==========

ARRAYS, POINTERS and REFERENCES
STATICALLY and DYNAMICALLY ALLOCATED MEMORY

==========

Pointers

- A pointer is a variable that holds a memory address.
- This address is the location of another variable (or object) in memory.
- If one variable contains the address of another variable,
  the first variable is said to point to the second.

Pointer variables

- General form of declaring a pointer variable:

```
typeName *varName;
```

- Examples:
  - `int *ptr1;`
    - OR
    -
    - `int * ptr1;`
    - `int* ptr1;`
  - **BE CAREFUL !**
    - `int* ptr3, ptr4;`
      - `ptr3` is of type "`int` pointer", but `ptr4` is of type "`int`"
  - `double *dPtr;`
  - `char *chPtr;`

Pointer operators

- There are 2 special pointer operators: **&** and ***** (both are unary operators).
  - **&** - returns the memory address of its operand
    - `int *xPtr;`
    - `xPtr = &x;  // xPtr  receives "the address of x"`
      - assume that the value of `x` is `10` and
        that this value is stored at address `2000` of the memory;
        then `xPtr` will have value `2000`.
  - ***** - returns the value located at the address that follows
    - `int y;`
    - `y = *xPtr; // y receives the "value at address xPtr"`
      `// or "the value pointed to by xPtr"`
      - considering the example above
        `y` takes the value stored at address `2000`, that is `10`.

- NOTE:
  - make sure that your pointer variables point to the correct type of data
  - Example: if you declare a pointer of type **int**, the compiler assumes that any address that it holds points to an integer variable, whether it actually does or not.

Pointer assignments

- As with any simple variable, you may use a pointer on the right-hand side of an assignment statement, to assign its value to another pointer.

```
int x;
int *p1, *p2;
p1 = &x;
p2 = p1;
```

Pointer arithmetic

- Only 2 arithmetic operations may be used with pointers:
  - addition and subtraction
  - operators ++ and -- can be used with pointers
- Each time a pointer is incremented / decremented it points to the next / previous location of its base type
- When a value **i** is added / subtracted to / from a pointer **p** the pointer value will increase / decrease by the length of **i * sizeof(pointed_data_type)**

```
int *p1, *p2;
p1 = 2000; // usually, you shouldn't do this
p2 = p1+3; // if sizeof(int) is 4,
           // p2 will point to address 2000+3*4=2012
```

Pointer comparison

- You can compare 2 pointers in a relational expression:

```
if ( p1 < p2 ) cout << "p1 points to lower memory than p2\n";
```

Pointers and arrays

- There is a close relationship between pointers and arrays.
- An array name without an index returns the address of the first element of the array.
- So it is possible to assign an array identifier to a pointer provided that they are of the same type.

```
int a[10];
int *p;
p = a;     // p points to the first element of array a[]
p = &a[0]; // equivalent to the previous assignment
           // taking into account these declarations and
           // the pointer arithmetic rules (above),
```

```
                    // the following two statements are equivalent
                    // (both access the 4th element of the array):
```

- `a[3] = 27;`
- `*(p+3) = 27;`

- Also, the following code is sintactically correct:

```
void showArray(const int *a, size_t size) { ... }
...
int values[10];
showArray(values,10);
```

- The following 2 declarations with initialization are equivalent:
  - `char s[ ] = "Hello!";  // s can be modified`
  - `char *s = "Hello!"; // s can't be modified`

  but the 1st string can be modified
  while the 2nd can't !!!
  (it is stored in non-modifiable memory)

Initializing pointers

- After a local pointer is declared but before it has been assigned a value, if contains an unknown value.
- Global pointers are automatically initialized to `NULL` (equal to zero).
  - Address zero can not be accessed by user programs.
  - Programmers frequently assign the `NULL` value to a pointer, meaning that it points to nothing and should not be used.
- **BE CAREFUL**: Should you try to use the pointer before giving it a valid value, you will probably crash your program.

Multiple indirection

- You can have a pointer that points to another pointer that points to the target value.
  - `int **p; // p is a pointer to pointer that points to an int`
  - Example:
    - `int x, *p1, **p2;`
    - `x = 5;`
    - `p1 = &x;`
    - `p2 = &p1;`
    - `cout << "x = " << **p2 << endl;`
- You can have multiple levels of indirection.
- See examples in the next pages on
  - 2D arrays with dynamic allocation
  - accessing command line arguments
    - `int main(int argc, char **argv) {.....};`
      OR
    - `int main(int argc, char *argv[ ]) {.....};`

Pointers to functions

- Even though a function is not a variable,
  it still has a physical adress in memory that can be assigned to a pointer.
- This address is the entry point of the function.
- Once a pointer points to a function,
  the functions can be called through that pointer.
- Example:

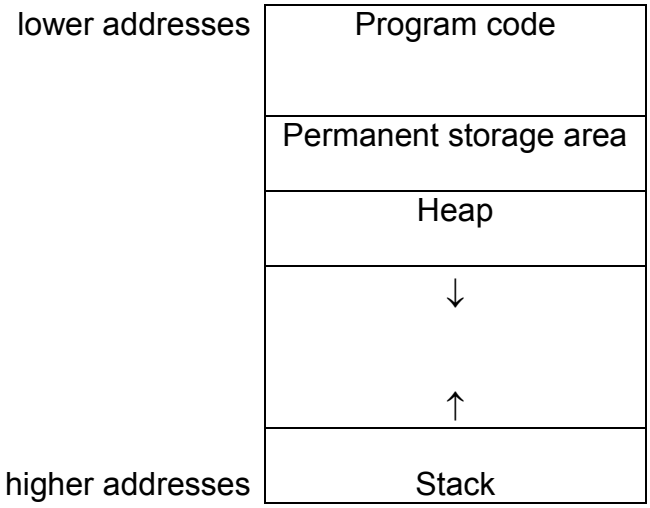
```
int sum(int x, int y)
{
  return x+y;
}

int main( )
{
  int result;
  int (*p) (int, int); // p is a pointer to a function that has
                       // 2 int parameters and returns int

  p=sum;               // now, p contains the starting address of sum()

  result = (*p)(2,3);  // sum() is called using pointer p !!!
  cout << result << endl;
}
```

References and Pointers

- A reference is essentially an implicit pointer.
- By far, the most common use of references is
  - to pass an argument to a function using call-by-reference (*already seen*)
  - to act as a return value from a function (*an example will be seen later*)
- A reference is a pointer in disguise
  - When you use references
    the compiler automatically passes parameters adresses
  - and dereferences the pointer parameters in the function body.
  - For that reason, in some situations,
    references are more convenient for the programmer than explicit pointers
- Example:
  - *see example of* `swap()` *functions in the next pages.*
- **NOTE**:
  - all independent references must be initialized in declaration
    - `int &r = x; // an independent reference`
  - independent pointers can be declared without being initialized
    - `int *p;`
    - ... but ... don't forget to initialize them before use
    - sometimes they are initialized as the result of a `malloc()` / `new` call
      (*see next pages*)

Dynamic memory allocation

- Pointers provide necessary support for C/C++ dynamic memory allocation system.
- Dynamic memory allocation is the means by which a program can obtain memory while it is running.
- Global variables are allocated storage at compile time.
- Local variables use the stack.
- However, neither global nor local variables can be added during program execution.
- Yet, there will be times where the storage needs of a program cannot be known when the program is being written.
  This is why dynamic memory allocation is useful.
- Dynamic memory allocation is particularly useful when you are programming in C.
  - As we have seen, some C++ data structures (ex: strings and vector) can change size dynamically.
- C++ supports 2 dynamic allocations systems:
  - the one defined by C
  - the one defined by C++
- Memory allocated by dynamic allocation functions is obtained from the heap.



