The g++ implementation of C++ Object Model

January 31, 2011 **programming language** **computer system**

This text discusses the C++ object model implemented by g++, including the memory layout of C++ objects and implementation mechanism of polymorphism, virtual inheritance, pointer to data member and member function and RTTI. I have never seen any official document on these issues. As far as I know *Inside the C++ object model* by Stanley Lippman is the only book which has detailed descriptions on the underlying mechanism. Unfortunately the materials in that book are mostly about some very old C++ compilers which are rarely used today. But it is a great help for me to study the details about g++.

The mechanisms are studied by disassembling the object code generated by g++ so I can not guarantee it is correct. All examples are tested under g++ 4.4.3 and Ubuntu 10.04.

The memory layout of C++ objects

The memory model of an object is composed of two parts: the data layout and the virtual table layout. The virtual table is used by g++ to implement polymorphism and virtual inheritance and is accessed via the virtual pointer stored in the object.

A class has at least one virtual pointer if and only if it has virtual base classes or virtual functions. The virtual pointer points to the slot beyond the RTTI pointer which points to the `typeinfo` object for this class. The virtual table slots can be accessed via positive, zero and negative index through virtual pointers. The slots with non-negative index are used to store addresses of virtual functions (or their thunks, see Call virtual functions or of a g++ defined function `__cxa_pure_virtual`, which will terminate the program, if a pure virtual function is declared. The RTTI pointer is at index -1. The inversion of offset of the virtual pointer in this object is at index -2.

Without inheritance

When considering no inheritance the rule for g++ to layout an object is as follows.

1. Layout the virtual pointer first if any.
2. Layout non-static data members in their declaration order.

The storage of data members must satisfy the alignment requirement so there may be padding between data members. But for simplicity this text do not take it into consideration. It is irrelevant whether a data member is a built-in type or class type. So in examples all data members are declared as `int` which occupy 4 bytes in my environment.

Static data members and non-virtual functions (static member functions and non-static member functions) do not have any impact on the memory layout of C++ objects and therefore ignored in this text.

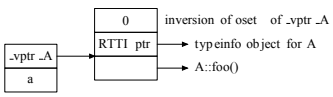
Without inheritance, the rule for g++ to create virtual tables, if any, is as follows.

1. The inversion of offset of the virtual pointer pointing to this sub-virtual-table is stored at index -2 from the virtual pointer.
2. RTTI pointer is stored at index -1.
3. Beginning from index 0 and so on are the virtual function slots, where the addresses of virtual functions or thunks are stored.

The following is a simple example.

```
struct A {
    int a;
    virtual void foo() { }
};
```

Its memory layout is shown as follows.



Add non-virtual inheritance

A derived object is created by concatenating all base class subobjects and appending data members declared by itself. The virtual pointers, if any, are updated appropriately.

After adding non-virtual inheritance the rule for g++ to layout an object of a derived class `D` is as follows.

1. If all base classes have no virtual tables, the rule is the same as the situation without inheritance (see last section);
2. Otherwise, the first base class subobject with virtual pointers is put at the beginning and other base class subobjects are put in their inheritance order;
3. The data members declared in `D` are put at the end.

Similarly, its virtual table is created by concatenating all base class virtual tables (which is referred to as sub-virtual-table in this text) after appending the virtual functions declared in this class (not inherited or overridden) to the first sub-virtual-table.

After adding non-virtual inheritance the rule for g++ to create virtual tables for a derived class `D` is as follows.

1. If all base classes have no virtual tables and the derived class do not declare any virtual function, no virtual tables are created.
2. Otherwise, if all base classes have no virtual tables and the derived class declares some virtual function, the rule for creating the virtual table is the same as the situation without inheritance (see Without inheritance).
3. Otherwise, if some base class has a virtual table, the virtual table of the derived class is created by concatenating sub-virtual-tables. The rule to create the first sub-virtual-table is as follows.
    1. copy the virtual table of the first base class which has one;
    2. if a virtual function is overridden in the derived class, update its corresponding virtual table slot to the address of the overriding function (which implements polymorphism);
    3. if a virtual function is declared (not overriding) in the derived class, append its address to the virtual table;
    4. update the RTTI pointer to points to the `typeinfo` object for class `D`.
4. The second and subsequent sub-virtual-tables are created in the order of the inheritance list, and the rules for creating them is as follows.
    1. copy the virtual table of a corresponding base class `B`;
    2. if a virtual function in `B` is overridden, update its corresponding virtual table slot to the address of thunk (see Call virtual functions to the overriding function (which implements polymorphism);
    3. update the RTTI pointer to points to the `typeinfo` object for `D`;
    4. update the value at index -1, which should equal to the inversion of offset of the `B` subobject in `D`;
5. After creating the virtual table of `D` update the virtual pointers in `D` object so that they correspond to the RTTI pointers one-by-one in increasing order of their addresses and each virtual pointer points to the slot beyond the RTTI pointer.

