```
#include <iostream>
#include <string> //provides definitions for string header
using namespace std;

        const double HOURLY_RATE_THRESHOLD = 12.00; //all constants must be initialized WITH A VALUE ASSIGNED
        cout.setf(ios::fixed);
        cout.precision(2);

int main()
{
        string personsName;
        string quest;
        int age;

        cout << "What is your name? ";
        getline(cin, personsName); //instead of cin >> personsName;, which only gives the first word of consecutive chars (minus blanks)

        cout << "How old are you? ";
        cin >> age;
        cin.ignore(10000, '\n'); //get rid of all new lines so getline is not satisfied immediately without user input
        //used almost only when you've read in a number, and the next operation is getline

        cout << "What is your quest? ";
        getline(cin, quest);
}


to read a number x from the input: cin >> x;
to read a string s from the input: getline(cin, s);
/       flash, forward slash
\       backslash

ints, doubles, floats
identifiers - variable names
case-sensitive
"code paragraphs"
comments

math in c++
        similarity - order of operations
        difference - no implied multiplication, i.e. (1+2)(3+4)

truncation
        int/int
        int a = 5.0/2.0

undefined
        dividing by zero
        overflow

outputting value with specified number of digits after decimal
        cout.setf(ios::fixed);
        cout.precision(2);


"string" type
getline()
pitfall of mixing cin and getline - cin doesn't consume trailing newline

cin.ignore(10000, '\n');

if-else statements

int main()
{
        string whee = "Wheeeeeeeeeeeeeeeeeeeeoooooeoooeoeoeoeoeoeoeoeeeeeeeeeeeeee!";
        int eCounter = 0;
        int ind=0;

        while(ind <= whee.length()-1)
        {
                if(whee[ind]=='E' || whee[ind]=='e') //same as: if(whee.at(ind)=='e')
                {
                        eCounter++;
                }
                ind++;
        }

        cout << "There are " << eCounter << " e's." << endl;
}
```

```
statement blocks
{
        everything within here is a statement block
        variables instantiated in this block with same name
        as one outside used if referred to (concept of scope)
}

//double:
//       +/- about 10 to the -308 to 10 to the 308
//       about 15 significant digits
//       anything with a dot '.' or 'e'

//int:
//       about -2 billion to 2 billion (historically, ints work must
faster; now, it affects readability; easier or makes more sense)

//identifier:
//       letter: letter/digit/underscore
//       hoursworked
//       hours_worked
//       hoursWorked ("camel case")
//       HoursWorked

//declaration:
//       type variable; (i.e., double doubleNameHere)
//       type variable = expression;

//arithmetic expressions: * / + -
//       (3+4)*(7-2) = 7*5 = 35
//       3+4*5 = 23
//       14.3/5.0 = 2.86
//       14.3/5 = 2.86
//       14/5.0 = 2.8
//       14/5 = 2 (integer result, since both operands are ints)
```

```
        int a = 10;
        int b = a*a;
        int c = 25/(b-100);

        double d;
        double e = d*d;
        cout << e;

        int f = 1000;
        int g = f*f*f;
        int h = f*g;

        //undefined behavior!


{ stmt; stmt; stmt; }

compound statement
block
```

```
void makeUpperCase(string& s)
{
	for(int k=0; k!=s.size(); k++)
	{
		s[k]=toupper(s[k]);
	}
}
#include <iostream>
#include <string>
#include <cctype>
using namespace std;

bool isValidPhoneNumber(string pn);
string cleanNumber(string pn);

int main()
{