C dynamic memory allocation

- The core of C's allocation system consists of the functions: `malloc()` and `free()`
  - => `#include <cstdlib>`
  - `void *malloc(size_t number_of_bytes);`
    - `number_of_bytes` is the number of bytes of memory you wish to allocate
    - the return value is a `void` pointer
      - **in C**, a `void *` can be assigned to another type of pointer; it is automatically converted
      - **in C++**, an explicit type cast is needed when a `void *` is assigned to another type of pointer
    - after a sucessful call, `malloc()` returns a pointer to the first byte of memory allocated from the heap
    - if there is not enough memory available `malloc()` returns a `NULL` pointer
    - Example:

```
int *p; int n;
cout << "n ? ", cin >> n;
p = (int *) malloc(n*sizeof(int)); // allocate space for
                                   // n consecutive integers
                                   // = an array of integers
if (p == NULL) {
  cout << "Out of heap memory !\n";
  exit(1);
}
```

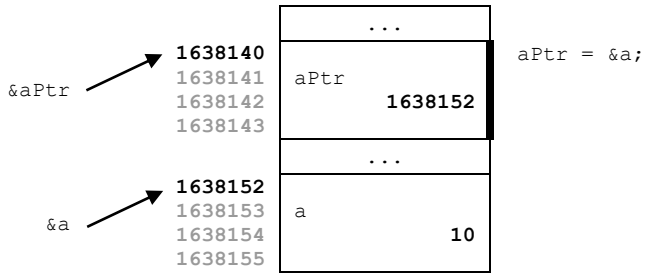
    - **NOTE**: the contents of the allocated memory is unknown
  - `void free(void *p)`
    - returns previously allocated memory to the system;
    - `p` is a pointer to memory that was previously allocated using `malloc()`.
    - **BE CAREFUL**: never call `free()` with an invalid argument

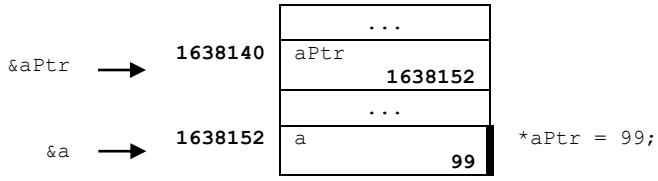
C++ dynamic memory allocation

- C++ provides two dynamic allocation operators: `new` and `delete`;
  - => `#include <new>`
  - *`p_var`* `= new` *`type`*`;`
    - `int *p = new int ;  // useful... ?`
  - *`p_var`* `= new` *`type`*`(`*`initializer`*`);`
    - `int *p = new int(0); //initialize the int pointed to by p with zero`
  - `delete` *`p_var`*`;`
    - `delete p;`
- Allocating **arrays** with new
  - *`p_var`* `= new` *`array_type`*`[`*`size`*`];`
    - `int *p = new int[10]; // allocate 10 integers array`
  - `delete [ ]` *`p_var`*`;`
    - `delete [ ] p;`

```
// Pointer concept
// JAS

#include <iostream>
#include <iomanip>
using namespace std;

void main()
{
	int a;
	int *aPtr; // OR  int * aptr;    OR int* aptr; // 'aPtr' is a pointer to an integer

	a = 10;
	aPtr = &a;  // '&a' means the address of 'a'

	cout << "   &a = " << &a << " (hexadecimal)\n";
	cout << "   &a = " << setw(8) << (unsigned long) &a << " (decimal)\n";
	cout << "&aPtr = " << &aPtr << " (hexadecimal)\n";
	cout << "&aPtr = " << setw(8) << (unsigned long) &aPtr << " (decimal)\n";
	cout << " aPtr = " << aPtr << " (hexadecimal)\n";
	cout << " aPtr = " << setw(8) << (unsigned long) aPtr << " (decimal)\n";
	cout << "   a  = " << a << endl;
	cout << "*aPtr = " << *aPtr << endl << endl;

	*aPtr = 99; // *aPtr - dereferencing pointer aPtr, using * operator
	cout << "   a  =  " << a << endl;
}
```

```
   &a = 0018FF08 (hexadecimal)
   &a =  1638152 (decimal)

&aPtr = 0018FEFC (hexadecimal)
&aPtr =  1638140 (decimal)

 aPtr = 0018FF08 (hexadecimal)
 aPtr =  1638152 (decimal)

   a  = 10
*aPtr = 10

   a  = 99

Press any key to continue . . .
```





```
// Pointers
// Using pointers - Passing parameters by reference (2 different ways):
// 1) using explicit pointers
// 2) using reference parameters
// JAS - Mar/2011

#include <iostream>
#include <iomanip>
using namespace std;

// Using explicit pointers for passing parameters by reference
void swap1(int *x, int *y)
{
	int temp;

	temp = *x;  // *x - dereferencing pointer x, using * operator
	*x = *y;
	*y = temp;
}

// Passing reference parameters (as we have seen before)
// A reference is a pointer "in disguise"
// When you use references the compiler automatically passes parameters adresses
// and dereferences the pointer parameters in the function body.
// For that reason , references are more convenient for the programmer
// than explicit pointers
void swap2(int &x, int &y)
{
	int temp;

	temp = x;
	x = y;
	y = temp;
}

void main(void)
{
	int a, b;

	a=10; b=20;
	cout << "a = " << a << ",  b = " << b << endl << endl;

	swap1(&a,&b);
	cout << "after swap1(): a = " << a << ",  b = " << b << endl << endl;

	swap2(a,b);
	cout << "after swap2(): a = " << a << ",  b = " << b << endl << endl;


}
```

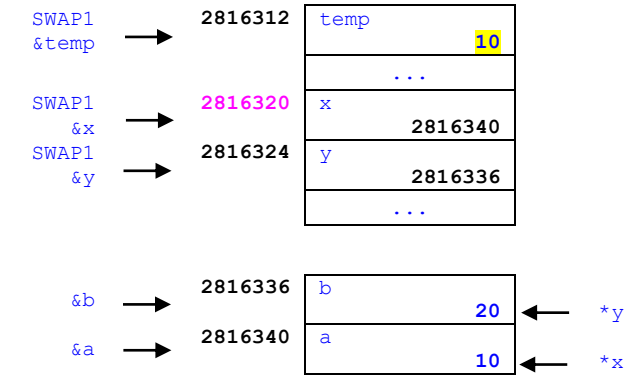
```
a = 10,  b = 20

after swap1(): a = 20,  b = 10

after swap2(): a = 10,  b = 20
```

TO DO BY STUDENTS: try to overload swap()

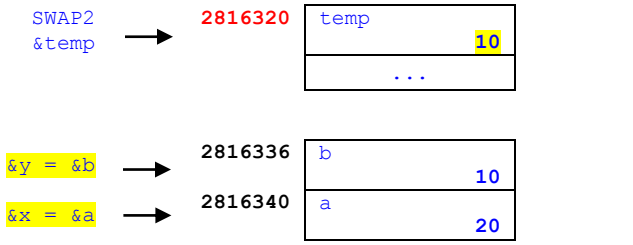
```
// Pointers
// Using pointers - Passing parameters by reference (2 different ways):
// (similar to the last example, but showing more information)
// 1) using explicit pointers
// 2) using reference parameters
// JAS - Mar/2011

#include <iostream>
#include <iomanip>
using namespace std;

void swap1(int *x, int *y)
{
	int temp;

	cout << "SWAP1_a\n";
	cout << "  &x = " << (unsigned long) &x << ", " <<
		" &y = " << (unsigned long) &y << ", " <<
		" &temp = " << (unsigned long) &temp << " (decimal)\n";
	cout << "   x = " << (unsigned long) x << ", " <<
		"  y = " << (unsigned long) y << " (decimal)\n";
	cout << " *x = " << *x <<
		" *y = " << *y << ", " <<
		" temp = " << temp << endl;

	temp = *x;
	*x = *y;
	*y = temp;

	cout << "SWAP1_b\n";
	cout << "  *x = " << *x <<
		" *y = " << *y << ", " <<
		" temp = " << temp << endl;
}

void swap2(int &x, int &y)
{
	int temp;

	cout << "SWAP2_a\n";
	cout << "  &x = " << (unsigned long) &x << ", " <<
		" &y = " << (unsigned long) &y << ", " <<
		" &temp = " << (unsigned long)&temp << " (decimal)\n";
	cout << "   x = " << (unsigned long) x << ", " <<
		"  y = " << (unsigned long) y << ", " << " (decimal)" <<
		"  temp = " << temp << endl;
	temp = x;
	x = y;
	y = temp;

	cout << "SWAP2_b\n";
	cout << "   x = " << x << ", " <<
		"  y = " << y << ", " <<
		"  temp = " << temp << ", " <<endl;
}

void main(void)
{
	int a, b;

	a=10; b=20;

	cout << "MAIN\n";
	cout << " &a = " << (unsigned long) &a << ", " <<
		" &b = " << (unsigned long) &b << " (decimal)\n";
	cout << "  a = " << a << ",  b = " << b << endl << endl;

	swap1(&a,&b);

	cout << "MAIN after swap1(): a = " << a << ",  b = " << b << endl << endl;

	swap2(a,b);

	cout << "MAIN after swap2(): a = " << a << ",  b = " << b << endl << endl;
}
```

```
MAIN
 &a = 2816340,  &b = 2816336 (decimal)
  a = 10,  b = 20

SWAP1_a
 &x = 2816320,  &y = 2816324,  &temp = 2816312 (decimal)
  x = 2816340,   y = 2816336 (decimal)
 *x = 10 *y = 20,  temp = -175524
SWAP1_b
 *x = 20 *y = 10,  temp = 10
MAIN after swap1(): a = 20,  b = 10

SWAP2_a
 &x = 2816340,  &y = 2816336,   &temp = 2816320 (decimal)   temp has same address as x in swap1()
  x = 20,   y = 10,  (decimal)  temp = 1759800264           just by chance; memory was reused
SWAP2_b
  x = 10,   y = 20,   temp = 20,
MAIN after swap2(): a = 10,  b = 20
```

```
void swap1(int *x, int *y)
{...}

void main(void)
{
       int a, b;
       a=10; b=20;
       ...
       swap1(&a,&b);
       ...
}
```

