3. Keeping in memory

Max Cattafi
m.cattafi@imperial.ac.uk

EE2-12 – Software Engineering 2
Object Oriented Software Engineering

Imperial College London

In this lecture

- Remarks, reminders and caveats re: pointers, references and dynamic memory.
- Destructor.
- More operator overloading.

Assigning through a pointer

```
int main(){
    int* a;
    *a = 10;
    cout << a << endl;
    return 0;
}
```

- Does it work?

Assigning through a pointer

```
int main(){
    int* a;
    *a = 10;
    cout << *a << endl;
    return 0;
}
```

- Compiles with (often) no warnings.
- Segmentation fault.
- Where are we storing our value (10)?

Assigning through a pointer (to something)

```
int main(){
    int* a;
    int v_a;
    a = &v_a;

    *a = 10;
    cout << *a << endl;

    a = new int;
    *a = 15;
    cout << *a << endl;

    return 0;
}
```

- Using some other (automatically allocated) variable’s address.
- Dynamic memory allocation (on the heap).

Dynamic memory allocation for objects

```
#include <iostream>
#include "point.hpp"
using namespace std;

int main(){
    Point* p1 = new Point();
    Point* p2 = new Point(1,2);

    cout << p1 -> get_x() << " " << p1->get_y() << endl;
    cout << p2->get_x() << " " << p2->get_y() << endl;

    return 0;
}
```

Returning a pointer

```
int* square(int n){
    int square_n = n*n;
    return &square_n;
}

int main(){
    int n = 10;
    cout << *square(n) << endl;
    return 0;
}
```

- Does it work?

Returning a pointer

```
$ g++ retpoint.cpp -o ret
retpoint.cpp: In function int* square(int):
retpoint.cpp:5: warning: address of local
variable square_n returned
$ ./ret
100
```

- Undefined behaviour.
- Very bad idea.

Returning a pointer (to something)

```
int* square(int n){
    int* square_n = new int;
    *square_n = n*n;
    return square_n;
}

int main(){
    int n;

    while(1){
        cin >> n;
        cout << *square(n) << endl;
    }
    return 0;
}
```

- Dynamic memory allocation on the heap.
- Is it a good idea?

Returning a pointer (to something)

```
int main(){
    int n;
    int *sq;
    while(1){
        cin >> n;
        sq = square(n);
        cout << *sq << endl;
        delete sq;
    }
    return 0;
}
```

- Beware of memory leaks.

The destructor

```
class Point{
    public:
    ...
    ~Point (){
       cout << "Point " << x << " " << y << " is leaving" << endl;
    }
    ...
};
```

The destructor

```
int main(){
    // the following curly brace is not a typo...
    {
        cout << "a new scope begins" << endl;
        Point p1;
        Point* p2 = new Point(1,2);

        cout << "the new scope ends" << endl;
    }

    cout << "goodbye everyone!" << endl;

    return 0;
}
```

```
a new scope begins
the new scope ends
Point 0 0 is leaving
goodbye everyone!
```

The destructor

```
int main(){
    {
        cout << "a new scope begins" << endl;
        Point p1;
        Point* p2 = new Point(1,2);

        cout << "the new scope ends" << endl;
    }
    // delete p2;
    // error: 'p2' was not declared in this scope

    cout << "goodbye everyone!" << endl;
    return 0;
}
```

The destructor

```cpp
int main(){
    {
        cout << "a new scope begins" << endl;
        Point p1;
        Point* p2 = new Point(1,2);

        delete p2;
        cout << "the new scope ends" << endl;
    }

    cout << "goodbye everyone!" << endl;
    return 0;
}
```

```
a new scope begins
Point 1 2 is leaving
the new scope ends
Point 0 0 is leaving
goodbye everyone!
```

The destructor

- In the previous example: defined our destructor in order to better understand when it is called, what happens to dynamic memory etc.
- As far as `Point` is concerned, even without our destructor, no memory leaks as long as `delete` is called on dynamically created objects.
- This is not always the case.
- There can also be other reasons to define a destructor.

Returning a pointer (to some field)

```
class Point{
    public:
    ...
    double* get_x(){
        return &x;
    }
    ...
};

int main(){
    Point p1(1,5);
    cout << *p1.get_x() << endl;
    return 0;
}
```

- Does it work? Is it a good idea?

