Lecture Material

Class hierarchies and casting

Run Time Type Information (RTTI)

Member pointers

Operators *new* and *delete*

Temporary objects

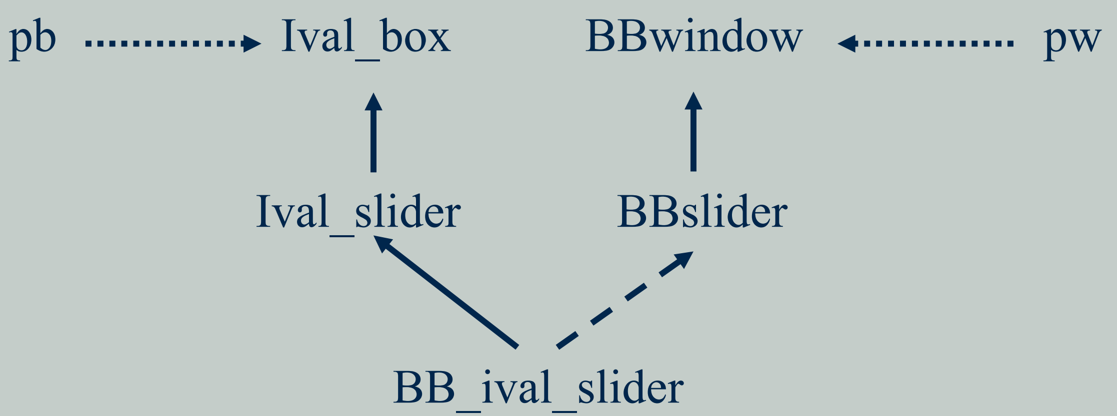
Casting

- A plausible use of the *Ival_box*es would be to hand them to a system that controlled a screen and have that system hand objects back to the application program whenever some activity had occurred.
- User interface system will not know about our *Ival_box*es. The system's interfaces will be specified in terms of the system's own classes and objects rather than our application's classes.
- We lose information about the type of objects passed to the system and later returned to us.
- We need the operation allowing to recreate lost information about the type of an object.

Operator *dynamic_cast*

- Operator *dynamic_cast* returns a valid pointer if the object is of the expected type and a null pointer if it isn't.

```
void my_event_handler(BBwindow* pw)
{
 if (Ival_box* pb = dynamic_cast<Ival_box*>(pw))
                                                   // does pw point to an Ival_box?
   pb->do_something() ;
 else {
      // Oops! unexpected event
      }
}
```



- Casting from a base class to a derived class is often called a downcast because of the convention of drawing inheritance trees growing from the root down. Similarly, a cast from a derived class to a base is called an upcast. A cast that goes from a base to a sibling class, like the cast from *BBwindow* to *Ival_box*, is called a crosscast.

Operator *dynamic_cast*

- The *dynamic_cast* operator takes two operands, a type bracketed by < and >, and a pointer or reference bracketed by (and).
- When using the conversion

  dynamic_cast<T>(p)*

  if *p* is a pointer to *T* or an accessible base class of *T* , the result is exactly as if we had simply assigned *p* to a *T* *, e.g.:

```
class BB_ival_slider : public Ival_slider, protected BBslider {
 // ...
};
void f(BB_ival_slider* p)
{
 Ival_slider* pi1 = p; // ok
 Ival_slider* pi2 =dynamic_cast<Ival_slider*>(p) ; // ok
 BBslider* pbb1 =p; // error: BBslider is a protected base
 BBslider* pbb2 = dynamic_cast<BBslider*>(p) ; // ok: pbb2 becomes 0
}
```

Operator *dynamic_cast*

- The previous example is the uninteresting case. However, it is reassuring to know that *dynamic_cast* doesn't allow accidental violation of the protection of private and protected base classes.
- The purpose of *dynamic_cast* is to deal with the case in which the correctness of the conversion cannot be determined by the compiler. In that case,

  dynamic_cast<T*>(p)

  looks at the object pointed to by *p* (if any). If that object is of class *T* or has a unique base class of type *T* , then *dynamic_cast* returns a pointer of type *T* * to that object; otherwise, 0 is returned.
- If the value of *p* is 0 , *dynamic_cast <T *>(p)* returns 0.
- Note the requirement that the conversion must be to a uniquely identified object. It is possible to construct examples where the conversion fails and 0 is returned because the object pointed to by p has more than one subobject representing bases of type *T*.

