

Review of Types in C++

- Two main kinds of types in C++
 - native and user-defined
- User-defined types
 - Classes, structs, unions, enumerations declared by code
 - includes those provided by C++ libraries, like `bad_alloc`
- Native types
 - Types “built in” to the C++ language itself: `int`, `long`, `float`, ...
- Type definitions
 - A typedef creates new type names for other types
 - We’ll use this idea a lot when we talk about templates

Overview of C++ Polymorphism

```
class Base {
public:
    Base (int i);
    Base (const Base & b);
    virtual ~Base ();
    void operator= (const Base & b);
    void i (int i); int i () const;
private:
    int i_;
};

class Derived : public Base {
public:
    Derived (int i, int j, int k);
    Derived (const Derived & d);
    virtual ~Derived ();
    void operator= (const Derived & d);
    void j (int j); int j () const;
    void k (int k); int k () const;
private:
    int j_; int k_;
};

int main (int, char *[]) {
    Base b (1);
    Derived d(2, 3, 4);
}
```

Poly (= many) + morphism (= forms)

- Inheritance creates sub-types
- Only applies to user-defined classes (and structs)
- A publicly derived class is-a subtype of its base class
 - Object **b** has only one type
 - Instance of **Base** only
 - Object **d** has two types
 - Instance of both **Base**, **Derived**

Aliasing and Substitution with Inheritance

```
class Base {
public:
    Base (int i);
    Base (const Base & b);
    virtual ~Base ();
    void operator= (const Base & b);
    void i (int i); int i () const;
private:
    int i_;
};

class Derived : public Base {
public:
    Derived (int i, int j, int k);
    Derived (const Derived & d);
    virtual ~Derived ();
    void operator= (const Derived & d);
    void j (int j); int j () const;
    void k (int k); int k () const;
private:
    int j_; int k_;
};

int main (int, char *[]) {
    Base b (1);
    Derived d(2, 3, 4);
    Base * bptr = d;
    Derived * dptr = d;
}
```

- Aliasing with inheritance
 - Pointer **dptr** can only point to **d**
 - Can't do **dptr = b; dptr->j ();**
 - Pointer **bptr** can point to **b** or **d**
 - But can only call Base functions
 - Can't say **bptr->j ();**
- Liskov Substitution Principle
 - if S is a subtype of T, then can use an S where a T is needed
 - But not the other way around
 - **bptr = &d;** ok, but not **dptr = &b;**

Static vs. Dynamic Types

```
class Base {
public:
    Base (int i);
    Base (const Base & b);
    virtual ~Base ();
    void operator= (const Base & b);
    void i (int i); int i () const;
private:
    int i_;
};
class Derived : public Base {
public:
    Derived (int i, int j, int k);
    Derived (const Derived & d);
    virtual ~Derived ();
    void operator= (const Derived & d);
    void j (int j); int j () const;
    void k (int k); int k () const;
private:
    int j_; int k_;
};
int main (int, char *[]) {
    Base b (1);
    Derived d(2, 3, 4);
    Base * bptr = d;
    Derived * dptr = d;
}
```

- Pointers have static types
 - References have them too...
 - Type with which it was declared
 - Most *general* type it can alias
 - **Base** is more general than **Derived**
 - Static type of **bptr** is **Base***
 - Static type of **dptr** is **Derived***
- What's pointed to / referenced has a dynamic type
 - Type of object currently aliased
 - Dynamic type of ***bptr** can be either **Base** or **Derived**
 - Dynamic type of ***dptr** is **Derived**
 - Expands if classes inherit from Derived
 - May change if pointer reassigned
 - May change if reference parameter is passed a different object

Declaring Virtual Member Functions

```
class A {
public:
    virtual foo () {cout<<"A";}
};

class B : public A {
public:
    // overrides A::foo definition
    virtual foo() {cout<<"B";}
};

int main (int, char *[]) {

    A *aptr = new B;

    // prints "B" : would print
    // "A" instead if non-virtual
    aptr->foo ();

    delete aptr;

    return 0;
};
```

- Enables polymorphism using pointers and references in C++
- Class declares a member function to be virtual
- Derived class overrides the base class's definition of the function
 - Good style to **say** it's virtual at re-declaration, but it's virtual anyway
 - **Overriding** only happens when signatures are the same
 - Otherwise it just **overloads** the function or operator name with a different signature
- Ensures function definition is resolved by dynamic type
 - E.g., that destructors farther down the hierarchy get called
- Non-virtual functions are resolved by static type

