

CSCI 241

C++ Course Notes

Basic Unix Commands:

cd go to home directory
cd name change directory to name subdirectory
cd .. step back one level in directory
chmod sets security (in your home dir type `chmod 700 .<.cr>` to set security)
mkdir make a directory
rmdir remove directory
rm remove filename
mv move file or directory
cp copy file or directory
ls list current directory files
ls -l adds details to file names listed
ls -R shows tree branch of directory
touch to "touch" a file will update the time stamp

Editor: (pico)

basic text editor. Can work at home with DOS editor, NotePad, or Turbo C++. Note that you should be careful to use tab setting of 8 spaces to match with pico. Be careful to use tabs and not spacebar when writing code to keep all things lines up properly. Tabs MUST be used when creating a Makefile.

Compiler: (g++)

Options:

-c creates object file
-o allows for specific naming of file (must be directly followed by name)
-g for debugging (puts debugging info into exe but makes it larger)
-O optimizer
-I include include_dir
-Wall gives all warnings
-W num turns off or on warnings (can specify)
-l name links library (ie -lm links math library)
-L library_dir

Makefile:

For each assignment you need a makefile. To generate, go to the directory with your source code and type... `pico Makefile <cr>`. This brings up the pico editor then you type the following:

```
format →      target : dependancies
                  command
```

```
example:      CC = g++
                  CCFLAGS = -Wall -O

                  test1 : test1.o
                        $(CC) $(CCFLAGS) -o test1 test1.o -lm

                  test1.o : test1.C
                        $(CC) -c $(CCFLAGS) test1.C

                  clean:
                        -rm *.o test1
```

When you want to compile and make an executable you type `..make <cr>`.

When you alter your source code and wish to make another executable make sure that you first type `..make clean<cr>` to remove all object files and your executable.

There are several changes between C and C++. In C you would code the following:

```
#include <stdio.h>

void main()
{
    printf("Hello World\n");
}
```

The differences in C++ are as follows:

#include

All standard libraries are available to use in C++ and will work. `<cslib.h>` is not available because it was written as a "crutch" in the 240 class. The way the libraries is called is slightly different :

<stdio.h>	becomes	<cstdio>
<stdlib.h>	becomes	<cstdlib>
<math.h>	becomes	<cmath>
<string.h>	becomes	<cstring>
<ctype.h>	becomes	<cctype>

Note as a standard, you add a `'c'` to the beginning and drop the `' .h'` at the end.

void main()

In C++ we will write as `int main()` and at the close of the program we will include `return(0);` before the last `}`.

Comments:

When writing comments in C++ you should no longer use the `//` method for commenting. From this point forward you should use `/* comment here */`. This is more acceptable across the board and less likely to cause problems for you if you get in the habit now.

printf:

The problem with `printf` is that you have to specify the exact format of the information you are going to print. The `printf` statement still works but there are problems with the `stdio` library so there is a better way called streaming. You will want to `#include <iostream>` in your program.

#include <iostream>

`iostream` contains 3 streams automatically available to you for use in your programs:

```
cin    - for input
cout   - for output
cerr   - for output (usually error codes)
```

cout:

Here's how it works. Say that you have coded:

```
....
int d = 7;
float f = 3.14;
```

to output using `iostream` you would eliminate the `printf` and replace it with:

```
cout << d;
cout << "\n";
cout << f;
```

`<<` is an output operator and basically you are "overloading" the function `cout` which can be done in C++ (remember part 12 of CSC1240).

You can also "*chain*" the `cout` statement to place all variables on the same line of code:

```
cout << d << "\n" << f;
```

This works because the computer will evaluate it similarly to the way it would evaluate the declaration of `int a=b=c=d=3; as a=(b=(c=(d=3)))`; So the program sees the chained statement

```
cout << d << "\n" << f; as ((cout << d) <<"\n" <<) <<f;
```

endl:

Using the `"\n"` is not the preferred way to add a new line return in C++. The manipulator `endl` is much better. Now we can write...

```
cout << d << endl << f;
```

cerr:

The stream `cerr` works much the same as `cout` with one important exception. When you use `cout` the information is sent to a buffer and when the buffer is full it is sent to the display. This is fine except if you are using `cout` statements to work on debugging your code. If you use `cerr` each line will print out directly. This is a slower output but will help you considerably when debugging.

The `endl` statement also flushes the buffer so put it at the end of each line.

Note: When using `cout` to print a character array named `s` you can write `cout << s` because it will point to the start of the character array and does the rest automatically.

cin:

The stream `cin` works much the same as `cout`. It is to replace the `scanf` statement in C. The `scanf ("%d", &variable);` statement is cumbersome because you have to pass the variable by reference to the function `scanf`. When you input, you can code as follows:

```
cin >> d >> f >> s;
```

Note that a string will input until it hits a "white space character" (space) so at this point we only know how to input one word at a time.

data re-direction:

Data re-direction works the same way as in C. at the prompt (mp%) type:

```
progl<p1.dat>p1.out
```

flush:

the command `flush` will empty the data buffer for the output. When you are printing statements and retrieving input from the user you have to flush at the end of the `cout` before the `cin` is written.

Sample Code:

```
#include <iostream>

int main()
{
    int d;
    float f;
    char s[80] = "Wow!";

    cout << "Hello World" << endl;

    cout << "Input a float. " << flush;
    cin >> f;

    cout << "Input an integer. " << flush;
    cin >> d;

    cout << "Input a string. " << flush;
    cin >> s;

    cout << "The value for d is " << d << endl;
    cout << "The value for f is " << f << endl;
    cout << "The value for s is " << s << endl;

    return (0);
}
```

bool:

You can use the variable type `bool` when you want something to have a logical operator of either true or false. The conditions `true` and `false` are designated to have the value of `true=1` and `false=0`.

Input/output formatting:

`get` is an overloaded function and can determine what the usage of the function will be depending on the arguments passed in. Therefore you can use the following variations...

`cin.get(char &ch);` This is the structure used to get a character from the keyboard if you were to use it in code, it would be written as `cin.get(ch);` This could also be coded as `cin >> ch;`

`cin.get(char *buf, int n);` This array of characters has 'n' elements in it and is guaranteed to put an end of line character at the end of the array or will go until it hits a new line character and then put a '\\0' character and return to the program.

`cin.get(char *buf, int n, char term);` This is more rare. It does the same as the one listed above but allows you to designate what the terminating character is.

Note: This form may create a problem because it leaves the "\\n" character there. This may sometimes be a problem but may also come in useful at times. Just be aware of the fact.

Example of problem:

```
...
while (cin)
{
    cin.get(buffer, 256);
    cout << buffer;
}
```

This will cause an infinite loop because it is thrown off by the "\\n"

More input functions:

`cin.getline(char *buf, int n);` Once it reaches a new line char, it will throw it away and not put it in the buffer. Both the `get` and the `getline` functions can have embedded or leading whitespace before or during the string.

`cin >> buffer` will throw away leading whitespace and stop when it encounters and additional whitespace.

format output:

Manipulators are used to replace many C functions.

`endl` -replaces the "\\n"
`flush` -flushes the print buffer but does not print new line.

There are many more...

setting width:

`cout.width(int n);` will set the width for the next output only. The integer n indicates the minimum number of characters used in the output and numbers are right justified by default.

*width is the only option that applies to only the next object to be printed to output. All other options remain until reset to another option.

`setw(n);` is an easier way to set the width of a field. You must use `#include<iomanip>`.

**width can also be used in input streams. Example: `cin.width(4);`

decimal output:

to force integers to print in decimal format...

```
cout.setf(fmt flags);  
cout.setf(fmt flags, fmt flags);
```

Use the first when only one option is available. Use the second when two or more options are available.

```
cout.setf(ios::dec, ios::basefield); -note no spaces. this prints in decimal format  
cout.setf(ios::oct, ios::basefield); this formats in octal format  
cout.setf(ios::hex, ios::basefield); this formats in hexadecimal format  
    **(ios:: is a scope identifier and tells the second thing where to look)
```

distinguish the outputs as follows:

```
    1234      - decimal  
    01234     - octal  
    0x1234    - hexadecimal
```

If you would like to print the output in a given format, you may use the flag names `showbase`:

```
cout.setf(ios::showbase); - this will option on and print in the above format.  
cout.unsetf(ios::showbase); - this will turn the option off.
```

Other flags in output stream classes:

by default, all numbers and strings are right justified.

```
cout.setf(ios::left, ios::adjustfield); - will format all output to left justified.  
cout.setf(ios::right, ios::adjustfield); - will format all output to right justified.  
cout.setf(ios::internal, ios::adjustfield); - will be used if you want to do something like  
    adding a '-' to the left side of a column of numbers indicating a negative number and still  
    justifying the numbers to the right.
```

There is more on this subject in Chapter 2 Section 13 of the Kalin textbook

```
cout.fill(char ch); - You can use this if you want to fill all blank spaces in a field with a '*' or any  
other character:
```

```
****123  
**26747  
***1234
```

```
cout.setf(ios::showpos); - this will attach a '+' sign to all positive numbers.  
cout.unsetf(ios::showpos); - this will turn off the option.
```

combinatorial |:

There is a difference between `||` which is an 'or' operator and `|` which combines.

```
cout.setf(ios::showbase|ios::showpos); - this will combine the two flags as one operation.
```

flags for floating point numbers:

```
cout.setf(ios::fixed, ios::floatfield); - fixed state.  
cout.setf(ios::scientific, ios::floatfield); - scientific notation  
cout.setf(0, ios::floatfield); -general notation
```

More output flags:

`cout.setf(ios::showpoint);` This will print out decimals so when it prints it will force it to print all trailing zero's.

`cout.setf(ios::uppercase);` when using scientific notation, `3.14+e25` becomes `3.14+E25` or when you are displaying in hex format, `0x325` becomes `0X325`.

Setting precision: default precision is set to 6 places

`cout.precision(int n);` This sets precision until you set it again.

general- this is actual digits not including the + or - but including the decimal.

float - how many to print after the decimal point.

`setprecision(4);` is another way to set the precision but if you use this you must use the `#include <iomanip>` manipulator.

`ios::showpos` only applies to decimal numbers. It will show a + sign on all positive numbers.

One more manipulator:

`cout << setiosflags(ios::showpos);` (to turn on)

`cout << resetiosflags(ios::showpos);` (to turn off)

Here is an example of formatting manipulators to try:

```
#include <iostream>
#include <iomanip>

int main()
{
    int d = 1234;
    double pi = 3.1415926;
    char msg[] = "This is really fun";

    cout.setf(ios::showbase|ios::showpos); //remember | is a combiner
    cout << d << oct << " " << d << hex << " " << endl;

    cout << pi << endl;
    cout << setprecision(20) << pi << endl;

    cout << setw(20) << msg << setw(6) << d << endl;

    cout.setf(ios::left,ios::adjustfield);
    cout.unsetf(ios::showpos);
    cout << setw(20) << msg << setw(6) << d << endl;
    cout << 3.5 << endl;
    cout << setiosflags(ios::showpoint) << 3.5 << endl;
    return (0);
}
```

There is a difference between integers in Turbo C and C on mp. In Turbo C an integer ranges from -62767 to +62767 (check this amount). In mp an integer ranges from -2 billion to +2 billion which is the same as a long int in Turbo C.

Control Structures:

Any program can be written with three main elements: statements, decisions, and loops.

If you need to break out of a program you can use the `break;` command. There are a few things to know though:

`break;` will get you out of a block of code one level of a for loop or switch statement. If you use `break;` in an `if` statement, it will kick you out of the current loop you are in. Break will kick you out of whatever LOOP you are in NOT just the decision which contains the break statement.

Example:

```
#include <iostream>

int main()
{
    int i, j;
    for (j=0;j<1;j++)
    {
        for (i=0;i<1;i++)
        {
            break;
            cout << "A";
        }
        cout << "B";
    }
    cout << "C";

    return (0);
}
```

The printout would be: BBC

Variable declarations:

In C++ you can put declarations of variables where you need them. Note that if you declare a variable mid-program, it only exists and available while the program is in the block of code where it is declared. Once you are out of that `{ }` it vanishes and is forgotten.

Example: `for (int j=0; j<10; j++)`

Continue:

`continue` is similar to the `break` command except that it will “bag” the rest of the body of a loop and go straight to the test (or condition). It can only be used in loops. Like `break`, it ignores `if` statements on exit and jumps out to the nearest `for`, `while`, or `do..while` loop.