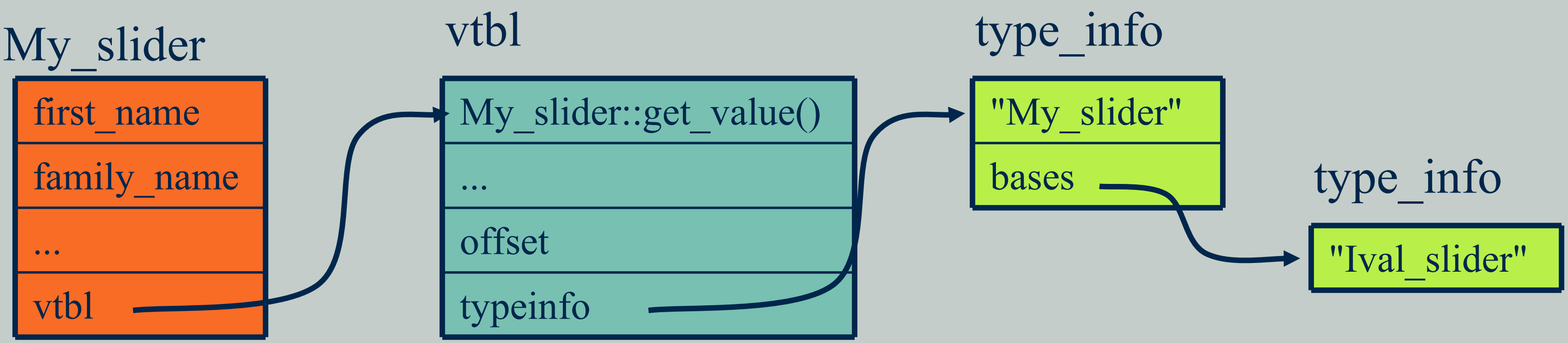
Operator *dynamic_cast*

- A *dynamic_cast* requires a pointer or a reference to a polymorphic type to do a downcast or a crosscast.

```
class My_slider: public Ival_slider { // polymorphic base
                                     //(Ival_slider has virtual functions)
 // ...
};
class My_date : public Date { // base not polymorphic
                              // (Date has no virtual functions)
 // ...
};
void g(Ival_box* pb, Date* pd)
{
  My_slider* pd1 = dynamic_cast<My_slider*>(pb) ; // ok
  My_date*pd2 =dynamic_cast<My_date*>(pd) ; // error: Date not polymorphic
}
```

Operator *dynamic_cast*

- Requiring the pointer's type to be polymorphic simplifies the implementation of *dynamic_cast* because it makes it easy to find a place to hold the necessary information about the object's type.
- A typical implementation will attach a "type information object" to an object by placing a pointer to the type information in the object's virtual function table.



- *offset* allows to find the beginning of the full object, having only a pointer to a polymorphic sub-object.

Operator *dynamic_cast*

- The target type of *dynamic_cast* need not be polymorphic. This allows us to wrap a concrete type in a polymorphic type, say for transmission through an object I/O system, and then "unwrap" the concrete type later.

```
class Io_obj{ // base class for object I/O system
 virtual Io_obj* clone() = 0;
};
class Io_date : public Date, public Io_obj{ };
void f(Io_obj* pio)
{
 Date* pd = dynamic_cast<Date*>(pio) ;
 // ...
}
```

- A *dynamic_cast* to *void* * can be used to determine the address of the beginning of an object of polymorphic type.

```
void g(Ival_box* pb,Date* pd)
{
 void* pd1 = dynamic_cast<void*>(pb) ; // ok
 void* pd2 =dynamic_cast<void*>(pd) ; // error: Date not polymorphic
}
```

dynamic_cast of References

- To get polymorphic behavior, an object must be manipulated through a pointer or a reference.

