#inlcude <iostream> //no semicolons! Provides cout, cin, endl, etc.

<string> //provides string methods such as size()

<cctype> //provides islower, isupper, isalpha, isdigit, toupper, tolower

<cmath> //provides sin, cos, exp, log, etc

‘\n’ is a newline, ‘\t’ is a tab

int a, double d, char c; //uninitialized string s; //initialized

cout.setf(ios::fixed);

cout.precision(2); //2 digits to the right

getline(cin, variable);

cin.ignore(10000, ‘\n’); //use if read number then next thing will be a string read with getline

int f = 1000;

int g = f\*f\*f;

int h = f\*g; //overflow int, since complement, gives negative result

note: if assign another value to const, you get compile error

**switch**(variable) {

case 1:… break;

case 2:

case 4:… break; //choice == 2 or choice == 4

case 3:

case 5:… break;

default:…; //break is optional in default case

}

!(a && b) == !a || !b string s = “Hello”;

!(a || b) == !a && !b s.size(); //5

!(a <= b) == a > b s[0]; //H

!(a < b) == a >= b s[-1], s[7], s[5] //undefined behavior

!(a >= b) == a < b for(size\_t k = 0; k != s.size(); k++)

!(a > b) == a <= b //to loop through a string

int k = ‘a’; //k is 97 since ‘a’ is 97 in ASCII int x = ‘G’; //x == 71

char c = 97; //c is ‘a’ x -= 1; //x == 70

c++; //c is now ‘b’ since 98 char y = x; //y == ‘F’

‘A’ < ‘B’ < ‘a’ < ‘b’ for ASCII int z = ‘5’; //z == 53

**string comparisons**

string s1 = “hello”; s1 < s2 true since ‘l’ < ‘p’

string s2 = “help”; s2 < s3 true since s2 runs out of characters first

string s3 = “helping”; s2 < s4 false since ‘e’ > ‘E’ in ASCII

string s4 = “hElp”;

array sizes must be known in compile time

int num = 10;

int arr[num]; //not allowed

**2D Arrays**  **passing 2D arrays in functions**

int attendance[5][7]; //5 rows, 7 columns must define size for 2D+ arrays from 2nd

int table[2][2] = {1,2}; //prints out 1 2 dimension beyond in parameters

0 0 int calc(char a[][SIZE], int n){…}

int table[2][2] = {1,2,3,4}; //prints out 1 2

int table[2][2] = {{1,2}, {3,4}}; 3 4 data type sizes

char, bool: 1 byte

**cstrings**  #include <cstring> short: 2 bytes

‘\0’ zero byte to terminate cstring float, unsigned int: 4 bytes

char t[10] = {‘h’,’e’,’l’,’l’,’o’}; double, long, long long: 8 bytes

char t[10] = “hello”; //’\0’ is implied, tacked on at t[5] long double: 10 bytes

strlen(t); //length of cstring excluding ‘\0’

strcpy(s,t); //copy string t to string s

strcat(s,“!!!”); //concatenate “!!!” to the cstring s and all ‘\0’ to the end

strcmp(s,t) //if s < t return negative, s == t return 0, s > t return positive

for(int i = 0; arr[i] != ‘\0’; i++) //loop to iterate through cstring

void removeNonAlpha(char str[]) {

for (int i = 0; str[i] != ‘\0’; i++)

while(!isalpha(str[i]) && str[i] != ‘\0’)

for (int j = i; str[j] != ‘\0’; j++)

str[j] = str[j + 1]

}

**pointers ☹ == memory address of a variable**

& operator gets address of a variable

\*operator gets value stored in variable that is being pointed to

double d = 10.0;

double \*dp; //pointer that holds the address of a double

dp = &d; //stores the address of d into dp

\*dp = 20.0; //changes the value of d from 10.0 to 20.0

**new operator used to create dynamic variables, accessed by pointers**

string \*p;

p = new string(“hello”);

**delete operator eliminates dynamic variables**

delete p; //p is now a dangling pointer, dereferencing is dangeraous and leads to undefined behavior, avoid by setting p to NULL after using delete

**dynamic array, useful bc size can be determined while program is running**

int \*ptr;

int arraySize;

cin >> arraySize;

ptr = new int[arraySize];

delete[] ptr; //destroy dynamically allocated array

**array traversal with pointers**

int sum(int \*head, int n) {

int total = 0;

for(int i = 0; i < n; i++)

total += \*(head + 1);

return total;

}

**cstring reversal with pointers**

void reverse(const char s[]) {

const char \*p = s; //create a new pointer for our traversal

while (\*p != ‘\0’) { //move the pointer to the end of the cstring

p++;

}

p--; //set p to point at the last char in the cstring

while (p >= s) {

cout << \*p;

p--;

}

cout << endl;

}

**structs: collection of data that is treated as its own special data type**

struct Person{

int age;

string name;

}; //must end with semicolon

struct Circle {

int radius;

