## Table of Contents

- 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;
cont int i = 3;
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

```cpp
int A[] = {10, 12, 5, 7, 9}; // an array of 5 integers
int* ip1 = A; // points to A[0]
ip1 = &A[1]; // now points to A[1]
ip += 2; // address arithmetic - now points to A[3]
if the integer pointer recast to 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
```
Pointers and const qualified

Variations:
- `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;` // cast ip4 new points to iA[1]

Memory allocation & deallocation

```
{ // create a new block - variables are known only within the block
    int iVal[30]; // reserves 30 conseq 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 conseq 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
}
```

Dynamic allocation in classes

```
class FString {
    FString(const char*); // rest of member functions & data
    FString& Assign(const char*); 
private:
    char* stg; short length; 
    FString(const char* s) {
        length = strlen(s); // get length to allot space
        new char[length + 1]; // allot space
        strcpy(stg, s); 
    } 
    FString Assign(const char* s) {
        if (stg) delete stg; // remove old stg
        char* temp = stg; 
        stg = new char[length + 1];
        strcpy(stg, temp); 
        return FString(stg); 
    }
};
```

```
Dynamic deallocation

```cpp
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
}
```

Common pointer errors

- Encapsulate the `new` operator - in class so destructor is automatically activated.
- Note: In complex problems - difficult - so remember to `delete`
- Uninitialized pointer - try to initialize with `new` expression at creation e.g., `int* myArray = new int;`
- Dereference null pointer - `int iVal = *myArray;`
- Dangling pointer - using pointer after `delete`

Copy constructors

- Copying in C++ is ubiquitous
- The problem with dynamic allocation
- 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
```cpp
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.
Case Study 3

- 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