  When a *dynamic_cast* is used for a pointer type, a 0 indicates failure. That is neither feasible nor desirable for references.

  If the operand of a *dynamic_cast* to a reference isn't of the expected type, a *bad_cast* exception is thrown.

```
void f(Ival_box* p, Ival_box& r)
{
 if (Ival_slider* is = dynamic_cast<Ival_slider*>(p)) {
                                       // does p point to an Ival_slider?
   // use 'is'
 } else {
   // *p not a slider
 }
 Ival_slider& is = dynamic_cast<Ival_slider&>(r) ;
                                       // r references an Ival_slider!
   // use 'is'
}
```

- If a user wants to protect against bad casts to references, a suitable handler must be provided.

```
void g()
{
 try {
   f(new BB_ival_slider,*new BB_ival_slider) ;
                                         // arguments passed as Ival_boxs
   f(new BBdial,*new BBdial) ; // arguments passed as Ival_boxs
 }
 catch (bad_cast) {
   // ...
 }
}
```

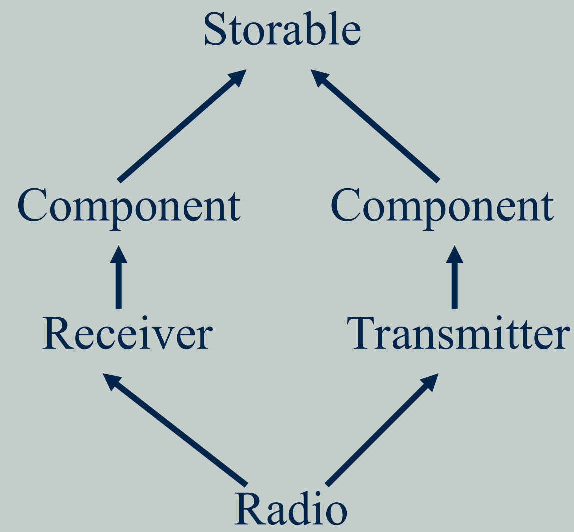
Navigating Class Hierarchies

- When only single inheritance is used, a class and its base classes constitute a tree rooted in a single base class.
- When multiple inheritance is used, there is no single root.
- If a class appears more than once in a hierarchy, we must be a bit careful when we refer to the object or objects that represent that class.

Navigating Class Hierarchies

- Consider the following lattice of classes:

```
class Component : public virtual Storable
{ /* ... */ };
class Receiver : public Component
{ /* ... */ };
class Transmitter : public Component
{ /* ... */ };
class Radio : public Receiver, public
Transmitter{ /* ... */ };
```



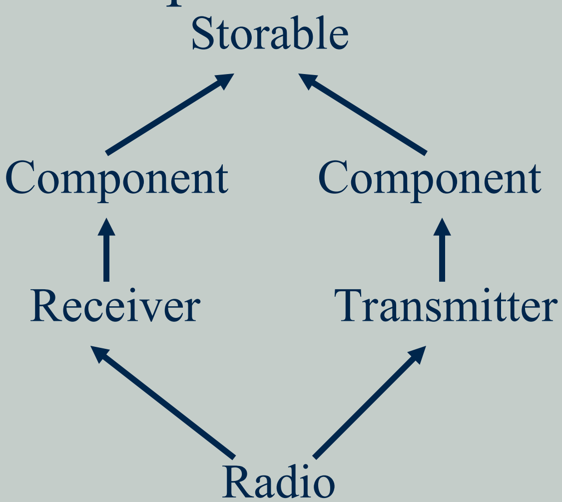
- A *Radio* object has two subobjects of class *Component*. Consequently, a *dynamic_cast* from *Storable* to *Component* within a Radio will be ambiguous and return a 0 . There is simply no way of knowing which *Component* the programmer wanted:

```
void h1(Radio& r)
{
 Storable* ps= &r;
 // ...
 Component* pc = dynamic_cast<Component*>(ps) ; // pc = 0
}
```

Navigating Class Hierarchies

- This ambiguity is not in general detectable at compile time:

