

C++ for Java Programmers – III

WL101.Ic

©Wilhansen Li 2010 <wil@nohakostudios.net>

Classes and Templates

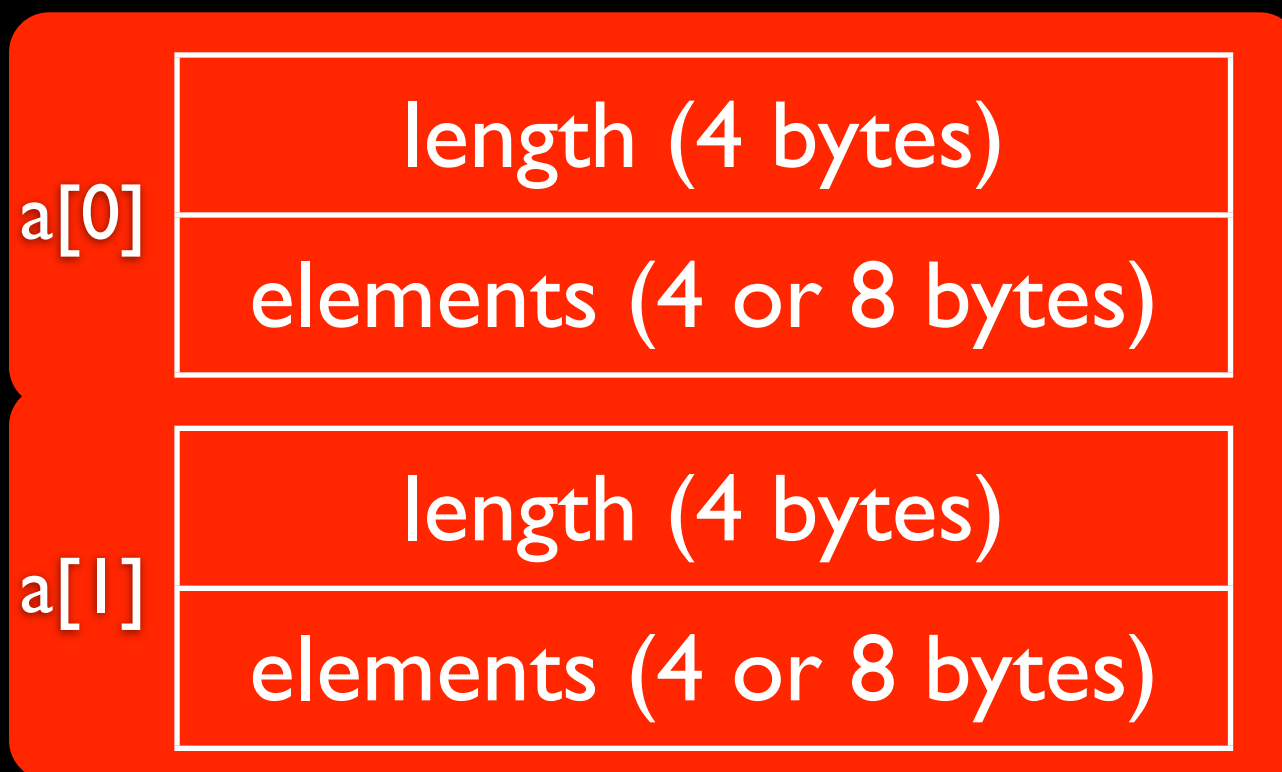
Outline

- POD Structs
- Constructor
- Copy Constructor
- Destructor
- RAI Technique
- Classes
- Access Modifiers
- Member Functions
- `this` pointer
- Const Member Functions
- Static members
- Inheritance
- `virtual` Functions
- Friends
- Operator Overloading
- Templates
 - Class templates
 - Function templates
 - Template Initialization

POD Structs

POD Structs

- POD (Plain Old Data) Structs group instances of variables together.
- Basically a Java class with only data.
- Memory layout is ***highly predictable***.



```
struct IntArray {  
    int length;  
    int *elements;  
};  
IntArray a[2];
```

Struct Declaration

```
struct StructName {  
    type fields;  
};
```

Note the semi-colon at the end.

