Miscellaneous Topics

Object-Oriented Programming with C++

Named casts

- The *C-style cast* is:
  - dangerous because it can do logically different conversion.
  - not search-friendly
- If you must cast things, use a *named cast*:
  - `static_cast`, less likely to make mistakes
  - `dynamic_cast`
  - `reinterpret_cast`
  - `const_cast`

Named casts

```cpp
double d = 7.1;
int a;

a = d;                         // implicit

a = (int) d;                   // explicit

a = static_cast<int>(d); // exact meaning
```

Named casts

```
int a = 7;
double* p;

p = (double*) &a; // ok (but a is not a double)

p = static_cast<double*>(&a); // error

p = reinterpret_cast<double*>(&a); // ok: I really
                                   // mean it
```

Named casts

```
const int c = 7;
int* q;

q = &c; // error

q = (int*)&c; // ok (but is *q=2 really allowed?)

q = static_cast<int*>(&c); // error

q = const_cast<int*>(&c); // I really mean it
```

Named casts

```
struct A {
  virtual void f() {}
};
struct B : public A {};
struct C : public A {};

int main()
{
  A *pa = new B;
  C *pc = static_cast<C*>(pa);  // OK: but *pa is B!
}
```

Named casts

```cpp
struct A {
  virtual void f() {}
};
struct B : public A {};
struct C : public A {};

int main()
{
  A *pa = new B;
  C *pc = static_cast<C*>(pa);  // OK: but *pa is B!
  C *pc = dynamic_cast<C*>(pa); // return nullptr
}
```

Named casts

```
struct A {
  // virtual void f() {}
};
struct B : public A {};
struct C : public A {};

int main()
{
  A *pa = new B;
  C *pc = static_cast<C*>(pa);  // OK: but *pa is B!
  C *pc = dynamic_cast<C*>(pa); // Error!
}
```

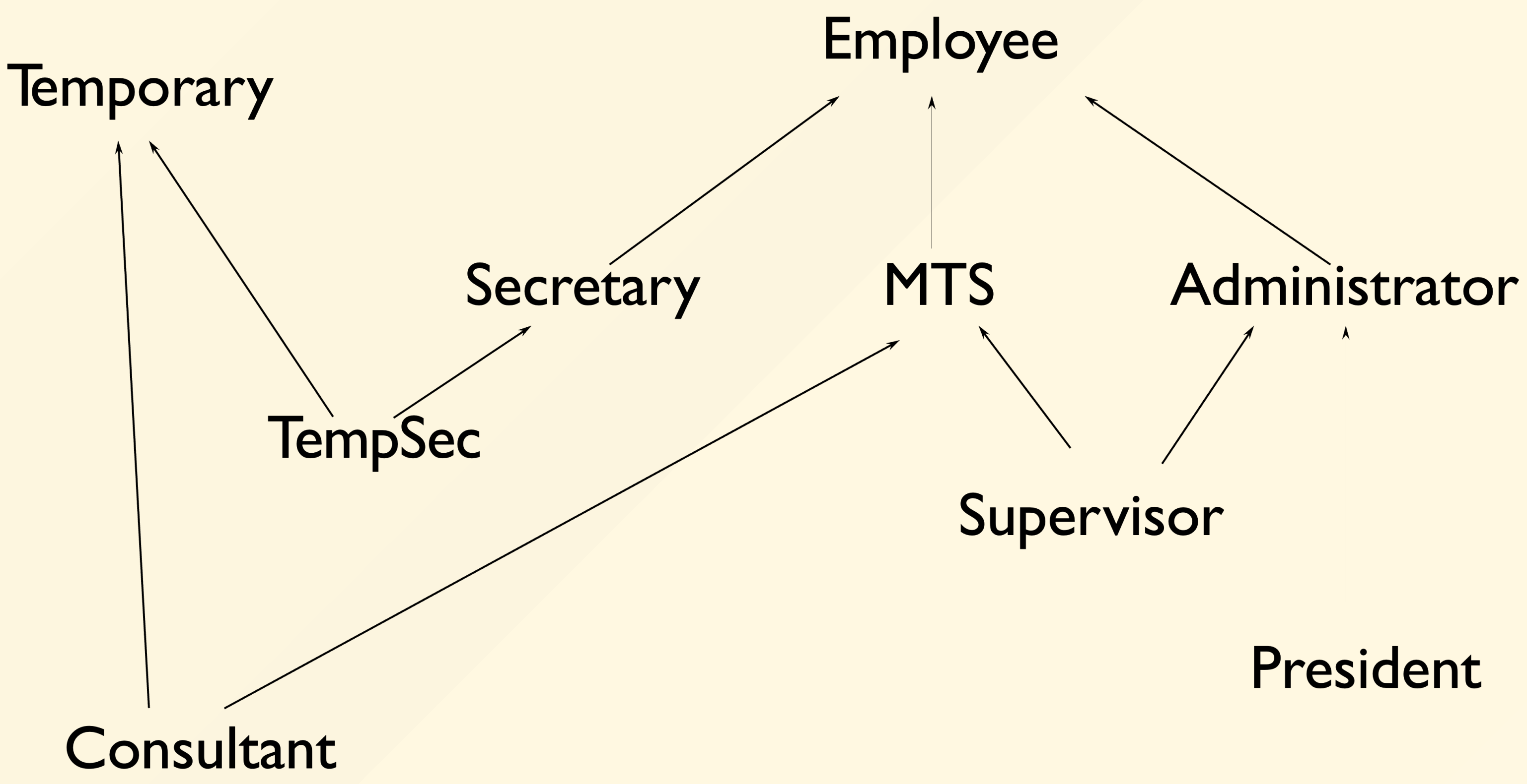
Named casts

```
struct A {
  // virtual void f() {}
};
struct B : public A {};
struct C : public A {};

int main()
{
  A *pa = new B;
  C *pc = static_cast<C*>(pa);  // OK: but *pa is B!
}
```