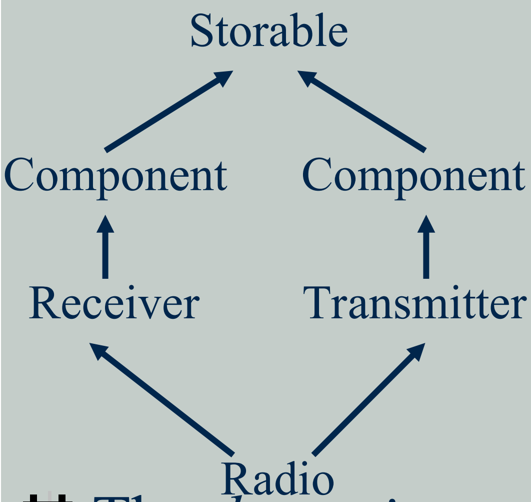
```
void h2(Storable* ps) // ps might or might not
                      // point to a Component
{
 Component* pc = dynamic_cast<Component*>(ps) ;
 // ...
}
```



- This kind of runtime ambiguity detection is needed only for virtual bases. For ordinary bases, there is always a unique subobject of a given cast (or none) when downcasting (that is, towards a derived class).
- The equivalent ambiguity occurs when upcasting (that is, towards a base) and such ambiguities are caught at compile time.

Static and Dynamic Casts

- A *dynamic_cast* can cast from a polymorphic virtual base class to a derived class or a sibling class. A *static_cast* does not examine the object it casts from, so it cannot:




```
void g(Radio& r)
{
 Receiver* prec= &r; // Receiver is ordinary base of Radio
 Radio* pr = static_cast<Radio*>(prec) ; // ok, unchecked
 pr = dynamic_cast<Radio*>(prec) ; // ok, runtime checked
 Storable* ps= &r; // Storable is virtual base of Radio
 pr = static_cast<Radio*>(ps) ;
        // error: cannot cast from virtual base
 pr = dynamic_cast<Radio*>(ps) ; // ok, runtime checked
}
```

- The *dynamic_cast* requires a polymorphic operand.
- There is a small runtime cost associated with the use of a *dynamic_cast*. If the programs provides other means to ensure, that the casting is correct, a *static_cast* can be used.

Static and Dynamic Casts

- The compiler cannot assume anything about the memory pointed to by a *void **. For that, a *static_cast* is needed.

```
Radio* f(void* p)
{
 Storable* ps = static_cast<Storable*>(p) ; // trust the programmer
 return dynamic_cast<Radio*>(ps) ;
}
```

Both *dynamic_cast* and *static_cast* respect *const* and access control:

```
class Users : private set<Person> { /* ... */ };
void f(Users* pu, const Receiver* pcr)
{
 static_cast<set<Person>*>(pu) ; // error: access violation
 dynamic_cast<set<Person>*>(pu) ; // error: access violation
 static_cast<Receiver*>(pcr) ; // error: can't cast away const
 dynamic_cast<Receiver*>(pcr) ; // error: can't cast away const
 Receiver* pr = const_cast<Receiver*>(pcr) ; // ok
 // ...
}
```

- It is not possible to cast to a private base class, and "casting away *const* " requires a *const_cast*. Even then, using the result is safe only provided the object wasn't originally declared *const*.

Cast Operators Summary

- *static_cast*
  - unchecked casting between related types
- *dynamic_cast*
  - checked casting between related types
- *const_cast*
  - removal of *const* attribute from the object
- *reinterpret_cast*
  - casting between unrelated types (e.g. *int* and pointer)
- C-style casting *(T)e*
  - any conversion, that can be expressed as a combination of operators *static_cast*, *reinterpret_cast* and *const_cast*

Class Object Construction and Destruction

- A class object is built from "raw memory" by its constructors and it reverts to "raw memory" as its destructors are executed.
- Construction is bottom up, destruction is top down, and a class object is an object to the extent that it has been constructed or destroyed.
- If the constructor for *Component* calls a virtual function, it will invoke a version defined for *Storable* or *Component*, but not one from *Receiver*, *Transmitter* or *Radio*. At that point of construction, the object isn't yet a *Radio*; it is merely a partially constructed object.
- It is best to avoid calling virtual functions during construction and destruction.

