Pointers in C++ - strict typing
Pointers - to any defined type
Pointers and const qualified
Memory allocation and deallocation
Dynamic allocation within class object
Copy constructors
Common pointer errors
Case Study 3
Pointers in C++ - strict typing

⇒ Enables sensible pointer arithmetic
  Pointer increment, decrement, assignment, etc.
⇒ General arithmetic doesn’t make sense

Example:

```cpp
class Rectangle;
const int i = 3;
Rectangle rectArray[5];
Rectangle* rect1 = &rectArray[0]; // assign.
Rectangle* rect2 = rect1++; // assign & increment
rect2 += i; // with increment variables
Rectangle* rect3 = rect1 - rect2; // nonsense
```
Pointers - to any defined type

Example - array elements
Can do arithmetic because strict typing

⇒ Generic pointers - casting needed

```c
#include <iostream>

int iA[] = {10, 12, 5, 7, 9}; // an array of 5 integers
int* ip1 = iA; // points to iA[0]
ip1 = &iA[1]; // now points iA[1]

ip += 2; // address arithmetic - now points to iA[3]

// the integer pointer recast go a float pointer
float* fp1 = (float*)ip1; // same address
float fV = *(fp1); // compiler ok but get info as a float

void* genPtr = (void*)ip1; // assumes byte size increments
int* p = (int*)genPtr; // requires casting
```
Variations:

```c
int iA[30];  // an array of 30 integers
int* ip1 = iA;  // not const - ip1 & iA modified thru ip1
int* const ip2 = iA;  // ip2 fixed, iA[0] modified thru ip2
const int* ip3 = iA;  // iA fixed thru ip3; ip3 pointing changeable
const int* const ip4 = iA;  // ip4 fixed, iA fixed relative to ip4
((int*) ip2)++;  // cast ip2 now points to iA[1]
```
{ // create a new block - variables are known only within the block
    int ival[30]; // reserves 30 consecutive integers is fixed memory
    int* p0 = ival; // points to ival[0]
    // reserve enough dynamic memory for an integer
    int* p1 = new int; // p1 points to location in memory
    // reserve enough dynamic memory for 30 consecutive integers
    int* p2 = new int[30]; // p2 points to first mem. location
    delete p1; } // frees *p1 space. later use of p1 can lead to problems
    // ival, p0, p1 & p2 gone. mem for p2 reserved but lost

⇒ Fixed vs dynamic allocation
⇒ New and delete operators
⇒ Scope and duration
Dynamic allocation in classes

```cpp
class FString
{
    FString(const char* ); // rest of member functions & data
    FString& Assign(const char* );

    private:
    char* stg; short length;

    FString::FString(const char* s) {
        length = strlen(s); // get length to allot space
        stg = new char[length + 1]; // allot space
        stgcpy(stg, s);
    }

    FString::Assign(const char* s) {
        if (stg) delete stg; // remove old stg
        // repeat same code as above
    }

    // class in dynamic memory
    FString* namePtr("John Jackson");
};
```
class FString {
    public:
    FString(char* );     // default constructor
    ~FString();         // destructor

    private: char *stg;   // rest of member list 
};

FString::FString(char* s) {
    stg = new char[strlen(s)+1]; // allocate dynamic memory
    strcpy(stg, s);             // make it a string
} // end constructor

FString::~FString() {
    delete [] stg;             // end destructor
Encapsulate the **new** operator - in class so destructor is automatically activated.

Note: In complex problems - difficult - so remember to **delete**

Uninitialized pointer - try to intialize with new expression at creation e.g.,

```cpp
int* myArray = new int;
```

Dereference null pointer - **int** iVal = *myArray;

Dangling pointer - using pointer after **delete**
Call by value, return value, copy constructors, temporary objects, etc.
C & C++ copy structures are byte transfer in memory
i.e., memcpy(s, ct, n); Hence, pointers are copied directly.

Solution - build your own copy constructor
MyClass::MyClass(const MyClass& aMem) {
/*
1. create new dynamic storage of size needed
2. copy values not memory into this’s new storage */
}

Special case for assignment e.g., overload operator =
need to delete the present value in dynamic memory.
Examine the implementation of Point, Circle and Cylinder as a basis for using derived classes, i.e., inheritance

Examine the FString to see how dynamic memory is allocated, released and how copying is affected