	cout << "Enter a phone number: ";
	string phone;
	getline(cin, phone);
	if(isValidPhoneNumber(phone))
		cout << "The digits in the number are " << cleanNumber(phone) << endl; //String is copied: "passed by value"
	else
		cout << "A phone number must contain 10 digits." << endl;
}

bool isValidPhoneNumber(string pn)
{
	int numberOfDigits=0;
	for(int k=0; k!=pn.size(); k++)
	{
		if(isdigit(pn[k]))
			numberOfDigits++;
	}

	return numberOfDigits==10;
}

string cleanNumber(string pn)
{
	string phone="";

	for(int k=0; k!=pn.size(); k++)
	{
		if(isdigit(pn[k]))
			phone += pn[k]; //concatenation (combine)
	}

	return phone;
}
```

```
#include <iostream>
#include <string>

using namespace std;

void flip (string& s)
{
	if(s.size()==0)
		return;
	int b=0;
	int e=s.size()-1;

	while(b!=e && b!=e-1)
	{
		char ch=s.at(b);
		s.at(b)=s.at(e);
		s.at(e)=ch;

		b++;
		e--;
	}
}
```

```
char c = 'A';
int k = 65;
char c2 = 65; //If ASCII is the encoding, this is 'A'
int k2 = 'A' //If ASCII is the encoding, this is 65
k++; //k is now 66
c++; //c is now 66; if ASCII is the encoding, this is 'B'
char d = '9'; //If ASCII is the encording, this is 57
char e = 9; //If ASCII is the encoding, this is '/t'
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 '6' '6'
                              //If ASCII, this is 54      54
cout << c; //calls the function for chars; writes 'B'
                              //If ASCII, this is 66
         code for ' 'is less than the code for any printable character
         code for 'A' is less than the code for 'B', 'B' is less than 'C', ...'Z'
         code for 'a' is less than the code for 'b', 'b' is less than 'c', ...'z'
         code for '0' is one less than code for '1', '1' is one less than '2', ...
We CANNOT assume that the codes for alphabetal letters are consecutive; this is only true in ASCII
(a is 1 less than b, which is 1 less than c, etc.)
```

```
"off-by-one-error" - screwing up a loop because the
index is wrong by 1 (OR "fencepost error")

i.e.:

int nTimes;
cin >> nTimes;

int n=0;
while(n<=nTimes)
{
	cout << "hello" << endl;
}

int n=1;

while (n<10)
;	//This program will keep running, since
the semi-colon counts as a "do nothing" under the
while loop; n is never incremented
{
	cout << "Hello" << endl;
	n++;
}
```

```
int longestRun(int a[], int n, int& value)
{
	int lastStreak=1;
	int maxStreak=1;
	int index=0;
	int lastVal = a[0];
	value = a[0];
	while (index<n-1)
	{
		lastStreak=1;
		lastVal = a[index];

		while(a[index]==a[index+1])
		{
			index++;
			lastStreak++;
		}
		if(lastStreak>maxStreak)
		{
			maxStreak=lastStreak;
			value=lastVal;
		}
		index++;
	}
	return maxStreak;
}
```

```
C strings

char s[100] = ""; //initiates an array with just a "zero-
byte"
char t[9] = {'H', 'e', 'l', 'l', 'o', '\0'}; //ends with
a zero-byte
	//ALTERNATIVE:
	char t[9] = "Hello"; //also adds a zero-byte

cout << t; //prints up-to, but NOT including, the zero-
byte: Hello
cin.getline(s, 100);

t[0] = 'J';

To find the string size in the array:

int strlen(const char a[])
{
	int k;
	for(k=0; a[k]!= '\0'; k++)
		;
	return k;
}
```

```
#include <cstring> //INCLUDES the strlen function!
s = t;   //Error! Won't compile!
strcpy(s, t); // strcpy(destination, source);
strcpy(t, "asdfjlj jalksdjflasjdflajsdlfjasldfjlsdjf"); /causes a problem!

If at any time, we try to access t[9], we have caused undefined behavior.
```

```
int main()
{
	int data[15] = {5, 8, 8, 2, 7, 7,
7, 7, 8, 8, 8, 3, 3, 3, 3};

	int v;
	int len = longestRun(data, 15, v);
	len = longestRun(data, 5, v);
	len = longestRun(data, 2, v)
}
```