Operator *typeid*

- The *typeid* operator yields an object representing the type of its operand.
- *typeid* behaves like a function with the following declaration:

```
class type_info;
const type_info& typeid(type_name) throw(bad_typeid) ;// pseudo declaration
const type_info& typeid(expression) ; // pseudo declaration
```

- *type_info* is defined in the standard library, in a header file *<typeinfo>*
- Most frequently *typeid()* is used to find a type of an object referred to by a pointer or a reference:

```
void f(Shape& r, Shape* p)
{
 typeid(r) ; // type of object referred to by r
 typeid(*p) ; // type of object pointed to by p
 typeid(p) ; // type of pointer, that is, Shape*
            // (uncommon, except as a mistake)
}
```

- If the value of a pointer is 0, *typeid()* throws a *bad_typeid* exception.

Operator *typeid*

- The implementation-independent part of *type_info* looks like this:

```
class type_info {
 public:
   virtual ~type_info() ; // is polymorphic
   bool operator==(const type_info&) const; // can be compared
   bool operator!=(const type_info&) const;
   bool before(const type_info&) const; // ordering
   const char* name() const; // name of type
 private:
   type_info(const type_info&) ; // prevent copying
   type_info& operator=(const type_info&) ; // prevent copying
   // ...
};
```

- The *before()* function allows *type_info*s to be sorted. There is no relation between the relationships defined by *before* and inheritance relationships.
- It is not guaranteed that there is only one *type_info* object for each type in the system.
  - we should use == on *type_info* objects to test equality, rather than == on pointers to such objects.

Operator *typeid*

- We sometimes want to know the exact type of an object so as to perform some standard service on the whole object (and not just on some base of the object).
- Ideally, such services are presented as virtual functions so that the exact type needn't be known.
- In some cases, no common interface can be assumed for every object manipulated, so the detour through the exact type becomes necessary.
- Another, much simpler, use has been to obtain the name of a class for diagnostic output:

```
#include<typeinfo>
void g(Component* p)
{
 cout << typeid(*p).name() ;
}
```

- The character representation of a class' name is implementation-defined.
- This C-style string resides in memory owned by the system, so the programmer should not attempt to *delete []* it.

Uses and Misuses of RTTI

- RTTI = Run Time Type Information
- One should use explicit runtime type information only when necessary
- Static (compile-time) checking is safer, implies less overhead, and – where applicable – leads to better-structured programs.
- For example, RTTI can be used to write thinly disguised switch-statements:

```
// misuse of runtime type information:
void rotate(const Shape& r)
{
 if (typeid(r) == typeid(Circle)) {
   // do nothing
 }
 else if (typeid(r) == typeid(Triangle)) {
   // rotate triangle
 }
 else if (typeid(r) == typeid(Square)) {
   // rotate square
 }
// ...
}
```

- Using *dynamic_cast* rather than *typeid* would improve this code only marginally.
- Virtual functions are the best solution here.

Pointers to Members

- Pointers to members are useful, when a class has many member function with the same arguments.

```
class X {
 double g(double a) { return a*a + 5.0; }
 double h(double a) { return a - 13; }
public:
 void test(X*, X);
};
typedef double (X::*pf)(double);// pointer to member
void X::test(X* p, X q) {
 pf m1 = &X::g;
 pf m2 = &X::h;
 double g6 = (p->*m1)(6.0); // call through pointer to member
 double h6 = (p->*m2)(6.0); // call through pointer to member
 double g12 = (q.*m1)(12); // call through pointer to member
 double h12 = (q.*m2)(12); // call through pointer to member
}
int main(){
 X i;
 i.test(&i, i);
}
```

- ->* and *. are the special operators to deal with pointers to members
- A pointer to a static member is a normal pointer

