



# CS 11 C++ track: lecture 3

---

- Today
  - `const` and `const`-correctness
  - operator overloading



# const and const-correctness (1)

---

- Simplest use of `const`:

```
// global variables:
```

```
const int max_height = 800; // e.g. pixels
```

```
const int max_width  = 600;
```

- `max_height` and `max_width` can't be changed later!
- Never use `#define` for this in C++ *e.g.*  
`#define max_height 800 // bad!`
- `const int` is checked at compile-time



## const and const-correctness (2)

---

- Protecting `const`-ness of function arguments

```
void print_int_safe(const int& i) {  
    cout << "i = " << i << endl;  
}
```

```
void print_int_dangerous(int& i) {  
    cout << "i = " << i << endl;  
    i++; // oops!  
}
```

- Second function can mutate `i` passed in, first one can't (compiler won't allow it)



## const and const-correctness (3)

---

- NOTE: `const int&` not same as `const int`

```
void print_int_safe(const int& i) {  
    cout << "i = " << i << endl;  
}
```

```
void print_int_safe2(const int i) {  
    cout << "i = " << i << endl;  
}
```

- First one gets *reference* to original `i`
- Second one get *copy* of `i` (`const` is superfluous)



## const and const-correctness (4)

---

- Rule of thumb for function arguments:
  - Pass by value for primitive types
    - `int`, `float`, `double`, etc.
  - Use `const <type>&` for other types
    - argument not copied or mutated (changed)
    - much cheaper for large objects
  - Use `<type>&` if you want to mutate the object



## const and const-correctness (5)

---

- Examples:

```
void print_int(int i) {  
    cout << "i = " << i << endl;  
}
```

```
void print_Thingy(const Thingy& t) {  
    // << has been overloaded for Thingys  
    cout << "thingy = " << t << endl;  
}
```

```
void bump_int(int& i) {  
    i++;  
}
```



# calling bump\_int

---

```
void bump_int(int& i) {  
    i++;  
}  
  
// later...  
int i = 10;  
bump_int(i);  
// now i == 11
```



# What if...?

---

```
void bump_int(const int& i) {  
    i++;  
}
```

- Won't compile – why not?



# Other uses of const

---

```
class Point {
    int x;
    Point(int x);
    Point(const Point& p); // can't change p
    int get_x() const; // can't change this Point
    int get_x2(); // can change this Point
    void foo(const Point& p) const;
        // can't change p or this Point
};
```



# Operator overloading -- basics

---

- What does *operator overloading* mean?
- What operators are *already* overloaded?
- Why is operator overloading useful?



# Operator overloading -- basics

---

```
Complex a, b, c; // Complex is a class
// ... initialize a and b ...
// want to say: c = (a + b) * (a - b);
c.set( (a.add(b)).multiply(a.subtract(b)) );
```

- ARRGH!
- Too gross to tolerate
- Operator overloading lets us express this "the nice way"
- Mainly useful for numeric types like **Complex**



# Interface to Complex class

---

```
// in Complex.hh:  
class Complex {  
    double re, im;  
    Complex(double re, double im);  
    // accessors:  
    double real() const;  
    double imag() const;  
    // overload addition operator:  
    Complex operator+(Complex other);  
};
```



# Implementation of Complex class

---

```
// in Complex.cc:  
// most of it is obvious  
Complex Complex::operator+(Complex other) {  
    double re1 = other.real();  
    double im1 = other.imag();  
    Complex result(re + re1, im + im1);  
    return result;  
}
```

- `c1 + c2` is sugar for `c1.operator+(c2)`



## Problem with `operator+` ?

---

- Copies `Complex` object (argument) unnecessarily
- If objects big, this is very wasteful
- How to fix?

```
Complex Complex::operator+(const Complex& other)
{
    // same as before
}
```

- Now `other` isn't copied



## Even better `operator+`

---

- `operator+` doesn't change current object...
- ... so declare it as `const`

```
class Complex {  
    // ...  
    Complex operator+(const Complex& other)  
        const;  
};
```

- `operator+` now can't change current object
- Rule: declare methods `const` when possible



# Overloading the assignment operator

---

- First attempt:

```
void operator=(const Complex& other);  
// ignore implementation for now  
// used like this:  
Complex c1(10, 20);  
Complex c2;  
c2 = c1; // same as c2.operator=(c1);
```

- OK but limited



# Problem with this implementation

---

- Assignment should support "chaining" e.g.

```
Complex c1(10, 20), c2, c3;
```

```
c3 = c2 = c1; // c2, c3 now == c1
```

```
// Same as:
```

```
// c3.operator=(c2.operator=(c1));
```

- Won't work with previous code

- Can't assign `void` to `c2`

- What do we return from `operator=` ?



## Second try

---

- Return a `Complex` object

```
Complex operator=(const Complex& other);
```

- What's the problem now?
  - Inefficient!
  - Makes a *copy* of `Complex` object before it returns!
- How to fix?



## Third try

---

- Return a `Complex&` (reference to `Complex`)  
`Complex& operator=(const Complex& other);`
- (Could return `const Complex&` too
  - but this limits what can be done with return value)
- This does *not* make a copy of return value
- This is the recommended strategy
  - use with `+=`, `-=`, etc. as well



# Implementation

---

`Complex&`

```
Complex::operator=(const Complex& other) {  
    if (this == &other) {  
        return *this; // saves effort  
    }  
    re = other.re;  
    im = other.im;  
    return *this; // for chaining  
}
```



# Notes

---

- With most objects, will need to allocate/deallocate memory as well
- Never, ever, **ever** return reference to local variables!
  - they don't exist after the function returns
  - great way to get core dumps



# Recall operator+

---

```
Complex Complex::operator+(const Complex&
    other)
{
    double re1 = other.real();
    double im1 = other.imag();
    // create local Complex object:
    Complex result(re + re1, im + im1);
    // return it!?
    return result;
}
```



# Huh?

---

- I said never to return reference to local variables
- But this *doesn't* return a reference:

```
Complex Complex::operator+(const Complex&
    other)
{
    // ...
    Complex result(re + re1, im + im1);
    return result;
}
```



## Bad alternative:

---

```
Complex& Complex::operator+(const Complex&
    other)
{
    // ...
    Complex result(re + re1, im + im1);
    // Return reference to local variable:
    return result; // BAD!
}
```

- Might want to do this to avoid copying
- If you do this, all hell will break loose



## Right way:

---

```
Complex Complex::operator+(const Complex&
    other)
{
    // ...
    Complex result(re + re1, im + im1);
    // Return copy of new Complex object:
    return result;
}
```

- Now local variable gets copied when returned
- Seems wasteful, but only way to do it right
- C++ is full of subtleties like this



# This lab / next week

---

- Get practice overloading operators for matrix class
- Next week:
  - memory management
  - the stack and the heap
  - inline functions
  - structs vs. classes