We don't have references in C language!

Pointers:

Another way to implement passing by reference
Traverse arrays
Manipulate dynamic storage (major reason for pointers)
Represent relationships in data structures

Declarations:

double-->double
double&-->reference to double (name for an already-existing double)
double*-->pointer to double (arrow pointing to double; address of some double)

Expressions:

&x-->generate a pointer to x (address of x)
*P -->(P some pointer) follow pointer p (object that P points to) "dereference p"

```
double a = 3.2;
double b = 5.1;
double* p = &a; //&a generates a pointer to a
        double* q = b; //Error! b is a double, not a pointer
a[   3.2   ]
b[   5.1   ]
p[]
double c = a; //Simply initializes c as a double with the same value as a
        double d = p; //Error! Different types, since p is initiated as a pointer
c[   3.2   ]
double d = *p; //Initializes double d to the value of a
double& dd = d; //Legal, but stupid. Declaring dd to be another name for d.
                        //Bad idea to use two names for one object in the same scope.
d[   5.2   ]dd
        p=b; //Error! p is a pointer to a double, but b is a double itself
p = &b; //Retargets p so that it points to b (and no longer a)
        OR
*p = b; //Assigns the p's double value that b holds. Changes a's value to b's value
*p+=4; //same as *p = *p + 4;
int k = 2;
        p = &k; //Error! &k is a pointer to an int
int* z = &k; //z declared as a pointer to ints, so works for k, but not a, b, c, or d
cout << (k*b); //Writes 10.2
cout << (k*p); //Error! Can't multiply an int by a pointer
cout << (k**p); //Works! Better to write as (k * *p);
cout << (*z**p); //Works! Better to write as (*z * *p);
```

```
double w = 3.2;
double x = 5.1;
double *p = &w;
        //p = 8.5; Error! Won't compile
*p = 8.5;
        //*p=&x; Error! won't compile
p = &x;
cout << *p; //writes 5.1
cout << w; //writes 8.5
double *q; //q is an unitialized pointer
        *q = 4.6; //undefined behavior
q = p;
double * r = &w;
*r = x;
if(p==r) //false --> comparing pointers
if(p==q) //true --> comparing pointers
if(*p == *r) //true --> comparing values referred to be pointers
```

```
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++)
        //Under certain circumstances, we can add an integer to a pointer
        *dp = 3.6;   //gives the same results: *dp=3.6, *&da[0]=3.6, da[0]=3.6

*&x ==> x
&a[i] + j ==> &a[i+j]    &a[i] - j ==> &a[i-j]
&a[i] < &a[j] ==> i < j                                 <=, >, >=
        //gives the same results: dp++, dp=dp+1, dp=&da[0]+1, dp=&da[0+1], dp=&da[1]
        //pointers can be assigned out-of-bounds in an array, but can't be addressed (?)
a <==> &a[0]
p[i] ==> *(p+i)
&a[i]-&a[j] ==> i-j

double a[]
double* a   //mean the same thing AS FUNCTION PARAMETERS
```

```
double* findFirstNegative(double a[], int n)
{
        for(double* dp=a; dp<a+n; dp++)
        //won't compile if parameter a is passed as const double array
        {
                if(*dp<0)
                        return dp;
        }
        return 0;    //alternatives: return a+n; OR return &a[0]+n; OR return NULL;
}

//null pointer is NOT an uninitialized pointer; it is a well-defined value
//ways to call the null pointer:
//      -use the integer constant 0 in a context where a pointer is required
//      -use NULL (return NULL or check for NULL in a pointer)
//      -Only in C++11: nullptr (

int main()
{       double da[5];
        ...//fill the array
        double* fnp = findFirstNegative(da, 5);

        if(fnp==0) //OR if(fnp==NULL)
                //Check if fnp is the null pointer?
                cout << "There are no negative values in the array" << endl;
        else
        {
                cout << "The first negative value is " << *fnp << endl;
                cout << "It's at element number " << fnp-da << endl;
        }       }
```

```
int findFirstNegative(const double a[], int n)
{       //same as    for(double* dp=&a[0]; dp<&a[0+n]; dp++)
        for(double* dp=a; dp<a+n; dp++)
            if(*dp<0)
            return dp-a; //&da[2]-&da[0] will give 2, the answer that we want; or dp-&a[0]
        return -1;   }

int main()
{       double da[5];
        ...//fill the array
        int pos = findFirstNegative(da, 5);
        if(pos==-1)
                cout << "There are no negative values in the array" << endl;
        else
        {
                cout << "The first negative value is " << da[pos] << endl;
                cout << "It's at element number " << Pos << endl;
        }       }
```

```
void f()
{       int n;
        cin>>n;
        double* a = new double[n]; //works during execution!
        //only way to give this memory back to OS is to call another function
        //even if pointer a goes away, any dynamically-called array is still there
(although it can't be called)
        a[0] = 10.3;
        a[1] = 4.7;
        delete [] a; //gives the memory back to the OS }
----------------------------------------------------------------
named local variables ("automatic variables") live on "the stack"
dynamic storage lives on "the heap"
variables declared outside of any function live in the "global storage area" //set up once
only at the start of execution, lasts for entire program lifetime ("the static storage area")
        "garbage"               "memory leak"
How to prevent? Don't lose the pointer to the memory allocation. Give the memory back!
After calling delete [] pointer, the pointer is called a "dangling pointer" - undefined
behavior to try to follow it.
It might appear to work, but storing something there doesn't guaranteed it to be unchanged.
----------------------------------------------------------------
Practical application looks like this:
double* getData(int n)
{       double* p = new double[n];
        //set the values of p[0], p[1], etc.
        return p;    }
void g()
{       int k;
        cin >> k;
        double* a = getData(k);
        //use the values of a[0], a[1], etc.
        delete [] a;  //getData's documentation should remind the user to later delete the
dynamically-allocated memory    }
```