Example:

```
struct Stuff {  
    int length;  
    int *elements;  
    char name[8];  
};
```

POD Struct

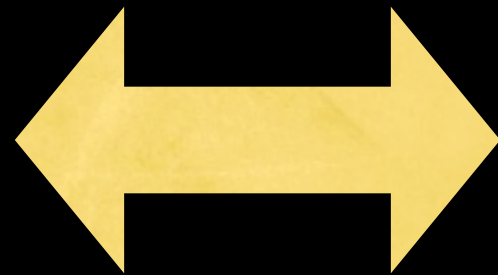
- A POD Struct (or structs in general) are types.
- Use and declare them as such.
- Array-type initializers can be used (the order follows the order of declaration in the struct)

```
struct IntArray {  
    int length;  
    int *elements;  
};  
IntArray a = {2, new int[2]};
```

Alternate instance declaration.

Declare variables upon structure definition.

```
struct IntArray {  
    int length;  
    int *elements;  
};  
IntArray a, b;
```



```
struct IntArray {  
    int length;  
    int *elements;  
} a, b;
```

Struct

- Structs should fully know the size of its members upon definition.
- There's no way to a struct to have a member of the same type.
- Pointers must be used since they more or less have fixed sizes.

Struct sizes

```
struct Node {  
    void *data;  
    Node next;  
    Node prev;  
};
```

```
sizeof(Node) == ?
```

```
struct Node {  
    void *data;  
    Node *next;  
    Node *prev;  
};
```

```
sizeof(Node) == sizeof(uintptr_t) * 3;
```

Access

- Use the `.` operator for values or references
- Use the `->` operator for pointers.

```
struct IntArray {  
    int length;  
    int *elements;  
};  
IntArray a, *b;
```

```
b = new IntArray();  
a.length = 1;  
b->length = 2;
```

Constructors

Constructors (ctor)

- Declared similar to Java
- A public constructor with no arguments is called the *default constructor*.

```
struct IntArray {  
    int length;  
    int *elements;  
    IntArray(int length) {  
        this->length = length;  
        this->elements = new int[length];  
    }  
    IntArray() {  
        this->length = 0;  
        this->elements = 0;  
    }  
};
```

```
IntArray a;  
IntArray *b;  
IntArray c(10);  
b = new IntArray(5);
```

Member Initialization List

- An alternative (and preferred) way to initialize members in constructors.
- The **only** way to initialize (non-static) const members and references.
- The initialization order is **not** according to the initialization list but according to declaration order.

```
struct IntArray {  
    int length, *elements;  
    IntArray(int arrLen) :  
        length(arrLen), elements(new int[arrLen]) {  
    }  
};
```

MIL Scope

- The syntax of the MIL is similar to how objects are initialized with parameters.
- The left-hand has class scope while the right-hand has parameter scope.
- You can do something like this:

```
struct IntArray {  
    int length, *elements;  
    IntArray(int length) :  
        length(length), elements(new int[length]) {  
    }  
};
```

Copy Constructor

Copy Constructor

- A special kind of constructor that's always called when a like object is copied upon construction.

```
IntArray b = a; //b copies a
```

- It is *only* called upon construction.

```
IntArray c;  
//does not call copy constructor  
c = a;
```


Copy Constructor

- They are also invoked when objects are copied to function parameters.

```
IntArray a(10);  
  
void f(IntArray p);  
f(a);
```

- As what was said before: *Passing objects as references eliminates copying.*

Copy Constructor

- A copy constructor is a construct that takes a ***const reference of the same type***.

```
struct IntArray {  
    int length, *elements;  
    IntArray(const IntArray &rhs) :  
        length(rhs.length), elements(new int[length]) {  
        for (int i = 0; i < length; ++i)  
            elements[i] = rhs.elements[i];  
    }  
};
```

Copy Constructor

- A class can be made non-copyable by making its copy constructor private.
- More details on access modifiers later.

```
struct NonCopyable {  
    int field;  
  
private:  
    NonCopyable(const NonCopyable &) {}  
};
```

Destructors

Destructors (dtors)

- C++ is a non-GC runtime.
- Destruction of objects is predictable (see WL101.1b).
- Destructors get called every time an object gets destroyed.

