**#include <iostream>**

using namespace std;

int main()

{

int a,b,c;

}

double a = n/**static\_cast<double>(b);**

**#include <string>**

string a = “eefa”;

string b = “”;

k = a.size(); //a.size() == a.length()

cout << a[k-1] <<endl;

**s.substr(position, length)**

**s.empty()** //Returns true if str is an empty string; returns false otherwise

**s.insert(pos, str2)** Inserts str2 into str beginning at position pos.

**s.remove(pos, length)**Removes substring of size length, starting at position pos.

**char** c = a[k-2];

**‘\\’ ,’\’’,’\”’**

b += a; //**append concatenate**

**cout.setf(ios::fixed);**

cout.precision(2);

amount \*=a+b 🡺 amount = amount \* (a+b)

**bool** (0-false; !0-true)

cout << “\n” <<endl;

cin >> a;

**cin.ignore(1000,’\n’);**

**getline(cin,a);**

**||, &&, !**

**if** (“a”**==**”b”)

{}

else if (a**!=**c)

{}

else

{}

if (t[k] == ‘e’ || **‘E’**)//will compile, a character in true or false, all the normal character have a non-zero starting code, so they are true.

if (t[2] **=** 0) //will compile, n => false

**switch (**choice**)** //only an integer or a character, constant

{

case 1: //constant label

…

break;

case 2:

case 3:

case 4:

…

break;

default:

…

}

int n = 0; //out of the loop

**while** (condition)

{

…

n++;

}

**do**

{

…

}

while (condition)**;**

**for (**initialization(1);condition(2);prep-for-next(4)**)**

stmt(3);

order: 1234 234 234

**void** greet(int n**,** string m);

**int** square(int k);

eg:

int main()

{

…

greet(3, “hello”);

int a = square(6);

}

void greet(int nTimes, string msg)

// it you put it here with out the function prototype above, this won’t compile.

{

if (nTimes < 0)

{

cout <<“do you want to greet me?” <<endl;

return; // we are done with the greet function, return back to where it is called.

}

for (int k = 1; k <= nTimes; k++)

cout << mag <<endl;

}

int square(int k)

{

return k\*k;

//return the value to where it is called

}

**bool** isValidPhoneNumber (string pn)

{

int numberOfDigits = 0

for (int k = 0; k != pn.size(); k++)

{

if (isdigit(pn[k]))

numberOfDigits++;

}

return (numberOfDigits == 10);

**#include <cctype>**

isdigit(), islower(), isupper(), isalpha(), isalnum(),

isspace(), ispunct(), etc.

t[0] = **toupper**(t[0]);

//only as expression does it effect t[0]

//only works for char

**#include <cmath>**

sin(), cos(), exp(), log(), sqrt() ...

**double&**

means reference-to-double or another-name-for-some-double

// this refers to an existing double

eg:

int square (int k); //better than below

void squareTheNumber (int& j);

**break;**

// abandon the current and remaining iterations of the loop

**continue;**

// abandon the rest of current iteration; goes on to the next iteration

**ARRAY**

int daysInMonth[Month] = {

31, 28, 31, 30, 31, 30,

31, 31, 30, 31, 30, 31 } ;

// the number of values in the array has to be known when the program is compiled

//so it can’t be a number you read in.

int attendance**[][NDAYS]**;

//any array’s length must be known at compile time except the first dimension

// in two-dimensional array, when you say a[k], it means the entire k row

// “two-dimensional array"

const string dayNames[NDAYS] = {

“Monday”, “Tuesday”, …, “Saturday”, “Sunday"

};

for (int d = 0; d < NDAYS; d++)

{

int total = 0;

for (int w = 0; w < NWEEKS; w++)

total += attendance[w][d];

cout << “The total for “ << dayNames[d] << “is” << total << endl;

grandTotal += total;

}

cout << “Over the course of “ << NWEEKS << “weeks, weekend attendance was “ <grandTotal << endl;

double computeMean**Function**(int a[], int n)

//big array can be expensive to copy, so if there is no &, it is not a copy

// a is only an indication of where it starts, it is not an array

// you have to trust the user not to input a number bigger than the array

**WARNING:**

… daysInMonth.**size()** … //NO! not in c++

int **a2[]** = { 30, 20, 30 };