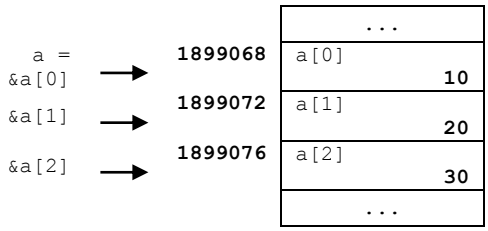


```
void swap2(int &x, int &y)
{...}

void main(void)
{
       int a, b;
       ... // after swap1() -> a=20; b=10
       swap2(a,b);
       ...
}
```



```
// Pointers and 1D arrays with static allocation
// Relationship between arrays and pointers
// Pointer arithmetic
// JAS - Mar/2011

#include <iostream>

using namespace std;


#define NMAX 3

void main(void)
{
        int a[NMAX];
        int *aPtr;
        int i;

        for (i=0; i<NMAX; i++)
                a[i] = 10*(i+1);

        for (i=0; i<NMAX; i++)
                cout << "&a[" << i << "] = " << (unsigned long) &a[i]
                     << ",  a[" << i << "] = " <<  a[i] << endl;

        aPtr = a; // an array identifier is a pointer to the 1st array element

        cout << "a     = " << (unsigned long) a << endl;
        cout << "&a[0] = " << (unsigned long) &a[0] << endl;
        cout << "aPtr  = " << (unsigned long) aPtr << endl;

        for (i=0; i<NMAX; i++)
                cout << "(aPtr+" << i << ") = " << (unsigned long) (aPtr+i)
                     << ",  *(aPtr+" << i << ") = " << *(aPtr+i) << endl;

}
```

```
&a[0] = 1899068,  a[0] = 10
&a[1] = 1899072,  a[1] = 20
&a[2] = 1899076,  a[2] = 30
a     = 1899068
&a[0] = 1899068
aPtr  = 1899068
(aPtr+0) = 1899068,  *(aPtr+0) = 10
(aPtr+1) = 1899072,  *(aPtr+1) = 20
(aPtr+2) = 1899076,  *(aPtr+2) = 30
```



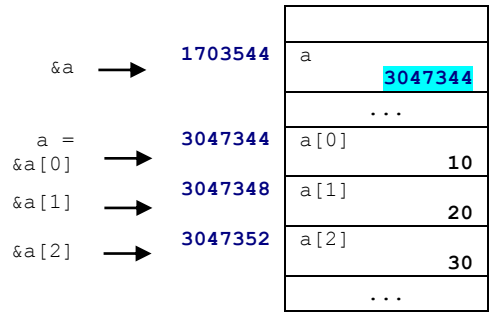
```
// Pointers and 1D arrays with static allocation
// Relationship between arrays and pointers
// Passing arrays as function parameters
// JAS - Mar/2011

#include <iostream>

using namespace std;


#define NMAX 3

void showArray1(const int v[], int nElems)
{
	cout << "showArray1()\n";
	for (int i=0; i<nElems; i++)
		cout << "v[" << i << "] = " << v[i] << endl;
}

void showArray2(const int *v, int nElems)
{
	cout << "showArray2()\n";
	for (int i=0; i< nElems; i++)
		cout << "v[" << i << "] = " << v[i] << endl;  // v[i] <=> *(v+i)
}


void main(void)
{
	int a[NMAX];

	for (int i=0; i<NMAX; i++)
		a[i] = 10*(i+1);

	showArray1(a,NMAX);

	showArray2(a,NMAX);

	showArray2(&a[0],NMAX);

	// showArray1(&a[0],NMAX);  // also possible

}
```

```
showArray1()
v[0] = 10
v[1] = 20
v[2] = 30
showArray2()
v[0] = 10
v[1] = 20
v[2] = 30
showArray2()
v[0] = 10
v[1] = 20
v[2] = 30
```

QUESTION:
is it possible to overload showArray(), giving this same name to both functions ?

```
// Pointers and 1D arrays with dynamic allocation
// Dynamic memory allocation:
// 1) C-style: malloc() & free()
// 2) C++-style: new & delete
// VERY IMPORTANT; NEVER MIX THE 2 KINDS OF DYNAMIC MEMORY ALLOCATION
// JAS - Mar/2011

#include <iostream>
// #include <cstdlib>
#include <new>

using namespace std;


//#define NMAX 3

void main(void)
{
	int *a; // OR int * a; OR  int* a;
	int nMax, i;

	cout << "nMax ? "; cin >> nMax;

	cout << "&a = " << (unsigned long) &a << endl;
	cout << "a (before dynamic memory allocation) = " << (unsigned long) a << endl;

	// dinamically allocate memory for array of integers
	// a = (int *) malloc(nMax * sizeof(int));  // C-style
	a = new int[nMax];  // C++-style

	cout << "a (after dynamic memory allocation)  = " << (unsigned long) a << endl;

	for (i=0; i<nMax; i++)
		a[i] = 10*(i+1);

	for (i=0; i<nMax; i++)
		cout << "a[" << i << "] = " << a[i]
		     << ", &a[" << i << "] = " << (unsigned long) &a[i] << endl;

	// free the dinamically allocate memory
	// free(a);   // C- style
	delete [] a;  // C++-style

	// a[0] = 100; // should not be done ... why ?
}
```

```
nMax ? 3
&a = 1703544
a (before dynamic memory allocation) = 9449524
a (after dynamic memory allocation)  = 3047344
a[0] = 10, &a[0] = 3047344
a[1] = 20, &a[1] = 3047348
a[2] = 30, &a[2] = 3047352
```

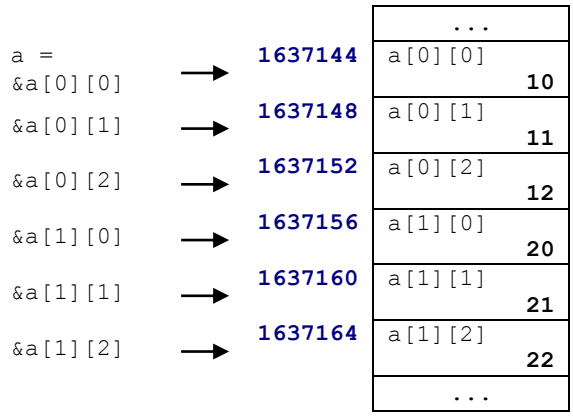


```
ANOTHER RUN ...

nMax ? 5
&a = 2750352
a (before dynamic memory allocation) = 8532020
a (after dynamic memory allocation)  = 10121352
a[0] = 10, &a[0] = 10121352
a[1] = 20, &a[1] = 10121356
a[2] = 30, &a[2] = 10121360
a[3] = 40, &a[3] = 10121364
a[4] = 50, &a[4] = 10121368

ANOTHER RUN ...

nMax ? 7
&a = 2946936
a (before dynamic memory allocation) = 0
a (after dynamic memory allocation)  = 688048
a[0] = 10, &a[0] = 688048
a[1] = 20, &a[1] = 688052
a[2] = 30, &a[2] = 688056
a[3] = 40, &a[3] = 688060
a[4] = 50, &a[4] = 688064
a[5] = 60, &a[5] = 688068
a[6] = 70, &a[6] = 688072
```

```
// 2D arrays with static allocation
// JAS - Mar/2011

#include <iostream>

using namespace std;

#define NLIN 2
#define NCOL 3

void main(void)
{
       int a[NLIN][NCOL];

       for (int i=0; i<NLIN; i++)
              for (int j=0; j<NCOL; j++)
                     a[i][j] = 10*(i+1)+j;

       for (int i=0; i<NLIN; i++)
              for (int j=0; j<NCOL; j++)
                     cout << "a[" << i << "][" << j << "] = " << a[i][j]
                          << ",  &a[" << i << "][" << j << "] = " << (unsigned long) &a[i][j]
                          << endl;
}
```