Returning a pointer (to some field)

```
class Point{
    public:
    ...
    double* get_x() /* look, no const here */ {
        return &x;
    }
    ...
};

int main(){
    Point p1(1, 5);
    cout << *p1.get_x() << endl;
    double* change_x = p1.get_x();

    *change_x = 20;
    cout << *p1.get_x() << endl;

    *(p1.get_x()) = 25;
    cout << *p1.get_x() << endl;

    return 0;
}
```

Outcome

- The value of `x` (from `get_x()`):

```
1
20
25
```

- Breaking encapsulation.

Returning a pointer (to const) to some field

```
class Point{
    public:
    ...
    const double* get_x() const {
        return &x;
    }
    ...
};

int main(){
    Point p1(1,5);
    cout << *p1.get_x() << endl;

//     double* change_x = p1.get_x();
//     *change_x = 20;
//     error: invalid conversion from 'const double*' to 'double*'

//     *(p1.get_x()) = 20;
//     error: assignment of read-only location
//     '* p1.Point::get_x()'
    return 0;
}
```

‘Pointer to const’

- ‘Pointer to const’ doesn’t mean that the pointed memory area is guaranteed to be constant.
- It means we cannot change its value *using that specific pointer*.
- It can still be changed, e.g. by other pointers not declared as const.
- Beware:

```
Point* p1 = new Point(1, 5);
const double* save_x = p1->get_x();
double copy_x = *(p1->get_x());
delete p1;
// copy_x is ok
// where does save_x point to now?
```

Pointers to const vs. const pointers

```
int main(){
    const Point* p1;
    p1 = new Point();
//      Pointer* const p2;
//      error: uninitialized const 'p2'
    Point* const p2 = new Point();

//      p1->origin_symmetric();
//      passing 'const Point' as 'this' argument of
//      'void Point::translate()' discards qualifiers
    p2->origin_symmetric();

    Point p3;
    p1 = &p3;
//      p2 = &p3;
//      error: assignment of read-only variable 'p2'

    delete p2;
    // delete p1;
    // not a good idea (why?)
    return 0;
}
```

Are we missing something?

```
p1 = new Point();
...
Point p3;
p1 = &p3;
```

- The address to the memory area allocated by `new` and previously pointed by `p1` is irremediably lost.
- We cannot access it anymore, we can't free it (using `delete`) either.

```
//    delete p1;
```

- Segmentation fault.
- `p1` now points to an automatically allocated variable (`p3`).
- We cannot 'dynamically free' memory which wasn't dynamically allocated (and why should we?).

Memory leaks

Q: How do I deal with memory leaks?

A: By writing code that doesn't have any.

Clearly, if your code has new operations, delete operations, and pointer arithmetic all over the place, you are going to mess up somewhere and get leaks, stray pointers, etc.

This is true independently of how conscientious you are with your allocations: eventually the complexity of the code will overcome the time and effort you can afford.

It follows that successful techniques rely on hiding allocation and deallocation inside more manageable types.

Good examples are the standard containers. They manage memory for their elements better than you could without disproportionate effort.

[B. Stroustrup, C++ FAQ]

Memory leaks

- Easy to miss un`delete`d memory areas here and there.
- Avoid ‘bare’ `new` (although sometimes useful).
- Use containers (e.g. `vector`) which take care of memory management for you (or write your own wrapper class).
- Garbage collection (default in other languages e.g. Java).

References

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

int main(){
    int a = 0;
    int& new_a=a; // initialization is necessary
    // with pointers: int *new_a = &a;

    cout << "a: " << a << endl;
    cout << "new_a: " << new_a << endl;
    // a: 0
    // new_a: 0

    new_a = 1;
    // with pointers: *new_a = 1;

    cout << "a: " << a << endl;
    cout << "new_a: " << new_a << endl;
    // a: 1
    // new_a: 1
```

References