Calling Virtual Member Functions

```
class A {  
public:  
    void x() {cout<<"A:x";};  
    virtual void y() {cout<<"A:y";};  
};
```

```
class B : public A {  
public:  
    void x() {cout<<"B:x";};  
    virtual void y() {cout<<"B:y";};  
};
```

```
int main () {  
    B b;  
    A *ap = &b; B *bp = &b;  
    b.x (); // prints "B:x"  
    b.y (); // prints "B:y"  
    bp->x (); // prints "B:x"  
    bp->y (); // prints "B:y"  
    ap->x (); // prints "A:x"  
    ap->y (); // prints "B:y"  
    return 0;  
};
```

- Only matter with pointer or reference
 - Calls on object itself resolved statically
 - E.g., `b.y()`;
- Look first at pointer/reference type
 - If non-virtual there, resolve statically
 - E.g., `ap->x()`;
 - If virtual there, resolve dynamically
 - E.g., `ap->y()`;
- Note that virtual keyword need not be repeated in derived classes
 - But it's good style to do so
- Caller can force static resolution of a virtual function via scope operator
 - E.g., `ap->A::y()`; prints "A:y"
 - E.g., `ap->B::y()`; prints "B:y"
 - But should use RTTI to be *type safe*

Virtual Destructors and the Class Slicing Problem

```
class A {
public:
    A () {cout<<" A";}
    virtual ~A () {cout<<" ~A";}
};

class B : public A {
public:
    B () :A() {cout<<" B";}
    virtual ~B() {cout<<" ~B";}
};

int main (int, char *[]) {

    // prints "A B"
    A *ap = new B;

    // prints "~B ~A" : would only
    // print "~A" if non-virtual
    delete ap;

    return 0;
};
```

- Derived class destructor called before base class destructor
- But, need to watch out for static versus dynamic type differences
- Making destructor virtual
 - ensures chain of destructor calls starts at dynamic type and moves up
 - Otherwise would start at static type (would “slice off” calls to destructors of more derived classes)
- “Class slicing problem” can also occur with pass/catch by value
 - Only static type gets copied
 - Another reason to always catch/pass by reference (using const if needed)

Declaring Pure Virtual Functions

```
class A {  
public:  
    virtual void x() = 0;  
    virtual void y() = 0;  
};
```

```
class B : public A {  
public:  
    virtual void x();  
    virtual void y() = 0;  
};
```

```
class C : public B {  
public:  
    virtual void y();  
};
```

```
int main () {  
    A * ap = new C;  
    ap->x ();  
    ap->y ();  
    delete ap;  
    return 0;  
};
```

- A is an Abstract Base Class
 - Similar to an interface in Java
 - Declares pure virtual functions (=0)
- Derived classes override those pure virtual member functions
 - B overrides `x()`, C overrides `y()`
 - Good style (but optional) to re-declare pure virtual functions that are not yet overridden
- Can't instantiate class with declared or inherited pure virtual functions
 - A and B are **abstract**, but C is **concrete**
 - Can create instances of C, but not B or A
- Can still have a pointer to an abstract class type
 - Very useful for polymorphism

Tips on Using Polymorphism

- Push common code/variables up into base classes
- Use public inheritance for polymorphism
 - Use private/protected inheritance for encapsulation
- Polymorphism depends on dynamic typing
 - Use a base-class pointer or reference if you want polymorphism
 - Use virtual member functions for dynamic overriding
- Use abstract base classes to declare interfaces
- Even though you don't have to, re-label each virtual (and pure virtual) function in derived classes

Type Casting in C++

- Casting up the inheritance type hierarchy, size conversion
 - Note that `static_cast` does not examine the dynamic type
 - Widens the static type to be more general, farther from dynamic type
 - Can also be used for situation-specific size conversions

```
B* p = static_cast<B*>(d); // d is D *
```

```
char c = static_cast<char>(i); // -128 <= i <= 127
```

- Casting down the inheritance type hierarchy
 - Narrows the static type to be less general, closer to dynamic type

```
D* p = dynamic_cast<D*>(b); // b is B *
```

- Casting away const protections

- Tells compiler: trust me, I know it's ok to modify this variable

```
T* p = const_cast<T*>(q); // q is const T *
```