Example:

```
while (1st condition...)
{
    ...
    ...
    ...
    if (2nd condition ...)
        continue; /* This will jump straight back to */
                 /* the while conditional statement */
    ...
    ...
    ...
}
```

Application example of continue:

```
#include <iostream>

int main( )
{
    int score[80];
    int sum=0, num=0;

    for (int j=0; j<80; j++)
    {
        if (score[j] <50)
        {
            continue;
        }
        sum += score[j];
        num++;
    }
    double average = double(sum)/num;
    cout << "The average for " << num << " people is "
         << average << endl;
    return (0);
}
```

goto:

If you need to break out of more than one level you can use the `goto` command. Unlike the `break` and `continue` statement, the `goto` is an *unconditional* statement. It is only to be used in certain instances and should be avoided unless completely necessary.

to use, you would type `goto label;` Label is any name you give indicating a section of code. At that section of code you would then type `label:` The program would then continue at that point.

recap:

```
continue; - stay in loop but skip to increment and test
break;    - break out of immediate loop (not if statement)
goto;     - unconditional jump to a specified label name.
```

switch:

`switch` statement is similar to cascading `if`, `else if`, `else` loop. It's great for char by char input.

```
switch(variable_name)
{
    case 1:
        ~~executable code ~~;
        break;
    case 2:
        ~~executable code ~~;
        break;
    default:
        ~~executable code ~~;
}
```

You can also have several cases back to back with one set of executable code for several choices. Note that all cases except `default` have to be scalar constants (single pieces of information) of `int` or `char` type. The order of cases is not important and `default` can be anywhere but there MUST be a `break` statement in each case to keep it from filtering down to the next condition.

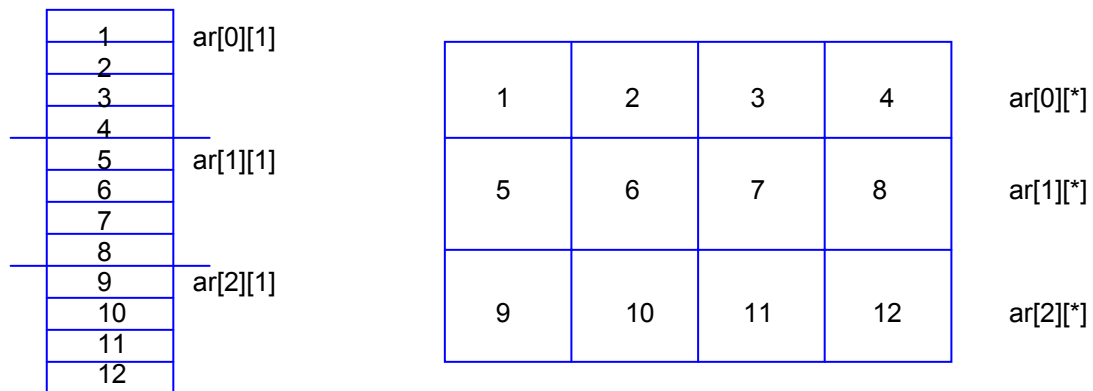
Arrays:

arrays are “packages” of information of the same type. In reading types in C and C++ we can use the “flip-flop” method. Start at the variable name and then go as far to the right then as far to the left as possible.

Example1: `int ar[5];` // ar is an array of 5 integers

Example2: `int ar[3][4];` // ar is an array of 3 arrays of 4 integers

following is an illustration of how a two dimensional array is stored in memory:



This type of order is called row major order and is a good idea to do calculations in this order. Make sure that your loops follow the “major order” of your two dimensional arrays in order to assure accuracy. If you always process in “row major” order you will preserve cache and speed up calculations.

Array Declaration/Initialization:

To declare an array: `ar [3][3];` // second array has to be given value always
`ar [][3];` // also valid because second is given value
`ar [][];` // ****illegal declaration****

Must give second a value because when it sets area up in memory it doesn't need to know how many rows but it does need to know how many elements in each row. It can always add more rows to the bottom but can't jam any more in a row that the allowed space.

To initialize: `ar2 [5] = {0,1,2,3,4};`
`ar3 [2][2] = {{3,4},{5,2}};`
`ar4 [] = {2,7,8,0,8};`

Passing arrays to functions:

```
void f1(int ar [5]);           // declaration prototype
void f2(int ar[ ], int n);    // n is number of elements in array
void f3(int [ ]);           // needs a "marker value" (\0) to show end
void f4(int ar*);           // really works the same as f3
```

now for 2D arrays...essentially the same thing is applicable:

```
void f5(int ar[5][5]);
void f6(int ar[ ][ ], int a, int b);
```

sleep();

the sleep function is a way to get your program to pause for a given period in seconds. It implement it's use you need to include the following:

```
#include <unistd.h>

sleep(3); // sets 3 second pause
```

rand();

the rand function will return a pseudo-random computer generated number at it's calling.

```
#include <cstdlib>

num = rand( ); // num is integer random number is assigned to.
```

srand();

The srand function will seed the random generator so that it will not always start with the same "random" numbers:

```
#include <cstdlib>

srand(n); // n is an unsigned int seed value
```

This is better but the seed has to be set by the user. If you really want to have a randomly picked number you need to seed the srand() with a different number everytime. In order to do this you can use a function called time() which gets time in seconds since 00:00:00 UTC Jan. 1, 1970

time();

```
#include <sys/types.h>
#include <time.h>

time_t time(time_t *tloc); // need clarification here
```

Pointer arithmetic:

following are several examples of pointer arithmetic and it's many uses:

```
int j;
char *p = "Hello"
```

cout << j would print out a hex number such as 0x8364BD indicating the location in memory of the beginning of the array.

if you take p+2 the result is another pointer which points to a place 2 characters further in memory.

```
if p points to "H"
p+2 points to "l"
```

in an array of integers: int q[]={-7,3,21,6};

-7	3	21	6	"\0"
----	---	----	---	------

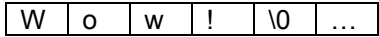
q points to -7 and q+2 points to 21

What's important is that the size of an integer is different from the size of a character:

p+2 is 2 bytes further in memory because it's a char
q+2 is 8 bytes further in memory because it's an int

the statement... `char *p = "Hello";`
 is the same as... `char p[] = {'H', 'e', 'l', 'l', 'o'};`

you can't add pointers to pointers because a pointer is an absolute location in memory. **BUT** you can do subtracting... Say that you have the array:



Assume that the array is `q[]`. The variable `char *q` is the address of the beginning of the array (or 'W')
 If you want to find how long the array is, you use another pointer, shall we say `char *r`, and step through the array until you find `'\0'` incrementing `r` as you go. Once you find the null terminator you take the difference between the two and that is the length of the string (`r-q=5`).

This will work as long as the pointers are of the same data type. You will get the # of elements between the two pointers.

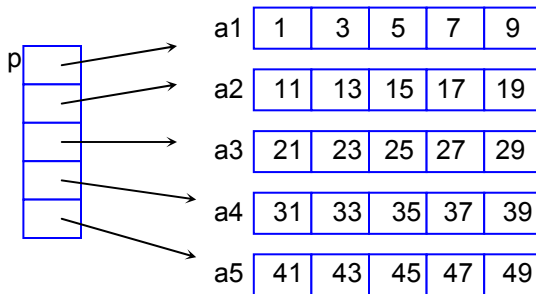
Let's create some arrays for pointer arithmetic practice:

```
int a1 [ ] = {1, 3, 5, 7, 9};
int a2 [ ] = {11, 13, 15, 17, 19};
int a3 [ ] = {21, 23, 25, 27, 29};
int a4 [ ] = {31, 33, 35, 37, 39}; // five separate arrays
int a5 [ ] = {41, 43, 45, 47, 49}; // of contiguous sets of numbers

int *p [ ] = {a1, a2, a3, a4, a5}; // array of pointer to ints Each
// points to start of array
```

Note: if you have arrays of different size this works well because the length of each row is different.

in Memory it looks like this:



- `p[2]` - gives address of pointer to int array a3
- `p[2][3]` - points to third element of second array (27)
- `(p+2)[3]` - gives address of start of a3 and third element (27)
- `(*p)[3]` - points to a1 position 3 (7)
- `*p[3]` - points to a4 then de-references to value (31)
- `*(p[3])` - same as above (31)
- `**p+2` - address of element #2 in p array (a3) then the de-ref. gives first element in a3 (21)
- `*(p[1]+3)` - goes to `p[1]` which is address to a2 then goes 3 spaces in and de-references (17)
- `*(p+3)[1]` - points to a4 and sees that as p so `[1]` points to a5 and de-references spot to (41)

Note: array subscripting comes before de-referencing.

L-value:

an l-value is anything you can put on the left side of a statement. It is someplace where you can actually store a value. In the case of `b[3]` ... `b[3]` has a value but `b` is not an L-value because you can't assign a value to `b`.

Swap Routines and improvements:

a function for a general swap routine would appear as follows:

```
void swap(int *a, int *b)
{
    int temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

to call this you would code... `swap(&ar[j], &ar[k]);`

This happens so often that in C++ they made a better way:

Reference:

now you can swap like this:

```
void swap(int &a, int &b)
{
    int temp;

    temp = a;
    a = b;
    b = temp;
}
```

to call this you would code... `swap(ar[j], ar[k]);`

Notice that when in the function you can use the values referenced into the function as you would outside the function and the actual values would be given.

pointers are L-values. References get you right to what you are referring to so when this is done the values are changed. Remember that you always have to use L-values when you swap. You can't call something like `swap(3,j);` // this is a no-no

examples of initializations...

```
int j;
int *p;

p = &j;

int &r = j    // a reference must be initialized to something
```

later on if you change `j` to `j=7` and then code `cout << r;` the output will be 7 because `r` points to `j`.

Const:

Whatever type you apply it to cannot be changed. It is much the same as the #define statement

Example:

```
#define MY_SYMBOL 3
is the same as...
const int MY_SYMBOL = 3;
```

What really happens is that when using the #define the compiler will go through the source code before compiling and change each instance of the variable to the value it holds. The problem with this is that in some other function, say a library function, if someone used the same symbolic constant using #define it could be changed having disastrous consequences.

Using const will not allow the variable value to be changed. It is much better to use in most cases. const can be used with floats, doubles, pointers...

The proper way to read const variables is from right to left.

Example:

```
const char p[ ] = "Wow!";
char const p[ ] = "Wow!";
char *const p = "Wow!"
```

These are essentially pretty much the same and are read:

1st p is an array of characters which are constant.

2nd p is an array of const char

3rd p is a constant pointer to characters

Difference:

You can change pointers but you can't change constants. For example:

```
const char *p = "hello";
const char *q = "wow";
```

you can't code `p[0] = "j";`
but you can say `p=q.`

But if you code:

```
char *const p = "hello";
char *const q = "wow";
```

now you might be able to code `p[0] = "j";`
but not `p=q.` This would give error because *p is declared as a constant pointer*

References:

the advantage of using references is that when you pass by copy and you are passing something very large it takes a lot longer. This isn't largely evident in things on a smaller scale but imagine if you were passing an array of structs holding phone numbers for the city of Chicago. When you pass by reference you only pass in a location instead of a copy of the entire array.

examples of prototypes and calling statements:

<u>Header</u>	<u>calling statement</u>
<code>void f1(Widget w)</code>	<code>uses f1(w1);</code>
<code>void f2(Widget *w)</code>	<code>uses f2(&w1);</code>
<code>void f3(Widget &w)</code>	<code>uses f3(w1);</code>
<code>void f4(const Widget &w)</code>	<code>uses f4(w1) //guarantees Widget isn't changed.</code>

if you de-reference a pointer to a class or struct you have to use either..

```
(*p).field or w->field
```

get in the habit of passing by reference and if you don't want it changed, make it `const.`

File Input/Output:

with streams, files are surprisingly easy. To use a file: `#include<fstream>` deals with streams involving files. There are three choices available.

```
ifstream // input file stream
ofstream // output file stream
fstream  // works for input or output
```

once opened the file can be used like any input or output stream.

```
fstream inp, out;

inp >> j; // this works reading int, float, double..
out << "Hello";
```

But how can I open the file??