Destructors

- Declared similar to ctors (just prepend the name with a ~) but...
- There can only be one dtor per class.
- Dtors accept no parameters.

```
struct IntArray {  
    int length, *elements;  
    ~IntArray() {  
        delete [] elements;  
    }  
};
```

Checkpoint 3.1

Given the following struct:

```
struct Test {  
    Test() { cout << "Default ctor\n"; }  
    Test(const Test &) { cout << "Copy ctor\n"; }  
    ~Test() { cout << "Dtor\n"; }  
};
```

What does the following print?

```
void foo(Test a) {  
    cout << "In foo\n";  
}  
  
int main() {  
    Test *a;  
    cout << "Start here\n";  
    Test b;  
    cout << "Pre foo\n";  
    foo(b);  
    cout << "Post foo\n";  
    return 0;  
}
```

The Big 4

- Usually, a custom implementation one of the following will result in the custom implementation of the rest:
 - Default Constructor (weaker case)
 - Copy Constructor
 - Destructor
 - Assignment Operator (later)

non-POD Structs

- Usually, implementing any of the Big 4 will result in the struct being non-POD.
- Additional overhead may appear, layout and addressing may not be predictable.
- Refer to other C++ references for specific conditions (they are changed in C++0x).

RAII Technique

Resource Acquisition Is Initialization

RAII

- RAII is one of the more powerful C++ tools for properly managing resources.
- Principle:
 - initialize resources on initialization
 - release on destruction
- “Abuses” the scoping system.

RAII Example

```
struct IntArray {  
    const int length;  
    int * const elements;  
    IntArray(int length) :  
        length(length), elements(new int[length]) {}  
  
    ~IntArray() {  
        delete [] elements;  
    }  
};
```

```
try {  
    int len;  
    cin >> len;  
    IntArray arr(len);  
    //Do something to arr...  
} catch ( ... ) {  
    //Will arr leak at this point?  
}
```

RAII

In Java, when using IO resources (especially network IO) , `close()` has to be called explicitly.

```
public static void main(String [] args) {  
    FileWriter f = new FileWriter("a.out");  
    //write something  
    f.close(); //important!  
}
```

In C++, when using IO resources with RAII, `close()` is called automatically.

```
int main() {  
    ofstream f("a.out");  
    //write something  
    return 0; //f.close() implicitly called.  
}
```

Garbage Collection doesn't solve everything.

Classes

Declaration

- Classes are declared much like structs.
- Classes are exactly like struct except that by default, all their members are *private*.
- Everything you've learned about structs also apply here.

```
class IntArray {  
    const int length;  
    int * const elements;  
    IntArray(int length) :  
        length(length), elements(new int[length]) {}  
};
```

Access Modifiers

- Unlike Java, C++ access modifiers are *fields*.
- They apply to all members below it.
- Changed by another access modifier.

```
class A {  
    //private members  
    int a;  
public:  
    //public members  
    int b;  
    A() {}  
protected:  
    //protected members  
    A(int a) : a(a) {}  
};
```

```
class B {  
    int a;  
    void doFoo() {}  
public:  
    A() {}  
private:  
    A(int a) : a(a) {}  
    int b;  
};
```


Access Modifiers

- Like Java, ctors can be declared private or protected.
- Disable object copying by declaring copy ctors private.

```
class A {  
    int a;  
    A() {}  
    A(const A&) {}  
public:  
    A(int a) : a(a) {}  
};
```

Member Functions

- Member functions are declared similarly to global functions.
- They are still subject to ordering rules.

```
class IntArray {  
    int length;  
    int *elements;  
public:  
    IntArray(int length) : length(length), elements(new int  
[length]) {}  
    void resize(int newLength) {  
        //resize the array.  
    }  
};
```

Member Functions

- Separating definition and declaration of member functions is a bit trickier.
- More info on WL101.Id

```
class Class {  
public:  
    void bar();  
};
```

```
void Class::bar() {  
    //stuff  
}
```

Scope resolution operator.