Named casts

```
struct A {
  // virtual void f() {}
};
struct B : public A {};
struct C : public A {};
struct D {};

int main()
{
  A *pa = new B;
  D *pd = static_cast<D*>(pa);  // Error!
  D *pd = reinterpret_cast<D*>(pa);  // Ok: but *pa is B!
}
```

Multiple inheritance



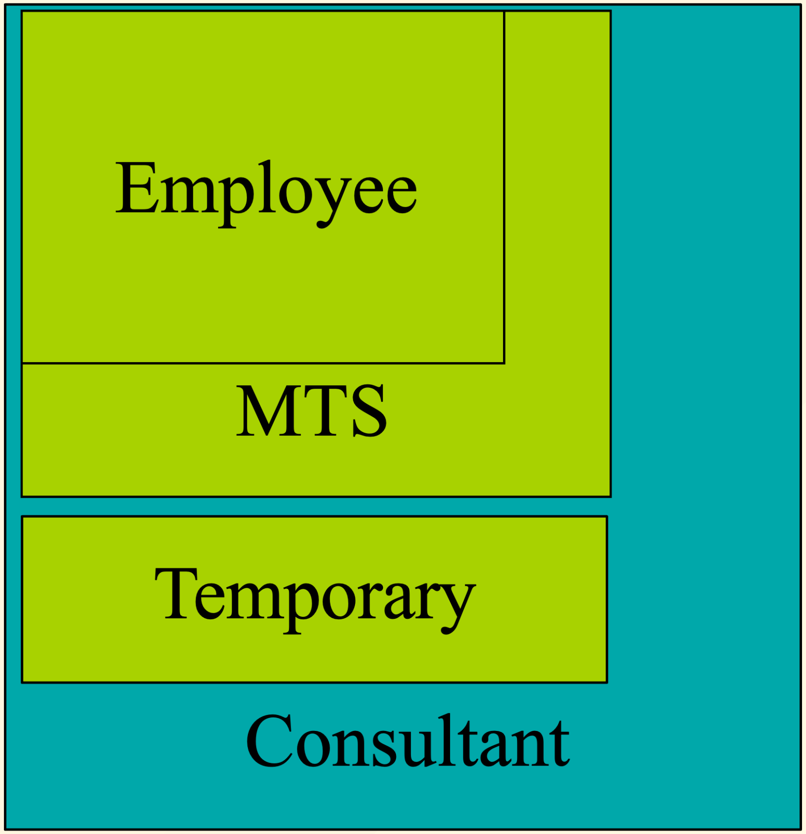
Mix and match

```
class Employee {
protected:
  String name;
  EmpID id;
};

class MTS : public Employee {
protected:
  Degrees degree_info;
};

class Temporary {
protected:
  Company employer;
};
```