```
inp.open("name", flags)
```

flags:

```
ios::in //opens for input
ios::out //opens for output
ios::binary // opens for binary input/output
ios::app // appends to the end of the file
ios::trunc // truncates and cleans everything out to zero length
```

So to open a file named mydata for binary input you would do the following:

```
ifstream inp; // given name for datafile
inp.open("mydata", ios::in|ios::binary); // literal name/path of file
```

You can also open a file for input and output at the same time.

After you are done with the file ALWAYS close it. Use the command, `inp.close()`;

There are two ways to read in C – `fscan` and `fread`. Same problem in C++ so there are other functions that work in conjunction with `in` and `out`... `read` is like `fread` in C (byte by byte)

```
inp.read(char *buf, int n); // char *buf , where to read to, n is number of bytes to read.
```

to write in binary fashion:

```
inp.write(char *buf, int n); // same specs apply
```

in C you use `fopen` to open a file and check for "NULL" In C++ you can use `if(cin)`..

Example:

```
inp.open("file.dat", ios::in);
if (!inp) // if it opened correctly the test will be OK
    cout << "error"; // if not test will be 0 or condition false
```

to test for end of file:

```
while(!inp.eof())
```

Hungarian notation:

When you name your variables try to follow the rule that places the first letter of the variable name as the first letter of the type of variable it is. This keeps you in a better frame of mind as to what you are working with.

Passing 2D arrays to functions:

arrays and pointers are very similar but arrays set aside memory block whereas pointers don't.

Classes:

Classes are similar to structs with the exception of a few things.

- 1) classes can have member functions.
- 2) they support public and private data

Example:

```
class myclass
{
private:    // by default all class members are private
    int x,y;
    char c;
    int func(void);
    Mytype acct;    // you can also have a member of another
                    // type that you created.
public:
    void setx(int x);
    myclass()    // default constructor to initialize
    ~myclass()    // destructor to clean up out of scope
};
```

in the main you can code the following:

```
#include <iostream>

int main()
{
    myclass c;
    c.setx(10); // can do because member function is public
    return 0;
}
```

You can't code `c.x = 10;` because `x` is a private variable.
...use `get` and `set` to retrieve and set values.

```
class Target
{
    int x,y;
    bool found;
};
```

Then in main to declare you would code...

```
Target p[3];    //declares an array of Targets
```

to get at the information you do similar to a struct:

```
p[2].x
p[2].y
p[2].found = true;
```

More Examples:

```
class T
{
public:
    int x,y;
    double f;

private:
    int j,k;
};
```

in main()...

```
T a,b;    // variables - objects - instances of type T

a.y = b.x; // OK
a.f = b.y ; // OK
a.f = b.j; // Can't because j is private
a.j = b.j  // Can't    "    "
```

****One helpful thing in C++ classes is that you can do direct assignment of class variables without worrying about copying each one.**

```
i.e.  a = b; // copies all elements of a into b
```

This is an easy way to copy arrays in one fell swoop.

Class Member Functions:

They only apply to the class that they are in.

any member function has access to all of the members of the class so when you are outside you can use a function to get a value when you cannot access the data members directly. This is where a `get` and `set` function come into play.

Member Function Headers:

```
int T::get_j(void)
{
    return j;    // sends current instance of j
}
```

first you have `int` as a return type

`T` distinguishes the function as a member of class `T`

`get_j` is the method name

when you call this function you use `a.get_j()` and it will call the function and return the value of `j` for the instance `a` of the class `T`

```
void T::set_j(int i)
{
    if(i<0) // lets you set positive values only
        return;
    j = i;
}
```

later on in the code you call...

```
a.set_j(3); // lets you set private value of j to value of 3 because of design of class
```

Constructors:

You always use a constructor to initialize variables. Even when you don't code it in, a default constructor is added. They always have the same name as class.

Example:

```
class M
{
    int x,y;

public:
    M();    // default constructor
    M(int) // constructor for different purpose
};
```

```

M::M( )
{
    x = 7;    // can do anything inside
    y = 3;    // now everytime the program calls class M
}            // x will be set to 7 and y set to 3

```

When calling..

```

M t;        // x and y members of M are given 7 and 3 respectively.

```

```

M::M(int j)
{
    x = 7;
    y = j;
}

```

One thing to be aware of is if you, for instance, have x in function header...

```

M::M(int x)
{
    x = 7;
    y = x;
}

```

So now with two constructors, there are multiple ways to call the constructors.

```

M a;        // Calls default constructor
M b(5);     // Calls second constructor
M c=5;     // also calls second constructor

```

Sample Code:

```

#include <iostream>

class A
{
    int x;

public:
    A();    // constructor is always public
    A(int);

    int get_x(void);
    void increment(void);
};

```

Sample code continued:

```

A::A()
{
    x = 0;
}

A::A(int s)
{
    x = s;
}

int A::get_x(void)
{
    return x;
}

```

```

void A::increment(void)
{
    x++;
}

int main()
{
    A a;
    A b(5);

    cout << a.get_x(x) << endl;
    b.increment();
    cout << b.get_x( ) << endl;
    return (0);
}

```

Breaking Up A Program:

To make your code manageable you should break it up into several files and then link them together when you compile. This will eliminate the problem of having to change something in the source code and then having to re-compile the entire project. This seems like small potatoes now but when the projects take hours to compile you will understand the importance more.

From now on in this class, each class should have it's own .C and .h files which will be linked to source code.

(A.h file)	(B.h file)	test.C
<pre> #ifndef A_H #define A_H class A { }; #endif /* header guards for A_H */ </pre>	<pre> #ifndef B_H #define B_H #include "A.h" class B { A a; ... }; #endif /*header guards for B_H */ </pre>	<pre> #include "B.h" #include "A.h" int main(); { B b1; A a1; return 0; } </pre>

Now in another file named **A.C...**

```
#include "A.h"

A::A( )      // default constructor
{
    code to initialize
}

A::A(int x)  // initialize for special use
{
    special initialization
}

void A::print(arguments)
{
    printing code
}
```

same follows for B.C file for all B class member functions.

so now you will have

```
test.C
A.C
B.C
A.h
B.h
```

all together they make test.o, A.o, and B.o which are linked to test, your executable
This can get rather sticky so use your Makefile to avoid errors...

```
CC = g++
CCFLAGS = -Wall -O

test : test.o A.o B.o
    $(CC) $(CCFLAGS) -o test\
        test.o A.o B.o -lm(if needed to link math)

test.o : test.C A.h B.h
    $(CC) $(CCFLAGS) -c test.C

A.o : A.C A.h
    $(CC) $(CCFLAGS) -c A.C

B.o : B.C B.h A.h
    $(CC) $(CCFLAGS) -c B.C

clean:
    -rm *.o test
```

note the \ in the first section... This is a way to extend the line in a Makefile. In order to implement this, you have to put a <cr> directly following the \.

Reading Files:

if you want to access a binary data file from the command line prompt you would include the following:

```
#include <fstream>
...

ifstream myfile;

myfile.open("filename", ios::in|ios::binary);
```

then at the command prompt you would type:

```
executable information.dat
```

when you write your main function it would look something like this...

```
int main(int argc, char * argv[ ])
```

`argc` returns the number of arguments(2) while `argv[]` returns `information.dat` in this case

to test if two arguments were entered you can code:

```
if (argc != 2)
    cout << "Error please include data file" << endl;

myfile.open(argv[1], ios::in|ios::binary)
if (!myfile)
    cout << "Error opening file" << end;
```

Debugging:

`make > errs` redirects output to filename

`make >& errs` redirects std. output and std. errs

Segmentation Fault:

This most often happens when you are trying to write outside the bounds of an array or pointer that wasn't properly set.

Core dump:

When your program fails when running, all information about your program is placed in a core file.

cerr <<:

remember to use the old faithful standby of print statements to check where you are in code.

Gnuu:

This is the debugger we can use in this system. To use it you must include the `-g` option in your makefile `g++` lines.

```
g++ -Wall -c -g test.C
g++ -o test -g test.o
```

A good way to do this is to make it part of the `CCFLAGS` statement but you must ALWAYS remove the `-g` before submitting assignment for grading.

at prompt type `gdb...` Example: `mp% gdb assign4<cr>`
the prompt will then change from `mp%` to `(gdb)`

gdb options:

r -(or run) will run debugger
bt -(backtrace) use u for up and d for down (gives list of all functions called in reverse order)
l -(list) lists code in stack with line numbers where you currently are ("l" again gives you next 10)

to run line by line:

b 35 -(break) sets a break at line 25
n -(or next) will step over and execute following line. every <cr> will execute another line
s -(or step) will step into a function
p -(or print) ie. p n.targets will print value in variable n.targets
c -(continue) after the break point
delete 1-will delete breakpoint #1
quit-will quit the debugger

you can even change values while running debugger.

Example:

```
print grid [0][3]<cr>          // prints out what's there
print grid [0][3] = 'J'<cr>   // changes value of variable
```

Sample Code:

following is an example of the .h and .C files to make a class of complex numbers and to implement them in a source code. This should help to make things clearer.

Complex Class:

in filename: complex.h

```
#ifndef COMPLEX_H
#define COMPLEX_H

class complex
{
    double re, im; // representing real and imaginary

public:
    complex();
    complex(double r);
    complex(double r, double i);

};
#endif /* COMPLEX_H */
```

in filename: complex.C

```
#include "complex.h"

complex::complex()
{
    re = im = 0.0; // initialize all to 0
}

complex::complex(double r)
{
    im = 0.0;
    re = r;
}

complex::complex(double r, double i)
{
    re = r;
    im = i;
}
```

in filename: test.C

```
#include <iostream>
#include "complex.h"
.
.
.
int main()
{
    complex a;      // calls default constructor
    complex b= -3.7; // calls second constructor
    complex c(2.5, 3.7) // calls third constructor
    .
    .
    .
    return (0);
}
```

default arguments:

in C++ class definitions you can just write one constructor that will take the place of all three.

example:

```
complex(double r=0.0, double i=0.0);
```

now you can call the same way...

```
complex a;
complex b= -3.7;
complex c(2.5, 3.7)
```

now all will call the same constructor using the default arguments given.

recall:

in main code:

```
get(char *p, int n);
get(char *p, int n, char term);
```

in header for class:

```
get(char *p, int n, char term = '\n');
```

this is an example of what we already do that uses default arguments. If you specify a terminating character or if you don't it will work either way.

this:

this is a keyword in C++ which refers to the current instance

How do we use this?

```
complex a(1,1), b(2,-1);
complex c;

c = a + b; // this would be nice...
```

Overloading operators:

in C++ you can overload operators as well. In example, we can write code to add two things we have created.. in the class..

```
complex operator+ (complex b);
                (const complex &)const; //const because constant instance
```

in a member method:

```
complex complex::operator+ (complex b);
{
    complex result;

    result.re = re + b.re;
    result.im = im + b.im;
    return result;
}
```

now you can just add the types together.

in regular function:

```
complex operator+ (const complex &a, const complex &b)
{
    complex result;

    result.re = a.re + b.re;    // must pass each value
    result.im = a.im + b.im;    // assuming they are public
    return result;
}
```

in member function:

```
void complex::print(void)
{
    cout << re << " " << im;
}
```

overloading the << operator

```
ostream & operator << (ostream &ostr, const complex &c)
{
    ostr << c.re;
    ostr << " ";
    ostr << c.im;
    return ostr;
}
```

inside a complex class, we have data members to public methods in public: that mention other functions.

friend:

friend says that here is a function that is not part of this class but is safe and allowed to access private parts of this class. Friend is used only inside the class definition.