} Defined *outside* of class definition.

Checkpoint 3.1

Implement an `IntArray` class:

- Private copy ctor (non-copyable).
- Dtor (releases resources).
- Public default ctor.
- Public `IntArray(const int length)` ctor.
- Public members:
 - `void resize(const int length)`
 - `void clear()`
 - `int& get(const int index)`
 - `void add(const int item)`
 - `int length()`
 - `int size()`

this Pointer

- In Java, `this` refers to the current object's.
- In C++, it's almost the same except that `this` is a pointer.
- Access `this` as a pointer (see earlier slide and WL101.1b).
- `this` can't be modified.

Example 3.1: Method Chaining

```
class Acc {  
    double a;  
public:  
    Acc(double a = 0) : a(a) {}  
    double get() { return a; }  
    Acc& plus(double b) { a += b; return *this; }  
    Acc& minus(double b) { a -= b; return *this; }  
    Acc& times(double b) { a *= b; return *this; }  
};
```

```
Acc().plus(1).minus(10).times(-2).plus(1).get();
```

Const Member Functions

Const Member Functions

- Declared by adding `const` after the function parameter list.

```
struct S {  
    void f() { cout << "normal\n"; }  
    void f() const { cout << "const\n"; }  
};
```


Const Member Functions

- `const` member functions are the only functions that can be called when the objects is `const`.

```
struct S {  
    void h() { cout << "h-normal\n" };  
    void f() { cout << "f-normal\n"; }  
    void f() const { cout << "const\n"; }  
};
```

```
S a;  
const S b;  
a.f();  
b.f();  
b.h(); //compiler error!
```

Const Member Functions

- `const` member functions cannot modify any non-`mutable` member variables.
- i.e. `this` is a `const` in a `const` m.f. regardless to whether the object is actually `const` or not.

```
struct S {  
    int a;  
    mutable int b;  
    void f() const { a = 10; } //error!  
    void h() const { b = 20; } //allowed  
};
```

Const Member Functions

- `const` member functions, if calling functions of itself, can only call other `const` member functions.

```
void g() { /*stuff*/ }

struct S {
    void notConst() { /*stuff*/ }
    void f() const { /*stuff*/ }
    void h() const {
        f(); //ok
        g(); //ok
        notConst(); //compiler error!
    }
};
```

Checkpoint 3.2

Get your `IntArray` code from Checkpoint 3.1

Which of the functions should be turned `const`?

Add (not replace) a `const` version of the following:

```
int& get(const int index)
```

Static Members

Static Members

- Behaves like those in Java.
- Static member functions are more straightforward to define.
- Static member fields are trickier to initialize.

Static Mem Fun

- When separating definition from declaration, the definition does not need the `static` keyword.

```
struct S {  
    static void foo();  
};  
  
void S::foo() {  
    //stuff  
}
```

Static Mem Fi

- Static member fields are initialized outside of the class.
- Initialization order not guaranteed across files (more on WL101.1d).

```
struct S {  
    static S *singleton;  
};  
  
S* S::singleton = 0;
```


Checkpoint 3.3

Implement the Singleton Design Pattern in C++.

Equivalent Java code:

```
class JavaSingleton {  
    private JavaSingleton() {}  
    private static JavaSingleton s = null;  
    public static JavaSingleton getInstance() {  
        if ( s == null )  
            s = new JavaSingleton();  
        return s;  
    }  
}
```

Inheritance

Inheritance

- The syntax is different.
- There is no distinction between interface and abstract classes (more info later).

```
class Base {  
};  
  
class Derived : public Base {  
};
```

Inheritance

- There are nine types of inheritance: `public`, `protected` and `private` and their `virtual` counterparts.
- Only `public` will be discussed, the rest are advanced and not really used often.

Inheritance

- By default all ctors are inherited.
- But the moment *any* ctor is defined in the class, the rest are “cancelled”.
- That is, if you want the same ctors to appear in the derived, they have to be repeated.
- Base dtors are automatically inherited and is automatically called when the derived dtor is returns.