//compiler will count initializers, so the array has 3 elements

int a3[3]; // 3 uninitialized integer values

int a4[]; // Error!! won’t compile

**comparison**

nt k =97;

char c = ‘a’;

int k2 = ‘a’; //if ASCII is the encoding, this is 97; if EBCDIC, 129

char c2 = 97; // if ASCII is the encoding, this is ‘a’; ifEBCDIC, ‘/'

k++; //Now 98

c++; // if ASCII is the encoding, was 97 now this is 98; in ASCII, this is ‘b’;

double x = 3.5;

cout << x; //calls the function for doubles; writes ‘3’ ‘.’ ‘5'

//if ASCII, this is 51 46 53.

cout << k; // calls the function for ints; writes ‘9’ ‘8'

//if ASCII, this is 57 56

cout << c; // calls the function for char; writes ‘b'

//if ASCII, this is 98

code for ‘ ‘ is less than the code for any printable character

code for ‘A’ is less than code for ‘B’; ‘B’ is less than ‘C’; …. ‘Z'

code for ‘a’ is less than code for ‘b’; ‘b’ is less than ‘c’; …. ‘z'

code for ‘0’ is one less than the code for ‘1’; ‘1’ is one less than ‘2’; … ‘9'

**#include <cstring>**

char csa[5][MAX\_NAME\_LENGTH**+1**] = {“cat”, “ferret”, “mouse” , “rat” , “horse" };

//the longest number of interesting character plus one for the zero byte.

char t[10]= {‘H’,’e’,’l’,’l’,’o’,’10’,…};

char s[100];

**/0** you don’t look at those beyond the zero byte, they will all be considered as zero.

empty string, just a zero byte

for(int k = 0**; t[k] != ‘\0'**; k++)

// stay in the loop until you check all the character in the string

cout << t[k] << endl;

cout << t;

cin.**getline**(s, **the position of \0 in the string);**

cin.getline(s, 100);

cout << **strlen**(t);

// count the position number of the \0 in the string //the number of character in the string

for (int k = 0; k != strlen(t); k++)

cout << t[k]<<endl;

**strcpy**(s, t);

**// strcpy(destination source string, string);**

char ca[4];

strcpy(ca,t) //undefined

strcpy(ca,"abc")

strcpy(ca,"abcd") //undefined, no \0

**strcat**(s,"!!!")// attach to the end

If(strcmp(t,s) < 0)

//compare the value of x and y

**strcmp(x,y)**

a negative value if x comes before y

0 if equal

a positive value if x comes after y

**Pointers**

**double\*** means     pointer-to-double or address-of-some-double

**double&**means     reference-to-double or another-name-for-some-double

**&x** means     “generate a pointer to x” or “address of x"

**\*p** means     “the object that p points to” or “follow the pointer p"

double\* p = &a;

//declare p as a pointer to a double, then generate a pointer “a” referring to the address of a

double\* q = 7.6;

//ERROR! assign a double to a pointer

double d = p;     //Error! d is a variable that can hold a double, but p is a pointer to a double

double& dd = d;

//dd is of type reference that is another name for already existing double d

//a pointer can be pointed to other things later

//Error, but a reference is always tied to the variable it is named for

p = &k;     //Error! not allowed, you can’t assign an int into a pointer to a double

\*q = 4.3;     //undefined behavior, q is uninitialized, run time error!

if (p == r) //false     comparing pointers means comparing the address

//&a[i]+j => &a[i+j]

//a<==>&a[0]

//add to the kind of type the pointer point to

//&a[i]<&[j] => i<j

//go forward a double, which has 8 bytes in memory

//&a[i] = a + i

//p[i] ==> \*(p+i)

// \*&x => x

//you are allow to generate the address of &da[5], but you can’t assign it.

if a is a pointer to sa

**a[k]; \*(a+k); \*(&sa[0]+k) \*(&sa[k]) sa[k]**

double\* p = **nullptr**;

double\* q;

if(q == nullptr);     // undefined behavior!!

\*p =        //undefined!!

p[i]     //undefined!!

// an object of some struct type .  the name of a member of that type

**(\*ep).age++;**      ‘.’ has higher priority than \*

**//ep->age++;**// a pointer to an object of some struct type -> the name of a member of that type

struct Target

//in struct, if you don’t say if the members are public or private, they are all public

//in class, if you don’t say if the members are public or private, they are all privat

//member functions:

     Target();   //constructor

                 //not allowed to put a return type

                 //automatically called when you declare a variable of this type

                 //it put the object into initial(well defined) state

     int position() const; //this function does not modify the object that you call on

                           //constant member function

void Target::int()     //the name of some struct type :: the name of a member of that type

{

     pos = 0;

     history = “”;

}

// if you are in a member function, then pos is in the member function that is a name of the target type

     cout << “There’s a target at position “ << x.position()<<endl;

//this does not change the value of pos, but Wont COMPILE!!