```
class complex // revisited
{
public:
    friend ostream & operator <<(ostream &ostr, const complex &);
};
```

class declaration:

ie. class ostream; can only be used when class is a reference. For example...
ostream & operator << (ostream & ostr...)

Abstract Data Types:

An abstract data type is a description of a collection of data and things you can do with that data. It is strictly a “pencil and paper” thing.

each operator should have a pre-condition and a post-condition
There is no definitive abstract data type (ADT)

Larger projects take planning about what kind of classes you have to make for the particular project so you really need to read and understand this in Data Structures book Chapter 2 – It's very important

Abstract data types(ADT's) include... strings, list, queue, stack to name a few

Example:

string:

- a sequential collection of characters.

operations of a string:

- altering
- clearing
- copying
- comparing
- concatenating
- character access
- deleting characters
- input
- output
- sorting
- sub-string
- string length

Various ways of representing strings

a)

W	e	l	c	o	m	e	'\0'
---	---	---	---	---	---	---	------

character array with terminating character

b)

H	e	l	l	o			
---	---	---	---	---	--	--	--

length = 5

character array with variable length determining end of string

c)

c	o	m	p
			*

u	T	e	r
			*

chained arrays of set size using pointers to chain together

Let's go with example b) since it is most like the strings used in C++ standard string class

String Class:

```
const int String::MAXLEN = 80;
    // above all else in class.C but here only for use in this class

/*****/
class String
{
    const int MAXLEN;
    char data[MAXLEN];
    int len;    // length of string

public:
    String();
    String(const char *);           // initialize from C style string
    int size(void) const;          // return size - const and
    int length(void) const;        // won't effect class members
    bool operator < (const String &) const;    // overloads < operator

    // friend function here overloads the << operator
    friend ostream & operator << (ostream &ostr, const String s);
};

/*****/
String::String()
{
    len = 0;
}

/*****/
String::String(const char *c)
{
    len = 0;

    while(*c != '\\0' && len < MAXLEN) // copies char until end of array
    {
        data[len] = *c;
        len++;
        c++;
    }
}

/*****/
int String::size(void) const
{
    return len;
}

/*****/
int String::length(void) const
{
    return len;
}
```

String Class Continued:

```

/*****
bool String::operator < (const String &s)const
{
    // Left side is the current instance
    int i=0;

    while(data[i] == data[s] && i<len && i<s.len)
    {
        if(i==len && i==s.len)
            return false;
        ... to be continued
    }
}

/*****
ostream & operator << (ostream &ostr, const String &s)
{
    for (int i=0; i<s.len; i++) // can access s.len because friend func.
        ostr << s.data[i];
    return ostr;
}

```

when you call the overloaded << operator in the main() you will call it by simply using it.

Example:

```
cout << '*' << s << '*';
```

will print out:

```
*Mine!*
```

Dynamic Memory Allocation

DMA allocates memory for use in program at the time the program runs. An example of this is a word processor, which allocates memory “on the fly”.

Routines for allocating memory are in:

```
<cstring> // contains info on memcpy
```

```
<stdlib> // info on malloc and free
```

malloc:

Malloc in C stands for memory allocation. It takes one argument, which is an integer representing the number of bytes you want to allocate, and returns a pointer to a void.

```
void * malloc (int n);
```

Example of use:

```
int *p = 0; // initializes pointer to 0
```

```
p = (int *) malloc(sizeof(int)) // sizeof assures correct number of bytes
```

now you can use like...

```
*p = 7;
```

**** IMPORTANT ****

Whatever you allocate, you are responsible and MUST free up when done using it to avoid memory leaks. This is done using the keyword `free`.

free:

`free` will return memory allocated using `malloc`. An example of it's use follows.

```
free(p); //this free's up memory allocated for *p
```

Note: If this memory was allocated inside a function and was not freed up before you left function, you would most likely have a memory leak. When you leave the function you lose your variable but then you have allocated memory that is not addressable from anywhere.

What about memory for more than one thing?? Like an array of integers? Let's try for 100 integers and only 100 integers...

```
int ar = 0;

int size;
.
.
.
size = 100;
ar = (int *) malloc (sizeof (int) *size);
```

This will allocate dynamic memory for 100 integers. Now you can take this pointer and reference it as an array.

Example:

```
cout << ar[7];
free (ar);
```

There is more information on "on the fly" memory allocation in the man pages under `realloc`.

Problem:

`Malloc` and `free` have one problem. `Malloc` does not initialize allocated memory. This is why C++ came up with a new way to allocate memory using the keyword `new`. This is the method we will use in this class from now on.

new:

`new` is an operator which takes on the right side, type of memory to allocate...

Example:

```
int *p;
p = new int;
```

notice that you didn't need anything like `sizeof`. The compiler takes care of this. Also, you don't have to do casting because it will also check for that. Another added bonus is that `new` will allocate memory and call the constructor to initialize it (yippee!).

Example:

```
class Widget
{
...
};

Widget *w, *v;

w = (Widget *) malloc (sizeof(Widget)); // this is the C way
v = new Widget; // this is the way for C++
```

It's much easier AND calls the constructor.

Freeing up the memory is now replaces the command `free`, with the keyword `delete`.

delete:

Delete takes a pointer to the type to free up memory for.

```
free (w);    // this is what you did in C using malloc
delete v;    // this is what C++ uses when using new
```

IMPORTANT:

Under no circumstances can you use `free` to replace memory that was allocated using `new` or `delete` for memory allocated using `malloc`. You can use either or both in C++ but you must use with corresponding types.

`new` and `delete` are only for C++ and are keywords so you don't have to "include" anything.

Example: now allocate an array of 10 Widgets...

```
Widget *x_ar;

x_ar = new Widget[10];
```

This allocates memory and calls constructor for all 10 Widgets. But there is one minor wrinkle. When you release memory for an array of Widgets it is slightly different.

```
delete [] x_ar;
```

What would happen if you were to code...

```
p = new int[10];
delete p;    // instead of delete [] p
```

or...

```
q = new int;
delete [] q;    // instead of delete q
```

not much at run time but it will catch up with you so make sure you use proper delete operator.

One more thing about the new constructor and taking arguments.

Example: to dynamically allocate memory for a complex variable...

```
complex *c, *d, *e;

c = new complex;
d = new complex(2.7);
e = new complex(3, 1.7);
```

*** you can call whatever constructor you want BUT you can't do it for arrays.

arrays will always call the default constructor so always make default constructor if you could possibly use an array in the class.

New Implementation of String class Using DMA:

if we were to use the String class we created earlier and ran over the size of memory that was set aside for the array we would be toast. So...let's create a new String class using DMA.

```
class String
{
private:
    char *data;
    int len;

public:
    String();
    String(const char *);          // initialize from C style string
    .
    .
};
/*****
String::String()
{
    len = 0;
    data = 0;    // makes sure it doesn't point to something important
}

/*****
String::String(const char *s)
{
    len = s.len;
    data = new char[len]

    // loop to copy all characters from s into data
    for (int i=0; i<len; i++)
        data[i] = s[i];
}
```

Tip: Look into `memcpy` in libraries manual for more information

But wait a minute... How are you supposed to free up memory here??

```
String s = "disaster";
```

when it's created you have...

```
*s [ len = 11 ] → [ d | i | s | a | s | t | e | r ]
```

when the function ends it will free up *s but it will now free up the memory used by the string. The solution for this is to use a destructor.

destructor:

in addition to the constructor in the class declaration you need to include...

```
~String();
```

and in the .C file you need the method:

```
String::~~String()
{
    delete [] data;    // use this way to delete memory for array of char's
}
```

the destructor will be called automatically so you don't need to worry about calling but you must have it or you will run into big problems.

delete will automatically call the destructor before it frees up memory. This is why we need to delete arrays with `delete [] ptr`. It will call destructor on every single position in the array and free them up.

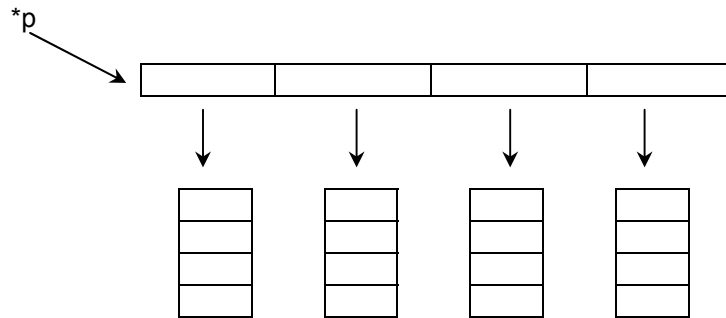
now we will make a slight change from the String class to a Vector class (very, very similar).

Vectors: (one-dimensional array of numbers)

say you have an array and call a function named void func()

```
void func()
{
    p = new Vector[10]; // allocates memory for array of 10 Vectors
}
```

This will call constructor and points to the beginning of the array. now you can create DMA for each instance of that array.



at the end of the function you will type

```
delete[] p;
```

This calls the destructor on each Vector element of p then it will free up the memory.

Operator overloading for arrays:

1st - []

- 1) takes two arguments
 - a) array to be indexed (outside the brackets)
 - b) integer indicating number of element to access)

in Vector class you have...

2nd - double & operator [] (int)

in the .C file you would have

```
double & Vector::operator [] (int)
{
    return data[i];
}
```

say that you want to create a constant Vector v2...

```
Vector v1;
const Vector v2;

cout << v2[3];
v1[5] = -1; // won't work - problem with l-value
```

if you change to 2nd way you could assign to v1[5] but now cout << v2[3] won't work because it's retrieving a reference which is an l-value. So... we have to create another operator subscript.

```
double & operator [] (int i);    // can change
double operator [] (int i) const; // current instance will remain constant
```

the first is not a constant instance but the second is. The second method overload is the same as the first but is returning a double.

we can use the first one to say v1[5] = -1;

how does this effect arrays of vectors?

```
Vector ar[7];

ar [3][5];
```

This gets element 5 out of an array of 3 Vectors. Not to be confused with C++ vectors so watch out!

Let's implement: (in Vector.h)

```
#ifndef VECTOR_H
.
.
.
class Vector
{
    int len
    double *data;
public:
    Vector();
    .
    .
    double & operator [] (int i);
    double operator [] (int i) const;
    ~Vector();
    friend ostream & operator <<(ostream &, const Vector &);
};
```

now in Vector.C

```
Vector::Vector()
{
    data = 0;
    len = 0;
}

/*****/
Vector::Vector(const double *data, int size)
{
    len = size;
    data = new double[size];

    for(int i=0; i<len; i++)
        data[i] = d[i];
}

/*****/
Vector::~~Vector()
{
    delete [] data;
}
```

```

/*****/
ostream & operator << (ostream & ostr, const Vector &v)
{
    for(int i=0; i<v.len; i++)
        ostr << v.data[i] << " ";
    return ostr;
}

/*****/
double Vector::operator [] (int index)
{
    return data[index];
}

```

Now the driver program so far...

```

#include <iostream>
#include "Vector.h"

double data[] = {...some numbers};

int main()
{
    Vector v1;
    Vector v2(data, sizeof(data); // gives size of (double)

    cout << v2[2];
    cout << "*" << v1 << "*" << v2;

    return (0);
}

```

say you have a function that takes a vector..

```

function (Vector v)
{
}

```

in the main ()...

```

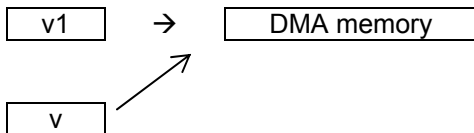
Vector v1;

function (v1);

```



it's not a reference so it's passed by copy. In calling the function it sets up memory for v including the value of the pointer so v points to the DMA memory of v1



When it is done with the function it calls destructor on v which kills memory allocated for v1. Now v1 points to something that doesn't exist. How can we fix this problem?? With the `copy` constructor.

Copy Constructor:

in class definition..

```
Vector(const Vector &); // what's different is the assignments
```

this is so that everything gets copied in, including DMA issues. Implement it in the .C file like this

```
Vector::Vector (const Vector &v)
// const because you're not changing value passed in
{
    len = v.len;
    data = new double[len];
    for (int i=0; i<len; i++)
        data[i] = v.data[i];
}
```

When is this called?? Three different places.

- 1) in variable instance creation when initializing from same type.

example:

```
Vector v1;
Vector v2 = v1;
```

- 2) copying variables of arguments

example:

```
func(Vector v);
```

- 3) copying return values

example:

```
Vector function ( ...)
{
    .
    .
    .
    return result;
}
```

this will copy into temporary location then cleanup is done then it will return value to variable. Most compilers will copy straight into variable if they are smart enough.

now look at this substitution...

Vector v1, v2;

v1 → DMA memory

v2 → DMA memory

v1 = v2;

what will happen?? now v1 points to v2.

v1 → DMA memory
↙
v2 → DMA memory

Problems:

```
float ar1[] = {3.0, 5.0, -1.2};
float ar2[] = {0.0, -7.5};

Vector v1 (ar1, 3);
Vector v2 (ar2, 2);
```

in memory would look like...

