ANALOG DEVICES **Engineer To Engineer Note EE-112**

Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Class Implementation in Analog C++

Contributed by Graham Andrews,
C/C++ Compiler Developer,
DSP Development Tools Product Line

Overview

A "class" is a type in the C++ programming language. The most simple form of this type is likened directly to a struct type in C. In fact, the keyword "class" can be replaced with the keyword "struct" with no effect on the layout of the internal representation of the object. The difference is that C++ can place restrictions on the accessibility of its class members. A struct that is valid in C is valid in C++, and the same rules of layout and parameter passing exist in both C and C++. Member functions declared within a class have no effect on the layout of a given class object unless the member function is declared virtual. When inheritance, virtual base classes, and virtual functions are present, the layout of C++ classes is more complicated.

A C struct is a valid C++ construct with the same object layout:

```
struct A{
   int x;
   double y;
};
```

If the keyword "struct" is replaced with "class" in C++, then the members `x` and `y` are implicitly defined as private . Access specifiers define the rules for using these members with the '`->`' and '`.`' operators. The keyword "struct" makes these members implicitly public in C++.

In C++, the previous example could be equally written as:

```
class A{
public:    // public access
   int x;
   double y;
};
```

All the examples described in this note show equivalent layouts in C++ and C terms. The examples demonstrate how an object representation in C++ translates to C, so there is a direct correspondence to the generated code.

Inheritance

A class can be derived from one or more classes. The classes from which it is derived are called base classes. An object of a derived class inherits all the members of the base classes. A derived class can have base classes, which themselves are derived, and each class may have more than one base class. The layout of class objects follows a simple pattern. This pattern is illustrated by the following examples, which show the underlying class layout in terms of C.

Example 1. Base & Derived Classes

```
class A{
   int a1;
   double a2;
};
class B:A{
   int b1;
   int b2;
};
```

C layout equivalents:

```
struct A{
   int a1;
   double a2;
};
struct B{
   struct A base_A;
   int b1;
   int b2;
};
```

Example 2. Inheritance Using Classes A & B from Example 1

```
class C{
   char c1[10];
};

class D : B{
   float d1;
   short d2;
};

class E: C ,D {
   unsigned int e1:2;
   unsigned int e2:6;
   unsigned e3:17;
};
```

C layout equivalents:

```
struct A {
   int a1;
   double a2;
};

struct B {
   struct A __b_A;
   int b1;
   int b2;
};

struct C {
   char c1[10];
};

struct D {
   struct B __b_B;
   float d1;
   short d2;
};

struct E {
   struct C base_C;
   struct D base_D;
   unsigned int e1: 2;
   unsigned int e2: 6;
   unsigned int e3: 17;
};
```

Virtual Base Classes

Every class in a hierarchy that specifies a base class to be virtual, shares a single object of that base class within a derived object. In terms of the layout, the compiler adds pointers to the class object layout to point to the instance of the base.

In Example 2, class `B` has class `A` as a base class, and class `C` has class `A` as a base class. If class `D` has both `B` and `C` as base classes, then there would be two class `A` parts of class `D`. However, if class `A` is declared to be a virtual base of class `B` and of class `C`, then only one `A` sub-object is present in a `D` object.

Example 3. Virtual Base Classes

```
class Z{
   int z1;
   int z2;
};

class A: Z{
   int a1;
};

class B : virtual A{
   int b1;
};

class C : virtual A{
   int c1;
};

class D: B, C{
   int d1;
};
```

C layout equivalents:

```
struct Z {
   int z1;
   int z2;
};

struct A {
   struct Z __b_Z;
   int a1;
};

struct B {
   int b1;
   struct A *__p_A;
   /* pointer to A part of a B
      object */
   struct A __v_A;
   /* instance of A part in B
      object */
};
```

```
struct __SO__B {
/* If a B is used as a base class,
   this struct is used instead of
   struct B (since the instance of
   an A is determined by the
   derived class).
*/
   int b1;
   struct A *_p_A;
   /* pointer to A part within the
      derived object */
};

struct C {
   int c1;
   struct A *__p_A;
   /* pointer to A part of a C
      object */
   struct A __v_A;
   /* instance of A part in C
      object */
};

struct __SO__C {
/* If a C is used as a base class,
   this struct is used instead of
   struct C (since the instance is
   determined by the derived class).
*/
   int c1;
   struct A *__p_A;
   /* pointer to A part within the
      derived object */
};

struct D {
   struct __SO__B __b_B;
   struct __SO__C __b_C;
   /* For Object D, note that
      base_B.__p_A and base_C.__p_A
      point at the instance of
      __v_A within the struct D
      declaration.
   */
   int d1;
   struct A __v_A;
};
```

Member functions

Member functions are functions that are declared within the scope of a class declaration, and their bodies maybe defined within or outside this declaration. Member functions that are not declared static have a hidden first parameter called the '`this`' parameter. The 'this' parameter is declared as a constant pointer to the class for which it is declared.

Example 4. Non-Static Member Functions

```
class X{
   int a;
   int func(int);
};

X obj;

obj.func(5);
```

This example translates in C terms as follows:

```
struct X{
   int a;
};

int X_func(struct X*, int);

struct X obj;

func(&obj, 5);
```

A static member function is an externally visible function that does not have a hidden '`this`' pointer. This function is general for all objects of this class and is able to access static data members (in most cases). Static data members are data members of a class declared with the '`static`' storage class specifier. These members do not form part of the class object; they may be used for counting objects of a class. Like any other external variable in a C++ program, static data members are externally visible objects and are defined once.

