Implementation Aspects of OO-Languages

- Allocation of space for data members: The space for data members is laid out the same way it is done for structures in C or other languages. Specifically:
  - The data members are allocated next to each other.
  - Some padding may be required in between fields, if the underlying machine architecture requires primitive types to be aligned at certain addresses.
  - At runtime, there is no need to look up the name of a field and identify the corresponding offset into a structure; instead, we can statically translate field names into relative addresses, with respect to the beginning of the object.
  - Data members for a derived class immediately follow the data members of the base class
  - Multiple inheritance requires more complicated handling, we will not discuss it here
Implementation Aspects of OO-Languages

class B {
    int i; double d;
    char c; float f;
}

// Integer requires 4 bytes
// pad,

// Double requires 8 bytes

// char needs 1 byte, 3 are padded
// float to be aligned on 4-byte
// require 4-bytes of space
class C {
    int k, l; B b;
}

int k
int l
int i
XXXXXXXXXXXXXXXX
double d
char c|XXXXXX
float f
class D: public C {
    double x;
}

<table>
<thead>
<tr>
<th></th>
<th>int k</th>
<th>int l</th>
<th>int i</th>
<th>XXXXXXXXXXX</th>
<th>XXXXXXXXXXXX</th>
<th>double d</th>
<th>char c</th>
<th>XXXXX</th>
<th>float f</th>
<th>double x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Implementation of Virtual Functions

• Approach 1:
  • Lookup type info at runtime, and then call the function defined by that type.
  • Problem: very expensive, require type info to be maintained at runtime.
• Approach 2:
  • Treat function members like data members:
    • Allocate storage for them within the object.
    • Put a pointer to the function in this location, and translate calls to the function to make an indirection through this field.
  • Benefit:
    • No need to maintain type info at runtime.
    • Implementation of virtual methods is fast.
  • Problem:
    • Potentially lot of space is wasted for each object.
    • Even though all objects of the same class have identical values for the table.
Implementation of Virtual Functions (Contd.)

• Approach 3:
  • Introduce additional indirection into approach 2.
  • Store a pointer to a table in the object, and this table holds the actual pointers to virtual functions.
  • Now we use only one word of storage in each object.
class B {
    int i;
    char c;
    virtual void g();
    virtual void h();
}

B b1, b2;
Impact of subtype principle on Implementation

- The subtype principle requires that any piece of code that operates on an object of type B can work "as is" when given an object belonging to a subclass of B.
- This implies that runtime representation used for objects of a subtype A must be compatible with those for objects of the base type B.
- Note that the way the fields of an object are accessed at runtime is using an offset from the start address for the object.
  - For instance, b1.i will be accessed using an expression of the form *(&b1+0), where 0 is the offset corresponding to the field i.
  - Similarly, the field b1.c will be accessed using the expression *(&b1+1)
Impact of subtype principle on Implementation (Contd.)

- an invocation of the virtual member function b1.h() will be implemented at runtime using an instruction of the form:

```call *(*(&b1+2)+1)```

- &b1+2 gives the location where the VMT ptr is located
- *(&b1+2) gives the value of the VMT ptr, which corresponds to the location of the VMT table
- *(&b1+2) + 1 yields the location within the VMT table where the pointer to virtual function h is stored.
The subtype principle imposes the following constraint:

- Any field of an object of type B must be stored at the same offset from the base of any object that belongs to a subtype of B.
- The VMT ptr must be present at the same offset from the base of any object of type B or one of its subclasses.
- The location of virtual function pointers within the VMT should remain the same for all virtual functions of B across all subclasses of B.
Impact of subtype principle on Implementation (Contd.)

- We must use the following layout for an object of type A defined as follows:

```c++
class A: public B {
    float f;
    void h(); // reuses implementation of G from B;
    virtual void k();}
```

```c++
A a;
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>c</th>
<th>VMT ptr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Float f</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Virtual Method Table (VMT) for class A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ptr to B’s g</td>
</tr>
<tr>
<td>Ptr to A’s h</td>
</tr>
<tr>
<td>Ptr to A’s k</td>
</tr>
</tbody>
</table>
```
Impact of subtype principle on Implementation (Contd.)

- In order to satisfy the constraint that VMT ptr appear at the same position in objects of type A and B, it is necessary for the data field f in A to appear after the VMT field.

- A couple of other points:
  - a) non-virtual functions are statically dispatched, so they do not appear in the VMT table
  - b) when a virtual function f is NOT redefined in a subclass, the VMT table for that class is initialized with an entry to the function f defined its superclass.