```
v1 [len = 3] → [ 3.0 | 5.0 | -1.2 ]
v2 [len = 2] → [ 0.0 | -7.5 ]
```

when you execute the statement...

```
v1 = v2;
```

there are two problems:

- 1) there is a memory leak
- 2) you have changed the value of where it points to.

Note: In any assignment operator there are five things you need to worry about. and they need to be performed in the following order:

- 1) check for assignment to self. If it is so, skip to step five.
- 2) delete/free memory associated with the instance
- 3) allocate new memory
- 4) make the copy
- 5) return *this (or this or current instance)

In Vector class...

```
Vector & Vector::operator = (const Vector &v)
{
    // step 1
    if (this == &v) // checks address of this instance and passed &v
        return *this;

    // step 2
    delete [] data;
    len = v.len;

    // step 3
    data = new float[len];

    // step 4
    for (int i=0; i<len; i++)
        data[i] = v.data[i];

    // step 5
    return *this;
}
```

notice that from the second line of step 2 through the assignment loop of step 4 is identical as the copy constructor.

The key difference is that the copy constructor assumes uninitialized instance whereas in the assignment operator, current instance is initialized because it is the left-hand side of the argument.

Example:

```
String s1 = "Hello"
```

This is not calling the assignment operator, it calls the constructor because s1 is not initialized.

```
String s2 = s1;
```

This calls the copy constructor because s2 is being created and not initialized.

```
String s3;  
s3 = s2;
```

First calls the default constructor for the first line. Then calls the operator assignment on the second line because it's already been initialized by the constructor.

Point 1 Explained:

say for example the following is set:

```
Vector v1(ar1, 3);  
Vector & v4 = v1; // any change to one will go to other too  
Vector v5;
```

what if you say `v4 = v1` or `v5 = v5` ? (assignment to self)

Step through the overloaded operator= starting with step 2.

- 1) omit step one for reason of explanation
- 2) delete memory
- 3) allocate new memory
- 4) copy?? Where can you copy from. The memory is deleted.

To fix this you need to check when you're assigning to self, do nothing. Hence, Step 1

Point 5 Explained:

recall what we know about chaining...

```
s1=s2=s3=s4; // chaining always evaluates right to left
```

this is the same as:

```
(s1 = (s2 = (s3=s4)));
```

because of this, we need the same in operator= overloading. so.. the return type is a reference to the same type.

```
Vector & Vector::operator = (const Vector &v)
```

this explains the need to return a de-referenced this in step 5 above.

To concatenate two vectors together:

```
Vector v5; // assume initialized and containing (6, 8, -9, -1.2)  
Vector v6; // assume initialized and containing (1, 2)  
Vector v7;
```

```
v7 = v5+v6; // want v7 to contain (6, 8, -9, -1.2, 1, 2)
```

if `v8 = v6`; and you want to code `v8 = v8+4.0`; and also `v8 += 3.0`; we need to overload the +operator. This will be done in two versions.

Version 1 (for v8 = v8 + 4.0)

```
Vector Vector::operator+ (float f) const
{
    Vector result;

    result.len = len + 1;
    result.data = new float[len+1];
    for(int i=0; i<len; i++)
        result.data[i] = data[i];
    result.data[len] = f;

    return result;
}
```

Version 2 (for v7 = v5 + v6)

```
Vector Vector::operator+ (float first, float second) const
{
    Vector result;

    result.len = first.len + second.len;
    result.data = new float[result.len];
    int i=0;
    for(; i<first.len; i++)
        result.data[i] = first.data[i];
    for(; i<result.len; i++)
        result.data[i] = second.data[i];

    return result;
}
```

operator += (for v8 += 3.0)

This is changing current instance by making DMA larger and tacking on another element. You can't free up memory first. You must copy over the data in v8 to a new pointer and then reallocate memory for v8 and copy original data into new memory and now add the 3.0. Then, and only then, free up original memory for v8 which was in the new pointer.

```
Vector Vector::operator +=(Vector v)
{
    // create temporary instance
    Vector temp;
    temp.len = len + v.len;
    temp.data = new double[temp.len];
    int i=0;

    // make copy
    for(; i<len; i++) // note empty first space, i is already zero
        temp.data[i] = data[i];

    // add the second vector
    for(; i<temp.len; i++) // i picks up where left off
        temp.data[i] = v.data[i];

    // re-allocate memory for original
    delete [] data;
    data = new double[temp.len];

    // copy back into original
    for(i=0; i<temp.len; i++)
        data[i] = temp.data[i];
    len = temp.len;
    return *this;
}
```

List: (new ADT) Can find example in Data Structures Book page 112-113

list:

- A List is a sequence of objects. All are of the same type similar to an array with one exception.

array - next to each other in memory

list - maybe yes but maybe no.

operations of a List:

- clear a list
- check if it's full
- check length
- retrieve an item from
- insert an item into
- delete an item from
- reset the list

Now we need to create and implement methods for these operations. Detailed examples are listed on page 119 of the Data Structures book.

```
const int MAX_SIZE = 100;

class IntList
{
    int data[MAX_SIZE];
    int count;    // counter
    int cpos;    // current position

public:
    IntList();
    void MakeEmpty();
    bool IsFull() const;
    int LengthIs() const;
    void Insert(int);
    void Retrieve(int &, bool & found);
    void Delete(int);
};

/*****/
IntList()
{
    count = 0;
    cpos = 0;
}

/*****/
void IntList::MakeEmpty()
{
    // same as constructor but can be called at anytime
    cout = 0;    // resets count to zero
    cpos = 0;    // resets current position to zero
}
```

```

/*****/
bool IntList::IsFull() const
{
    return (count >= MAX_SIZE);
    // can do this because results of relational expressions are bools
}

/*****/
int IntList::LengthIs() const
{
    return count;
}

/*****/
void IntList::Insert(int item)
{
    if (count == MAX_SIZE)
        return;
    data[count] = item;
    cpos = count;
    count++;
}

/*****/

```

What is it that we want to do in the retrieve function? Assume that you have a list as follows:

2	6	5	3	9
---	---	---	---	---

if we want to find the position of 5 we start at the beginning and step through list until we find it. If it's not in the list we exit.

```

void IntList::Retrieve(int &item, bool &found)
{
    int index = 0;
    while(item != data[index] && index < count)
        index++;
    if (index == count);
    {
        found = false;
        return;
    }
    item = data[index];
    cpos = index;
    found = true;
}

```

Note: You can also return a bool and "86" the bool &found in the header if you would like. That would be my reference but the book does it this way so we'll stick with it for now.

Now we need to do some removal and cleanup...

If you want to delete the item 6 in the list above, you have a problem if using an array. What you need to do is to delete the item and then "shake down" all remaining items one position to create a list containing the integers (2, 5, 3, 9).

This will take three parts:

- 1) does the element you want to delete exist in the list
- 2) delete the element
- 3) copy or "shake down" items over

```

void IntList::Delete(int item)
{
    // first do the search
    int index = 0;

    while(index < count && item != data[index])
        index++;

    // if none is found do nothing and return
    if (index == count)
        return;

    // found so perform move
    cpos = index;
    count--; // fix boundary position because of deleted item
    while(index < count);
    {
        data[index] = data[index+1]; // shake down
        index++;
    }
}

```

You can also make a second Insert method which will insert an item at a certain position of the list. To do this you need to move to the end, add one position and do a position swap moving backwards until you find the position you want to insert the item into.

```

void IntList::Insert2(int &item, int pos)
{
    // test for room to insert an item
    if(count == MAX_SIZE)
        return; // can't insert because there isn't room

    // check for other errors
    if(pos < 0 || pos > count)
        return;

    // create an open position
    while(index > pos)
    {
        data[index] = data[index-1];
        index--;
    }

    // insert new item into list
    data[index] = item;
    cpos = pos;
    count++;
}

```

C++ string class

You have a string class available to you in C++ and can access it using `#include<string>` (Note that it is not `<cstring>`. `cstring` is for C style strings which are different as follows:

C string - null terminated

C++ string - not null terminated. they contain...

1. pointer to character array
2. length of string (properties same as what we have been doing)

Note that when you declare a C++ string you use all lowercase characters (string NOT String).

There are many things you can do with C++ strings. For example:

Declaration and Assignment

examples:

```
string s;           // empty string with length of 0
string s1 = "Hello"; // can initialize as array of characters
string s2 = s1;     // can initialize to another string
```

comparison operators (==, !=, <, >, <=, >=). All will return bool.

examples:

```
string s3;
cin >> s3;
if (s3 == "sort")
    ...do sort routine
else if (s3 == "print")
    ... do print routine
```

Multiple Constructors available

examples:

```
string s4(7, 'a'); // makes string = "aaaaaaa"
string s5 = "Frodo"; // uses C style string
string s6 = s5;
string s7(s5, 3, 2); // result will be "do"
```

s7 will use s5, starts at element 3, goes length of 2. *Remember* - always use **zero based indexing** so just remember (source, starting point, number of positions)

```
char msg[] = "This is fun!";
string s8(msg+5, 2); // result is "is"
string s9(msg, 5, 3); // essentially the same thing
```

they will point to the 5th position of msg ("i") and take two characters ("is")

This idea of two integer arguments (position and length) is common in C++ strings but beware! This is a common source of errors.

operator =

examples:

```
string s1,s2;
s1 = s2;
s1 = "second"; // converts C style string to C++ string
s1 = 'a'       // this will convert character to C++ string
               // because s1 is already declared
```

BUT...

```
string s3 = 'b'; // ERROR - can't declare and initialize this way
```

string class functions:

.c_str()

Takes no arguments and returns a pointer to a C style string. It enables you to access characters in a data array and change them. The best way to use this is when you need to read, like using atof...

example:

```
string s4 = "123.4";
cout << atof(s4.c_str());
```

This takes s4 and converts it to a C style string and passes it back to atof which converts it to a float which can then be used in calculations.

operator[]

example

```
s4[1]          // will return character #1 ("2")
s1[0] = 'X'    // can do but be careful with []
```

There are two forms of subscript overloading same as in the class we wrote so just use them whenever you normally would

operator +

you can use for:

```
string + char
string + string
string += char
string += string
```

.append(char) or .append(string)

This will append to the end of a string. Use as follows:

```
string s4 = "zig";
s4.append("zag");    // will give you "zig zag"
```

.insert(int, string)

can have many forms. It allows you to place other strings anywhere inside a string. It has the form:

```
.insert(int position, string)
```

Note: position - start inserting BEFORE this integer position

forms of the .insert function:

```
.insert(int pos, string);           // inserts all of string
.insert(int pos, string, int, int) //from string, starting..., for ...chars
.insert(int pos, char *)           // inserts C style string
.insert(int pos, char* int)        // int chars of C style string
```

examples of .insert

```
string s1 = "hot dog";
s1.insert(4, "diggety");
```

```
cout << s1;    // yields - hot diggety dog
```

```
string s2 = "slam";
string s3 = "grand";
```

```
s2.insert(0, s3);    // will make "grand slam"
s2.insert(0, s3, 2, 4); // makes "and grand slam"
```

.replace

We can also replace substrings within strings (like changing a name in a message that goes out to several people)

```
.replace(position, length, replace_with)
```

you can use a C style string or C++ string in the replace. It will adjust the string for different length of replacement.

```
examples of .replace
string s5 = "High dive!";
s5.replace(5, 4, "jump");           // "High jump!"
s5.replace(5, 4, "interest rate"); // "High interest rate!"
s5.replace(5, 4, "sky");           // "High sky!"
```

.erase(pos, len)

Will erase a given number of characters starting at assigned location and adjust length of the string

```
.erase(position, length);
```

so now...

```
s5.erase(5, 4); // "High !"
```

.find(string)

Returns an integer indicating location of the first occurrence of a string.

examples:

```
s6 = "Ooga Booga";  
s6.find("Boo"); // will return 5 (case matters!!)
```

```
s6.find("Book", 3, 1);
```

this will search using the 3rd position of Book for length of 1 character ("k") and will exit when not found.

.length() and .size()

Both functions do the same thing and return an integer which is the length of the string.

.rfind(string)

Does the same as find except it will start from the end of the string and search backwards.

example:

```
string s7 = "High road";  
s7.rfind("gh") // returns 2 as the position
```

This is useful if...