Inheritance

- MIL are used for calling base ctors.
- If nothing was specified, the default base ctor will be called.

```
class Base {  
    int a;  
public:  
    Base(int a) : a(a) {}  
};
```

```
class Derived : public Base {  
public:  
    Derived() : Base(0) {}  
};
```

Let's try it!

```
class Base {
public:
    Base() {
        cout << "Base ctor\n";
    }
    ~Base() {
        cout << "Base dtor\n";
    }
};

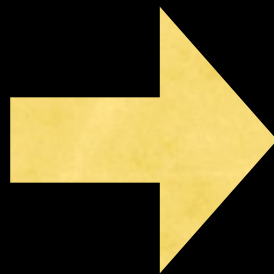
class Derived : public Base {
public:
    Derived() {
        cout << "Derived ctor\n";
    }
    ~Derived() {
        cout << "Derived dtor\n";
    }
};
```

```
int main() {
    {
        cout << "pre init\n";
        Derived d;
        cout << "post init\n";
    }
    cout << "post dtor\n";
}
```

Inheritance

- When functions with the same name but different signature are declared in the derived class, the base implementation is “hidden”.
- Expose them again with **using**.

```
struct Base {  
    void foo() {}  
};  
  
struct Derive  
: public Base {  
    void foo(int a) {}  
};
```



```
struct Base {  
    void foo() {}  
};  
  
struct Derive  
: public Base {  
    using Base::foo;  
    void foo(int a) {}  
};
```


Virtual Functions

Virtual Functions

- Unlike Java (and like C#), methods are not automatically overridden.
- This is because adding overriding functionality to a class will create overhead.
- If you want a function to be overridable, declare them `virtual` in the base class.
- If you tried to override a function without declaring it `virtual`, weird things will happen.

What “weird things”?

```
struct Base {  
    int n;  
    Base(int n) : n(n) {}  
    virtual void foo() {  
        cout << n << " Base::foo\n";  
    }  
};  
  
struct Derived : public Base {  
    Derived(int n) : Base(n) {}  
    void foo() {  
        cout << n << " Derived::foo\n";  
    }  
};
```

```
int main() {  
    Base b(0);  
    b.foo();  
    Derived d(1);  
    d.foo();  
    Base *p = &d;  
    d.foo();  
}
```

Then try removing `virtual` from `foo`
and see what happens.

Virtual Functions

- Functions are not automatically *virtual*... even the **dtor**.
- If a root class is meant to be inherited, *always always* declare a *virtual* dtor even if it doesn't do anything.
- If you don't, the derived dtor won't get invoked when deleting polymorphically.

Virtual Dtors

```
struct Base {  
    int n;  
    Base(int n) : n(n) {}  
    ~Base() {  
        cout << n << " Base dtor\n";  
    }  
};  
  
struct Derived : public Base {  
    Derived(int n) : Base(n) {}  
    ~Derived() {  
        cout << n << " Derived dtor\n";  
    }  
};
```

```
int main() {  
    {  
        Base b(0);  
    }  
    {  
        Derived d(1);  
    }  
    Base *p = new Derived(2);  
    delete p;  
}
```

Now declare the Base dtor as **virtual** and see....

Pure Virtual Functions

- A “virtual function” may be declared pure virtual by setting it to 0

```
virtual void foo() = 0;
```

- This is akin to declaring the class abstract and setting that function as abstract.
- There’s no distinction between an interface against an abstract class in C++

Friends

Friends

- Friends are entities that have access to a class' private members.
- Declared in the class.
- The entity should at least be declared beforehand.
- Some people say they should not be used because they break encapsulation.

Friend Examples

```
void a(); //fxn declaration

class B; //class declaration

class C {
private:
    void foo() {}
    friend a; //function friend
    friend class B; //class friend
};

void a() {
    C c;
    c.foo(); //ok
}
```

Friends

- Just because I'm your friend doesn't mean you're my friend. (friending is not symmetric)
- Your friend is not automatically my friend. (friending is not transitive)
- Just because I'm your friend doesn't mean that your child is my friend. (friending is not inherited)
- Just because I'm your friend doesn't mean our parents are friends. (converse of the previous point)



Friends

They can access your privates.

