

Deriving a new class from `IntArray`

Class declaration:

```
#include "IntArray.h"
class IntArrayRC : public IntArray
{ public:
    IntArrayRC( int );
    int& operator[]( int );
private:
    void rangeCheck( int );
};
```

Class implementation:

```
IntArrayRC::IntArrayRC( int s ) :IntArray( s ) {}
int& IntArrayRC::operator[]( int index )
{ rangeCheck( index );
  return IntArray::operator[]( index );
}
void IntArrayRC::rangeCheck( int index )
{ if ( index < 0 || index >= getSize() )
  { cerr << "Bad index for IntArrayRC: "
    << "\n\tsize: " << size
    << "\n\tindex: " << index << "\n";
    exit( 1 );
  }
}
```

19 April 2005

A demonstration run of `IntArrayRC`:

```
earth% cat >s.c
```

```
#include <stream.h>
#include "IntArray.h"
// ... definitions of IntArrayRC ...
const size = 12;
main()
{
    IntArrayRC A( size );
    for( int i = 1; i <= size; ++i )
        A[ i ] = i;
}
```

```
earth% g++ s.c IntArray.a
```

```
earth% a.out
```

```
Bad index for IntArrayRC:
        size: 12        index: 12
```

Considering

- classes as **sets**
- derived classes as **subsets**,
 $\text{IntArrayRC} \subset \text{IntArray}$
 $x \in \text{IntArrayRC} \Rightarrow x \in \text{IntArray}$

i.e., an instance of `IntArrayRC` can be used wherever an instance of `IntArray` is expected.

19 April 2005

That's because an `IntArrayRC` **is a kind of** `IntArray`.

For example, if we define

```
#include "IntArray.h"
void swap( IntArray &a, int i, int j )
{ int temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
```

then the following is perfectly valid:

```
IntArray A;
IntArrayRC R;
swap( A, 3, 7 );
swap( R, 3, 7 ); // because R is an IntArray
```

The other way around, however, is not valid:

```
#include "IntArray.h"
void swapRC( IntArrayRC &r, int i, int j )
{ int temp = r[i];
  r[i] = r[j];
  r[j] = temp;
}
IntArray A;
IntArrayRC R;
swapRC( A, 3, 7 ); // type error here
swapRC( R, 3, 7 ); // OK
```

Even in the valid case, there's a problem: `swap()` always uses `IntArray::operator[]`, so indexes are *not range-checked*.

19 April 2005

We can fix this by changing `IntArray`'s declaration to make `operator[]` a **virtual function**:

```
class IntArray
{
public:
    ...
    virtual int& operator[]( int );
    ...
}
```

After this change, `swap()` calls the version of `operator[]` that belongs to its argument.

Unlike the overloading we've seen previously, this kind of polymorphism is **resolved at run-time**.

It's called **dynamic polymorphism**.

The cost of dynamic polymorphism is the overhead of virtual functions:

- function code is accessed through a table (small cost: two extra memory references)
- in-line expansion is no longer possible (function-call-and-return overhead is unavoidable, and some optimizations cannot be obtained).

Note: In Java, all member functions are automatically "virtual" (unless declared "final").

19 April 2005

How are virtual functions implemented?

Example:

```
class A
{
public:
    int a;
    virtual void f( A& );
    virtual void g( int );
    virtual void h( double );
};
```

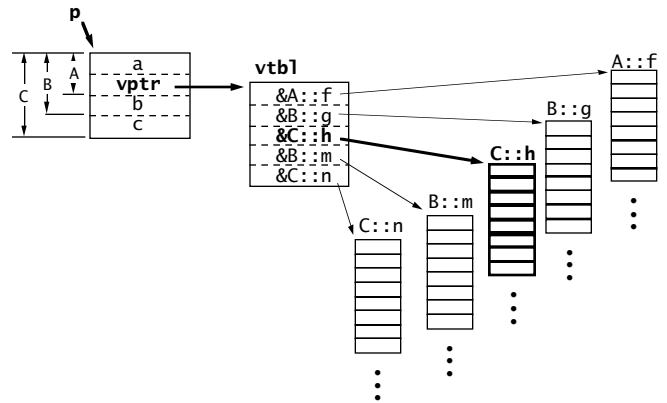
```
class B : public A
{
public:
    int b;
    void g( int ); // overrides A::g
    virtual void m( B* );
};
```

```
class C : public B
{
public:
    int c;
    void h( double ); // overrides A::h
    virtual void n( C* );
};
```

Virtual-function calls: An example

```
void F1( A& p ){ p.h(9.5); }
...
C x; ... F1(x) ...
```

F1(x) is handled by indirection:



The call `p.h(9.5)` becomes something like `((p->vptr)[2])(p, 9.5)`

The class-`C` `vtbl` is *shared* by all instances of `C`.

Virtual functions work *only* for *reference* and *pointer* arguments.

By-*value* arguments are always monomorphic: Whatever the argument's actual type, it is treated like a value of the parameter's type (which it is).

Pointer arithmetic is not compatible with dynamic polymorphism.

When

- a base-class pointer `A *p` actually points to an array of derived-class objects `C` and
- `sizeof(C) > sizeof(A)`

arithmetic on `p` is performed in units of `sizeof(A)`, i.e.,

$$p+1 = p + \text{sizeof}(A)$$

A virtual-function exercise:

```
class A
{ public:
    virtual char f() {return 'A';}
    char g() {return 'A';}
    virtual char h() {return 'A';}
    char testF() {return f();}
    char testG() {return g();}
    char testH() {return h();}
};

class D : public A
{ public:
    char f() {return 'D';}
    char g() {return 'D';}
    // D inherits testF, testG, testH
};

void main()
{ D d;
  cout << d.testF() << d.testG() << d.testH();
}
```

Result: _____

What would be printed if `virtual char f(){...}` were omitted from `A`?