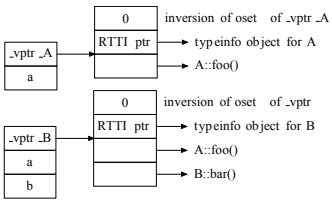
For multiple inheritance hierarchies these rules can be applied recursively.

According to above rules when considering no virtual inheritance the offset of each data member is fixed no matter how many times the class is inherited, which means that the offset of a data member of a class relative to the address of the class object or subobject of its some derived class is never changed.

Example 1: class `B` inherits from `A`.

```
struct A {
    int a;
    virtual void foo() { }
};
struct B : A {
    int b;
    virtual void bar() { }
};
```

Their memory layouts are shown as follows.

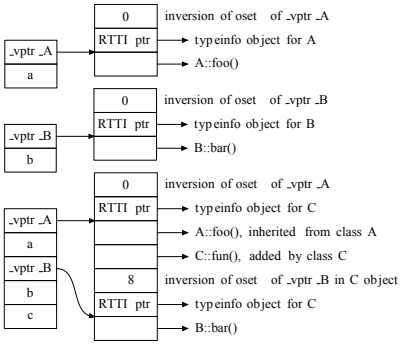


Note that the offset of `a` in class `A` is 4 and in the `A` subobject of class `B` is also 4.

Example 2: multiple inheritance. Classes `A` and `B` have their own virtual pointer and class `C` inherits from `A` and `B`.

```
struct A {
    int a;
    virtual void foo() { }
};
struct B {
    int b;
    virtual void bar() { }
};
struct C : A, B {
    int c;
    virtual void fun() { }
};
```

Their layouts are shown as follows.

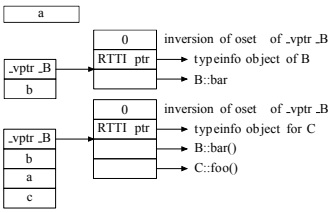


Note that the offset of class `B` data member `b` in class `B` is 4 bytes and its offset in the class `B` subobject of class `C` is also 4 bytes.

Example 3: multiple inheritance with the first base class lacking virtual pointer.

```
struct A {
    int a;
};
struct B {
    int b;
    virtual void foo() { }
};
struct C: A, B {
    int c;
    virtual void bar() { }
};
```

Their layouts are shown as follows.

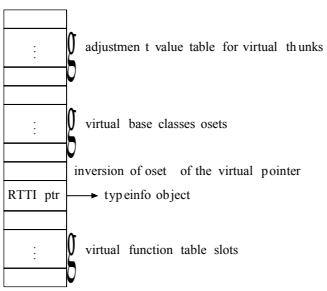


Add virtual inheritance

The virtually inherited base class subobject occurs only once in the most derived class object, which results in that virtual base class subobject does not has fixed offset in derived class object. So the virtual base class subobject must be accessed indirectly. g++ achieves this goal by putting virtual base class offset table, which contains offsets of each virtual base class subobject, in the virtual table. (“Virtual inheritance” of 3.4 of *Inside the C++ Object Model*)

In addition, as discussed in Call virtual functions, the virtual function has to adjust the `this` pointer sometimes, which is done by the thunk. So the thunk needs a way to calculate the adjustment value. For the situation with non-virtual inheritance this can be done at compile time because the offset of the subobject is fixed. But for the situation with virtual inheritance this can only be done indirectly. Similarly g++ solves this problem by putting the adjustment value in the virtual table, one for each virtual function inherited from a virtual base class, for which g++ will generate a virtual thunk to it. Note that only virtual thunk needs the adjustment value and thunk is never stored in the first sub-virtual-table because the `this` pointer do not need to be adjusted when the virtual function is called through the first sub-virtual-table, so the first sub-virtual-table does not have these values. For the second and subsequent sub-virtual-tables each virtual function inherited from a virtual base class has a corresponding adjustment value even if it is not a thunk. If it is not a thunk the adjustment value is 0, which is used as a placeholder, because it may be needed if the virtual function is overridden by some derived class and then a virtual thunk is generated.