	0	1	2
0	10	11	12
1	20	21	22

```
a[0][0] = 10,  &a[0][0] = 1637144
a[0][1] = 11,  &a[0][1] = 1637148
a[0][2] = 12,  &a[0][2] = 1637152
a[1][0] = 20,  &a[1][0] = 1637156
a[1][1] = 21,  &a[1][1] = 1637160
a[1][2] = 22,  &a[1][2] = 1637164
```



```
// 2D arrays with static allocation
// 2D arrays as function parameters
// JAS - Mar/2011

#include <iostream>

using namespace std;

#define NLIN 2
#define NCOL 3


void showArray(int a[][NCOL], int numLines, int numCols)
// WHY DOES THE COMPILER NEED TO KNOWN THE NUMBER OF COLUMNS, "NCOL" ?
{
	for (int i=0; i< numLines; i++)
	{
		for (int j=0; j< numCols; j++)
			cout << a[i][j] << " ";
		cout << endl;
	}

}

void main(void)
{
	int a[NLIN][NCOL];

	for (int i=0; i<NLIN; i++)
		for (int j=0; j<NCOL; j++)
			a[i][j] = 10*(i+1)+j;

	showArray(a, NLIN, NCOL);
}
```

```
10 11 12
20 21 22
```

==========

CHALLENGE

Implement a similar program using 2D dynamicaly allocated array

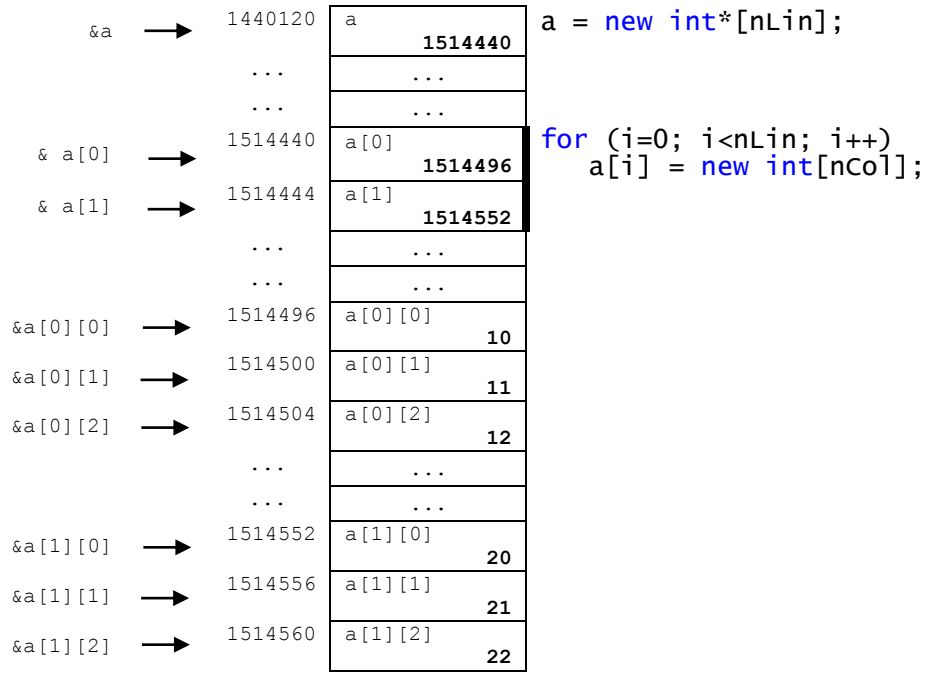
```
// Pointers and 2D arrays with ("bidimensional") dynamic allocation
// "C-like": using malloc / free

#include <iostream>
#include <cstdlib>
//#include <new>

using namespace std;

void main(void)
{
  int **a;   // <-- NOTE THIS
  int i, j, nLin, nCol;

  cout << "nLin ? "; cin >> nLin;
  cout << "nCol ? "; cin >> nCol;

  // allocate memory for 2D array
  a = (int **)malloc(nLin * sizeof(int *));
  for (i = 0; i < nLin; i++)
    a[i] = (int *)malloc(nCol * sizeof(int)); // allocate memory for each line of the array

  // use the array
  for (i = 0; i < nLin; i++)
    for (j = 0; j < nCol; j++)
      a[i][j] = 10 * (i + 1) + j;


  cout << "&a = " << (unsigned long)&a << endl;
  cout << " a = " << (unsigned long)a << endl;
  for (i = 0; i < nLin; i++)
    cout << "&a[" << i << "] = " << (unsigned long)&a[i] << endl;
  for (i = 0; i < nLin; i++)
    cout << " a[" << i << "] = " << (unsigned long)a[i] << endl;

  for (i = 0; i < nLin; i++)
    for (j = 0; j < nCol; j++)
      cout << "a[" << i << "][" << j << "] = " << a[i][j] << ", &a[" << i << "][" << j << "] = " <<
(unsigned long)&a[i][j] << endl;

  // free all allocated memory (in reverse order of allocation)
  for (i = 0; i < nLin; i++)
    free(a[i]);
  free(a);
}
```

```
nLin ? 2
nCol ? 3
&a = 1440120
 a = 1514440
&a[0] = 1514440
&a[1] = 1514444
 a[0] = 1514496
 a[1] = 1514552
a[0][0] = 10, &a[0][0] = 1514496
a[0][1] = 11, &a[0][1] = 1514500
a[0][2] = 12, &a[0][2] = 1514504
a[1][0] = 20, &a[1][0] = 1514552
a[1][1] = 21, &a[1][1] = 1514556
a[1][2] = 22, &a[1][2] = 1514560
Press any key to continue . . .
```

```
// Pointers and 2D arrays with ("bidimensional") dynamic allocation
// "C++-like": using new / delete

#include <iostream>
// #include <cstdlib>
#include <new>

using namespace std;

void main(void)
{
	int **a;  // <-- NOTE THIS
	int i, j, nLin, nCol;

	printf("nLin ? "); cin >> nLin;
	printf("nCol ? "); cin >> nCol;

	// allocate memory for 2D array
	a = new int*[nLin];
	for (i=0; i<nLin; i++)
		a[i] = new int[nCol];// allocate memory for each line of the array

	// use the array
	for (i=0; i<nLin; i++)
		for (j=0; j<nCol; j++)
			a[i][j] = 10*(i+1)+j;


	cout << "&a = " << &a << endl;
	cout << " a = " << a  << endl;
	for (i=0; i<nLin; i++)
		cout << "& a[" << i << "] = " << &a[i] << endl;
	for (i=0; i<nLin; i++)
		cout << "  a[" << i << "] = " <<  a[i] << endl;

	for (i=0; i<nLin; i++)
		for (j=0; j<nCol; j++)
			cout << "a[" << i << "][" << j << "] = " << a[i][j] << ", " <<
			"&a[" << i << "][" << j << "] = " << &a[i][j] << endl;

	// free all allocated memory (in reverse order of allocation)
	for (i=0; i<nLin; i++)
		delete[] a[i];
	delete[] a;
}
```

```
nLin ? 2
nCol ? 3
&a = 1440120
 a = 1514440
& a[0] = 1514440
& a[1] = 1514444
  a[0] = 1514496
  a[1] = 1514552
a[0][0] = 10, &a[0][0] = 1514496
a[0][1] = 11, &a[0][1] = 1514500
a[0][2] = 12, &a[0][2] = 1514504
a[1][0] = 20, &a[1][0] = 1514552
a[1][1] = 21, &a[1][1] = 1514556
a[1][2] = 22, &a[1][2] = 1514560
```

```
nLin ? 2
nCol ? 3
&a = 1440120
 a = 1514440
& a[0] = 1514440
& a[1] = 1514444
  a[0] = 1514496
  a[1] = 1514552
a[0][0] = 10, &a[0][0] = 1514496
a[0][1] = 11, &a[0][1] = 1514500
a[0][2] = 12, &a[0][2] = 1514504
a[1][0] = 20, &a[1][0] = 1514552
a[1][1] = 21, &a[1][1] = 1514556
a[1][2] = 22, &a[1][2] = 1514560
```



```
/*
POINTERS TO STRUCT'S
How to access the members of a struct using a pointer to the struct ?
*/

#include <iostream>
#include <iomanip>
#include <cctype>
#include <string>
#include <sstream>

using namespace std;

struct Fraction
{
  int numerator;
  int denominator;
};

bool readFraction(Fraction &f)  // readFraction() is overloaded (see below)
{
  string fractionString;
  char fracSymbol;
  int numerator;
  int denominator;
  bool success;

  cout << "n / d ? ";
  getline(cin,fractionString);

  istringstream fractionStrStream(fractionString);
  if (fractionStrStream >> numerator >> fracSymbol >> denominator)
    if (fracSymbol == '/')
    {
      f.numerator = numerator;
      f.denominator = denominator;
      success = true;
    }
    else
      success = false;
  else
    success = false;
  return success;
}

bool readFraction(Fraction *f)  // readFraction() is overloaded (see above)
{
  string fractionString;
  char fracSymbol;
  int numerator;
  int denominator;
  bool success;

  cout << "n / d ? ";
  getline(cin,fractionString);
```

```
  istringstream fractionStrStream(fractionString);
  if (fractionStrStream >> numerator >> fracSymbol >> denominator)
    if (fracSymbol == '/')
    {
      f->numerator = numerator;
      //(*f).numerator = numerator;
      f->denominator = denominator;
      //(*f).denominator = denominator;
      success = true;
    }
    else
      success = false;
  else
    success = false;
  return success;
}


Fraction multiplyFractions(Fraction f1, Fraction f2)
{
  Fraction f;

  f.numerator = f1.numerator * f2.numerator;
  f.denominator = f1.denominator * f2.denominator;
  return f;

}


void showFraction(Fraction f)
{
  cout << f.numerator << "/" << f.denominator;
}



int main()
{
  Fraction f1, f2, f3;

  cout << "Input 2 fractions:\n";
  //if (readFraction(f1) && readFraction(f2))
  if (readFraction(&f1) && readFraction(&f2))
  {
    f3 = multiplyFractions(f1,f2);

    cout << "Product: ";
    showFraction(f3);
  }
  else
  {
    cout << "Invalid fraction\n";
  }
  cout << endl;

  return 0;
}
```