- Reinterpretation of what is being pointed to

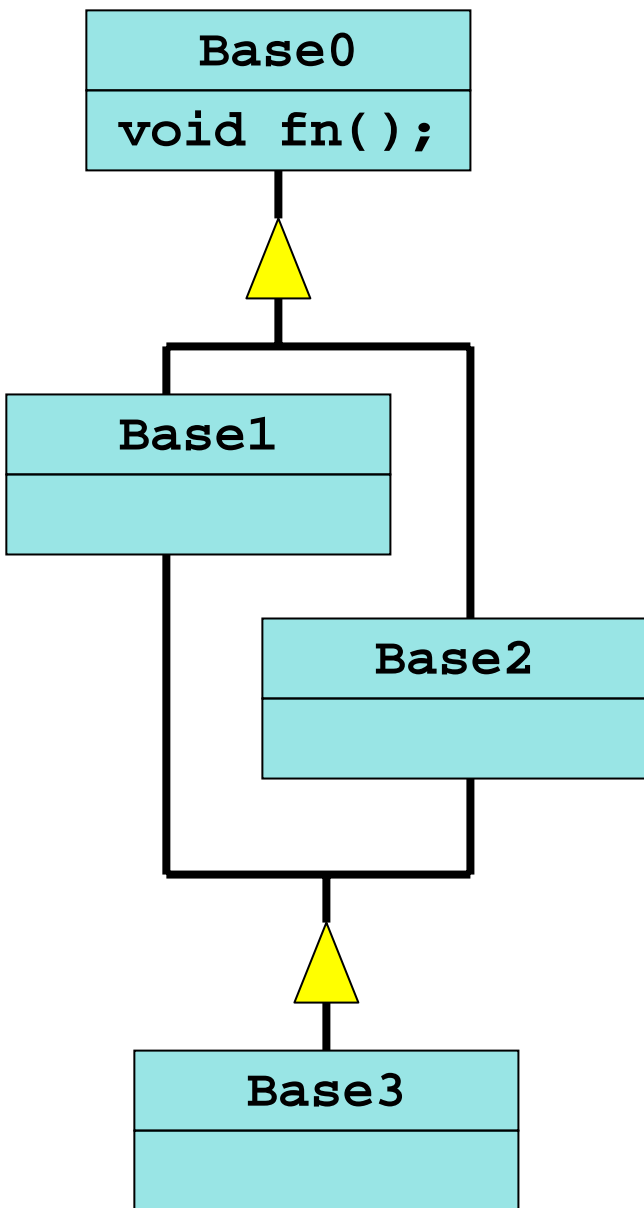
- Tells compiler: trust me, I **really really** know what I'm doing

```
T* p = reinterpret_cast<T*>(v); // v is void*
```

Run-Time Type Information (RTTI)

- Using `dynamic_cast<>` operator template
 - Examples
 - `dynamic_cast<T*>(p)`
 - `dynamic_cast<T&>(r)`
 - If p points to (sub-)type T then address is returned, else 0
 - If r references (sub-)type T then a reference to T is returned
 - otherwise an exception is thrown
- Using `typeid()` operator and `type_info` class
 - Only use when appropriate (see Prata 5th ed. pp. 847-848)
 - Need to include `<typeinfo>` header file to use them
 - returns type information (including a name string)
 - if polymorphic then the dynamic type is returned
 - otherwise the static type is returned.

Basic Ideas of Multiple inheritance



- Override resolution is not applied across class scope boundaries
- So if two different base classes define a function with the same signature, you have an ambiguity (compiler cannot figure it out)
 - Can be explicit, `Base1::fn()` or `Base2::fn()`
 - Can use a using-clause:
`using Base1::fn();` Or `using Base2::fn();`
- What about replicating the base classes?
 - Eliminates ambiguities related to which base is being referenced (but may duplicate code)
 - If you do this you should write an overriding function in the derived class
- Making `Base0` a virtual base class
 - `Base1` and `Base2` say `: virtual public Base0`
 - Only one copy of base object is constructed
 - Initialization of virtual base class is responsibility of the most derived object
 - Only time you'd initialize above immediate parent
`Base3::Base3 (int i)`
`: Base0 (i), Base2(i), Base3(i) {}`

For Next Time

- Topic: C++ memory models and memory management idioms
 - auto_ptr
 - singleton
 - guard
 - reference counting
 - copy-on-write
- Assigned Readings
 - pages 393-420
 - pages 873-877