Take all situations into consideration the general layout of a sub-virtual-table is like the following.



A sub-virtual-table of a class `D` is composed of four or five parts:

1. Adjustment value table for virtual functions, one slot for each virtual function. The first sub-virtual-table and sub-virtual-tables which is not inherited virtually do not have this part.
2. Virtual base class offset table, one slot for each virtual base class including all virtual base classes in the inheritance hierarchy. Its index begins from -3 and go on decreasing.
3. Inversion of offset of the virtual pointer pointing to this virtual table, whose index is always -2. This is necessary for `dynamic_cast<void*>` in particular.
4. RTTI pointer pointing to `typeinfo` object for `D`, whose index is always -1.
5. Virtual function table which stores addresses of each virtual function, whose index begins from 0 and go on increasing. The virtual pointer is pointing to the first entry of this table.

Generally speaking the rule for g++ to layout an object is as follows.

1. First layout the non-virtually inherited subobject as usual;
2. Then layout the virtually inherited subobject. The rule is same.

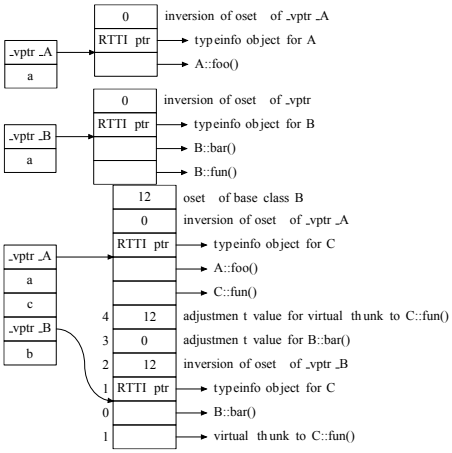
And the rule for g++ to create virtual tables for a derived class `D` is same as usual except

1. The sub-virtual-tables from non-virtual base classes are created first and then those from virtual base classes, corresponding to the layout of the object.
2. The first sub-virtual-table is copied from the the first non-virtual base class which has a virtual table. If all non-virtual base classes do not have a virtual table then the first sub-virtual-table is created for class `D` and the rule is the same as in Without inheritance.
3. Then the first sub-virtual-table is prepended with virtual base class offset table.
4. The second and subsequent sub-virtual-tables are prepended with a adjustment value table if its corresponding base class is virtually inherited and has declared some virtual functions and the values in their virtual base class offset table are updated.

Let's see an example.

```
struct A {
    int a;
    virtual void foo() { }
};
struct B {
    int b;
    virtual void bar() { }
    virtual void fun() { }
};
struct C: A, virtual B {
    int c;
    void fun() { }
};
```

The layout of classes `A`, `B` and `C` is shown in Figures 11, 12 and 13 respectively.



In Figure 13 the sub-virtual-table `_vptr_A` points to has no adjustment values because it is the first sub-virtual-table. The second sub-virtual-table, which is pointed to by `_vptr_B`, has two adjustment values because class `B` is virtually inherited and has declared two virtual function. One is `B::bar()` at index 0 and its adjustment value is 0 which is at index -3 and is used as a placeholder. The other is virtual thunk to `C::fun()` at index 1 and its adjustment value is -12 which is at index -4. We can conclude that the more the index of a virtual function slot is the less the index of its adjustment value is. Putting it another way the index of a virtual function slot and the index of its corresponding adjustment value are symmetric.

How to access data members

Through objects

The compiler knows the offset of each data member in the object, even the data member inherited from a virtual base class. And the dynamic type and the static type of the object are the same so we can access the data member directly. The following code

```
T obj;
obj.x = 1;  // assign the data member x of class T
```

which access the data member `x` of class `T`, is translated to the following pseudocode

```
*(&obj + offset_of_x) = 1
```

Through pointers and references

Accessing through pointers to objects is the same as through references, so we only discuss the situation of pointers.

