## 2D, 3D arrays

Instantiation: <type> arr[i][j] (etc. can have as many braces as you like)

i is rows, j is columns

Indexing: arr[row][column]

Passing to functions:

void checkTimes(int arr[][3]); // all but first item must have a defined size  
void f(int arr[][4][3]);

## Structures

Example:

struct Student {  
 int id;  
 string name;  
}; // don't forget to close with semicolon!

Arrays of structures:

Product myProducts[3] {  
 {123, 123, 1.34},  
 {1234, 1234, 2.34},  
 {567, 543, 34.56}  
};

## Pointers

\* - dereference operator

& - address of operator

delete frees up memory for other programs to use. Causes program to lose ownership of memory.

After deleting, set the pointer to nullptr to avoid accidental assumption of ownership later.

-> - dereference + member variable access

Pointer to array of pointers: int\*\* arr = new int\*[4];

## Files

Requires <fstream>

Usage:

fstream data;  
data.open("info.txt", ios::in|ios::out)

Types of file streams: ifstream, ofstream, fstream

Flags: ios::in Allow reads; - ios::out Allow writes; ios::binary; ios::ate Begin at end of file. - ios::app Always write to end.. - ios::trunc Delete all contents of file on open.

Reading/writing data:

// Write to data  
data << "Line 1" << endl;  
// Write binary data to file  
data1.write(reinterpret\_cast<char\*>(&dataToWrite), sizeOf(dataToWrite));  
  
// Read data  
data.getline();  
data.get(); // read one byte or character  
data.put(); // write one byte or character  
// read size bytes into address addr\_to\_write\_to  
data.read(addr\_to\_write\_to, size);

Test whether the file successfully opened using fail() (returns true if failed)

## Destructors

Destructor:

~HasPointer() {  
 // release memory, set to null  
 delete ptrMember;  
 ptrMember = nullptr;

// make sure connections are closed  
 file.close();  
}

## Class Access Specifiers

When inheriting: class DerivedClass : accessSpecifier BaseClass

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Inheritance access specifier | | |
|  |  | Public | Private | Protected |
| Base  access  specifier | Public | Public | Private | Protected |
| Private | Unavailable | Unavailable | Unavailable |
| Protected | Protected | Private | Protected |

## Declaring classes

// on the stack  
Student Sue;  
Student Sue("Sue"); // with a name  
// on the heap  
Student\* Sue = new Student();  
Student\* Sue = new Student("Sue");

## Operator Overloading

// can be any operator  
ClassName operator+(ClassName &rhs);  
// rhs does \*not\* have to be the same as ClassName  
ClassName operator-(int &rhs);

The number of arguments must be the same as it would be for the normal operator.

## Templates

Function templates: generic function that will work with any data type, including user-defined data types.

template <class T1, class T2>  
void print(T1 item, T2 item2) {  
 cout << item << " " << item2;  
}

Class templates: generic class for any data type.

template <class T>  
class ItemHolder {  
 T item;  
}  
  
// initialize:  
ItemHolder<int>\* itemHolder = new ItemHolder<int>();

## Exceptions

Used to signal *unexpected* events that occur while a program is running. Should *not* be used for normal execution.

Unhandled exceptions will cause the program to abort execution.

To throw an exception: throw argument; // any argument

try {  
 // some code here  
} catch (ExceptionType one) {  
 // handle an ExceptionType  
} catch (...) {  
 // handle \*any\* type of exception  
}

* Exceptions should not be a replacement for condition checking.
* Try blocks should be kept as short as possible.
* Throw exceptions instead of using error codes.

## Stacks, Queues, Lists

### Stacks

LIFO data structures

Push:

template <class T>  
void Stack<T>::push(T\* item) {  
 if (next == maxLength) {  
 cout << "Failed to push.";  
 throw FullStackException();  
 }  
 stack[next] = item;  
 next++;  
}

Pop:

template <class T>  
T\* Stack<T>::pop() {  
 if (next == 0) {  
 throw EmptyStackException();  
 }  
 next--;  
 return stack[next];  
}

### Queues

Lines of items. FIFO.

Dequeue:

template <class T>  
T\* Queue<T>::Dequeue() {  
 if (Length() == 0) {  
 throw EmptyQueueException();  
 }  
 T retval = list[0];  
  
 for (int i = 0; i < back - 1; i++) {  
 list[i] = list[i+1];  
 }  
 back--;  
 return retval;  
}

Enqueue:

template <class T>  
T\* Queue<T>::Enqueue(T\* item) {  
 if (IsFull()) {  
 throw FullQueueException();  
 }  
 list[back] = val;  
 back++;  
}

Operations: Enqueue, Dequeue, Peek, Length, IsEmpty, IsFull, MakeEmpty

### Ordered lists

Adding to the list:

template <class T>  
void OrderedList<T>::AddItem(T\* item) {  
 if (IsFull()) {  
 // cannot add item to a full list  
 throw FullListError();  
 } else {  
 // walk through the list from the beginning  
 // until we find a spot for the item  
 int i = 0;  
 while (i < next && \*item < \*list[i]) {  
 i++;  
 }  
 // now shift the array to the right of the  
 // correct position right one  
 for (int j = next; j > i; j--) {  
 list[j] = list[j-1];  
 }  
 // increase the length  
 next++;  
 // and place the item into the correct position  
 list[i] = item;  
 }  
}

Removing from the list:

template <class T>  
T\* OrderedList<T>::RemoveItem(T\* item) {  
 if (IsEmpty()) {  
 // item cannot be in the list  
 throw EmptyListError();  
 } else {  
 // walk through the list to find item  
 int i = 0;  
 while (i < next && \*item != \*list[i]) {  
 i++;  
 }  
 if (i == next) {  
 // item isn't in the list  
 throw NotFoundError();  
 } else {  
 // store item at the found position  
 T\* temp = list[i];  
  
 // now shift the list right of the item left  
 for (int j = i; j < next-1; j++) {  
 list[j] = list[j+1];  
 }  
  
 // reduce the length  
 next--;  
 // and return the item  
 return temp;  
 }  
 }  
}