10. Program Set 7

1. Stack vs heap

```cpp
#include <iostream>
int main() {
    int a = 5;
    int* p = new int(5);
    std::cout << a << " " << *p << "\n";
    delete p;
}
```

2. Dangling from local return, don't do this

```cpp
#include <iostream>
int* bad() {
    int x = 5;
    return &x;
}
int main() {
    int* p = bad();
    std::cout << "Dangling demo; do not dereference p!\n";
}
```

3. Correct heap return

```cpp
#include <iostream>
int* make_five() {
    int* p = new int(5);
    return p;
}
int main() {
    int* p = make_five();
    std::cout << *p << "\n";
    delete p;
}
```

4. new[] / delete[] for arrays

```
#include <iostream>
int main() {
    int n = 5;
    int* a = new int[n];
    for (int i = 0; i < n; ++i) {
        a[i] = i * i;
    }
    for (int i = 0; i < n; ++i) {
        std::cout << a[i] << " ";
    }
    std::cout << "\n";
    delete[] a;
}
```

5. Leak demo

```
int main() {
    int* a = new int[1000000];
    // forgot: delete[] a;
    return 0;
}
```

6. Double delete causes undefined behaviour

```
#include <iostream>
int main() {
    int* p = new int(7);
    delete p;
    // delete p; // UB if uncommented
    std::cout << "Freed once; second delete would be undefined
behaviour.\n";
}
```

7. Delete expression does not match data type, a wrong match can cause undefined behaviour

```
int main() {
    int* a = new int[3];
    // delete a; // WRONG
    delete[] a; // RIGHT
}
```

8. new without initializer, then assign value

```
#include <iostream>
int main() {
    int* p = new int;
    *p = 42;
    std::cout << *p << "\n";
    delete p;
}
```

9. Constructors on new / arrays call default constructor

```
#include <iostream>
class X {
public:
    X() {
        std::cout << "X()\n";
    }
};
int main() {
    X* p = new X;
    delete p;
    X* a = new X[3];
    delete[] a;
}
```

10. Dangling after delete

```
#include <iostream> int main() {
    int* p = new int(9);
    delete p;
    std::cout << *p << "\n"; // would be dangling
```

```
    p = 0;
}
```

11. Minimal RAII wrapper

```
#include <iostream>
class IntOwner {
    int* p;
public:
    explicit IntOwner(int v)
        : p(new int(v)) {
    }
    ~IntOwner() {
        delete p;
    }
    int get() const {
        return *p;
    }
};
int main() {
    IntOwner x(10);
    std::cout << x.get() << "\n";
}
```

12. Rule of Three (deep copy)

```
#include <iostream>
#include <cstring>

class Name {
    char* s;
public:
    Name()
        : s(0) {
    }
    explicit Name(const char* t) {
        s = new char[std::strlen(t) + 1];
        std::strcpy(s, t);
    }
    Name(const Name& o) {
```

```
        if (o.s) {
            s = new char[std::strlen(o.s) + 1];
            std::strcpy(s, o.s);
        } else {
            s = 0;
        }
    }
    Name& operator=(const Name& o) {
        if (this != &o) {
            char* ns = 0;
            if (o.s) {
                ns = new char[std::strlen(o.s) + 1];
                std::strcpy(ns, o.s);
            }
            delete[] s;
            s = ns;
        }
        return *this;
    }
    ~Name() {
        delete[] s;
    }
    const char* c_str() const {
        return s ? s : "";
    }
};
int main() {
    Name a("Alice");
    Name b = a;
    std::cout << b.c_str() << "\n";
}
```

13. Shallow copy pitfall

```
#include <iostream>
class Bad {
public:
    char* s; // unclear ownership
};
int main() {
    char buf[] = "hi";
    Bad a;
    a.s = buf;
```

```
    Bad b = a;
    b.s[0] = 'm';
    std::cout << a.s << "\n";
}
```

14. Returning a heap array from a factory

```
#include <iostream>
int* make_arr(int n) {
    int* a = new int[n];
    for (int i = 0; i < n; ++i) {
        a[i] = i;
    }
    return a;
}
int main() {
    int* a = make_arr(5);
    for (int i = 0; i < 5; ++i) {
        std::cout << a[i] << " ";
    }
    std::cout << "\n";
    delete[] a;
}
```

15. Ownership transfer by raw pointer

```
#include <iostream>
int* make_val() {
    int* p = new int(7);
    return p;
}
void take_ownership(int* p) {
    std::cout << *p << "\n";
    delete p;
}
int main() {
    int* p = make_val();
    take_ownership(p);
    p = 0;
}
```

16. Copying an owning buffer (deep copy)

```cpp
#include <iostream>
class Buffer {
    int* data;
    int n;
public:
    explicit Buffer(int N)
        : data(new int[N])
        , n(N) {
        for (int i = 0; i < n; ++i) {
            data[i] = 0;
        }
    }
    Buffer(const Buffer& o)
        : data(new int[o.n])
        , n(o.n) {
        for (int i = 0; i < n; ++i) {
            data[i] = o.data[i];
        }
    }
    Buffer& operator=(const Buffer& o) {
        if (this != &o) {
            int* nd = new int[o.n];
            for (int i = 0; i < o.n; ++i) {
                nd[i] = o.data[i];
            }
            delete[] data;
            data = nd;
            n = o.n;
        }
        return *this;
    }
    ~Buffer() {
        delete[] data;
    }
    int& operator[](int i) {
        return data[i];
    }
    int size() const {
        return n;
    }
};
int main() {
```

```
    Buffer a(3);
    a[1] = 42;
    Buffer b = a;
    std::cout << b[1] << "\n";
}
```