The difference between accessing through pointers and through objects is that the dynamic type and static type of the object which the pointer points to may be different. The offset of data members, which is not inherited from a virtual base class, is never changed so we can access the data member directly just like through objects. Given the following inheritance hierarchy.

```
struct A {
    int a;
};
struct B: A {
    int b;
};
B b;
B *pb = &b;
A *pa = &b;
```

Note that `pa` points to `A` subobject of `B`. Then the accessing code is translated as follows.

```
pb->b = 1;  // --> *(pb + offset_of_b_in_B) = 1
pb->a = 2;  // --> *(pb + offset_of_a_in_B) = 2
pa->a = 1;  // --> *(pa + offset_of_a_in_A) = 1
```

Here comes the difficulty when accessing a data member of a virtual base class through pointers and references. The difficulty lies in that the offset the data member is not fixed any more. We can only access the data member indirectly. First we must find the address of the virtual base class subobject, which can be done by accessing the virtual base class offset table.

Given the following inheritance hierarchy.

```
struct A {
    int a;
};
struct B: virtual A {
    int b;
};
B b;
A *pa = &b;
B *pb = &b;
```

Then

```
pa->a = 1;
```

is translated to

```
*(pa + offset_of_a_in_A) = 1;
```

and

```
pb->a = 1;
```

is translated to

```
A *pa_t = pb + pb->_vptr_B[-3];  // find the address of class A subobject;
                                 // its index is -3;
                                 // _vptr_B is the virtual pointer of B
*(pa_t + offset_of_a_in_A) = 1;  // the same as above
```

In summary the algorithm of accessing the data member through a pointer or reference is as follows.

> If the data member is inherited from a virtual base class find the address of the subobject through virtual base class offset table and then access it through that address. Otherwise access the data member directly.

Through pointer to data member

According to above discussion we know we need two information to access a data member: the address of the object and the offset of the data member. In fact the pointer to data members (PMD) is just an offset. So we can access the data member through pointers to data members just like through pointers to objects.

But we must be careful with some special situations. For example,

```
struct A {
    int a;
};
struct B {
    int b;
};
struct C: A, B {
    int c;
};
... &A::a ...  // has type int A::*
... &C::a ...  // has type int A::* because a is the member of class A
               // (see clause 2 of 5.3.1 of C++ standard.)
```

When an assignment involves a pointer to member of a base class and a pointer to member of a derived class the conversion is required because the offset in the base class may be different from that in the derived class. In the following code

```
int C::*pmc = &C::b;
```

`&C::b` has type `int A::*`, which is different from the type of left hand side (and have different offsets), so comes in the conversion:

```
int C::*pmb = &C::b + sizeof(A);
```

But a pointer to member of a virtual base class can not be converted to a pointer to member of its derived class (See clause 2 of 4.11 of “ISO C++98 Standard”). It is easy to agree with this when considering that the virtual base class subobject do not have fixed offset.

How to call member functions

There are three kinds of member functions: static member functions, non-static member functions and virtual functions. Non-static and virtual Member functions are converted to an non-member function by adding an additional parameter as the first parameter (a.k.a the `this` pointer, which is used to access the member of this object). The first parameter for non-static and virtual functions must be a pointer to the class object or subobject in which the function is declared (not inherited from a base class), which means the pointer must point to the subobject of a base class if the called function is from the base class.

Static member functions are not bound to an object and can’t access non-static members so they don’t require the `this` pointer. They are called just like global functions.

So when calling non-static and virtual member functions the following two tasks must be done correctly:

1. the entry of the called function must be found;
2. an appropriate `this` pointer must be passed as the first parameter.

The first task is easy without polymorphism. The compiler knows the address of each function at compile time and knows which function is called. When involving polymorphism the compiler does not know which function is actually called at compile time.

The second task means that if calling a function inherited from a base class the `this` pointer must be adjusted to point to the subobject of the base class which declares the function.

Call static member functions

Static member functions can be called through class name, objects, pointers and references. These different forms have the same semantic and the compiler will generate codes that jump to the entry of the function directly, which is the same as calling a global function.