Operator Overloading

“See that plus? I can turn that to a times.”

Operator Overloading

- Remember the C++ type principle.
- Operator Overloading is a way to make your classes behave like primitives.
- Allows you to define what operators do.
- One of the targets must be a *class* type (you can't redefine what operators do between primitives!)

Basic Principles

- Looks like functions with names starting with **operator**
- Most can be declared in-class or off-class.
- The in-class is similar to the off-class with the first parameter to be implicitly **this**.

```
class FiniteField {  
    int a;  
public:  
    FiniteField(int a) : a(a%7) {}  
    FiniteField& operator=(int r) { a = r % 7; return *this; }  
    FiniteField& operator+=(int r) { a = (a+r) % 7; return *this; }  
};  
FiniteField operator+(int lhs, const FiniteField &rhs) {  
    return FiniteField(lhs + rhs.a);  
}
```

Preamble

- Nearly all operators are overloadable.
- Specified returns values are optional but highly recommended.
- Some legends:
 - **C** the class the operator is on.
 - **S, T, U** other classes or primitives.

Assignment Op.

- `=, +=, -=, *=, /=`
- Need to return a reference to self to allow chaining (e.g. `a = b = c`).
- RHS can be the same class or a different one.
- The `const` reference is optional.
- in-class:

```
C& operator=(const S &rhs)
```

- off-class:

```
C& operator=(C &lhs, const S &rhs)
```


Checkpoint 3.4a

Take out your `IntArray` code again.

Implement the marked field.

```
class IntArray {  
    /* private members */  
    IntArray(const IntArray &rhs) { /*copy ctor*/ }  
public:  
    IntArray() { /*default ctor*/ }  
    IntArray(const int len) { /*...*/ }  
    IntArray& operator=(const IntArray& lhs) { /* implement this */ }  
}
```

In the assignment, do a **deep copy**.
Also, **check for self-assignment**.

Checkpoint 3.4b

Implement a `Vector2` class (a 2-dimensional vector)

Use the template below and implement:

- Assignment.
- Vector subtraction (base it on the addition code).

```
struct Vector2 {  
    double x, y;  
    Vector2() {}  
    Vector2(double x, double y) : x(x), y(y) {}  
    Vector2(const Vector2& rhs) : x(rhs.x), y(rhs.y) {}  
  
    Vector2& operator+=(const Vector2 &rhs) {  
        x += rhs.x;  
        y += rhs.y;  
        return *this;  
    }  
    //also do = and -=  
};
```

Checkpoint 3.4c

Implement scalar multiplication and division in `Vector2`

Use the template below (implement `/=` as well)

```
struct Vector2 {  
    double x, y;  
    Vector2() {}  
    Vector2(double x, double y) : x(x), y(y) {}  
    Vector2(const Vector2& rhs) : x(rhs.x), y(rhs.y) {}  
  
    /* operators =, += and -= */  
    Vector2& operator*=(const double s) { /* implement this */ }  
};
```

Comparison Op.

- `==`, `!=`
- Returns `bool` (for comparison, duh)
- You can compare two different things.
- Usually declared as `const` functions.
- in-class:

```
bool operator==(const S &rhs) const
```

- off-class:

```
bool operator==(const C &lhs, const S &rhs)
```

Binary Arithmetic Op.

- `+`, `-`, `/`, `*`, `%`, `&&`, `||`
- Usually returns an instance of a new object (optionally a `const`).
- Also implemented as a `const` function.
- in-class:

```
C operator+(const S &rhs) const
```

- off-class:

```
C operator+(const C &lhs, const S &rhs)
```

Checkpoint 3.5a

Implement binary arithmetic operator versions of vector subtraction, scalar multiplication and scalar division.

Try to implement them off-class (to get a feel).