To make a new class:
//no required format for the name, but convention to start with a capital letter

```
struct Employee
{       string name;
        int age;
        double salary;
                //string address;
                //string title;
                //these are instance variables (or members or fields)
};
```
//DON'T FORGET THE SEMICOLON!!!!!!!!!!!!!!

//Pass by reference is cheaper; no copy is made (important for huge structures)
//add const to make sure no modifications are made

To make sure no modifications are made:
-Pass by copy (modifications do not affect original)
-Pass by constant reference (modifications cannot be made)

(an object of some struct type) . (name of a member of that struct type)
(pointer to an object of some struct type) -> (name of a member of that struct type)

Class vs. Struct:
-Same thing except Class assumes any variables as private (if not specified); Struct assumes public
-Struct: use for simple conglomeration of data
-Class: use for more complicated things
-In general, they are the same. Java has no struct. C# has both but they are different!

```
delete p;    //for a single object
delete [] p; //for an array

double* dp = new double[n];
...
delete [] dp;
```

VS.

```
Target* tp = new Target;
...
delete tp;
```

```
//"constructor"
class Target
{public:
        bool move(char dir);
        int position();
        void replayHistory();
private:
        //class invariant: history consists only of R's and L's
        //pos == number of R's minus # of L's in history
        int pos;
        string history;         };
//position_, history_
//m_position, m_history
```

//Using the wrong delete is undefined behavior; program will crash!
//Example: if we make array of Target targets[100], then call delete [] target, the program will crash!
//target is a single object, a POINTER to the first element of the array

```
int *p1 = new int[10];
int *p2[15];
for (int i = 0; i < 15; i++)
p2[i] = new int[5];
int **p3 = new int*[5];
for (int i = 0; i < 5; i++)
p3[i] = new int;
int *p4 = new int;
int *temp = p4;
p4 = p1;
p1 = temp;
```

```
delete p1; // "delete temp;" works too.
for (int i = 0; i < 5; i++)
delete p3[i];
delete[] p3; // This must happen AFTER
// the above for loop.
for (int i = 0; i < 15; i++)
delete[] p2[i];
delete[] p4;
```

```
Target targets[100];
int nTargets = 0; //to keeep track of actual number of targets
//Why waste space (creating 100 targets) when we only use a few?
Solution: dynamically allocate a new target
void h()
{       Target* targets[100];
        int nTargets=0;
        ...
        targets[nTargets] = new Target;
        nTargets++;
        ...
        targets[k]->move('R'); //targets is a pointer to a Target!
        //targets[k] is a pointer, so use ->, not .
        ...
        //for(int k=0; k<100; k++)
//delete target[k]; //attempt to delete every target pointer
//doesn't work: we didn't make all 100!
        for(int k=0; k<nTargets; k++)
                delete target[k];       }
```

```
class Pet
{public:
        Pet(string nm, int initialHealth);
        //Pet(string nm, int initialHealth, Toy* favoriteToy)
        void addToy();
        void cleanup();
        ~Pet(); //Destructor for the Pet class
private:
        int m_health;
        string m_name;
        Toy* m_favoriteToy;  };

Pet::Pet(string nm, int initialHealth)
{       m_health = initialHealth;
        m_name = nm;
        m_favoriteToy = NULL;           }

void Pet::addToy()
{       delete m_favoriteToy;
        m_favoriteToy = new Toy;        }

void Pet::cleanup()
{       //do something...or not         }

Pet::~Pet()
{       delete m_favoriteToy; //works
        //EVEN if m_favoriteToy was not added/NULL          }

void f()
{       Pet p("Fluffy", 10);
        //Pet p2; //Won't compile; no initialization values
        ...
        p.addToy();
         //After f(), didn't delete memory allocation to toy!
        ...
        p.addToy();
        //If we add another toy, then we lose the pointer to the first!
        //MEMORY! addToy should get rid of any previous toys
        ...
        //p.cleanup();
}       <====destructor called on p!

Deconstructor-like constructor, but called when object about to go away
```