Call non-static member functions

Non-static member functions can be called through objects, pointers and references. Calling non-static member functions is different from calling static member functions because it has to pass the `this` pointer as the first parameter, which must points to the object or subobject of the class which declares the function. That means the following code

```
obj.fun();  // obj is an object of class T
```

is translated to

```
_T_fun(&obj); // _T_fun is the mangling name of T::fun()
```

if `fun()` is declared in class `T`, and translated to

```
B *p = &obj + offset_of_B_in_T; // adjust p to point to B subobject in T
_T_fun(p);  // _T_fun is the mangling name of T::fun();
```

if `fun()` is declared in some base class `B` of class `T`.

Calling through pointers and references is similar. When calling a non-static member function `fun()` through a pointer `T *p`, if `fun()` is declared in class `T` the pointer `p` is passed as the first parameter, and if `fun()` is declared in some base class `B` of `T` then the pointer `p` is adjusted to point to the subobject of class `B` before passed as the first parameter.

The difference from calling through objects lies in the method of adjusting the `this` pointer when the called function is declared in some virtual base class, which is via the virtual base class offset table, because the offset of virtual base class subobject is not fixed.

Call virtual functions

Calling virtual functions through objects is same as calling non-static member functions because it does not exhibit polymorphism.

The difficulty arises when calling through pointers and references because it must exhibit polymorphism.

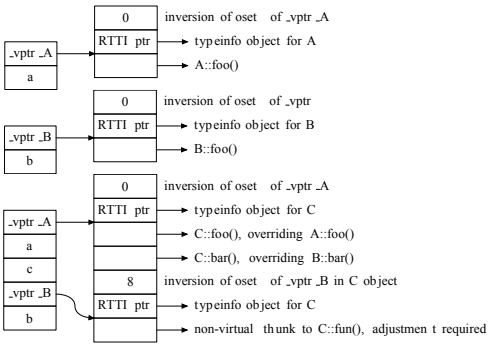
The first is to find the actually called function. This can be solved via virtual table, which can be found via the virtual pointer stored in each object. The address of each virtual function is stored in the virtual table slots. Each virtual function is associated with an index of the table, which is never changed in the inheritance hierarchy and is known at compile time. First the pointer must be adjusted to point to the subobject of the class in which the called virtual function is declared (see Call non-static member functions for how to do the adjustment). Then the virtual pointer can be found through the adjusted pointer and the address of the virtual function can be found by indexing the virtual table.

The second task is passing an appropriate `this` pointer, which the actually called virtual function expects to receive. The compiler does not know which virtual function is called at compile time if it is invoked through a pointer so it has no idea how to adjust the pointer before passing it as the first parameter. g++ solves this problem by storing a pointer to a thunk to the virtual function if the adjustment is required or the virtual function itself if not. A thunk is a snippet of assembly code which can adjust the pointer correctly. Only when a virtual function from a base class is overridden is the adjustment required. When the adjustment is required two situations must be considered. If the overridden virtual function comes from a non-virtual base class the adjustment value can be calculated at compile time, but if it comes from a virtual base class the adjustment value can only be obtained by accessing the adjustment value table (see vinheri). So g++ generates two kinds of thunks, one for the non-virtual base class called non-virtual thunk and one for the virtual base class called virtual thunk, respectively. In this arrangement for each base class `B` of `D` the entry point in the B-in-D sub-virtual-table expects its `this` pointer to point the `B` subobject of `D`. The thunk can do the adjustment if required.

The following example can show when the adjustment is required.

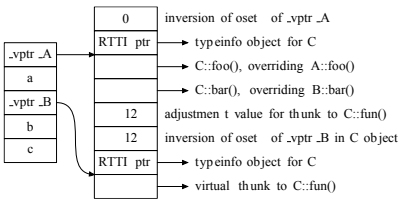
```
stuct A {
    int a;
    virtual void foo() {
        cout << "A::foo()\n";
    }
};
struct B {
    int b;
    virtual void bar() {
        cout << "B::bar()\n";
    }
};
struct C: A, B {
    int c;
    void foo() {
        cout << "C::foo()\n";
    }
    void bar() {
        cout << "C::bar()\n";
    }
}
```

The memory layout corresponding to the above code is shown as below.