==========

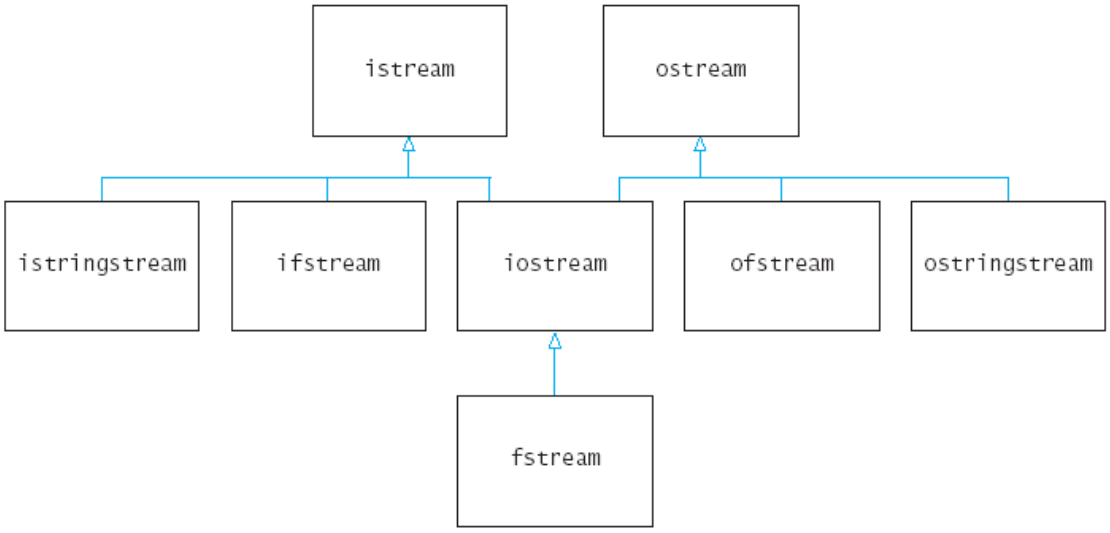
STREAMS / FILES

==========

I/O Streams

- I/O refers to program Input and Output
- I/O is done via stream objects
- A **stream** is a flow of data.
- Input stream: data flows into the program
    - Input can be from
        - the keyboard
        - a file
- Output stream: data flows out of the program
    - Output can be to
        - the screen
        - a file
- Input and Output stream: data flows either into or out of the program
    - only possible with files

- **The C++ input/output library** consists of several classes that are related by inheritance (*inheritance will be treated later in this course*)
- The inheritance hierarchy of stream classes:



- The standard **cin** and **cout** objects
  belong to specialized system-dependent classes with nonstandard names.
- You can assume that
    - **`cin`** belongs to a class that is derived from **`istream`** and
    - **`cout`** belongs to a class derived from **`ostream`**.

cin & cout streams

- `cin`
  - input stream connected to the keyboard
- `cout`
  - output stream connected to the screen
- `cin` and `cout` are declared in the `iostream` header file
  - => `#include <iostream>`
- You can declare your own streams to use with files.

Why use files?

- • Files allow you
  - to use input data over and over
  - to deal with large data sets
  - to access output data after the program ends
  - to store data permanently

Text files vs. Binary files

- Usually files are classified in two categories:
  - ASCII (text) files
  - and binary files.
- While both binary and text files contain data stored as a series of bits,
  - the bits in text files represent characters,
  - while the bits in binary files represent other types of data (`int`, `float`, `struct`, ...)
- Simple text files are usually created by using a text editor like **notepad**, **pico**, etc. (not Word or OpenOffice)
- We work with binary files all the time.
  - **executable files**, **image files**, **sound files**, ... are binary files.
- In effect, ASCII files are basically binary files, because they store binary numbers.
- `cin` & `cout` "behave like" text files.

Accessing file data

- Open the file
  - this operation associates the name of a file in disk to a stream object.
  - NOTE: cin and cout are open automatically on program start.
- Use read/write calls or extraction/insertion operators, to get/put data from/into the file.
- Close the file.

Declaring Stream Variables

- Like other variables, a stream variable must be …
  - declared before it can be used
  - initialized before it contains valid data
    - Initializing a stream means connecting it to a file
- Input-file streams are of type `ifstream`
- Ouput-file streams of are type `ofstream`
- These types are defined in the `fstream` library
  - => `#include <fstream>`

- Example:

```
#include <fstream>
using namespace std;
...
ifstream in_stream;
ofstream out_stream;
```

Connecting a stream to a file / Opening a file

- The opening operation connects a stream to an external file name
  - An external file name is the name for a file that the operating system uses
  - Examples:
    - `infile.txt` and `outfile.txt` used in the following examples
- Once a file is open, it is referred to using the name of the stream connected to it.
- A file can be opened using
  - the **`open()`** member function associated with streams
  - the **constructor** of the stream classes
- Examples:
  - `ifstream in_stream;`
  - `ofstream out_stream;`
  - `in_stream.open("infile.txt");`
    - connects `in_stream` to `"infile.txt"`
  - `out_stream.open("C:\\Mieic\\Prog\\programs\\outfile.txt");`
    - connects out_stream to `"oufile.txt"`
      that is in directory `"C:\Mieic\Prog\programs"`
    - note the double backslash in the string argument
      - necessary in Windows systems
        where the directories of the path are separated by `'\'`
  - Alternatively:
    - `ifstream in_stream("infile.txt");`
    - calls the constructor of `ifstream` class
      that automatically tries to open the file
- The filename does not need to be a constant, as in the previous examples.
  Program users can enter the name of a file to use for input or for output.
  - in this case it must be stored in a string variable
  - In C++11, you can use a `std::string` as argument to **`open()`**
    or to the constructor

```
std::string filename;
cout << "Filename ?"; cin >> filename;
myFile.open(filename);
```

  - In the previous C++ standard,
    `open()` only accepts a C-string for the first parameter.
    The correct way of calling it would then be:
    - `myFile.open(filename.c_str());`
- Note:
  - The name of a text file does not necessarily have the extension `'.txt'`

open() method (C++11)

- `void ifstream::open(const string &filename, ios::openmode mode = ios::in);`
- `void ofstream::open(const string &filename, ios::openmode mode = ios::out);`
- `void fstream::open(const string &filename, ios::openmode mode = ios::in | ios::out);`
    - **filename** is the name of the file (must be a C-string, in pre-C++11 compilers)
    - **mode** determines how the file is opened; can be the OR (|) of several constants
        - **ios::in** – the file is capable of input
        - **ios::out** – the file is capable of output
        - **ios::binary** – causes file to be opened in binary mode;
          by default, all files are opened in text mode
        - **ios::ate** – cause initial seek to end-of-file;
          I/O operations can still occur anywhere within the file
        - **ios::app** – causes all output to the file to be appended to the end
        - **ios::trunc** – the file is truncated to zero length

Using input/output stream for reading/writing from/to text files

- It is very easy to read from or write to a text file.
- Simply use the **<<** and **>>** operators the same way you do when performing console I/O, except that, instead of using cin and cout,
  use a stream that is linked to a file.
- Example 1:

```
ifstream in_stream;
in_stream.open("infile.txt");
int one_number, another_number;
in_stream >> one_number >> another_number;
```

- Example 2:

```
ofstream out_stream;
out_stream.open("outfile.txt");
out_stream << "Resulting data:";
out_stream << one_number << endl << another_number << endl;
```

Closing a file

- After using a file, it should be closed.
  This disconnects the stream from the file
    - Example: `in_stream.close();`
- The system will automatically close files if you forget
  as long as your program ends normally
- Files should be closed:
    - to reduce the chance of a file being corrupted
      if the program terminates abnormally.
    - if your program later needs to read input from the output file.