```
string s8 = "www.cs.niu.edu";  
int p = s8.rfind('.'); // will find the last period and returns value 10
```

then you can use...

```
s8.replace(p+1, 3, "com"); // makes "www.cs.niu.com"
```

You could also code:

```
s8.replace(s8.rfind('.')+1, 3, "com");
```

The second way works but is not a good idea. You can get into trouble because order of evaluation is not portable. It may work on the machine you write the code on and not the machine you implement it on. If you don't find '.' on the second way you will get a bogus position so it's best to split up the function calls and always do error checking.

.empty()

This function will takes no arguments and returns a bool indicating if the string is empty or not.

.substr()

The sub-string function is used to extract a sub-string within a string. It takes as arguments, an integer indicating position and a length of characters and returns a string

example:

```
string s1 = "submarine";  
string s2;  
s2 = s1.substr(3,3); // this extracts "mar"
```

additional information:

you can also use `cout << s4` and `cin >> s4` with C++ strings

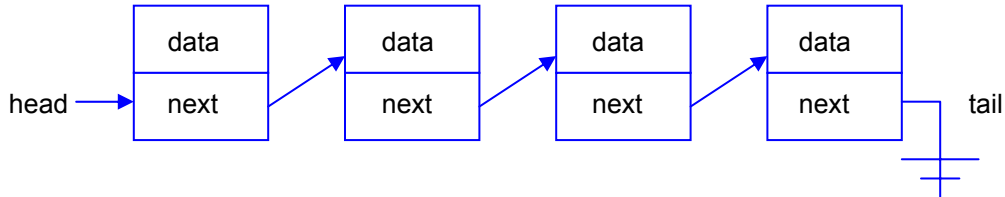
Linked Lists

Recall the properties of a list which were covered earlier. There are a few drawbacks.

- 1) insertion into a list required moving items to accommodate the insert items.
- 2) in doing this it is possible to exceed the array size.

This leads to a new version of the list using DMA called a linked list.

The idea: (using a singularly linked list)



Node:

a node consists of data and structuring information(head, next, tail)

Things to do with nodes:

- find
- create
- remove
- cleanup

Let's create a node in C++ form:

```
class Node
{
public:
    int data;
    Node *next;
    Node(int d=0, Node *p = 0); // constructor with default args.
};
```

Note that one of your data members is Node which is the type that you are currently creating. This is a special case where this is permitted.

```
Node::Node (int d, Node *p)
{
    data = d; // this creates a completely
    next = p; // initialized Node
}
```

The idea of Linked lists is that you can allocate memory as you need them "on the fly". Nodes are pointers. To implement you would use the following as an example:

```
Node *head = new Node;
head->next = 0; // same as (*head).next
Node *n = new Node;
n->next = 0;
Node *t = head; // declares and sets to Node head
while(t->next != 0)
    t = t->next;
t->next = n;
```

There is one major difference between a class and a struct in C++ :

```
class - by default, all members are private
struct - by default, all members are public
```

knowing this information and wanting to be able to access the nodes within a list, lets write a struct called LNode:

```
struct LNode
{
    int data;
    LNode *next;
    LNode(int d=0, LNode *n) // uses default arguments
};

LNode::LNode(int d, LNode *n)
{
    data = d;
    next = n;
}
```

Now let's write the List Class:

```
class List
{
    LNode *head;

public:
    List();
    ~List();
    int insert(int record); // return is error code
    int retrieve(int record); // return is error code
    int remove(int record);
    void clear();
    bool empty();
};
```

we won't write out the constructor or the empty functions again. Just need to know that:

constructor - sets head to zero

empty - checks if *head is NULL. If so, it's empty and returns true

clear()

Let's first concentrate on the clear(). It will be responsible to get rid of all the elements in the List. It will go down the list one by one and free up. *** keep track of the pointers!! ***

- 1) set the additional pointer and point at each node.
- 2) use that pointer to get at the next thing in the list and mark head.
- 3) then delete the node

```
void List::clear( )
{
    LNode *rest; // meaning rest of the list

    // check for empty list already
    if (head==0)
        return;

    // loop and delete
    while (head) // NULL would return false and indicate empty
        rest = head->next;
    delete head;
    head = rest;
}
```

Note: Once you delete memory it's free game for use by other persons or program executions on the computer. Watch where you place your delete in this case. It may not be a problem but if something else takes that memory space before you retrieve the information, you're toast. It's good practice to get the pointers straightened out first and then to delete memory.

You also need a destructor that does the same thing.

```
List::~~List()
{
    clear();
}
```

You can often code like this when you are using DMA

There is a positive side effect of using the head pointer in the function above. By using it and "stepping back" as we delete, when we kick out of the loop head = NULL so it has re-initialized back to zero.

There is another way to implement the destructor using recursion.

Recursion: recursion is an analytical tool of problem approach. *It is a function that calls itself.* an example of a recursive function is as follows:

```
void demo(int a)
{
    int b;
    demo (a+1);
}
```

Look closely. This function will cause BIG problems if you were to run it. What really happens is that when the function is called it will allocate memory. The recursive nature will continue to call the function on itself and continue to allocate memory for each additional instance. The last call will be executed first so the memory keeps being chunked away until all system resources are annihilated

- Q) So how can we make recursion useful for our purposes??
- A) It allows you to break up problems into smaller problems of the same type.

Example: n!

$n! = n * (n-1) * (n-2) * (n-3) * \dots * 3 * 2 * 1$ so we could then say...
 $n! = n * (n-1)!$

when coding this **you must think about where you are going to stop and how to do it**. You will stop when you reach 1! which is equal to 1. That is the smallest you can encounter.

implementation:

```
int factorial(int n)
{
    if(n==0)
        return 1;
    return n * factorial(n-1);
}
```

This is key: Remember back when we were looking at memory used by each one of the function calls recursively, it was setting up a whole new memory space for the local variables of that function. Even though we are passing new values into the factorial function which are going to be assigned to the int for that invocation. When you come back from that function the number it is going to pass back as the value of factorial(n-1) the value of n in this invocation will remain unchanged. Now, we can multiply the value of $n * factorial(n-1)$.

- Q) Why are we bringing up recursion now?
- A) Because deleting a list can be turned into a recursion.

Recursive version of deleting list: (recursive clear) You want to put this in the private section so other people can't call this. They shouldn't even know it's there.
 You are going to want to stop this recursion when the pointer is null

```
List::r_clear(LNode *n)
{
    if(n==0)
        return;
    r_clear(n->next);
    delete n;
}
```

BUT.. to use this we must first re-write the clear () too. The r_clear () is a private function so you need to have a way to call it.

```
List::clear()
{
    r_clear(head); // this jump starts the private method r_clear
    head = 0
}
```

Creating a List (inserting nodes)

When you want to insert a node into a list, you need to know these things:

- 1) is the list sorted or unsorted?
- 2) do you have control of the location of the insert?

Let's implement a version tacking on a node to the beginning of a list. Remember that this operation is all about changing pointer values. We are going to create another piece of dynamically allocated memory and then change the pointers. The order of operation is extremely important so you don't lose something along the way.

Inserting at the beginning of a list:

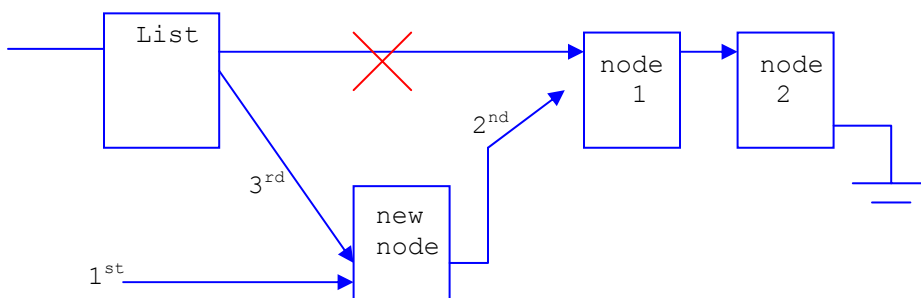
```
void List::insert(int record)
{
    LNode *p;

    p = new LNode(record) // using default arguments here
    p->next = head;
    head = p;
}
```

What's really going on here?

1. you are creating a pointer and assigning it DMA for the size of the record
2. you are telling the pointer to point to what head points to
3. you are telling head to point to the new pointer you created.

Here is a visual example of what has just happened and the order of completion:



Inserting at the end of a list:

The only real difference for inserting at the end of the list is that you have to loop through the list and check to see when you are at the end. You need to start at the beginning of the list and step down to the bottom. This is done in the following steps:

1. You should create two pointers. One as a search pointer and one for your previous location.
2. start off with search = head and pred = 0.
3. use the slinky effect to safeguard against Buss errors which are caused by "stepping off the end" of your list.

Once you get to the end of the list (where search ==0) then you know you are at the end of the list and can insert the node in the position of the pred->next. Always be careful of the order of switching the pointers so you don't lose your nodes.

Example:

```
Node *search = head;
Node *pred = 0;
Node n;
n = new Node(pass arguments here);

// check to see if DMA created
if (n==0)
    return ERROR_CODE;    // return error code

// search for end of list
while (search !=0)
{
    pred = search;
    search = search->next;
}

//check if head of list
if(pred==0)
{
    n->next = search;
    head = n;
    return OK;    // return error code
}

// tack onto end of list
n->next = search;
pred->n = n;
```

This code will take care of inserting a node at the end of a list. It can be easily modified to insert at any given place in the list by changing the search condition and adding additional checks. Always remember to check for errors and all possible conditions. It's really easy to lose information when changing the pointers so be sure where you are before you change them.

Deleting a node from a list:

A deletion from a list is accomplished in much the same way as the insert. The only difference is in your pointer manipulation once you find the node to remove. Step through as follows:

1. create two pointers (search and pred) and step through list same as in the insert.
2. once you find the node to delete you set the pred->next to the search->next
3. delete the search

make sure you do it in this order otherwise you will lose all nodes after the one you are deleting. The following is an example of deleting from the end of a list.

Example:

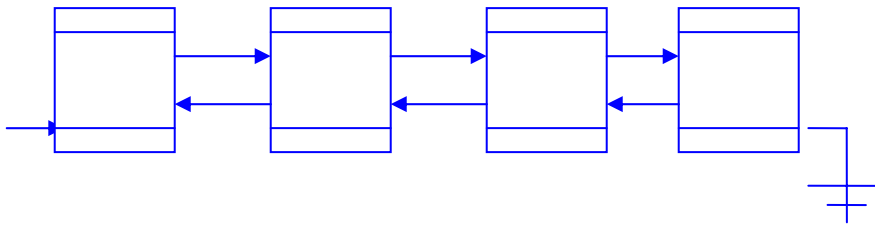
```
{
Node *search, *pred;
search = head;
pred = 0;

//search for end of list
while(search!=0)
{
    pred = search;
    search = search->next;
}
// check for at head of list
if(pred == head)
{
    head = search->next;
    delete search;
    return OK; // Error code
}

pred->next = search->next;
delete search;
}
```

Doubly Linked Lists:

If you create a doubly linked list it is much the same as previously mentioned except for the fact that each node has a previous and well as a next pointer. It looks like this:



Example:

```
struct Node
{
    T data;
    Node *next;
    Node *perv;
};
```

deleting an entire list is done recursively the same as with a singly linked list.

Inserting into Doubly Linked List:

1. Search through your list same as before until you establish the position you wish to insert.
2. establish links from the new node first
3. then and only then sever the original links.

Don't let it confuse you. Just remember that each node now has a previous and a next pointer.

Inserting into the middle of a list takes making the pointer switch like this:

```
// connect new node
n->next = search;
n->prev = search->prev;

//now sever ties
search->prev->next = n;
search->prev = n;
```

Inserting at the head of a list takes making the pointer switch like this:

```
n->next = head;
n->prev = 0;
head->prev = n;
head = n;
```

Inserting at the end of a list takes making pointer switch like this:

```
// search to find end
while(search!=0)
{
    pred = search;
    search = search->next;
}

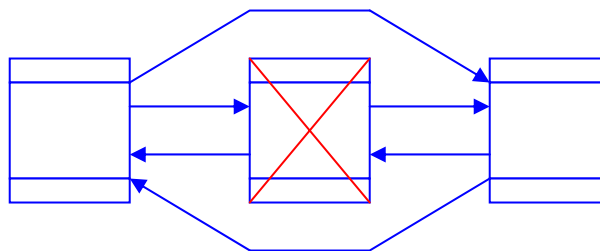
n = new Node();
n->prev = search;
n->next = 0;
search->next = n;
```

Removing Node from Doubly Linked List:

When you are deleting a node from a doubly linked list you only have two pointers to take care of. If you have three nodes named 1, 2, and 3 respectively and you want to remove node 2 from the list, you need to take the next pointer from node one to point to node 3 and the previous pointer from node three to point to node 1. This is accomplished in the following manner using only one search pointer:

```
search->next->prev = search->prev;
search->prev->next = search->next;
delete search;
```

This is diagramed below:



Stacks and Queues:

Stack: (FILO or first in last out)

Stack ADT definition:

- A stack is a linear sequence of data of the same type. It can be likened to a stack of plates at a diner. The plates are taken off in a “last in, first out” order.

operations of a stack:

- they have top and bottom
- can put on top and take off top ONLY

key words to remember:

- push** – to push is to place an item on the top of a stack.
- pop** – to pop from a stack is to remove an item from the top.
- top** – refers to the top of the stack.

Pop does not return the value of the top of stack. It just removes it.

Example of Stack class:

```
class Stack
{
    char ar[50];    // type of data in stack
    int n;         // number of elements in stack

public:
    Stack();
    void push(char);
    void pop();
    char top() const;
    bool empty() const;
    bool full() const;
    int size() const;
};
```

Note: because this is a linked implamentation you don't have to worry about copy constructor, destructor, assignment operator.

