Dynamic Array

DIGITAL ARTS & ENTERTAINMENT

Memory organisation

- The memory a program uses:
  - Code area → compiled code area
  - Global area → global and static variables
  - Heap → dynamically allocated memory (now)
  - Stack → function arguments and local variables

Memory allocation

- Three basic types of memory locations:
  1. Global memory area:
     - for static and global variables
     - allocated once when the program is started
     - persist throughout the lifetime of the program
  2. Stack memory area:
     - for function arguments and local variables
     - allocated on the stack when the relevant block is entered
     - released when the block is exited
  3. Heap memory area:
     - Dynamic memory area → topic of today

Limitations of static array?

```
void main()
{
  const int arraySize{ 10 };
  int intArray[arraySize];

  for (int i = 0; i < arraySize; i++)
  {
    intArray[i] = rand();
  }
}
```

- size of array must be known at compile time.
- the size of the array is limited by the size of the stack.

Dynamic Array

- Dynamic array:
  - Uses heap memory
  - Manual management at runtime (new / delete)
  - Size must NOT be known at compile time.
  - Size is NOT limited by the available stack memory.

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize];

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- Number of elements can be determined at run time.
- arraySize does not need to be const.

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize];

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- It is allocated at run-time.
- The operator new is used to allocate dynamic memory.
- new is followed by the data type specifier
- and the number of elements between square brackets [ ].

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize];

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- The operator new is used to allocate dynamic memory.
- new is followed by the data type specifier
- and the number of elements between brackets.

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize];

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- The new operator returns a pointer to the first element of the new array.
- The elements are not automatically initialized. (!)
- Allocation can fail, resulting in an exception. (if nothrow, then a nullpointer)

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize];

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- Working with the dynamic array is similar to working with a static array.

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize];

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- When no longer needed, the memory must be freed.
- Using the delete[] operator.
- Failing to delete results in unreleased memory, and if in a loop: a memory leak

DIGITAL ARTS & ENTERTAINMENT

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize];

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- delete[] does not delete nor clear the pointer variable pIntArray, it still points at the freed memory location (!) → dangling pointer
- Using a dangling pointer leads to undefined behavior
- Avoid dangling pointers by setting the pointer to nullptr:

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize];

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- Don't delete[] twice
- second time delete[] happens on a dangling pointer
- resulting in a delete-twice-error

Dynamic Array

```
int arraySize{ };
std::cin >> arraySize;
int* pIntArray = new int[arraySize]{};

for (int i = 0; i < arraySize; i++)
{
  pIntArray[i] = rand();
}
delete[] pIntArray;
pIntArray = nullptr;
```

- Add the uniform initializer braces to initialize the array to the type default value using uniform initialization.

Dynamic Array

```
//The returned pointer is the only way to access the dynamic array:
int *pNumbers { new int[42]{} };

//Dereferencing the pointer to access the elements is not very readable
std::cout << *(pNumbers + 2);

//Use the array index [ ] operator instead
std::cout << pNumbers[2];
pNumbers[5] = 100;
int i { pNumbers[12] };
```

Dynamic Array

- No need to check a pointer for value nullptr before deleting.
  - A nullptr value of a pointer means that no memory address has been allocated to this pointer.
  - Deleting a nullptr has no effect.

```
if(pIntArray != nullptr) delete[] pIntArray;
```

Redundant code

Dynamic Array

- Memory leaks
  - What if:

```
void Draw()
{
        int* pIntArray = new int[arraySize]{};
}
```

  - Dynamically allocated memory has no scope
  - The pointer variable has (!)
  - When the function ends, the only link to the memory is lost, resulting in a memory leak.

Dynamic Array

- Memory leaks
  - What if:

```
int * pNumbers { new int[8]{ } };
pNumbers = new int[5]{ };
```

  - The pointer variable is reassigned to another value. The first memory allocation becomes a leak.

Dynamic Array

- Memory leaks
  - The program can "eat away" all available computer memory, leading to a crash !
  - Fortunately, when a program is terminated, the operating system will release all memory, including the leaked memory.
  - If it happens in the game loop, sooner or later the game will crash

```
while (true)
{
  new int[100];
}
```

don't !!

Dynamic Array

When to use dynamic arrays

- When size is not known at compile time.
- When array will not fit on the stack and should not be global.

Dynamic Array

Advantages compared to static array

- control over lifetime
- size determined at runtime
- larger array sizes are possible

Disadvantage compared to static array

- Allocating takes more time.
- Needs to be manually created/deleted.

Dynamic array

- `int * pDynArray { new int[125] }; -> array`
- `int * pDynArray { new int{125} }; -> no array!!`
- `int * pDynArray { new int(125) }; -> no array!!`

- !! Attention !!

Dynamic array

- Initializing dynamic arrays using uniform initialization

```
int *pNumbers{ new int[5]{} }; // to 0,0,0,0,0

int *pNumbers{ new int[5]{ 5,8,7,4,1 } }; // to values

int *pNumbers{ new int[5]{ 4,5 } }; // to 4,5,0,0,0
```

Dynamic array

- The pointer is the only way to access the dynamic memory:
  - In case of an array, use the [ ] operator to access the elements.
  - The pointer behaves like a decayed array: it does not know the size of the array.

```
int *pNumbers { new int[15] {5} };
std::cout << pNumbers[0] << '\n'; // prints 5

std::cout << sizeof(pNumbers) << '\n'; // prints size of the pointer: 4 (32 bit) or 8 (64 bit)
```

Dynamic array

- Example

```
int main()
{
  std::cout << "Enter a positive integer: ";
  int length{};
  std::cin >> length;

  int *pArray { new int[length] };

  pArray[0] = 5;

  delete[] pAray;
  pArray = nullptr;

 return 0;
}
```

References

- http://www.learncpp.com/cpp-tutorial/69-dynamic-memory-allocation-with-new-and-delete/
- https://www.cplusplus.com/doc/tutorial/dynamic/