Dynamic (heap) memory and pointers

- many times we do not know how much data we'll need to store until the program is already running
- if we try to pick a fixed size we run the risk of either it being too small (so the program can't do its job) or of being way too big and wasting memory resources
- dynamic memory allocation refers to programs requesting memory as the program runs, then releasing that space once they no longer need it
- these requests are allocated from the “heap” space, and their locations in memory are tracked through pointers

Dynamic array allocation

- one of the most common uses of dynamic allocation is to allocate arrays of just the right size
  - as the program runs we determine what the “right” size is, then request an array of exactly the right size
  - if the request is granted we are given back a pointer to its location in memory
  - we use the array (through the pointer) for as long as it is needed
  - when finished we deallocate the array

Array allocation/deallocation

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

int main()
{
   int size;
   cout << “Enter the desired array size”;
   cin >> size; // error checking needed

   // request space for size doubles using new
   double* arr = new (std::nothrow) double[size];
   if (arr == NULL) {
      cout << “Not enough memory”;
   }
   else {
      // use the array normally, e.g.
      for (int i = 0; i < size; i++) {
         cin >> arr[i];
      }

      // delete when done
      delete [ ] arr;
   }
}
```

Memory allocation requests

- new, malloc, and calloc can each be used to request dynamically allocated memory
- we'll focus on new, since it offers extra functionality that is useful when we get to the use of classes/objects
- not all requests succeed: sometimes there just isn't enough memory available in one contiguous block
- malloc and calloc return NULL if they fail, new either throws an *exception* (to be discussed later) or , if we add an *std::nothrow* option, returns NULL

Notes about new and delete

- as we'll cover exceptions later, for the moment we'll have new return NULL if it cannot fulfill the request
- NULL (aka memory address 0) is an unusable memory address, and is often used as a sentinel or marker value
- after calling new we'll check for NULL to see if it worked or not
- once we're totally done using the dynamically-allocated array we'll call *delete [ ]* to release the memory

Functions returning points

- sometimes we use a function to carry out the allocation and return the pointer to the new array, e.g.

```
// try to allocate an array of the given size
// return the resulting pointer
int* allocArrInt(int size)
{
  int *ptr = NULL;
  if (size > 0) {
    int *ptr = new (std::nothrow) int[size];
    if (ptr == NULL) {
      cout << “Insufficient memory” << endl;
    }
  } else {
     cout << “Invalid array size” << endl;
  }
  return ptr;
}
```

```
int main()
{
   int arrsize = 0;
   cout << “Enter the desired array size”;
   cin >> arrsize;
   int* array = allocArrInt(arrsize);
   if (array != NULL) {
      // use the array normally,
      // ....
      // but delete it when you're done
      delete [] array;
   }
}
```

Passing pointers by reference

- we can even have the function take the pointer as a pass by reference parameter, and return a bool specifying whether the allocation succeeded or failed

```
bool allocate(int* &arr, int size)
{
   if (size > 0) {
       arr = new (std::nothrwos) int[size];
   } else {
       arr = NULL;
   }
   if (arr == NULL) {
      return false;
   }
   return true;
}
```

```
int main()
{
   int *array = NULL;
   int size;
   cin >> size;
   if (allocate(array, arrsize)) {
      //.... use array normally then delete
      delete [] array;
   } else {
      cout << “Allocation failed” << endl;
   }
}
```

Dynamically allocating 2d array

- if we want to dynamically allocate a 2d array, we dynamically allocate an array of pointers for the rows, then go through each row and dynamically allocate a pointer for the columns in that row, e.g.

```
int rows, cols;
float **array2d;

cout << “Enter num rows and cols”;
cin >> rows >> cols;
// alloc array of ptrs for the indiv rows
array2d = new (float*)[rows];
if (array2d == NULL) {
   cout << “Alloc of rows failed”;
}
```

```
else {
   for (int r = 0; r < rows; r++) {
       // alloc the array of floats for the current row
       array2d[r] = new float[cols];
       if (array2d[r] == NULL) {
          cout << “Alloc failed for row “ << r << endl;
       }
   }
}
```

Deallocating dynamic 2d array

- must deallocate the individual arrays of floats first, then the array of pointers
- remember to check for nulls first, calling delete [] on a null pointer can cause a crash

```
if (array2d != NULL) {
    for (int r = 0; r < rows; r++) {
        // delete the current row of floats (if it isn't NULL already)
        if (array2d[r] != NULL) {
            delete [ ] array2d[r];
        }
    }
    // delete the array of pointers
    delete [ ] array2d;
}
```

Dynamic data structures

- another form of dynamic allocation comes into play when we need to incrementally add new items to data storage as the program goes along
  - e.g. we have a system that queues and processes requests for ticket purchases online: as each new request comes in we allocate space for it and add it to the queue, and as we process them we remove them from the queue (and deallocate them after processing)
- we'll consider two specific dynamic data structures shortly: linked lists and trees

Common pointer bugs

- pointer bugs can lead to many odd program crashes
  - wild pointers: using an uninitialized pointer variable, that could thus point anywhere
  - null pointers: dereferencing a pointer that has been set to null, always causes a crash (as does trying to delete a null pointer)
  - dangling pointers: dereferencing a pointer after we have deleted the memory it points to, the memory could now be in use for something else
  - memory leaks: forgetting to delete dynamically allocated memory before we use the pointer for something else

Buggy examples

- wild pointer

```
int x;
int* iptr; // uninitialized, could point anywhere
x = (*iptr);
```

- trying to delete a null pointer

```
iptr = NULL;
delete [] iptr; // crash
```

- dereferencing null pointer

```
iptr = NULL;
x = (*iptr); // crash
```

More buggy examples

- dangling pointer

```
int* arr = new (std::nothrow) int[10];
// .... does stuff with arr
delete [ ] arr;
cout << arr[i]; // arr could already have been reallocated
```

- memory leak

```
int *arr = new (std::nothrow) int[10];
arr = NULL; // have lost all access to the allocated space
```