```
/*******/
Stack::Stack()
{
    n=0; // says no elements in array
}
/*******/
bool Stack::empty()
{
    return (n==0); // evaluates ==0 for true and != 0 for false
}
/*******/
bool Stack::full()
{
    return (n==50); // or use symbolic constant
}
/*******/
char Stack::top()
{
    if (n==0)
        return '\0'; // returns arbitrary value
    return ar[n-1];
}
```

Note: we can't put in n because n refers to the first element available. We have to return the element at n-1. When we set a counter it is on a 1 based index but remember that the stack is on a 0 based index.

```
/******  
void Stack::push(char c)  
{  
    if(n==50)  
        return; // stack is full  
  
    // insert onto stack  
    ar[n] = c; // or optional ar[n++] = c  
    n++;  
}  
/******  
Stack::pop()  
{  
    if (empty)  
        return;  
    n--;  
}
```

Post and pre incrementing (++ and --):

Note that in the push function we used the n++ inside the subscripted array. It is important to note that there are two versions of both of these operators (n++ or ++n). The difference between post and pre increment is n++(post-increment). Every expression in C returns a value. The question is when is it returned. In a post increment situation it will return the value of n before the increment and then the value can be used in the rest of the expression. Pre-increment (++n) will do the increment first and then return as a value of the expression the new value of the variable.

Enumerated Data Type:

keyword: enum

an enumerated data type is that you are counting all the objects. It is similar to a set.

Anonymous enumerated data type:

Example:

```
enum {LIST_OK, LIST_NO_MEM, ...}
```

you don't need to specify numbers for the error codes. That is done automatically. The first one is automatically assigned the value of "0" and the next on "1" and so on. (LIST_OK = 0, LIST_NO_MEMORY = 1, ...)

Later on in the program:

```
int err;  
.  
.  
err = LIST_OK;
```

enumerations can also be given names.

Named enumerated data type

Example:

```
enum ErrorCode {LIST_OK, LIST_NO_MEM, ...};
```

This is a completely new data type. Now you can create functions that return ErrorCode's

```
ErrorCode err = LIST_OK;
```

You can also choose your own values for enumerations:

Example:

```
enum set{A=8, B=10, C, D};
```

if a value is not set, the value takes up where you last specified a value. For instance, in the example above, C=11 and D=12 because the last value you specified was for B.

Another interesting fact is that you can double-up values.

```
enum set{A=-1, B, C, D=0};
```

B, by default is given the value "0" but you can still specify that D also has the value of "0".

How can this be useful to us in a specific way? Think about the situation when you have a class named Date.

```
class Date
{
public:
    enum Month {Jan=1, Feb, Mar, Apr,... Dec};
    int day;
    Month month;    // must place after enum setting
    int year;

    set_date(Month, int, int);
};
```

Now, if someone wants to use these month symbols outside the class method, it would be done in the following manner:

```
Date d;

d.set_date(Date::Apr, 17, 2001);
```

With this information we can definitely get around the array size problem we had earlier where we declared a const int outside the class.

Example: (recall the String class we created)

```
class String
{
    enum {max_size = 100};    // note that it is private
    char data[max_size];

public:
    func();
};

/*****
String::func()
{
    . . .
    a = max_size;    // use just as expected
}
*****/
```

Note that the enumerated max_size is not an L-value and you cannot assign values to them

Ways of allocating memory:

automatic variables:

keyword: auto

automatic variables are declared but memory for those variables does not exist until that function is called. When the function is called that contains them, the memory will be allocated. This is why we can get away with recursion, because every time a function is called, it sets up a new copy of those variables

Example:

Standard implementation

```
func()
{
    int a;
    int *p;
}
```

Alternate Implementation

```
func()
{
    auto int a;
    auto int *p;
}
```

global variables:

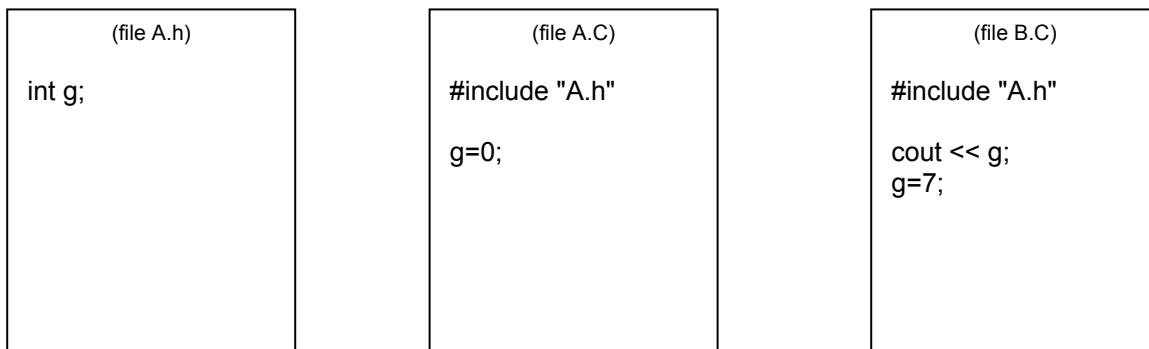
Global variables are variables you can have outside of a function. They are available to everybody. global variables are static variables.

local variables:

variables that are inside a function are called local variables and can only be used within the scope of the function. They are dynamically created when the function is called. There is a very subtle difference between this and the dynamic memory allocation when you call new or malloc.

static variables:

a static variable is a global variable which is set up at the beginning of a program. They have no idea who is going to use them first. They are called static because their allocation is not dynamically changed.



If you set `g=0` in A.C you want to be able to print out and set `g` from within B.C. You want a global variable available to both A.C and B.C so that it can be shared back and forth. The only problem is... Where is the memory for this? When you compile and create object files for A and B, it says that there is a global variable in A.C and so it creates memory for it. Likewise in B.C the compiler creates static memory for the variable. So, both A.o and B.o have memory set aside for `g`. Which memory is correct??

As it turns out, both are correct. This means that when we make changes to `g` in A and then do the same thing in B, B is going to access something entirely different. How do we get around this problem?

Two keywords: are available in C and C++

extern -

static -

extern - keyword that you put in front of a variable declaration which tells the compiler there is a symbol of this type but the memory is somewhere else.

so in A.h declare g as: `extern int g;`

now when A.C sees it and B.C sees it, they don't allocate memory automatically.

****Note that you must have memory declared in one and only one of the source files also.**

You would think that this would be a compiler error because you are declaring the variable in both the A.h and the A.C file but in this special case it is not.

static - when applied to a global variable means that this symbol can only be seen inside this source code file.

is also useful when you want a variable inside a function to preserve it's value between the times it gets called and in the function is the only place that you are going to use it. All variables in a function are destroyed when you leave the function. BUT if you declare the variables inside the function as static, they be used just like any other variable with the exception

that the memory associated with that function is in the static memory space. In other words, the memory that is initialized when the program is started and not when the function is called. It's like a global variable but that symbol is valid *only* within *that* function. It will only be initialized once.

```
M.C
f1( );
{
}
f2( );
{
}
e( ); // want unseen by
others
```

if you want a function such as e() to be made unseen by any other function including source code that has included the M.h file, you need to make it a static function.

Example of the use of static:

say that you have a function called f();

```
f( )
{
    int a;
    static int b;
    .
    .
}
```

memory associated with the function symbol is valid only in the function.

initializing it here (b=0) will initialize it only once. How does this effect us with classes?

Example:

```
class W
{
    static int a; // takes this member and allocates it out of static
                // memory before the program is started
};
```

This will allocate static memory before the program runs. You may wonder how it does this if you don't know how many instances of the variable you will have. The answer is you don't. This member has only one instance of that member made for the class.

Note that there is only one instance of 'a' for the entire class. All instances of the class have the same 'a' in common. Whenever anyone accesses that member 'a' they will get the same value as anyone else who accesses it.

But there is a problem. You can't initialize it within a class so how do you initialize it?

If you declare a static data member of a class, you need to allocate memory for it. Before the constructor and the destructor in your source code file:

```
static int W::a = 0; // allocates memory and initializes
```

If you are dealing with constants you can declare it as follows:

```
class W
{
    static const int SIZE = 50;
    int ar[SIZE];
    ...
};
```

Note: this is the ONLY time you can initialize a member within a class. (if it is a static const)

You can initialize within a class if you declare it as a const int. The SIZE will then remain in the class. This declares a constant variable without having to be a global constant. This is a better way than using enumeration for declaring constants.

static says that nobody outside this source code file knows about this instance of the variable.

Sample code for this section to be posted on the official course site under [test13.C](#) and [test13a.C](#)

You can't have a const int in a class because it will be initialized for every instance in class. You can't do this. But if you declare it as static... for example:

```
static const int SIZE = 50;
```

then it allows you to have the constant variable declared within the class without being a global const int. If you code:

```
static int a: // need to declare to make memory
static const int a; created here
.
.
enum {SIZE = 50};
```

Key Note:

- Memory for static variables is created when program is run not when function is called
- Static variables *don't* go away when you leave the function.

Queue: (FIFO or first in first out)

- an abstract data type which is a sequence of data of the same type
- it has a front and a back.
- works well for list of things to do like print spoolers
 - 1) array implementation is for a set size of queue
 - 2) linked list implementation which are much more efficient

functions of a queue:

- push() to push a value onto the queue.
- pop() to pop a value off of the queue.
- front() similar to top() in a stack so you can examine.

In an array implementation of a queue you can push something onto the front of the list and then push another element on. You can keep track of the list size.

Problem with the array implementation is that you can keep track of size but when you pull off the front, you have to move all other values down. This takes time and resources.

Solution is to keep track of the first and last numbers in the queue. As you add in, the back moves. And as you pop things off, the front moves as well. This is still a problem because you eventually run out of room in the queue. Or... an even better solution to this problem is a circular queue. This brings us to yet another new data structure:

deque: (pronounced deck)

a deque is a linear data structure with a sequence of elements. It is a special case of a queue has both a front and a back. It is essentially a double ended queue. The main difference is that you can push and pop off of either the front or back.

Templates:

recall back to the sorting routines we wrote way back. There are many times that a sort could be used for several different types of data. This is where generic programming comes into play.

A template is related to a macro in that in a C macro, you can use #define to replace a given symbol with a set value throughout a program.

Example of C macro:

```
#define sqr(x)    x*x
```

this is a function that will return the square of an input. To use it you would code the following:

```
int a = 7;
cout << sqr(a);
```

when this is compiled, the compiler will replace the (a) with (a*a)

what if you coded:??