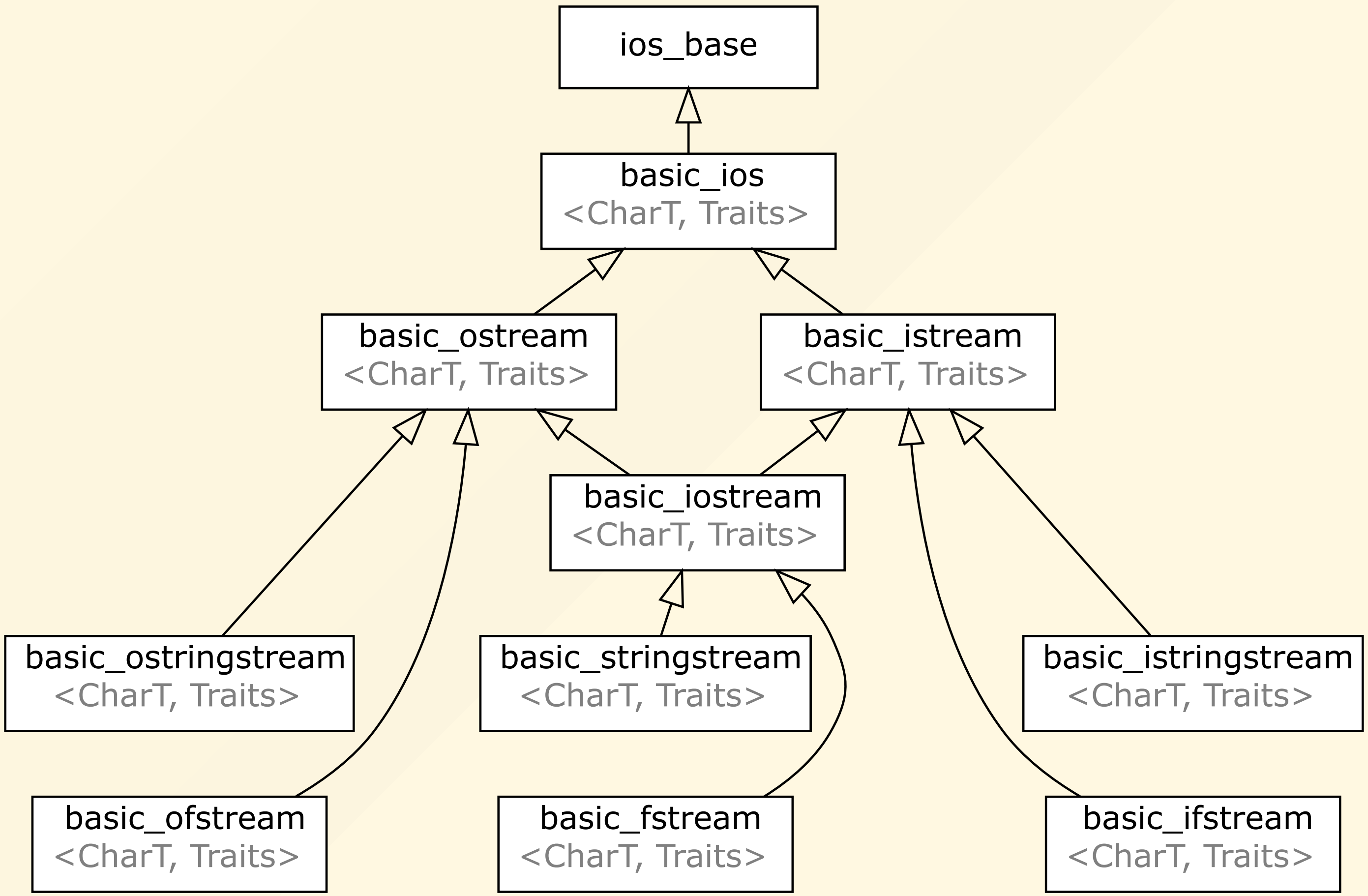
```
class Consultant:
 public MTS,
 public Temporary {
  /* ... */
};
```