//compiler don’t know if it modifies the target or not, unless the function is concatenated by const

int main()

{

     pos++;    //won’t compile!!, it is private only for target type

}

class Target     //you are free to change the implementation as long as the visible interface is the same

                    //dont have to worry about breaking anybody else’s code in a group work

void playGame()

{

     Target**\*** targets[100];     //array of pointers

     int nTargets = 0;

     ...

     if (...)

          addTargets(targets, nTargets, 3);     //dont have to create the target object that you don’t need

     ...

     int i;

     … i get some value in the range 0 to nTarget-1 (e.g 1)

     target[i]->move(‘R’);  //call the member function on the ith Target.

     ...

     delete targets[1];    //delete the object pointed to here that is given by “new”, that has not been deleted

                           //give the object that the pointer is pointing to, back to the operating system

     count << \*target[1]     //undefined behavior

     delete target[1];        //undefined behavior!! You can’t delete it again since there is no new in it

     //target[1] is called a dangling pointer, that is a pointer that points to a object that you deleted, but it, still point to that part of memory, we can’t count on it to have a decent value

     targets[1] = nullptr;   //unnecessary, but confirming

          ta[nt] = new Target;     //dynamic storage allocation, a new target object, return a pointer to it

//. is when you have an object on the left;

//-> is when you have a pointer on the left

named **local variable** (“automatic variables")

 live on “the stack" //go away automatically

**variables declared outside of any function** live in the “global storage area” (“the static storage area") //go away when the program ends

**dynamic storage** lives on “the heap" //it doesn’t go away unless you ask it to.

class Pet

{

     public:

         Pet(string nm, int initialHealth);

         Pet();//you are allowed to have different function with the same name as long as they have different

// this is a constructor that takes no argument: "default constructor/ zero-argument constructor"

          ~Pet();    //destructor: automatically called when we hit the end of the function

          Toy\* n\_favoriteToy;      //use pointer when you may or may not have a favorite toy

Pet::~Pet()     //automatically called whenever an object is about to go away

{

     delete m\_favoriteToy;     //if you call delete and pass a nullptr, it does nothing

}

     Pet p2; // Error! won’t compile! no default constructor for it

//if you don’t declare any constructor at all, the compiler will write a default constructor for you, which leaves any data member uninitialized, and any object of a class type, it calls that class default constructor.

//if you declare a constructor that is not a default constructor, the compiler won’t declare a default constructor for you

//a nullptr is a testable pointer, not initialized pointer

myCompany.hire(); this == myCompany

this: a key word to an object that you call the function on

class Company;//incomplete type declaration. to make sure the circular relation between

              //Company and Employee

class Employee

{

          Employee(Company\* cp, string nm, double sal, int ag);

};

class Company

{

...

};

     m\_employees[nEmployees] = new Employee(this, nm, sal, ag);

               //how to pass parameters to a constructor when you dynamically allocate something with new.

               //type of thing you are creating + pass all the constructor parameters in the parenthesis

void Company::setBonusRate(double pct)

{

     m\_bonusRate = pct;     //error, checking omitted

}

struct B;

struct A    //size of A = 4 + size of B = size of A +4   //infinite object, won’t compile!

{           //size of B = size of A

     B b;

};

struct B

{

     A a;

};

Target t;    //calls the default constructor;

Taget u();  // does not declare a Target object. u is a function that takes no argument and return a Target

//if you don’t declare any constructors for a class at all,

//the compiler will write a default constructor for you

//that leaves members of a builtin type uninitialized,

//and default-constructs members of a class type.

     if(m\_rock != nullptr && m\_rock->weight() > 50)     //make sure you test with the right order

int main()

{

     Complex z;

     ...

     Complex z = x + y     //this is equal to call the function operator+ with parameter x and y

}

Complex **operator+(**Complex a, Complex b)

{

     Complex result(a.real() + b.real(), a.imag() + b.imag());

     return result;

}

//you can’t overload existing operation type

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

bool operator==(Complex a, Complex b)

{

     return (a.real() == b.real() && a.imag() == b.imag());

}

bool operator!=(Complex a, Complex b)

{

     return !(a == b);     //inverting the result of the corresponding operation

}

     draw(c1);     //the right function will get called based on the type of the argument

void f(double d);          //the int 3 here can be converted automatically to a double type

void f(stirng s);

void f(char ch);

void f(int i);

void g(int i, double d);

void g(double d, int i);

int main()

{

     f(3);

     g(10, 20);    //ambiguous!!

}