Errors on opening files

- Opening a file could fail for several reasons.
  Common reasons for open to fail include
  - the file does not exist (or the path is incorect)
  - the external name is incorrect
  - the file is already open
- Member function **`is_open()`**, can be used to test whether the file is already open
- May be no error message if the call to open fails.
  Program execution continues!
- Member function **`fail()`**, can be used to test the success of a stream operation
  (not only the open() operation)
  - Example:

```
in_stream.open("numbers.txt");
if( in_stream.fail( ) )
{
    cerr << "Input file opening failed.\n";
    exit(1) ;  //  sometimes, it is best to stop the program,
               //  with an exit code != 0
}
```

Reading from text files – additional notes

- Stream input is performed with the stream extraction operator **`>>`**, which
  - skips white space characters (`' '`, `'\t'`, `'\n'`)
  - returns **`false`**, after end-of-file (EOF) is encountered
  - Example:

```
double next, sum = 0;
while(in_stream >> next)
{
   sum = sum + next;
}
```

- Stream input causes some stream state flags to be set when an error occurs:
  - **`failbit`** - improper input (internal logic error of the operation)
  - **`badbit`** - the operation failed (failure of I/O on the stream buffer)
  - **`eofbit`** - EOF was reached on the input stream
    - EOF can be tested using the eof() member function
      - **`while( ! in_stream.eof() ) ...`** (*see later*)
  - **`goodbit`** to be set when no error has occured
- Member function **`ignore()`** can be used to skip characters, as with **`cin`** stream.
- NOTE:
  - be careful when mixing **`>>`** and **`getline()`** OR **`>>`** and **`cin.get()`**
  - *remember what has been said about this, in the string section*

How To Test End of File

- In some cases, you will want to know when the end of the file has been reached.
  - For example, if you are reading a list of values from a file, then you might want to continue reading until there are no more values to obtain.
  - This implies that you have some way to know when the end of the file has been reached.
  - C++ I/O system supplies such a function to do this: **`eof( )`**.
  - To detect EOF involves these steps:
    - 1. Open the file being read for input.
    - 2. Begin reading data from the file.
    - 3. After each input operation, determine if the end of the file has been reached by calling **`eof( )`**.
- NOTE:
  - **`eof()`** returns false only when the program tries to read past the end of the file
- Example:
  - This loop reads each character, and writes it to the screen

```
in_stream.get(next);          // NOTE:first, input must be tried
while ( ! in_stream.eof( ) ) // then you can test for EOF
{
   cout << next;
   in_stream.get(next);
}
```

Formatting output to text files

- As for **`cout`**, formatting can be done using:
  - manipulators (defined in **iomanip** library => #include <iomanip>)
    - **`setw()`**
    - **`fixed`**
    - **`setprecision`**
    - ... and some other
  - using **`setf()` member function** of output streams
    - **`out_stream.setf(ios::fixed);`**
    - **`out_stream.setf(ios::showpoint);`**
    - **`out_stream.precision(2);`**
    - ... and some other
- Note:
  - A manipulator is a function called in a nontraditional way used after the insertion operator (<<) as if the manipulator function call is an output item
  - Manipulators in turn call member functions
    - **`setw`** does the same task as the member function **width**
    - **`setprecision`** does the same task as the member function **`precision`**
    - ...
  - Any flag that is set, may be unset, using the **`unsetf`** function
    - Example: **`cout.unsetf(ios::showpos);`** causes the program to stop printing plus signs on positive numbers

Stream names as arguments

- Streams can be arguments to a function
- The function's formal parameter for the stream must be call-by-reference
    - Example:

```
void make_neat(ifstream &messy_file, ofstream &neat_file);
// make_neat() code will be presented in the following pages
```

- Take advantage of the inheritance relationships between the stream classes whenever you write functions with stream parameters.
    - **ifstream** as well as **cin** are objects of type **istream**
    - **ofstream** as well as **cout** are objects of type **ostream**
    - Example:
        - **double get_max(istream &in);**
        - You can now pass parameters of types derived from **istream**, such as an **ifstream** object or **cin**.
            - **max = get_max(in_stream);**
            - **max = get_max(cin);**
            - both **cin** and **in_stream** can be used as arguments of a call to **get_max()**, whose parameter is of type **istream &**

Binary I/O

- While reading and writing text files is very easy it is not always the most efficient way to handle files.
- There will be times when you need to store information in binary format: **int**'s, **double**'s, **struct**'s, ... or **char**'s
- When performing I/O of binary data be sure to open the file using the **ios::binary** mode specifier
- I/O can be performed using the
    - **get()** and **put()** member functions
    -
        - **istream & get(char &ch);**
        - **ostream & put(char ch);**
    - NOTE:
        - In a text stream, some character translations may take place. For example, when the newline character is output, using **<<**, it may be converted into a carriage-return / linefeed sequence.
        - The reverse happens when a carriage-return / linefeed sequence is input from a file: it is converted into a newline char.
        - No such translations occur on binary files:
            - using **get()** you can "see" the carriage-return/linefeed chars in a text file

- `read()` and `write()` member functions
  - can be used to read/write blocks of binary data
    - `istream & read (char *buf, streamsize num);`
      - reads `num` characters from the invoking stream ans puts them into the buffer pointed to by `buf`
    - `ostream & write (const char* buf, streamsize num);`
      - writes `num` characters to the invoking stream from the buffer pointed to by `buf`
- Example: (*see next pages*)

Random access

- The C++ I/O system manages 2 pointers associated with a file:
  - the get pointer, which specifies where in the file the next input operation will occur
  - the put pointer, which specifies where in the file the next output operation will occur
- You can perform random access (in a nonsequential fashion) by using the `seekg()` and `seekp()` functions.
- Generally, random access I/O should only be performed on those files opened for binary operations. WHY?
- Their most common forms are:
  - `istream& seekg (streamoff offset, ios_base::seekdir origin);`
  - `ostream& seekp (streamoff offset, ios_base::seekdir origin);`
    - `origin` can take one of the values: `ios::beg`, `ios::end`, `ios::cur`
    - `offset` is an integer that specifies the displacement of the get/put pointer relative to the specified `origin`
- `seekg()` and `seekp()` are interchangeable for file streams. However, this is not true for other types of streams (ex: stringstreams, *see next pages*), as they may hold separate pointers for the put and get positions.
- `tellg()` and `tellp()` can be used to obtain the current position of the pointers.
- Note: you can't call **seekp**/**tellp** on an instance of **ifstream** and you can't call **seekg**/**tellg** on an instance of **ofstream**. However, you can use both on an instance of **fstream**.

==========

INPUT/OUTPUT – TEXT FILES

==========

```
/**
   INPUT FROM TEXT FILE
   Reads numbers from a file and finds the maximum value
   @param in the input stream to read from
   @return the maximum value or 0 if the file has no numbers

   (from BIG C++ book)
*/
#include <iostream>
#include <string>
#include <fstream>

using namespace std;

double max_value(ifstream &in) //stream parameters must always be passed by reference
{
   double highest;
   double next;
   if (in >> next) // if file contains at least 1 element
      highest = next;
   else
      return 0;   // If file is empty.  Not the best solution ...!!!

   while (in >> next)
   {
      if (next > highest)
         highest = next;
   }

   return highest;
}

int main()
{
   string filename;
   cout << "Please enter the data file name: "; // numbers.txt
   //located in C:\Users\jsilva\......\Project_folder\numbers.txt
   cin >> filename;

   ifstream infile;
   infile.open(filename);

   if (infile.fail())   // OR if (! infile.is_open())  OR if (! infile)
   {
      cerr << "Error opening " << filename << "\n";
      return 1;   // exit(1);
   }

   double max = max_value(infile);
   cout << "The maximum value is " << max << "\n";

   infile.close();
   return 0;
}
```

==========

```
/**
INPUT FROM TEXT FILE   OR   KEYBOARD
Reads numbers from a file and finds the maximum value
@param in the input stream to read from
@return the maximum value or 0 if the file has no numbers

(adapted from BIG C++ book, by JAS)
*/

#include <iostream>
#include <string>
#include <fstream>

using namespace std;

double max_value(istream &in) // can be called with 'infile' or 'cin'
{
      double highest;
      double next;
      if (in >> next)
            highest = next;
      else
            return 0;

      while (in >> next)
      {
            if (next > highest)
                  highest = next;
      }

      return highest;
}

int main()
{
      double max;

      string input;
      cout << "Do you want to read from a file? (y/n) ";
      cin >> input;

      if (input == "y")
      {
            string filename;
            cout << "Please enter the data file name: ";
            cin >> filename;

            ifstream infile;
            infile.open(filename);

            if (infile.fail())
            {
                  cerr << "Error opening " << filename << "\n";
                  return 1;
            }

            max = max_value(infile);
            infile.close();
      }
```

```
	else
	{
		cout << "Insert the numbers. End with CTRL-Z." << endl;
		max = max_value(cin);
	}

	cout << "The maximum value is " << max << "\n";

	return 0;
}
```