} ring, hoop; //creates two Circle structs named ring and hoop

**classes: like a struct but with member function as well as member variables**

**make mutator and accessor functions public and member variables private**

class Person {

public: //member variables and member functions are private by default

void setAge(int yrs);

int getAge();

void setName(string nm);

string getName();

void doubleMoney() { money \*= 2; };

double getMoney();

private:

int age;

string name;

double money;

};

Person::doubleMoney() { //outside of the class definition

money \*= 2;

}

void Person::setAge(int yrs) {

if (yrs >= 0)

age = yrs;

}

int Person::getAge() { return age; }

void Person::setName(string nm) {

if (nm.length > 0)

name = nm;

}

string Person::getName() { return name; }

void Person::doubleMoney() { money \*= 2; }

double Person::getMoney() { return money; }

**constructor: member function that has the same name as the class, no return type, and automatically performs initialization when we declare an object:**

* A default constructor is one with no arguments; the compiler generates an empty one by default.
* The “default default” constructor leaves primitive member variables (int, double, etc.) uninitialized and calls the default constructors of class members

class Person {

public:

Person();

// Same stuff as before!

};

Person::Person() {

age = 0;

name = “”;

money = 0.0;

}

**initializer lists are required to initialize const and reference type member variables. They are also preferred for class member variables (like name); otherwise, the default constructor for that class is called.**

**Because of this, they are required for classes with no default constructor.**

Person::Person()

: age(0), name(""), money(0.0) {}

Person::Person(int yrs, string nm, double cash)

:age(yrs), name(nm), money(cash) {}

**destructor: member function that is called automatically when an object of a class passes out of scope**

* **uses delelte to eliminate any dynamically allocated variables created by the object**

Person::Person() {

// Same stuff as before ...

fluffy = new Pet(“Steve”);

}

Person::~Person() {

delete fluffy;

}

**friend functions: grant special access to a function not owned by a class, to allow that function access to the class’s private data members**

class Y {

int dataInY;

friend std::ostream& operator<<(std::ostream& out, const Y& o);

// This doesn’t declare a member function! It just marks a function with this specific

// header as being a “friend” of the Y class.

};

std::ostream& operator<<(std::ostream& out, const Y& y) {

return out << y.dataInY;

}

**friend classes**

class Storage {

int nValue;

friend class Display; // Now all members of Display can access private members of Storage.

};

class Display {

…

void displayItem(Storage &storage) {

std::cout << storage.nValue << ‘\n’;

}

};

**copy constructor: constructor that takes one parameter that is of the same type as the class**

Person::Person(const Person &other) {

age = other.age;

name = other.name;

money = other.money;

fluffy = new Pet; // If we just had ‘fluffy = other.fluffy’

\*fluffy = \*(other.fluffy); // these two people would share the same

// Pet object in memory!

}

**assignment operator: called when an already initialized object is assigned to new value from another existing onject, returns the object on the left side of the = sign**

Person p1(10, “Jim”, 10.0);

Person p2(15, “Tim”, 50.0);

p1 = p2; // assignment operator called

Let’s define an assignment operator for our Person class!

Person& Person::operator=(const Person& rtSide) {

if (this == &rtSide) // The left side and the right side of the = sign are the same!

return \*this;

else {

age = rtSide.age;

name = rtSide.name;

money = rtSide.money;

delete fluffy; // Delete the old instance of Pet before creating a new one.

fluffy = new Pet;

\*fluffy = \*(rtSide.fluffy);

return \*this; // Return the calling object (left side of the = sign).

}

}

**arrow operator -> used to access an object’s member variables and functions, dynamically allocated, used when we have a pointer to the object whose members we are trying to reference**

int main() {

Person\* p = new Person;

p->setAge(20);

p->setName(“Bob”);

double money = p->getMoney();

p->age = 10; // ERROR: age is not a public variable!

}

Note: p->setAge(20) and (\*p).setAge(20) are equivalent statements!

**this pointer: predefined pointer that points to the calling object**

two below are same but top method is clear and better stylistically

int Person::getAge() {

return age;

}

int Person::getAge() {

return this->age;

}

**function overloading: multiple definitions for the same function name in the same scope, definition of the functions must differ from each other by the types and/or number of arguments in the argument list**

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(int i, double f) {

cout << "Printing numbers: " << f << ‘ ‘ << i << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print numbers

pd.print(42, 500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

**operator overloading: overload basic operators in C++ (such as +, -, <<, etc.) so they work with user-defined structs and classes as well**

class Vector {

public:

double getX() {return m\_x;}

double getY() {return m\_y;}

void setX(int x) {m\_x = x;}

void setY(int y) {m\_y = y;}

}

Vector operator+(const Vector& vec) {

Vector newVec;

newVec.setX(this->m\_x + vec.getX());

newVec.setY(this->m\_y + vec.getY());

return newVec;

}

private:

double m\_x; // x component of vector

double m\_y; // y component of vector

};