```
int c,d;
cout << sqr(c+d);
```

You would think this would work but in fact it will not give you the result you may think. Look closely... when it is compiled, the compiler replaces what is within the parentheses with what is specified in the macro. In this case it would be (c+d*c+d). Due to the hierarchy of operands this would be equivalent to saying (c+(d*c)+d) instead of ((c+d)*(c+d))

This leads us to the use of templates. Lets recall a standard swap function:

Standard function:

```
void swap(string &a, string &b)
{
    string temp;
    temp = a;
    a = b;
    b = temp;
}
```

Now to convert this to a template, the only thing we need to change are the highlighted variables; A template is basically a macro on steroids. It will even let you take entire functions and classes and treat them as macros.

Template:

```
template <class T>           // must ALWAYS have this line for a template
void swap(T&a, T&b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

Note that whenever you see 'T' , it represents a type of some sort. Whenever it is called it will insert the type of the call and generate code for a swap of that type. This won't compile at this stage. You first need to create an instance for it's use because the compiler doesn't know what the type T is.

Templates will go in your .h file (or above the main() in a small coding where you would generally put your class definitions. Then in the main() you can code:

```
main()
{
    int a,b;
    double c,d;
    string e,f;

    swap(a,b);
    swap(c,d);
    swap(e,f);
}
```

When this compiles, the compiler will see the first swap function call and determine that a swap function is needed for integers. Since it doesn't find one but does find the template, it will then generate the code for an integer swap function. Likewise, on the following two lines it will do the same for a double swap and a string swap. It will only generate code for templates when it is needed.

There is another nice aspect of using templates. Say you wanted to swap (e,b) which is a string and an integer. It won't let you do this because your template function header states that it is taking a T &a, and a T &b. So the two types have to be the same. Trying to do this would give you a compile time error.

But.. You can write a template to take more than one type.

Example:

```
template <class T, class S>
func(T &a, S &b)
{
    whatever code ...
}
```

Note: your symbols for the parameters can be any legal variable but T is by far the most common one used.

Let's go one step further and make an entire class into templates (List class) so we can use it for integers, doubles, strings, or CD's.

Standard List class

```
struct Node
{
    int data;
    Node *next;
};

class List
{
    Node *head;

public:
    List(); // constructor
    List(const List &); // copy constructor
    List & operator =(const List &);
    ~List();
    ...
};
```

Template List class

```
template <class T>
struct Node
{
    T data;
    Node <T> *next;
};

template <class T>
class List
{
    Node <T> *head

public:
    List <T>();
    List <T>(const List <T> &);
    List <T> & operator =(const List <T> &);
    ~List <T>();
    int insert(const T&);
};
```

Now we have a template class definition which we can use for any type of data but we need to implement the functions...

```
template <class T>                                     // constructor
List <T>::List <T>()
{
    whatever you need to initialize
    head = 0;
}

template <class T>
List <T>::List <T>(const List <T> &)                 // copy constructor
{
    ...
}

template <class T>
int List <T>::insert(const T & value)
{
    ...
}
```

Notice that in the insert function you are just passing in a const T instead of List <T>. This may seem confusing at first but remember that you are passing a data type such as a double or a string but you are using it in a template so it is of class type T. This is different than in the copy constructor where you are passing in a reference to a List. In that case you need to say what type of List you are passing. Hence the parameterized List<T>.

Important Note: This won't compile in the C file like normal. You must put all code for templates in the .h file including class definition, implementations, ...

Now in your Test.C file...

```
#include "List.h"
...
...

main()
{
    List <int> list1;
    // when you get here, the compiler will break loose
    // and start to create a list class for integers but it will
    // only generate code for parts of the List class that
    // are needed in this program.

    List <string> list2;
    List <CD> list3;
    list1.insert(3); // generates insert function for int now
    list2.insert("Hello");
    list3.insert(my_cd);
}
```

<p style="text-align: center;">List.h file</p> <p>header guards global symbolic constants function prototypes template class definition</p> <hr/> <p>include definitions of all template functions and code including member functions of the template class.</p>
--

In each case, the compiler will generate a new function for each type of List but since it's a template, the compiler will do it for you automatically. Now you can see how you can take advantage of the fact that you don't need to change any code.

How does this effect your Makefile?

You can't compile a template unless you have a particular type to compile.

```
test:test.o
    g++ -Wall -o test test.o

test.o:test.C List.h
    g++ -Wall -c test.C
```

Sorting:

You may think that one sorting routine is better than another but the speed of the sort really depends on what you are sorting.

There are two types of sorting 1) internal sorting 2) external sorting

Internal sorting is done in the main memory of the program while external sorting is done by bringing in and sending data to a file of some sort. In this class we will only be dealing with internal sorting.

A sort is an algorithm that deals with a sequence of elements. It can be dealt with in two ways.

- 1) as an array
- 2) as a linked list

functions in sorting

	<u>internal vs external</u>
representation:	array vs. linked list
data:	sorted vs. unsorted data

we are going to deal with array sorting.

Bubble Sort:

be careful when using a bubble sort. It has advantages in only a few cases but in general is a lousy sort. We use it because it is fairly easy to understand how it works.

You start with an array of numbers. (5,3,7,9,0)

You move down the list until you find one out of order and change it. In this instance you swap position of the 5 and 3 which gives the array (3,5,7,9,0)

you then continue down the array until you encounter another instance which needs to be swapped. This happens when you get to the 9 and 0 (3,5,7,0,9)

This completes the first pass. Continue until the smallest number (0) has "bubbled up" to the top of the list. The list then moves to loop starting at the next number in list and continues through the entire array.

Let's code this using templates for practice:

```
#include <iostream>
#include <cstdlib>

template <class T>
void swap (T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

template <class T>
void bubble_sort( T ar[], int n)
{
    bool swapped = true;

    while (swapped)
    {
        swapped = false;
        for (int i=0; i<n-1; i++) // note n-1 because of checking i to i+1
            below
            {
                if (ar[i] > ar[i+1])
                {
                    swap(ar[i], ar[i+1]);
                    swapped = true;
                }
            }
    }
}
```

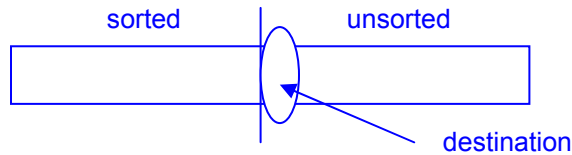
You can alternate the outer loop of a bubble sort to cut down the number of iterations it performs by replacing the while loop with:

```
for (end = n-2; end >=0; end--)
{
    for (int i=0; i<= end; i++)
    {
        same code as before...
    }
}
```

Now you don't have to go through the entire loop each time. It will go one position less each time through.

In general, a bubble sort is something you would really never want to use because it is an n^2 algorithm. The only time you would really want to use this is when you are sorting a list that is already sorted and are inserting something into it. In this case the modification we showed above would fail.

Insertion Sort:



An array is broken into two parts. Sorted and unsorted. Insertion sort will copy the first element in the unsorted section of the array into a temporary variable. It will then find out where to put it in the list. It has to move everything down, one at a time, until it finds the position to place the item and then inserts it.

Example:

```
for (int i=0; i<n; i++)
{
    temp = ar[i];

    for (j=i; j>0; j--)
    {
        if (temp < ar[j-1])
            ar[j] = ar[j-1];
        else
            break;
    }
    ar[j] = temp;
}
```

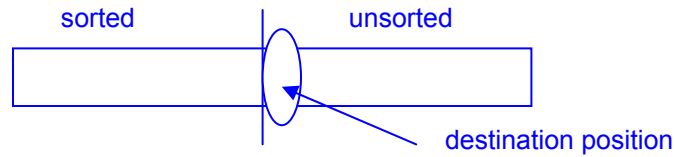
To demonstrate how this works, lets say that \wedge will indicate the separation between the sorted and the unsorted sections of the list.

```

 $\wedge$  5 3 0 7 9    - starting position of all numbers
5  $\wedge$  3 0 7 9    - (first pass) it places the five in the top position
3 5  $\wedge$  0 7 9    - (second pass) checks and swaps in the following order:
    1) it pulls out the first unsorted number (3) into a temporary variable.
    2) checks the temporary variable with the first in the sorted list (5)
    3) sees it's smaller and moves 5 to the old position of 3
    4) inserts temporary variable (3) into the j position
0 3 5  $\wedge$  7 9    - (third pass)
```

Even though at this point you see that the list is sorted, the computer won't know that because it hasn't looked at the 7 or 9 yet. When your data is already sorted you still have to copy the value out but only have to do one comparison and then re-insert it back into the list and go to the next one. This is still an n^2 algorithm and is fairly awful as algorithm efficiency goes.

Selection Sort:



Selection sort works on the same principle of dividing the array into a sorted and unsorted section. What it does is to search for the smallest value in the entire unsorted list and swap it with the first element in the unsorted section of the array. It goes without saying that the smallest of the unsorted values will be greater than all of the sorted values.

```
for (int i=0; i< n-1; i++)
{
    min_val = ar[i];
    min_index = i;

    for (j=i+1; j<n; j++)
    {
        if (ar[j] < min_val)
        {
            min_val = ar[j];
            min_index = j;
        }
    }
    temp = ar[i];
    ar[i] = min_value;
    ar[min_index] = temp;
}
```

if you started with the same array:

^ 5 3 0 7 9

0 ^ 3 5 7 9

When you start this sort:

- 1) the min_value takes on the value of 5 and min_index is set to 1
- 2) you start to loop from the position of min_index +1 to the number of elements
- 3) if the next item is less than your min_value, you set min_value to that item and the min_index to that position of the array. Then continue through the rest of the unsorted list.
- 4) once you reach the bottom, swap the min_value with ar[i] and continue another loop

Index

A		free	28	pop	56
abstract data types (ADT)	25	fstream	15	push	56
String	25, 30	G		Q	
List	37	gdb	22	Queue	56
.append	42	global variables	54	R	
arrays	10	Gnuu	21	rand()	11
passing to functions	10,16	goto	9	read	15
automatic variables	53	H		reading files	21
B		header file	19	recursion	46
basic commands	1	Hungarian notation	15	references	13,14
bool	4	I		removing nodes	50
break	8	ifstream	15	.replace	42
breaking up a program	19	#include	2	.rfind	43
bubble sort	61	#include <iostream>	3	S	
C		insertion sort	62	segmentation fault	21
cerr	3	inserting nodes	47,49	set width	5
cin	4	.inspect	42	setting precision	7
classes		integers	7	selection sort	63
complex class Ex.	22,23	input/output formatting:	5-7	sleep	11
constructors	17	decimal		sorting	60
function headers	17	fill		bubble sort	61
header files	18,19	get / getline		insertion sort	62
member functions	17	hex		selection sort	63
String class Ex.	26	internal		srand()	11
using DMA	30	left		Stacks	50
C++ type	40	octal		Stack class	51
clear()	45	right		static variables	54
compiler options	1	setf / unsetf		String Class	26
concatenating vectors	36	setiosflags /reset...		string (c++ type)	40-43
continue	8,9	ios flags	15	functions:	
copy constructor	34	in, out, binary		.c_str	
cout	3	app, trunc		.append	
const	14	L		.insert	
core dump	21	.length	43	.replace	
D		List (ADT)	38-40	.erase	
data redirection	4	Linked Lists	44	.find	
debugger (Gnuu)	21,22	local variables	54	.length/ size	
default arguments	23	L-value	13	.rfind	
delete	29	M		.empty	
deque	57	main()	2	.substr	
destructor	30, 46	malloc	27	swap routines	13
Doubly Linked Lists	49	Makefiles	2,20	switch	9
Dynamic Memory		manipulators	7	T	
Allocation (DMA)	27	N		templates	57
E		new	28	swap function template	57
endl	3	nodes	44-45	list class template	58
.empty	43	inserting	47, 49	this	23
enumerations	52	deleting	48, 50	V	
.erase	42	O		vector class	31
extern	55	ofstream	15	void main()	2
F		open	15	W	
factorial	46	overloading operators	24	write	15
files:		[] overload	31-32		
input/output	15	<< overload	24, 33		
I/O flags	15	+ overload	24, 37		
reading	21	+= overload	37		
.find	43	operator=	42		
flags		operator[]	42		
adjusted	6	operator+	42		
basefield	6	P			
floatfield	6	pointer arithmetic	11,12		
showpos	6				
showpoint	7				
flush	4				
friend	24				
front	56				