Pointers to Members

- The virtual functions work as usual

```
class X
{
protected:
  int val;
public:
  X(int v) : val(v) {};
  virtual void f (double a)
  {
    cout << a + val <<endl;
  }
  virtual ~X(){};
};


class Y: public X
{
public:
  Y(int v) : X(v) {};
  void f (double a)
  {
    cout << 2 * a + val <<endl;
  }
};
```

```
typedef void (X::*pf) (double);

void
test (X * p, X * q)
{
  pf m = &X::f;
  (p->*m)(6.0);
  (q->*m)(7.0);
}

int
main ()
{
  X i(3);
  Y j(4);
  test (&i, &j);
}
```

- A pointer to a virtual member isn't a pointer to a piece of memory the way a pointer to a variable or a pointer to a function is. It is more like an index into an array (virtual function table).
- A pointer to a virtual member can therefore safely be passed between different address spaces as long as the same object layout is used in both.

Pointers to Members and Inheritance

- A derived class has at least the members that it inherits from its base classes. Often it has more.
- This implies that we can safely assign a pointer to a member of a base class to a pointer to a member of a derived class, but not the other way around.

```
class X {
 public:
   virtual void start() ;
   virtual ~X() {}
};
class Y : public X {
 public:
   void start() ;
   virtual void print() ;
};
void (X::* pmi)() = &Y::print; // error
void (Y::*pmt)() = &X::start; // ok
```

Operators *new* and *delete*

- The operators dealing with the free store (*new*, *delete*, *new []* and *delete[]*) are implemented using functions:

```
void* operator new(size_t) ; // space for individual object
void operator delete(void*) ;
void* operator new[](size_t) ; // space for array
void operator delete[](void*) ;
```

- When operator *new* needs to allocate space for an object, it calls *operator new()* to allocate a suitable number of bytes. Similarly, when operator *new* needs to allocate space for an array, it calls *operator new []()*.
- When *new* can find no store to allocate, the allocator throws a *bad_alloc* exception.
- We can specify what *new* should do upon memory exhaustion. When *new* fails, it first calls a function specified by a call to *set_new_handler()* declared in <*new*>, if any.

```
void out_of_store() {
  cerr << "operator new failed: out of store\n";
  throw bad_alloc() ;
}
int main() {
  set_new_handler(out_of_store) ; // make out_of_store the new_handler
  for (;;) new char[10000] ;
    cout << "done\n";
}
```

Operators *new* and *delete*

- A *new_handler* might do something more clever than simply terminating the program.
- If a programmer knows how *new* and *delete* work – for example, because he provided his own *operator new()* and *operator delete()* – the handler might attempt to find some memory for *new* to return.
- Operator *new()* implemented using *malloc* can look like follows:

```
void* operator new(size_t size)
{
 for (;;) {
   if (void* p =malloc(size)) return p; // try to find memory
   if (_new_handler == 0) throw bad_alloc() ; // no handler: give up
   _new_handler() ; // ask for help
 }
}
```

- The *new_handler* can do one of the following things:
  - find more memory and return
  - throw *bad_alloc*

Placement *new*

- We can place an object at any address, using the placement *new* operator

```
void* operator new(size_t, void* p) { return p; }
                                             // explicit placement operator

int main()
{
 char buf[sizeof(string)];
 string* s = new(buf) string; // construct an string at 'buf;' invokes:
                              // operator new(sizeof(string),buf);
 *s="hello";
 cout << *s<<endl;
 s->~string();
};
```

- It is one of the rare cases, when explicit call of a destructor is used
- This code still has alignment problems with character buffer. Should use *std::aligned_storage_t*.
- This is the simplest version of placement *new* operator. It is defined in a header file <*new*>

Placement *new*

- The placement *new* construct can also be used to allocate memory from a specific arena:

```
class Arena {
 public:
   virtual void* alloc(size_t) =0;
   virtual void free(void*) =0;
   // ...
};
void* operator new(size_t sz,Arena* a) {
 return a->alloc(sz) ;
}
```

- Now objects of arbitrary types can be allocated from different *Arena*s as needed.

```
extern Arena*Persistent;
extern Arena* Shared;
void g(int i) {
 X* p = new(Persistent)X(i) ; // X in persistent storage
 X* q = new(Shared) X(i) ; // X in shared memory
 // ...
}
```

- The destructor has to be called explicitly

```
void destroy(X* p,Arena* a) {
 p->~X() ; // call destructor
 a->free(p) ; // free memory
}
```

Placement *delete*

- The placement *delete* operator is invoked, if an exception is thrown in the object constructor.

```
void operator delete (void *s, Arena * a)
{
  a->free (s);
};
```

- Apart from scalar placement *new* and *delete* operators we can define similar operators for arrays.