Example 5. Static Member Functions

```
class X{
   int a;
   static int count;
   static void counter();
};

X obj;

int  X::count = 0;
/* define and initialize static
   data */
```

```
X::counter();
```

```
obj.counter();
/* same as line above */
```

This example translates to C equivalent:

```
struct X{
   int a;
};

struct X obj;

int X_count = 0;

void X_counter();

X_counter();
X_counter();
```

Virtual functions

A virtual function does have an effect on the object layout. A virtual function is a non-static member function that is declared with the specifier keyword 'virtual'. The calling mechanism allows invocation of a member function declared in a derived class even if its access is denied through a pointer or a reference to a base class. The mechanism works as an additional pointer placed in the object to a virtual table. The virtual table contains the address of the function to be invoked and an associated index to modify the object pointer to point to the whole-derived object.

```
class B{
public:
   void func(int);
   virtual void func(double);
};

class D: public B{
public:
   void func(int);
   void func(double);
   /* this func is a virtual
      function inherited from
      class B */
};

B b_obj;
D d_obj;

B *p_b1 = &b_obj;
B *p_b2 = &d_obj;
/* points to the B part of a D
   object d_obj */

p_b1->func(5);
/* calls B::func(int) as
   B_func(p_b1,5); not virtual */

p_b1->func(2.0)
/* calls B::func(double).
   Since it is a virtual function
   call, the compiler has to index
   the object pointed at by
   p->b1 and to look up the virtual
   table. This consists of a
   modifying offset for the 'this'
   parameter to point to the
   correct part of the object along
   with the address of the
   appropriate function:
   *(p->b1+tab[offset]_vfaddr)
   (p->b1+tab[offset]_thismod,2.0);
*/

p->b2->func(5);
/* calls B::func(int) as
   B_func(p_b2,5); not virtual */

p->b2->func(2.0)
/* calls D::func(double).
   This works using the same
   methodology described for
   p_b1->func(2.0), but the table
   placed in the B part is
   different as it was created as
   part of a D object.
 */
```

Example 6. Virtual Functions

```
class B1{
public:
   int b1;
   void func(int);
   virtual void func(double);
};

class B2{
public:
   int b2;
   virtual void func(char *);
};
```

```
class D: public B1, public B2{
public:
   int d1;
   void func(int);
   void func(double);
   /* this func is a virtual
      function inherited from
      class B1 */
   void func(char *);
   /* this func is a virtual
      function inherited from
      class B2 */
};

B1 b1_obj;
B2 b2_obj;
D d_obj;

B1 *p_b1 = &b1_obj;
B1 *p_b1d = &d_obj;
/* points to the B1 part of a D
   object d_obj */
B2 *p_b2 = &b2_obj;
B2 *p_b2d = &d_obj;
/* points to the B2 part of a D
   object d_obj */

void B1::func(double){};
void D::func(double){};
void B2::func(char *){};
void D::func(char *){};
```

The generated code in terms of C:

```
struct _VTABLE {
   short mod;
   /* modifies the this parameter
      by adding an offset */
   short unused;
   /* unused hangover from cfront
      and compatibility */
   void (*f)();
   /* address of virtual function
      to be called */
};

struct B1 {
   int b1;
   struct _VTABLE *__vptr;
};

struct B2 {
   int b2;
   struct _VTABLE *__vptr;
};

struct D {
   struct B1 __b_B1;
   struct B2 __b_B2;
   int d1;
};

extern void func__2B1Fd(struct B1
*const, double);

static struct B1 *__ct__2B1Fv
(struct B1 *const);

extern void func__2B2FPc(struct B2
*const, char *);

static struct B2 *__ct__2B2Fv
(struct B2 *const);

extern void func__1DFd(struct D
*const, double);

extern void func__1DFPc(struct D
*const, char *);

struct _VTABLE __vtbl__2B1[2] =
{{((short)0),((short)0),((void
(*)())0)},
{((short)0),((short)0),((void
(*)())func__2B1Fd)}};

struct _VTABLE __vtbl__2B2[2] =
{{((short)0),((short)0),((void
(*)())0)},
{((short)0),((short)0),((void
(*)())func__2B2FPc)}};

struct _VTABLE __vtbl__1D[3] =
{{((short)0),((short)0),((void(*)()
)0)},
{((short)0),((short)0),((void(*)())
func__1DFd)},
{((short)0),((short)0),((void
(*)())func__1DFPc)}};

struct _VTABLE __vtbl__2B2__1D[2] =
{{((short)0),((short)0),((void
(*)())0)},
{((short)(-8)),((short)0),((void
(*)())func__1DFPc)}};
```

In the previous example, an object of type B1 has its _vptr member initialized to the address of __vtbl__2B1[0]. An object of type B2 has its __vptr initialized to &__vtbl_2B2[0]. An object of type D is more complicated and has a virtual table for its B1 and B2 parts. An object of type D requires a virtual table pointer for itself as a D, for its B1 and B2 parts. In fact, D shares the virtual table with B1 by pointing its __vptr to &__vtbl__1D[0], and it sets the __vptr for its B2 part to &__vtbl__2B2__1D[0].