**Memory Related Implementation**

**1. Aligned malloc and free**

#include <stdlib.h>

#include <stdio.h>

void\* aligned\_malloc(size\_t n\_bytes, size\_t alignment)

{

void\* p1; // original block

void\*\* p2; // aligned block

int offset = alignment - 1 + sizeof(void\*); // sizeof(void\*) to store p1’s address

if ((p1 = (void\*)malloc(n\_bytes + offset)) == NULL)

{

return NULL;

}

p2 = (void\*\*)(((size\_t)(p1) + offset) & ~(alignment - 1)); // size\_t return the address

p2[-1] = p1; // keep the original malloc address to free

return p2;

}

void aligned\_free(void \*p)

{

free(((void\*\*)p)[-1]);

}

You need an offset if you want to support alignments beyond what your system's malloc() does. For example if your system malloc() aligns to 8 byte boundaries, and you want to align to 16 bytes, you ask for 15 bytes extra so you know for sure you can shift the result around to align it as requested. You also add sizeof(void\*) to the size you pass to malloc() to leave room for bookkeeping.

~(alignment - 1) is what guarantees the alignment. For example if alignment is 16, then subtract 1 to get 15, aka 0xF, then negating it makes 0xFF..FF0 which is the mask you need to satisfy the alignment for any returned pointer from malloc(). Note that this trick assumes alignment is a power of 2 (which practically it normally would be, but there really should be a check).

It's a void\*\*. The function returns void\*. This is OK because a pointer to void is "A pointer to any type," and in this case that type is void\*. In other words, converting void\* to and from other pointer types is allowed, and a double-pointer is still a pointer.

The overall scheme here is to store the original pointer before the one that's returned to the caller. Some implementations of standard malloc() do the same thing: stash bookkeeping information before the returned block. This makes it easy to know how much space to reclaim when free() is called.

All that said, this sort of thing is usually **not useful, because the standard malloc() returns the largest alignment on the system**. If you need alignment beyond that, there may be other solutions, including compiler-specific attributes.

**2. Aligned (32 bytes) memcpy and memmove**

void myMemCpy(void \*dest, void \*src, size\_t n)

{

// Typecast src and dest addresses to (char \*)

char \*csrc = (char \*)src;

char \*cdest = (char \*)dest;

for (int i=0; i<n; i++) // Copy contents of src[] to dest[]

cdest[i] = csrc[i];

}

// It assumes that dst and src are misaligned at the same degree

void aligned\_memory\_copy(void\* dst, void\* src, unsigned int bytes)

{

unsigned char\* b\_dst = (unsigned char\*)dst;

unsigned char\* b\_src = (unsigned char\*)src;

// Copy bytes to align source pointer

while ((b\_src & 0x3) != 0) {

\*b\_dst++ = \*b\_src++;

bytes--;

}

unsigned int\* w\_dst = (unsigned int\*)b\_dst;

unsigned int\* w\_src = (unsigned int\*)b\_src;

while (bytes >= 4) {

\*w\_dst++ = \*w\_src++;

bytes -= 4;

}

if (bytes > 0) { // Copy trailing bytes

b\_dst = (unsigned char\*)w\_dst;

b\_src = (unsigned char\*)w\_src;

while (bytes > 0) {

\*b\_dst++ = \*b\_src++;

bytes--;

}

}

}

void myMemMove(void \*dest, void \*src, size\_t n)

{

// Typecast src and dest addresses to (char \*)

char \*csrc = (char \*)src;

char \*cdest = (char \*)dest;

// Create a temporary array to hold data of src, to allow to overlap dest and src

char \*temp = new char[n];

for (int i=0; i<n; i++) // Copy data from csrc[] to temp[]

temp[i] = csrc[i];

for (int i=0; i<n; i++) // Copy data from temp[] to cdest[]

cdest[i] = temp[i];

delete [] temp;

}

**3. 2D malloc to enable ‘arr[i][j]’**

// Create a 1D array of pointers. Then, for each array index, we create a new one-dimensional

// array. This gives us a two-dimensional array that can be accessed via array indices.

int\*\* My2DAlloc(int rows, int cols){

int \*\*arr = (int\*\*)malloc(rows\*sizeof(int\*)); // array of pointers for each row

for(int i=0; i<rows; ++i)

arr[i] = (int\*)malloc(cols\*sizeof(int)); // allocate each row

return arr;

}

void My2DDealloc(int \*\*rowptr, int rows) {

for(int i=0; i<rows; i++) {

free(rowptr[i]); // deallocate each row pointer

}

free(rowptr); // deallocate the array of “row” pointers

}

// One malloc call version

int\*\* My2DAlloc1(int rows, int cols){

int header = rows \* sizeof(int\*); // storage for each row's starting address

int data = rows \* cols \* sizeof(int); // array data

int \*\*arr = (int\*\*)malloc(header + data);

int \*buf = (int\*)(arr + rows); // starting address of array data

for(int i=0; i<rows; ++i)

arr[i] = buf + i \* cols; // store each row's staring address in the header array

return arr;

}

void My2DDealloc1(int \*\*arr) {

free(arr);

}

int main(){

int \*\*arr = My2DAlloc1(4, 5);

arr[2][3] = 23;

cout<<arr[2][3]<<endl;

My2DDealloc1(arr);

return 0;

}

4. Smart Pointer Class

#include <iostream>

#include <cstdlib>

using namespace std;

template <typename T>

class SmartPointer{

protected:

T \*ref;

unsigned \*ref\_count; // track the number of pointers to one object

public:

SmartPointer(T\* ptr){ // constructor

ref = ptr;

ref\_count = (unsigned\*)malloc(sizeof(unsigned));

\*ref\_count = 1;

// or ref\_count = new unsigned(1);

}

// this constructor creates a new smart pointer that points to an existing object.

// We will need to first set obj and ref\_count to pointer to sptr's obj and ref\_count.

// Then, because we created a new reference to obj, we need to increment ref\_count

SmartPointer(SmartPointer<T> &sptr){

ref = sptr.ref;

ref\_count = sptr.ref\_count;

++\*ref\_count;

}

// create by Assignment operator

// Decrement its reference count, then copy the pointers to obj and ref\_count over.

// Finally, seince we created a new reference, we need to increment ref\_count

SmartPointer<T>& operator=(SmartPointer<T> &sptr){

if (this != &sptr) {

if (--\*ref\_count == 0){

clear();

}

ref = sptr.ref;

ref\_count = sptr.ref\_count;

++\*ref\_count;

}

return \*this;

}

// Destructor: destory a reference to the object.

// Decrement ref\_count. if ref\_count is 0, then free the memory created by the integer and

// destory the object

~SmartPointer(){

if (--\*ref\_count == 0){

clear();

}

}

T getValue() { return \*ref; }

private:

void clear(){

delete ref;

//free(ref\_count);

delete ref\_count;

ref = NULL;

ref\_count = NULL;

}

};

int main(){

int \*ip1 = new int();

\*ip1 = 11111;

int \*ip2 = new int();

\*ip2 = 22222;

SmartPointer<int> sp1(ip1), sp2(ip2);

SmartPointer<int> spa = sp1;

sp2 = spa;

return 0;

}