5;
double da[MAXSIZE];
for(double* dp = da; dp < da + MAXSIZE; dp++)
*dp = 7.7;

Pointers of objects
Target* t = new Target(10);
cout << t->pos << endl;     // arrow notation is equivalent to (*t).pos
delete t;                   // IMPORTANT! OR YOU GET MEMORY LEAKS!
delete nullptr;             // harmless, but accessing a nullptr is bad

Array pointers
int* arr = new int[26];
delete[] arr;               // this is how you delete a dynamically allocated array

Pointers are also pass by value, but if you dereference a pointer within a function
passed-by-value, you still modify the value stored in that pointer address in memory
```

18 Structs/Classes

```
Structs: by default, member variables/functions public
Classes: by default, member variables/functions private

struct Employee {
string name;
int age;
double salary;
};

Employee e1;
e1.name = "Fred";
e1.age = 47;
e1.salary = 60000;
Employee company[100];

DON'T FORGET THE SEMICOLON!!!

class Target {
public:
Target(int score);
void printScore() {cout << score << endl;}
int position() const;

private:
int score;
int pos;
};

Target::Target(int score) {
score = 0;
pos = 0;
}

int Target::position() const
{return pos;}

You can have multiple constructors in a class/struct!
```