Module 10: Programming in C++

Dynamic Memory Management

Intructors: Abir Das and Sourangshu Bhattacharya

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

- Understand the dynamic memory management in C++

1. Dynamic Memory Management in C
   - `malloc` & `free`

2. Dynamic Memory Management in C++
   - `new` and `delete` operator
   - Dynamic memory allocation for Array
   - Placement `new`
   - Restrictions

3. Operator Overloading for Allocation and De-allocation

4. Module Summary

Program 10.01/02: `malloc()` & `free()`: C & C++

Module 10

Intructors: Abir
Das and
Sourangshu
Bhattacharya

Objectives &
Outline

Memory
Management in C
malloc & free

Memory
Management in
C++
new & delete
Array
Placement new
Restrictions

Overloading new
& delete

Module Summary

C Program	C++ Program
| ```c
#include <stdio.h>
#include <stdlib.h>


int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    printf("%d", *p);  // Prints: 5

    free(p);
}
``` | ```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    cout << *p;  // Prints: 5

    free(p);
}
``` |

- Dynamic memory management functions in `stdlib.h` header for C (`cstdlib` header for C++)
- `malloc()` allocates the memory on heap or free store
- `sizeof(int)` needs to be provided
- Pointer to allocated memory returned as `void*` – needs cast to `int*`
- Allocated memory is released by `free()` from heap or free store
- `calloc()` and `realloc()` also available in both languages

- C++ introduces operators `new` and `delete` to dynamically allocate and de-allocate memory:

Functions `malloc()` & `free()`	`operator new` & `operator delete`

``cpp
#include <iostream>
#include <cstdlib>
using namespace std;
 	
int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;
    cout << *p;   // Prints: 5
 	
    free(p);
}
``
	``cpp
	#include <iostream>
	
	using namespace std;
	
	int main() {
	    int *p = new int(5);
	
	    cout << *p; // Prints: 5
	
	    delete p;
	}
	``

• Function `malloc()` for allocation on heap • `sizeof(int)` needs to be provided • Allocated memory returned as `void*` • *Casting* to `int*` *needed* • *Cannot be initialized* • Function `free()` for de-allocation from heap • Library feature – header `cstdlib` needed	• `operator new` for allocation on heap • *No size* specification needed, *type suffices* like here int • Allocated memory returned as `int*` • *No casting needed* • *Can be initialized* • `operator delete` for de-allocation from heap • Core language feature – no header needed

- C++ also allows `operator new()` and `operator delete()` functions to dynamically allocate and de-allocate memory:

Functions `malloc()` & `free()`	Functions `operator new()` & `operator delete()`

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

int main() {
    int *p = (int *)malloc(sizeof(int));
    *p = 5;

    cout << *p;   // Prints: 5

    free(p);
}
```

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

int main() {
    int *p = (int *)operator new(sizeof(int));
    *p = 5;

    cout << *p;   // Prints: 5

    operator delete(p);
}
```

- Function `malloc()` for allocation on heap
- Function `free()` for de-allocation from heap

- Function `operator new()` for allocation on heap
- Function `operator delete()` for de-allocation from heap

There is a major difference between `operator new` and function `operator new()`. We explore this angle later

Program 10.05/06: `new[]` & `delete[]`: Dynamically managed Arrays in C++

Module 10

Intructors: Abir
Das and
Sourangshu
Bhattacharya

Objectives &
Outline

Memory
Management in C

malloc & free

Memory
Management in
C++

new & delete

Array

Placement new

Restrictions

Overloading new
& delete

Module Summary

Functions `malloc()` & `free()`	operator `new[]` & operator `delete[]`
| ```cpp
#include <iostream>
#include <cstdlib>
using namespace std;

int main() {
    int *a = (int *)malloc(sizeof(int)* 3);
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
             << a[i] << "       ";

    free(a);
}
-----
a[0] = 10     a[1] = 20     a[2] = 30
``` | ```cpp
#include <iostream>
using namespace std;


int main() {
    int *a = new int[3];
    a[0] = 10; a[1] = 20; a[2] = 30;

    for (int i = 0; i < 3; ++i)
        cout << "a[" << i << "] = "
             << a[i] << "       ";

    delete [] a;
}
-----
a[0] = 10     a[1] = 20     a[2] = 30
``` |
| • Allocation by `malloc()` on heap