Memory Management for Classes

- It is possible to take over memory management for a class by defining *operator new()* and *operator delete()* as class members.

```
class Employee {
 // ...
 public:
 // ...
    void* operator new(size_t) ;
    void operator delete(void*, size_t) ;
};
```

- Member *operator new()*s and *operator delete()*s are implicitly *static* members.

```
void* Employee::operator new(size_t s)
{
 // allocate 's' bytes of memory and return a pointer to it
}
void Employee::operator delete(void* p, size_t s)
{
 // assume 'p' points to 's' bytes of memory
 // allocated by Employee::operator new()
 // and free that memory for reuse
}
```

Memory Management for Classes

- Using *size_t* argument in a *delete* operator, the memory allocation function can avoid storing the information about the size of allocated block at every allocation.
- When the object is freed via the pointer to its base class, we need to pass the right size to *operator delete*:

```
class Manager : public Employee {
 int level;
 // ...
};
void f()
{
 Employee* p = new Manager; // trouble (the exact type is lost)
 delete p;
}
```

- To avoid the problem, the base class needs a virtual destructor. Even the empty destructor will do.

```
class Employee {
 public:
   void* operator new(size_t) ;
   void operator delete(void*, size_t) ;
   virtual ~Employee() ;
   // ...
};
Employee: :~Employee() { }
```

Memory Allocation for an Array of Objects

- The class can also define array allocators and deallocators, used when dealing with arrays of objects:

```
class Employee {
 public:
   void* operator new[](size_t) ;
   void operator delete[](void*, size_t) ;
   // ...
};
void f(int s)
{
 Employee* p = new Employee[s] ;
 // ...
 delete[] p;
}
```

- The memory needed will be obtained by a call,

$$Employee::operator\ new[\,]\,(sizeof(Employee)*s+delta)$$

  where *delta* is some minimal implementation-defined overhead, and released by a call:

$$Employee::operator\ delete[\,]\,(p,\ s*sizeof(Employee)+delta)$$

Temporary Objects

- Temporary objects most often are the result of arithmetic expressions. For example, at some point in the evaluation of $x*y+z$ the partial result $x*y$ must exist somewhere.
- Unless bound to a reference or used to initialize a named object, a temporary object is destroyed at the end of the full expression in which it was created. A full expression is an expression that is not a subexpression of some other expression.

```
void f(string& s1, string& s2, string& s3)
{
 const char* cs= (s1+s2).c_str() ;
 cout << cs;
 if (strlen(cs=(s2+s3).c_str())<8 && cs[0]==´a´) {
      // cs used here
 }
}
```

Pointer to freed memory

- A temporary object of class *string* is created to hold *s1+s2*. Next, a pointer to a C-style string is extracted from that object. Then – at the end of the expression – the temporary object is deleted.
- The condition will work as expected because the full expression in which the temporary holding *s2+s3* is created is the condition itself. However, that temporary is destroyed before the controlled statement is entered, so any use of *cs* there is not guaranteed to work.

Temporary Objects

- A temporary can be used as an initializer for a *const* reference or a named object.

```
void g(const string&, const string&) ;
void h(string& s1, string& s2)
{
 const string& s = s1+s2;
 string ss = s1+s2;
 g(s,ss) ; // we can use s and ss here
}
```

- A temporary object can also be created by explicitly invoking a constructor. Such temporaries are destroyed in exactly the same way as the implicitly generated temporaries.

```
void f(Shape& s, int x, int y)
{
 s.move(Point(x,y)) ; // construct Point to pass to Shape::move()
 // ...
}
```