- Consultant picks up the attributes of both `MTS` and `Temporary`
  - name, id
  - degree_info
  - employer

MI complicates data layouts

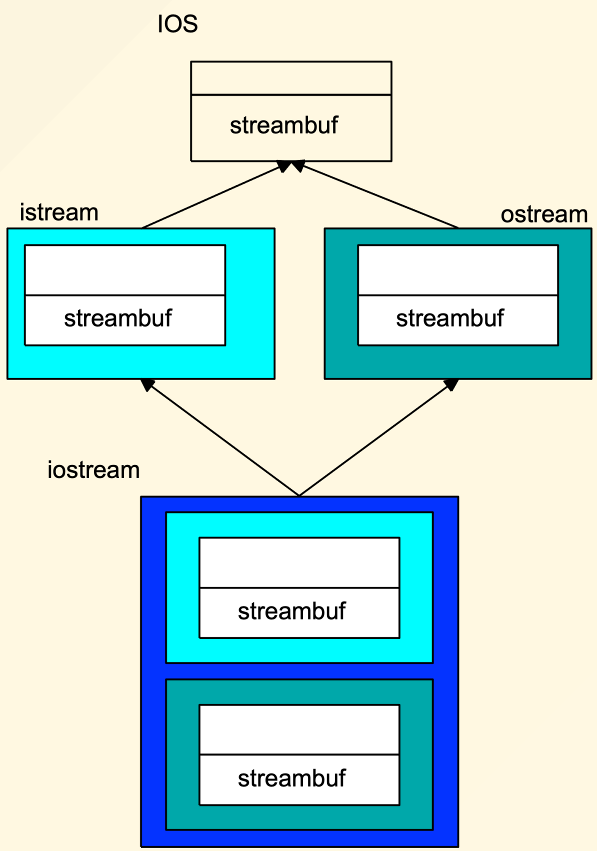


`iostream` package



Vanilla MI

- Members are *duplicated*
- Derived class has access to full copies of each base class
- This *can* be useful!
  - Multiple links for lists
  - Multiple streambufs for input and output



More on MI ...

```
struct B1 { int m_i; };
struct D1 : public B1 {};
struct D2 : public B1 {};
struct M : public D1, public D2 {};

int main() {
  M m; // OK
  B1* p = &m; // ERROR: which B1???
  B1* p1 = static_cast<D1*>(&m); // OK
  B1* p2 = static_cast<D2*>(&m); // OK
}
```

`B1` is a *replicated* sub-object of `M`.

Replicated bases

- Normally replicated bases aren't a problem (usage of `B1` by `D1` and `D2` is an implementation detail).
- Replication becomes a problem if replicated data bring in confusing logic:

```
M m;
m.m_i++; // ERROR: D1::B1.m_i or D2::B1.m_i?
```

Safe uses

- Protocol classes

Protocol / Interface classes

- Abstract base class with
  - All non-static member functions are *pure* virtual except destructor
  - Virtual destructor with empty body
  - No non-static member variables, inherited or otherwise
    - May contain static members