When calling `C::foo()` through a pointer to `A` the adjustment is not required, because the `A` subobject in `C` has zero offset, the value of the pointer is already right (`C::foo()` receives a pointer to `C` as the first parameter because it is declared in `C`). But when calling `C::fun()` through a pointer to `B` the adjustment is required, because the `B` subobject in `C` has non-zero offset (8 bytes in this example), the pointer must be subtracted by 8 before passing to `C::fun` (which receives a pointer to `C` as the first parameter) as the first parameter. So the compiler generates a non-virtual thunk (class `B` is non-virtually inherited) to do the adjustment, as shown in the figure where a pointer to the thunk is stored in the virtual table. Because class `B` is non-virtually inherited the compiler knows the adjustment offset value (8 bytes in the above example) at compile time. So the second sub-virtual-table (which comes from `B`) does not have a adjustment value table. But when involving virtual inheritance, due to the fact that the offset is not fixed any more, the adjustment must be done indirectly and a adjustment value table is created, as shown in the following example.

If all is the same as above except that `B` is virtually inherited then `C` will have the layout in Figure 17.



Note that `B` subobject is put at the end because it is virtually inherited. As shown in the figure the adjustment value is stored in the virtual table with index -12 (and the value is -12 because `B` subobject has an offset of 12), which is used by the virtual thunk to adjust the pointer.

In summary, if class `B` is some base class of `D` and `fun()` is a virtual function declared in `B` with index `idx` in the virtual table, then the following code

```
pb->fun();  // pb is a pointer to B
```

is translated to

```
pb += offset(_vptr);  // Find the _vptr which can address the virtual table
                      // which fun resides in.
_vptr = *pb;          // Get the virtual pointer.
(*_vptr[idx])(pb);    // Find the entry to fun() and call it.
```

Pointer to member function

When taking the address of a non-static member function of a class by class name the type of the result is “pointer to member function” (PMF) (See section 5.3.1 clause 2 of [C++98]). It is incomplete because it can not be used to call a function by itself unless bound to the address of a class object.

Remember that two kinds of information are required to call a member function: the entry point to the function and the `this` pointer. For non-virtual functions the entry point can be obtained at compile time and the `this` pointer is the address of the object or subobject in which the function is declared, so an offset must be provided to get this address. For virtual functions the entry point can only be obtained via the virtual table. A class may have multiple virtual tables (and thus multiple virtual pointers), so a offset must be provided to get the virtual pointer which can access the virtual table and a index into the virtual table must also be provided to get the entry point. The `this` pointer is just the address of the virtual pointer, which equals to the address of the subobject in which the function is declared.

According to above discussion, both situations need an offset to calculate the address of the subobject in which the function is declared. For the non-virtual function the address of the function is required and for the virtual function an index to the virtual table is required. In fact PMF is a structure declared as

```
struct PMF {
    union {
        UINT32 faddr;  // entry point to non-virtual function
        UINT32 index;  // index to the virtual table plus 1
    };
    UINT32 offset;     // offset of the subobject
};
```

The member `faddr` of PMF is the address of the function if PMF points to a non-virtual function. The member `index` is the index of this function in the virtual table plus 1 if PMF points to a virtual function. In both cases the member `offset` is the offset of the subobject of the class in which this function is declared. Curiously! Why plus 1? Because we have to distinguish whether a PMF pointer is pointing to a virtual function or not. Calling virtual and non-virtual functions is very different. `faddr` and index of virtual functions are always even, so after adding 1 to the index field of PMF we can now decide if a PMF pointer is pointing to a virtual function or not by checking if the first field of PMF is even or not. The real index can be obtained by subtracting 1 from it. The second reason why plus 1 is that we must distinguish whether a PMF pointer is a null pointer or not. Zero value of `index` or `faddr` represents a null PMF pointer independent of the offset value.

If `pmf` is a PMF pointer of type `void (T::*)()` and `p` is a pointer to class `T`, then the following code

```
(p->*pmf)();
```

is translated to

```
if (pmf.index & 1 != 0) {
    // virtual function invocation
    p += pmf.offset;          // find the vptr
    vptr = *p;
    (*vptr[pmf.index-1]))(p); // call the virtual function
} else {
    // non-vritual function invocation
    (*pmf.faddr)(p + pmf.offset);
}
```

If `pmf` is a PMF pointer of class `B` and `p` is a pointer to a class `D` derived from `B` then calling `pmf` through `p` must be done after `p` is adjusted to point to `B` subobject.

Like pointer to data member, a pointer of type `pointer to member of `B`, where `B` is a class type, can be converted to a pointer of type`pointer to member of `D`, where `D` is a derived class of `B`. The member `faddr` or `index` does not have to adjusted because it never changes. Only the member `offset` needs to be adjusted just like the situation of PMD.