TO DO BY STUDENTS:

- What is the output when the file is empty or the user types CTRL-Z as first input?
- Modify the program to solve the 'problem'.

```
// INPUT/OUTPUT - TEXT FILES
// Reads all the numbers in the file rawdata.dat and writes the numbers
// to the screen and to the file neat.dat in a neatly formatted way.
// Illustrates output formatting instructions.
// Adapted from Savitch book

// DON'T FORGET TO PUT FILE rawdata.txt IN THE PROJECT DIRECTORY
// OR IN THE CURRENT DIRECTORY (IF YOU RUN THE PROGRAM FROM THE COMMAND PROMPT)

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <iomanip>

using namespace std;


/*
The numbers are written one per line, in fixed-point notation
with 'decimal_places' digits after the decimal point;
each number is preceded by a plus or minus sign and
each number is in a field of width 'field_width'.
(This function does not close the file.)
*/
void make_neat(ifstream& messy_file, ofstream& neat_file,
               int field_width, int decimal_places);



int main( )
{
      const int FIELD_WIDTH = 12;
      const int DECIMAL_PLACES = 5;

    ifstream fin;
    ofstream fout;

    fin.open("rawdata.txt");
    if (fin.fail( ))           //Could have tested if(fin.is_open())
    {
        cerr << "Input file opening failed.\n";
        exit(1);
    }

    fout.open("neatdata.txt");
    if (fout.fail( ))
    {
        cerr << "Output file opening failed.\n";
        exit(2);
    }

    make_neat(fin, fout, FIELD_WIDTH, DECIMAL_PLACES);

    fin.close( );
    fout.close( );

    cout << "End of program.\n";
    return 0;
}
```

```
//Uses iostream, fstream, and iomanip:
void make_neat(ifstream &messy_file, ofstream &neat_file,
               int field_width, int decimal_places)
{
      double next;

    neat_file.setf(ios::fixed);           // not in e-notation
    neat_file.setf(ios::showpoint);       // show decimal point ...
                                          // ... even when fractional part is 0
    neat_file.setf(ios::showpos);         // show + sign
    neat_file.precision(decimal_places);
/*
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.setf(ios::showpos);
    cout.precision(decimal_places);
*/
    while (messy_file >> next)
    {
        //cout << setw(field_width) << next << endl;
        neat_file << setw(field_width) << next << endl;
    }
}


/*

rawdata.txt

10.37     -9.89897
2.313   -8.950  15.0

  7.33333   92.8765
-1.237568432e2

neatdata.txt

   +10.37000
    -9.89897
    +2.31300
    -8.95000
   +15.00000
    +7.33333
   +92.87650
  -123.75684

*/
```

```
//FILES
//Detecting the end of a file with eof() method
//Copies file code.txt to file code_numbered.txt,
//but adds a number to the beginning of each line.
//Illustrates the use of get() member function of istream/ifstream
//Assumes code.txt is not empty.

#include <fstream>
#include <iostream>
#include <cstdlib>

using namespace std;

int main( )
{
  ifstream fin;
  ofstream fout;

  fin.open("code.txt");
  if (fin.fail( ))
  {
    cerr << "Input file opening failed.\n";
    exit(1);
  }

  fout.open("code_numbered.txt");
  if (fout.fail( ))
  {
    cerr << "Output file opening failed.\n";
    exit(1);
  }

  char next;
  int n = 1;
  fin.get(next); //THE ARGUMENT OF get() IS PASSED BY VALUE OR BY REFERENCE?
  fout << n << " ";
  while (! fin.eof( )) //returns true if the program has read past the end of the input file;
                       //otherwise, it returns false
  {
    fout << next;  //NOTE: get() READS SPACE AND NEWLINE CHARACTERS
    if (next == '\n')
    {
      n++;
      fout << n << ' ';
    }
    fin.get(next);
  }

  fin.close( );
  fout.close( );

  return 0;
}

TO DO BY STUDENTS:
try with an empty file; see what happens; solve the "problem"
TIP: investigate the use of get()
```

```
//Appending data to the end of a text file

#include <iostream>
#include <fstream>

using namespace std;

int main( )
{
  ofstream fout;
  fout.open("numbers.txt", ios::app); //TO DO: try with a non-existing file
  fout << "Appended data:\n";
  for (int i=10; i<=19; i++)
    fout << i << endl;
  fout.close( );
  return 0;
}
```

==========

INPUT/OUTPUT – BINARY FILES

==========

```
// A binary file for storing integer values
// JAS - 2015/04/09

#include <iostream>
#include <fstream>

using namespace std;


int main()
{
  fstream f; // read and write stream

  f.open("numbers.dat", ios::out | ios::binary);  // create the file

  //streampos place = 5 * sizeof(int); // start writing at position of 5th integer
  //f.seekp(place); // random access

  for (int x = 65; x <= 65 + 25; x++)   // write 26 integers, starting with 65
    f.write((char *)&x, sizeof(int));

  f.close();

  //--------------------

  // USUALLY THIS PART WOULD BE DONE BY ANOTHER PROGRAM …

  f.open("numbers.dat", ios::in | ios::binary); // open the file for reading

  //streampos place = 7 * sizeof(int); // start reading at position of 7th integer
  //f.seekg(place); // random access

  for (int i = 1; i <= 26; i++)
  {
    int y;
    f.read((char *)&y, sizeof(int));
    cout << "y= " << y << endl;

    // FOR RETRIEVING THE INTEGERS AS CHARS !!!
    // char c;
    // f.read(&c, sizeof(char));
    // cout << "c= " << c << endl;
  }

  f.close();

  return 0;
}
```

```
// Random access to a binary file
// a very crude example
// JAS

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib> // for exit()
using namespace std;

const int MAX_NAMELEN = 10;
const char file[] = "name_age.dat";

//-----------------------------------------------------------------------
typedef struct
{
  char name[MAX_NAMELEN]; //why not "string name" instead of char name[MAX_NAMELEN] ?
  unsigned int age;
} Person;

//-----------------------------------------------------------------------
bool fileExists(const char *filename)
{
  bool exists = false;
  ifstream ifile;
  ifile.open(filename);
  if (ifile.is_open())
  {
    exists = true;
    ifile.close();
  }
  return exists;
}

//-----------------------------------------------------------------------
void showPerson(const Person &p)
{
  cout << setw(MAX_NAMELEN) << p.name << " " << p.age << endl;
}

//-----------------------------------------------------------------------

bool writeFileRecord(fstream &f, const Person *rec, unsigned int recNum)
{
  streampos place = recNum * sizeof(Person); // convert to streampos type
  //cout << "WRITE: place = " << place << endl;

  f.seekp(place); // random access
  if (f.fail()) return false;

  f.write((char *) rec, sizeof(Person)) << flush;
  // flush the output to guarantee that the file is updated before proceeding
  // NOTE: this syntax is possible because write() returns an "istream &"
  if (f.fail()) return false;
  return true;
}
```

```
//---------------------------------------------------------------------
bool readFileRecord(fstream &f, Person *rec, unsigned int recNum)
{

  streampos place = recNum * sizeof(Person); // convert to streampos type
  //cout << "READ: place = " << place << endl;

  if (f.eof())
    f.clear(); // clear flags if last read attempt returned end of file

  f.seekg(place); // random access
  if (f.fail()) return false;

  f.read((char *) rec, sizeof(Person));
  if (f.fail()) return false;

  return true;

}

//---------------------------------------------------------------------
void showFileContents(fstream &f)
{
  Person p;
  int n = 0;

  if (f.eof())
    f.clear();

  f.seekg(0); // go to beginning
  cout << "CONTENTS OF THE FILE: \n";

  // could have used readFileRecord() above; TO DO by students
  while (f.read((char *) &p, sizeof(Person))) //compare w/other blue code
  {
    n++;
    if (f.fail())
    {
      cerr << "Error in reading " << file << endl;
      exit(EXIT_FAILURE);
    }
    if (p.age != -1) // SEE HOW p2 WAS INITIALIZED
      cout << n << ": " << setw(MAX_NAMELEN) << p.name << " " << p.age <<
endl;
  }
}
```

// CONTINUES ON NEXT PAGE

```
int main()
{
  Person p1={"Ana", 20}, p2={"", -1}, p3={"Rui", 21} ;
  // TO DO by students:
  // 1) use an array of struct's
  // 2) alternatively, read data from keyboard

  Person p;
  unsigned int numRec;

  fstream finout; // read and write streams

  if (! fileExists(file))  // if file does not exist ...
  {
    cout << "File does not exist. An empty file will be created.\n";
    finout.open(file, ios::out | ios::binary);  //... create the file
    finout.close();
  }
  // open file in input/output + binary modes
  finout.open(file, ios::in | ios::out | ios::binary);
  if (finout.is_open())
  {
    if (!writeFileRecord(finout,&p1,0)) {cerr << "Write error\n"; exit(1);}
    if (!writeFileRecord(finout,&p2,1)) {cerr << "Write error\n"; exit(1);}
    if (!writeFileRecord(finout,&p3,2)) {cerr << "Write error\n"; exit(1);}
    //if (!writeFileRecord(finout,&p3,10)) {cerr << "Write error\n";
exit(1);}

    // TO DO: use an array of Persons instead of p1, p2, and p3

    showFileContents(finout);

    cout << "numRec ? "; cin >> numRec; //try with other numRec's
    if (readFileRecord(finout,&p,numRec))
    {
      cout << "numRec = " << numRec << ": ";
      showPerson(p);
    }
    else
    {
      cerr << "Read error\n";
      exit(1);
    }

  }
  else
  {
    cerr << file << " could not be opened\n";
    exit(EXIT_FAILURE);
  }

  finout.close();

  return 0;
}
```

QUESTION:
what will happen if you try to see the contents of file name_age.dat using a text editor ?