Example interface

- Unix character device

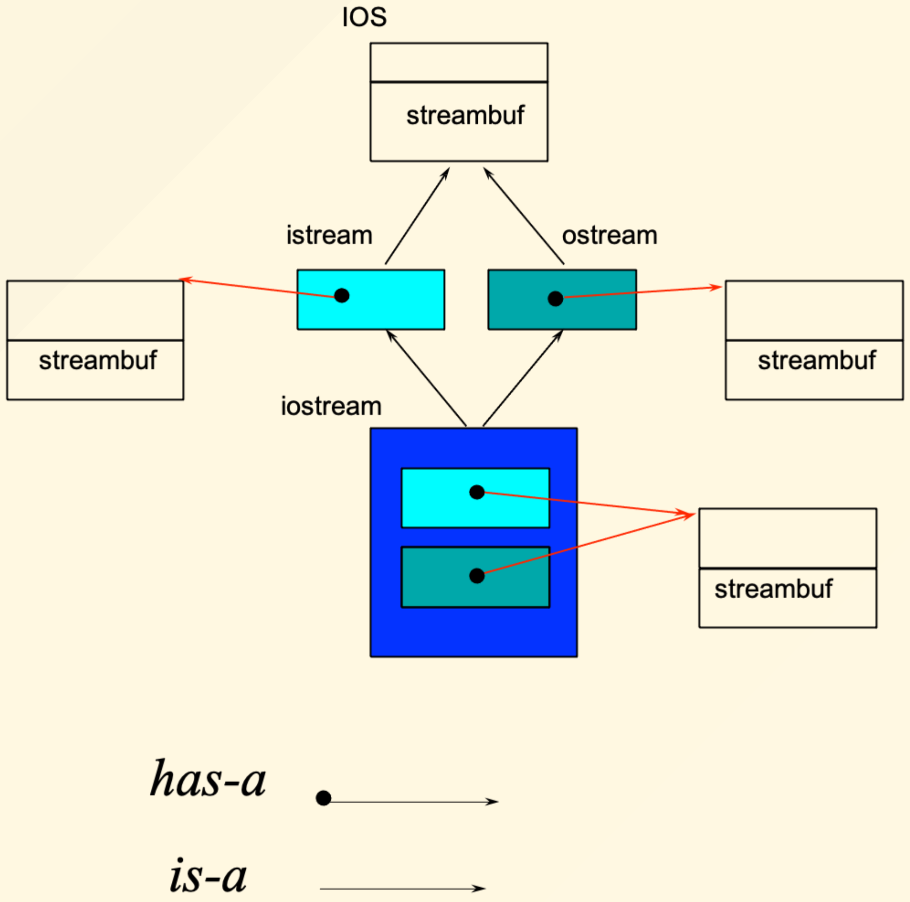
```
class CDevice {
public:
    virtual ~CDevice() = default;

    virtual int read(...) = 0;
    virtual int write(...) = 0;
    virtual int open(...) = 0;
    virtual int close(...) = 0;
    virtual int ioctl(...) = 0;
};
```

What about sharing?

- How do you avoid having two `streambufs`?
- Base classes can be *virtual*
  - To C++ people, “virtual” means “indirect”
- Virtual member functions have dynamic binding
  - They use pointer indirection
- Virtual base classes are represented indirectly
  - They use pointer indirection

Using virtual base classes

- Virtual base classes are *shared*
- Derived classes have *a single copy* of the virtual base
- Full control over sharing
  - Up to you to choose
- Cost is in complications



Virtual bases

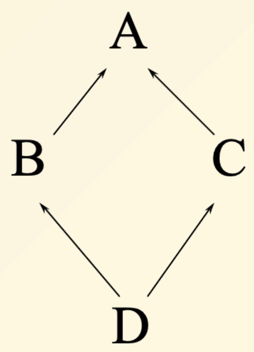
```
struct B1 { int m_i; };
struct D1 : virtual public B1 {};
struct D2 : virtual public B1 {};
struct M : public D1, public D2 {};
int main() {
   M m;  // OK
   m.m_i++; // OK, there is only one B1 in m
   B1* p = new M; // OK
}
```

Virtual bases

- Use of virtual base imposes some runtime and space overhead.
- If replication isn't a problem then you don't need to make bases virtual.
    - Abstract base classes (that hold no data except for a vptr) can be replicated with no problem – virtual base can be eliminated.

Complications of MI

- Name conflicts
  - Dominance rule
- Order of construction
  - Who constructs virtual base?
- Virtual bases not declared when you need them



- Code in virtual bases called more than once
- Compilers are still iffy
- Moral:
  - Use sparingly
  - Avoid diamond patterns
    - expensive
    - hard

TIPS for MI

- In general, SAY NO

Avoiding name clashes

- Including duplicate names at global scope is a problem

```
// old1.h
void f();
void g();

// old2.h
void f();
void g();
```

Avoiding name clashes ...