17. Manual cleanup in catch

```
#include <iostream>
void might_throw(bool t) {
    if (t) {
        throw 42;
    }
}
int main() {
    int* p = new int(5);
    try {
        might_throw(true);
    }
    catch (...) {
        delete p;
        p = 0;
    }
}
```

18. RAII beats manual cleanup because destructor automatically runs on end of scope. The memory can be deleted in the destructor.

```
#include <iostream>
class Holder {
    int* p;
public:
    Holder()
        : p(new int(5)) {
    }
    ~Holder() {
        delete p;
    }
};
void might_throw() {
    throw 1;
```

```
}
int main() {
    try {
        Holder h;
        might_throw();
    }
    catch (...) {
        std::cout << "no leak\n";
    }
}
```

19. nothrow new. std::nothrow is used to allocate memory like new. But if new fails, then an error is thrown. If std::nothrow fails, then the pointer variable has a null value and no error is thrown.

```
#include <iostream>
#include <new>
int main() {
    int* p = new (std::nothrow) int[1000000000];
    if (!p) {
        std::cout << "allocation failed\n";
    } else {
        delete[] p;
    }
}
```

20. calling delete more than once on a 0 pointer is safe. There is no undefined behaviour.

```
int main() {
    int* p = 0;
    delete p;
    delete[] p;
}
```

21. Here, Team class has an object for Person. i.e. Team is composed of Person. This concept is called composition. This can also be called has-a. The lifetime of Person is tied to lifetime of Team. When Team is constructed and destroyed, Person is also constructed and destroyed respectively. Here, when Team is destroyed, the destroyer in Person is also executed.

```
#include <iostream>
#include <cstring>
class Person {
    char* name;
public:
    Person()
        : name(0) {
    }
    explicit Person(const char* n) {
        name = new char[std::strlen(n) + 1];
        std::strcpy(name, n);
    }
    ~Person() {
        delete[] name;
    }
    const char* get() const {
        return name ? name : "";
    }
};
class Team {
    Person lead;
public:
    explicit Team(const char* n)
        : lead(n) {
    }
    void print() const {
        std::cout << lead.get() << "\n";
    }
};
int main() {
    Team t("Ada");
    t.print();
}
```

22. Here, the new and delete operator is overloaded. The Tracked* t = new Tracked; line does two things: first the overloaded new operator is called, the default constructor is called. The line delete t; does the reverse, first the default destructor is called, then the overloaded delete is called.

```
#include <iostream>
#include <new>
class Tracked {
public:
```

```
    static void* operator new(std::size_t sz) {
        std::cout << "new " << sz << " bytes\n";
        return ::operator new(sz);
    }
    static void operator delete(void* p) {
        std::cout << "delete\n";
        ::operator delete(p);
    }
};
int main() {
    Tracked* t = new Tracked;
    delete t;
}
```

23. Mini string (deep copy, no leaks)

```
#include <iostream>
#include <cstring>
class MiniStr {
    char* s;
public:
    MiniStr()
        : s(0) {
    }
    explicit MiniStr(const char* t) {
        s = new char[std::strlen(t) + 1];
        std::strcpy(s, t);
    }
    MiniStr(const MiniStr& o) {
        if (o.s) {
            s = new char[std::strlen(o.s) + 1];
            std::strcpy(s, o.s);
        } else {
            s = 0;
        }
    }
    MiniStr& operator=(const MiniStr& o) {
        if (this != &o) {
            char* ns = 0;
            if (o.s) {
                ns = new char[std::strlen(o.s) + 1];
                std::strcpy(ns, o.s);
            }
```

```
            delete[] s;
            s = ns;
        }
        return *this;
    }
    ~MiniStr() {
        delete[] s;
    }
    const char* c_str() const {
        return s ? s : "";
    }
};
int main() {
    MiniStr a("hi");
    MiniStr b = a;
    std::cout << b.c_str() << "\n";
}
```

24. Deleting non-owned memory, which should not be done in practice.

```
#include <iostream>
class Bad {
public:
    const char* s;
    // ~Bad() { delete[] s; } // would be WRONG here
};
int main() {
    Bad b;
    b.s = "literal";
    std::cout << b.s << "\n";
}
```

25. Splitting allocate, initialize, free of memory. If this is done separately without RAII, then the programmer has to keep track of the memory's status (it's allocation, if it exists, or it's freeing)

```
#include <iostream>
void alloc(int*& p, int n) {
    p = new int[n];
}
void init(int* p, int n) {
    for (int i = 0; i < n; ++i) {
```

```
        p[i] = i;
    }
}
void free_(int*& p) {
    delete[] p;
    p = 0;
}
int main() {
    int* a = 0;
    alloc(a, 5);
    init(a, 5);
    for (int i = 0; i < 5; ++i) {
        std::cout << a[i] << " ";
    }
    std::cout << "\n";
    free_(a);
}
```