==========

STRINGSTREAMS

==========

String Streams

- We saw how a stream can be connected to a file.
- A stream can also be connected to a string.
- With stringstreams you can perform input/output from/to a string.
- This allows you to convert numbers
  (or any type with the << and >> stream operators overloaded) to and from strings.
- To use stringstreams =>
  - `#include <sstream>`
- The **`istringstream`** class reads characters from a string
- The **`ostringstream`** class writes characters to a string.

Stringstream uses

- A very common use of string streams is:
  - to accept input one line at a time and then to analyze it further.
    - by using stringstreams you can avoid mixing **`cin >>`** ... and **`getline()`**
    - *see examples in the following pages*
  - to use standard output manipulators to create a formatted string

istringstream

- Using an **`istringstream`**,
  you can read numbers that are stored in a string by using the >> operator:

```
string input = "March 25, 2014";
istringstream instr(input); //initializes 'instr' with 'input'

string month, comma;
int day, year;
instr >> month >> day >> comma >> year;
```

- Note that this input statement yields **`day`** and **`year`** as integers.
  Had we taken the string apart with **`substr`**, we would have obtained only strings.
- Converting strings that contain digits to their integer values is such a common operation
  that it is useful to write a helper function for that purpose:

```
int string_to_int(string s)
{
   istringstream instr;
   instr.str(s); // ALTERNATIVE way to initialize 'instr' with 's'
                 // to the initialization mode used above
   int n;
   instr >> n;
   return n;
}
```

ostringstream

- By writing to a string stream, you can convert numbers to strings.
- By using the << operator, the number is converted into a sequence of characters.

```
ostringstream outstr;
outstr << setprecision(5) << sqrt(2);
```

- To obtain a string from the stream, call the `str` member function.
  - `string output = outstr.str();`
- Example: (builds the string "`January 23, 1955`")

```
string month = "January";
int day = 23;
int year = 1955;
ostringstream outstr;
outstr << month << " " << day << "," << year;
string output = outstr.str();
```

- Converting an integer into a string is such a common operation that is useful to have a helper function for it.

```
string int_to_string(int n)
{
   ostringstream outstr;
   outstr << n;
   return outstr.str();
}
```

String ←→ Number conversion in C++11

- **C++11** introduced some standard library functions that can directly convert basic types to `std::string` objects and vice-versa.
- These functions are declared in declared in `<string>`.
- `std::to_string()` converts basic numeric types to strings.
  - Example:

```
int number = 123;
string text = to_string(number);
```

- The set of functions
  - `std::stoi, std::stol, std::stoll` - convert to integral types
  - `std::stof, std::stod, std::stold` - convert to floating-point values.
  - Example:

```
text = "456"
number = stoi(number);
```

```
/**
READ TIME IN SEVERAL FORMATS
ex:
21:30
9:30 pm
10 am
and show it in "military format" (HH:MM) and "am/pm format" (HH:MM am/pm)
*/
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

/**
Converts an integer value to a string, e.g. 3 -> "3".
@param s an integer value
@return the equivalent string
*/
string int_to_string(int n)
{
      ostringstream outstr;
      outstr << n;
      return outstr.str();       //convert stringstream into string
}
/**
Reads a time from standard input
in the format hh:mm or hh:mm am or hh:mm pm
@param hours filled with the hours
@param minutes filled with the minutes
*/
void read_time(int &hours, int &minutes)
{
      string line;
      string suffix;
      char ch;

      getline(cin, line);

      istringstream instr(line);  //initialize stringstream from string
                                  // ALTERNATIVE:
                                  // istringstream  instr;
                                  // instr.str(line);

     instr >> hours;

      minutes = 0;

      instr.get(ch);   // do {instr.get(ch);} while (ch==' ');  // EFFECT ?
                       // try with 18:45 and 18: 45 and 18 :45 and 18 : 45
      if (ch == ':')
            instr >> minutes;
      else
            instr.unget(); // OR instr.putback(ch);

      instr >> suffix;
      if (suffix == "pm")
            hours = hours + 12;
}
```

```
/**
Computes a string representing a time.
@param hours the hours (0...23)
@param minutes the minutes (0...59)
@param military
 true for military format,
 false for am/pm format,
*/
string time_to_string(int hours, int minutes, bool military)
{
      string suffix;
      string result;

      if (!military)
      {
            if (hours < 12)
                  suffix = "am";
            else
            {
                  suffix = "pm";
                  hours = hours - 12;
            }
            if (hours == 0) hours = 12;
      }

      result = int_to_string(hours) + ":";
      if (minutes < 10) result = result + "0";
      result = result + int_to_string(minutes);

      if (!military)
            result = result + " " + suffix;

      return result;
}


int main()
{
      int hours;
      int minutes;

      do
      {
            cout << "Please enter the time\n";
            cout << "HH[:MM] or HH[:MM] am or HH[:MM] pm (0:0 => END): ";

            read_time(hours, minutes);

            cout << "Military time: "
                  << time_to_string(hours, minutes, true) << "\n";
            cout << "Using am/pm:   "
                  << time_to_string(hours, minutes, false) << "\n";
            cout << endl;

      } while (hours!=0 || minutes!=0);  // TO DO by students

      return 0;
}
```

```
/*
Read fractions and do arithmetic operations with them

STRINGSTREAMS
By using STRINGTREAMS you can avoid mixing cin << ... and getline(cin, ...)
You may always use getline()
*/

/*
TO DO:
Fraction sumFractions(Fraction f1, Fraction f2)
Fraction subtractFractions(Fraction f1, Fraction f2)
Fraction divideFractions(Fraction f1, Fraction f2)
*/

#include <iostream>
#include <iomanip>
#include <cctype>
#include <string>
#include <sstream>

using namespace std;

struct Fraction
{
	int numerator;
	int denominator;
};

/*  // READING / WRITING    DIRECTLY FROM / TO    cin / cout
bool readFraction(Fraction &f)
{
	char fracSymbol;
	int numerator;
	int denominator;
	bool success;

	cout << "n / d ? ";
	cin >> numerator >> fracSymbol >> denominator;
	if (cin.fail())
	{
		cin.clear();
		success = false;
	}
	else
		if (fracSymbol == '/')
		{
			f.numerator = numerator;
			f.denominator = denominator;
			success = true;
		}
		else
			success = false;

	cin.ignore(1000,'\n');

	return success;
}
*/
```

```
bool readFraction(Fraction &f)
{
  string fractionString;
  char fracSymbol;
  int numerator;
  int denominator;
  bool success;

  cout << "n / d ? ";
  getline(cin,fractionString);

  istringstream fractionStrStream(fractionString);
  if (fractionStrStream >> numerator >> fracSymbol >> denominator)
    if (fracSymbol == '/')
    {
      f.numerator = numerator;
      f.denominator = denominator;
      success = true;
    }
    else                // TO DO: write these tests in a different way
      success = false; // suggestion: initialize 'success'
  else
    success = false;
  return success;
}

Fraction multiplyFractions(Fraction f1, Fraction f2)
{
      Fraction f;

      f.numerator = f1.numerator * f2.numerator;
      f.denominator = f1.denominator * f2.denominator;
      return f;

}

void showFraction(Fraction f)
{
      cout << f.numerator << "/" << f.denominator;
}

int main()
{
      Fraction f1, f2, f3;

      cout << "Input 2 fractions:\n";
      if (readFraction(f1) && readFraction(f2))
      {
            f3 = multiplyFractions(f1,f2);

            cout << "Product: ";
            showFraction(f3);
      }
      else
      {
            cout << "Invalid fraction\n";
      }
      cout << endl;

      return 0;
}
```