Here's something to get started:

```
Vector2 operator+(const Vector2 &lhs, const Vector2 &rhs) {  
    return Vector2(lhs.x + rhs.x, lhs.y + rhs.y);  
}
```

Checkpoint 3.5b

Given **a** and **b** which are 2d vectors...

Implement the dot product:

$$\text{*(a,b)} = \mathbf{a.x} * \mathbf{b.x} + \mathbf{a.y} * \mathbf{b.y}$$

Implement the cross product:

$$\text{%(a,b)} = \mathbf{a.x} * \mathbf{b.y} - \mathbf{a.y} * \mathbf{b.x}$$

Unary Op.

- **!**
- Can return anything (usually returns a **bool**)
- Also usually a **const** function.
- In-class:

```
bool operator!() const
```

- Off-class:

```
bool operator!(const C &rhs)
```


Index Op

- `[]`
- Makes your class behave like an array.
- This is how `std::vector` does it.
- `const` is optional (depends on your use).
- Accepts a single parameter and returns anything.
- In-class:

```
S operator[](T parameter)
```

- Off-class: no way.

Function Op

- `()`
- Makes your class behave like a function.
- Objects that behaves like a functions are called `functors`.
- Very important when dealing with `<algorithms>`
- In-class:

```
S operator() ( /*parameters*/ )
```
- Off-class: no way.

Post Incr./Decr. Op

- Postfix increment (**++**) and decrement(**--**)
- Don't make them return a reference to the current object (remember how they work).
- In-class:

```
C operator++(int dummy)
```

- Off-class:

```
C operator++(C& lhs, int dummy)
```

Pre Incr./Decr. Op

- Prefix increment (**++**) and decrement(**--**)
- In-class:

```
C& operator++()
```

- Off-class:

```
C& operator++(int dummy, C& lhs)
```

Pointer Resolution Op.

- `*`, `->`
- Both have to return pointers of non-void.
- Automatically gets dereferenced.
- Used to have classes that disguises as pointers (like managed pointers).
- In-class:

```
S* operator*()
```

- Off-class:

```
S* operator*(C& rhs)
```

Templates

Templates

- Equivalent to generics in Java.
- Usage of template type are similar to Java (how do you use `java.util.ArrayList`?)
- Trickier, but more powerful.
- It's how `std::vector` can be used to store any type.

```
template <typename T>
void fill_zero(std::vector<T> &v) {
    for (int i = 0; i < v.size(); ++i )
        v[i] = static_cast<T>(0);
}
```

Basics

- Before the start of the declaration, add `template <`
- Follow it with a list of parameters which could be `typename` or `int`.
- end with `>`

```
template <typename T, int N>  
T foo() { /*...*/ }
```


Let's try it!

Take out your `IntArray` code again...

Let's make it type-agnostic!

1. Rename `IntArray` to just `Array`
2. Add `template<typename T>` before `class` (`T` is an identifier, so you can replace it with any other name).
3. Replace declarations of `int` with `T` (which ones? not all of them should be replaced).

Default Template Params.

- Like default function parameters, it's possible to specify default template parameters.
- Same as how default function parameters are done.
- The `<>` are still needed even if all parameters are defaulted.

```
template <typename T = int>
struct S {
    T n;
};

S<> a;
```

Function Templates

- Function templates are similarly declared to class templates.
- Function templates can resolve automatically.

```
template <typename T>  
T cross(T x0, T y0, T x1, T y1) {  
    return x0 * y1 - x1 * y0;  
}
```

```
cross(1, 2, 3, 4);  
cross(1.0f, 2.0f, 3.0f, 4.0f);  
cross(1.0, 2.0, 3.0, 4.0);
```

Template Instantiation

- If classes are “skeletons” of objects...
- Templates are “skeletons” of classes.
- A declared class template isn’t really “initialized” unless it’s been “instantiated” (i.e. upon first declaration of a variable).
- Over-instantiation of templates *may* cause code bloat.

A Template Trick

- It's possible to “reflect” template types via `typedefs`.
- Useful at times.

```
template <typename T>
class Array {
public:
    typedef T Type;
};

typedef Array<int> IdList;

IdList::Type studentId;
```