• # of elements implicit in size passed to `malloc()`
• Release by `free()` from heap | • Allocation by operator `new[]` (**different from** operator `new`) on heap
• # of elements explicitly passed to operator `new[]`
• Release by operator `delete[]` (**different from** operator `delete`) from heap |

```cpp
#include <iostream>
using namespace std;
int main() { unsigned char buf[sizeof(int)* 2]; // Byte buffer on stack
    // placement new in buffer buf
    int *pInt = new (buf) int (3);
    int *qInt = new (buf+sizeof(int)) int (5);

    int *pBuf = (int *)(buf + 0);           // *pInt in buf[0] to buf[sizeof(int)-1]
    int *qBuf = (int *)(buf + sizeof(int)); // *qInt in buf[sizeof(int)] to buf[2*sizeof(int)-1]
    cout << "Buf Addr  Int Addr" << pBuf << " " << pInt << endl << qBuf << " " << qInt << endl;
    cout << "1st Int  2nd Int" << endl << *pBuf << "          " << *qBuf << endl;

    int *rInt = new int(7); // heap allocation
    cout << "Heap Addr  3rd Int" << endl << rInt << "   " << *rInt << endl;
    delete rInt;            // delete integer from heap
    // No delete for placement new
}
```

```
-----
Buf Addr   Int Addr
001BFC50   001BFC50
001BFC54   001BFC54
1st Int   2nd Int
3         5
Heap Addr   3rd Int
003799B8   7
```

- Placement operator new takes a buffer address to place objects
- These are not dynamically allocated on heap – may be allocated on stack or heap or static, wherever the buffer is located
- Allocations by Placement operator new must not be deleted

Mixing Allocators and De-allocators of C and C++

Intructors: Abir
Das and
Sourangshu
Bhattacharya

Objectives &
Outline

Memory
Management in C
malloc & free

Memory
Management in
C++
new & delete
Array
Placement new
Restrictions

Overloading new
& delete

Module Summary

- Allocation and De-allocation must correctly match.
  - Do not free the space created by new using free()
  - And do not use delete if memory is allocated through malloc()

These may result in memory corruption

Allocator	De-allocator
malloc()	free()
operator new	operator delete
operator new[]	operator delete[]
operator new()	No delete

- Passing NULL pointer to delete operator is secure
- Prefer to use only new and delete in a C++ program
- The new operator allocates exact amount of memory from Heap or Free Store
- new returns the given pointer type – no need to typecast
- new, new[ ] and delete, delete[ ] have separate semantics

```cpp
#include <iostream>
#include <stdlib.h>
using namespace std;

void* operator new(size_t n) { // Definition of Operator new
    cout << "Overloaded new" << endl;
    void *ptr = malloc(n);      // Memory allocated to ptr. Can be done by function operator new()
    return ptr;
}
void operator delete(void *p) { // Definition of operator delete
    cout << "Overloaded delete" << endl;
    free(p);                    // Allocated memory released. Can be done by function operator delete()
}
int main() { int *p = new int; // Calling overloaded operator new
    *p = 30;            // Assign value to the location
    cout << "The value is :" << *p << endl;
    delete p;           // Calling overloaded operator delete
}
-----
Overloaded new
The value is :  30
Overloaded delete
```

- operator new overloaded
- The first parameter of overloaded operator new must be size_t
- The return type of overloaded operator new must be void*
- The first parameter of overloaded operator delete must be void*
- The return type of overloaded operator delete must be void
- More parameters may be used for overloading
- operator delete should not be overloaded (usually) with extra parameters

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

void* operator new [] (size_t os, char setv) { // Fill the allocated array with setv
    void *t = operator new(os);
    memset(t, setv, os);
    return t;
}
void operator delete[] (void *ss) {
    operator delete(ss);
}
int main() {
    char *t = new('#')char[10]; // Allocate array of 10 elements and fill with '#'

    cout << "p = " << (unsigned int) (t) << endl;
    for (int k = 0; k < 10; ++k)
        cout << t[k];

    delete [] t;
}
-----
p = 19421992
##########
```

- operator new[] overloaded with initialization
- The first parameter of overloaded operator new[] must be size_t
- The return type of overloaded operator new[] must be void*
- Multiple parameters may be used for overloading
- operator delete [] should not be overloaded (usually) with extra parameters

- Introduced `new` and `delete` for dynamic memory management in C++
- Understood the difference between `new`, `new[]` and `delete`, `delete[]`
- Compared memory management in C with C++
- Explored the overloading of `new`, `new[]` and `delete`, `delete[]` operators