```
    int b=2;
    new_a = b;
    // caution: this is like *new_a = b;
    // not like new_a = &b;
    // (references cannot be re-assigned)

    cout << "a: " << a << endl;
    cout << "new_a: " << new_a << endl;
    cout << "b: " << b << endl;
    // a: 2
    // new_a: 2
    // b: 2

    new_a = 3;
    cout << "a: " << a << endl;
    cout << "new_a: " << new_a << endl;
    cout << "b: " << b << endl;
    // a: 3
    // new_a: 3
    // b: 2

    return 0;
}
```

Pointers and references

```
int main(){
    int a = 10;
    int* point_a = &a;
    int& new_a = *point_a;

    cout << "a: " << a << " *point_a: " << *point_a << " new_a: " ↩
         << new_a << endl;
    cout << "&a: " << &a << " point_a: " << point_a << " &new_a: "↩
          << &new_a << endl;
```

```
a: 10 *point_a: 10 new_a: 10
&a: 0xbfc6fa5c point_a: 0xbfc6fa5c &new_a: 0xbfc6fa5c
```

Pointers and references

```
//    int a = 10;
//    int* point_a = &a;
//    int& new_a = *point_a;

    new_a = 11;

    cout << "a: " << a << " *point_a: " << *point_a << " new_a: " ↩
        << new_a << endl;
    cout << "&a: " << &a << " point_a: " << point_a << " &new_a: "↩
         << &new_a << endl;
```

```
a: 11 *point_a: 11 new_a: 11
&a: 0xbfc6fa5c point_a: 0xbfc6fa5c &new_a: 0xbfc6fa5c
```

Pointers and references

```
//     int a = 10;
//     int* point_a = &a;
//     int& new_a = *point_a;

//     new_a = 11;

    int b = 12;
    point_a = &b;

    cout << "a: " << a << " *point_a: " << *point_a << " new_a: " ↩
         << new_a << endl;
    cout << "&a: " << &a << " point_a: " << point_a << " &new_a: "↩
          << &new_a << endl;
```

```
a: 11 *point_a: 12 new_a: 11
&a: 0xbfc6fa5c point_a: 0xbfc6fa50 &new_a: 0xbfc6fa5c
```

Returning a reference to a field

```
class Point{
    public:
    ...
    const double& get_x() const {
        return x;
    }
    ...
};

    ...
    Point* p1 = new Point(1, 5);
    const double& save_x = p1->get_x();
    double copy_x = p1->get_x();

    delete p1;
    // copy_x is ok
    // save_x is a reference to...?
    ...
```

Returning pointers and references

- Returning references, like returning pointers, can break encapsulation.
- We have already seen how to avoid it: `const` references.
- In some cases exposing ‘what’s inside’ to change or assignment is the desired behaviour (for instance when?).
- The syntax for pointers can be cumbersome, references are quite smooth.
- Accessor/getter member functions for object member data often return `const` references.

Parameters by pointer to const

```
double Point::distance(const Point& p2) const {
    return sqrt(
        pow( (x - p2.x), 2) +
        pow( (y - p2.y), 2)
    );
}

double Point::distance( const Point* p2) const{
    return sqrt(
        pow( (x - p2 -> x), 2) +
        pow( (y - p2 -> y), 2)
    );
}
```

Parameters by pointer to const

```
Point p, p2(1,2);
p.translate(Point(2, 3));
// ok
p.translate(&p2);
// ok
p.translate(&Point(2,3));
// warning: taking address of temporary
```

- Const references extend the life cycle of temporaries.

Overloading `operator«`

```
cout << "this is a string, numbers will follow: " << 42 << 3.14 <<↩
     endl;
```

- Defined (respectively for `int`, `double`, `string`) as:
- `ostream& std::ostream::operator« (int val);`
- `ostream& std::ostream::operator« (double val);`
- `ostream& std::operator« (ostream& os, const string& str);`

Printing Points

```
// in point.hpp

class Point{
  ...
  friend ostream& operator<< (ostream& out, const Point& cPoint);
  ...
};

// somewhere else

ostream& operator<< (ostream& out, const Point& cPoint){
    out << "(" << cPoint.x << ", " << cPoint.y << ")";
    return out;
}
```

Why the return?

```
cout << "hi" << "bye" << endl;

// this is equivalent to:

operator<<(
    operator<<(cout, "hi"),
    /* returns reference to cout */
    "bye"
);
```

Wrap-up

- C++ inherits from C all the pointer power/responsibility (and clumsiness).
- References help in terms of syntax.
- Memory leaks in C++ are a serious issue (and a productivity bottleneck).