- Wrap declarations in namespaces

```
// old1.h
namespace old1 {
  void f();
  void g();
}

// old2.h
namespace old2 {
  void f();
  void g();
}
```

Namespace

```
namespace Math {
  double abs(double);
  double sqrt(double);
  int trunc(double);
  ...
} // Note: No terminating end colon!
```

- Expresses a logical grouping of classes, functions, variables, etc.
- A namespace is a scope just like a class
- Preferred when name encapsulation is needed

Defining namespaces

- Place namespaces in include files:

```
// Mylib.h
namespace MyLib {
  void foo();
  class Cat {
  public:
    void Meow();
  };
}
```

Defining namespace functions

- Use normal scoping to implement functions in namespaces.

```cpp
// MyLib.cpp
#include "MyLib.h"

void MyLib::foo() {
  cout << "foo\n";
}
void MyLib::Cat::Meow() {
  cout << "meow\n";
}
```

Using names from a namespace

- Use scope resolution to qualify names from a namespace.
- Can be tedious and distracting.

```
#include "MyLib.h"
int main()
{
  MyLib::foo();
  MyLib::Cat c;
  c.Meow();
}
```

`using`-declarations

- Introduces a local synonym for name
- States in one place where a name comes from.
- Eliminates redundant scope qualification:

```
int main() {
  using MyLib::foo;
  using MyLib::Cat;
  foo();
  Cat c;
  c.Meow();
}
```

`using`-directives

- Makes *all* names from a namespace available.
- Can be used as a notational convenience.

```
int main() {
  using namespace std;
  using namespace MyLib;
  foo();
  Cat c;
  c.Meow();
  cout << "hello" << endl;
}
```

Ambiguities

- *using*-directives may create *potential* ambiguities.
- Consider:

```
 // Mylib.h
namespace XLib {
  void x();
  void y();
}

namespace YLib {
  void y();
  void z();
}
```

Ambiguities (cont)

- *using-directives* only make the names available.
- Ambiguities arise only when you make calls.
- Use scope resolution to resolve.

```
int main() {
  using namespace XLib;
  using namespace YLib;
  x();        // OK
  y();        // Error: ambiguous
  XLib::y(); // OK, resolves to XLib
  z();        // OK
}
```

Namespace aliases

- Namespace names that are too short may clash
- Names that are too long are hard to work with
- Use aliasing to create workable names
- Aliasing can be used to version libraries.

```
namespace supercalifragilistic {
  void f();
}
namespace short_ns = supercalifragilistic;
short_ns::f();
```

Namespace composition

- Compose new namespaces using from others.
- *using-declarations* can resolve potential clashes.
- Explicitly defined functions take precedence.

```
namespace first {
  void x();
  void y();
}
namespace second {
  void y();
  void z();
}
```

Namespace composition

```
namespace mine {
  using namespace first;
  using namespace second;
  using first::y; // resolve clashes
  void mystuff();
  /* ... */
}
int main() {
  mine::x();
  mine::y(); // call first::y()
  mine::mystuff();
}
```

Namespace selection

- Compose namespaces by selecting a few features from other namespaces.
- Choose only the names you want rather than all.
- Changes to “orig” declaration become reflected in “mine”.

```
namespace mine {
  using orig::Cat; // use Cat class from orig
  void x();
  void y();
}
```

Namespaces are open

- Multiple namespace declarations add to the same namespace.
    - Can be *distributed* across multiple files.

```
// header1.h
namespace X {
  void f();
}

// header2.h
namespace X {
  void g(); // X how has f() and g();
}
```

Final Exam

Object-Oriented Programming with C++

Question types (online test)

- Fill in Blank (text)
  - write the output of the given code
- Fill in blank (program)
  - single line code completion
- Code Completion (function)
  - class design
  - function implementation
- Multiple Choice (single answer)
  - miscellaneous

IDE is *NOT* allowed.