As same as PMD, a pointer to member function of a virtual base class can not be converted to a pointer to member function of its derived class (See clause 2 of 4.11 of [c++98]). The reason is same.

Type identification

According to the standard (see clause 3 of 5.2.8 of [c++98]) when the operand of `typeid` is not polymorphic class type, the result refers to a `typeinfo` object representing the static type of the operand. In this situation the address of the `typeinfo` object can be calculated at compile time.

When the operand of `typeid` is a polymorphic class type the result refers to a `typeinfo` object representing the dynamic type of the operand. In this situation the address of the `typeinfo` object can not be calculated at compile time, and the RTTI pointer stored in the virtual table is just for addressing this problem.

In summary the following code

```
typeid(exp);
```

is translated to

```
T obj = exp;
obj._vptr[-1];  // access the RTTI pointer
```

if the type of `exp` is a polymorphic class type `T`, or return the address of the `typeinfo` object if the type of `exp` is not polymorphic.

Note that even if the result of an expression is a class type which has a RTTI pointer but is not polymorphic (the RTTI pointer is obtained through virtual inheritance) `typeid` applied on it still gets a result refers to a `typeinfo` object representing the static type, not the dynamic type, of the operand.

Dynamic cast

Until now I only learned that the dynamic cast expression `dynamic_cast<T>(v)` is rewritten by g++ as an invocation of a function `__dynamic_cast(v, p1, p2, value)`, where `p1` is a pointer to the `typeinfo` object representing the static type of `v` and `p2` is a pointer to the `typeinfo` object representing the type pointed to by `T`. The meaning of `value` is unknown. It seems like an integer value.

Addition: The code for printing the virtual table

The following is the code for printing the virtual table of a class, which depends on the Linux tools `objdump`, `AWK`, `cut`, `grep` and `c++filt`.

```
/*
 * print_vtable --
 *   Print the virtual table of a class given its name.
 */
void print_vtable(const char *name)
{
    char s[100];

    /*
     * Get the beginning and end addresses of the virtual table
     * by parsing the disassembled code.  The label for a virtual
     * table of a class whose name is "xyz" is "_ZTV3xyz" where 3
     * is the length of the class name.
     */
    FILE *fp;
    unsigned int *vp, *vpe;
    snprintf(s, sizeof(s),
             "objdump -D ./a.out | "
             "awk '/<_ZTV%d%s>: *$/{print $1; f=1; next}"
             "/^0[0-9]+/ && f==1 {print $1; exit}'",
             strlen(name), name);
    fp = popen(s, "r");
    if (fgets(s, sizeof(s), fp) == NULL)
        return;
    vp = (unsigned int *) strtol(s, NULL, 16);
    fgets(s, sizeof(s), fp);
    vpe = (unsigned int *) strtol(s, NULL, 16);
    pclose(fp);

    /*
     * Print the virtual table and the virtual function name
     * which is obtained by using c++filt to decode the mangled
     * name in the object code.
     */
    printf("vtable for %s:\n", name);
    printf("-------------------\n");
    for (; vp < vpe; vp++) {
        UINT32 t = *vp;
        if (t != 0 && ((t<<1)>>1) == t) {
            printf("%#10x ", t);
            snprintf(s, sizeof(s),
                     "objdump -D ./a.out | grep \'^%08x\' |"
                     "cut -d'<' -f2 | cut -d'>' -f1 | c++filt", *vp);
            fp = popen(s, "r");
            if (fgets(s, sizeof(s), fp) != NULL)
                fputs(s, stdout);
            else
                printf("\n");
            pclose(fp);
        } else
            printf("%10d\n", t);
    }
    printf("\n");
}
```

Note

After I have completed this text I encountered a standard effort to specify the layout of C++ objects among other things, that is Itantium C++ ABI (). That document is more detailed than this one and has more authority.

References

- [Lippman] Stanley Lippman, *Inside the C++ object model*, Addison Wesley, 1996.
- [C++98] ISO C++ Standard, 1998 edition.
- [c++filt] c++filt(1), a g++ name demangling tool.

Linux程序的加载、运行和终止» (/2011/08/20/loading-running-and-termination-of-linux-program.html)

Comments