From C function pointers to C++ objects

Hayo Thielecke
University of Birmingham
www.cs.bham.ac.uk/~hxt

July 2, 2014

Abstract

This note explains how to program objects and virtual functions in C (not C++) by building them by hand from structures and function pointers. This serves two purposes: the C++ object model is explained by being reduced to more primitive constructs, and an advanced example of function pointers is presented.

Contents

1 Introduction
2 Dynamic dispatch in C++ (and Java)
3 Building objects from structures and function pointers in C
4 The same classes written in C++
5 C++ objects and language design
6 Conclusions

1 Introduction

These notes are intended for readers who are familiar with basic procedural programming in a C-like syntax (such as Java). They were written with the students on my second-year C/C++ course in mind, who have been taught Java in their first year. See http://www.cs.bham.ac.uk/~hxt/2014/c-plus-plus/index.shtml

There are many conflicting characterizations of what “object-oriented” means. In fact, the C++ standard [C++] Section 1.8] has a rather minimalist notion of object:

An object is a region of storage.

And according to some, Lisp is object-oriented whereas C++ is not. But the latter is what we are interested in here.
Here we will assume that polymorphism, as given by virtual functions in C++, is the defining feature of objects. For our present purposes, we will use the following definition of the word *object* in the context of C++:

An object is a structure in memory that contains at a known position a pointer to another structure, called a vtbl, short for virtual function table. The members of the vtbl are function pointers. All the functions that the vtbl members point at expect as their first parameter a pointer to the object.

This definition is close to the view a C++ compiler has of objects, and one that should be familiar to C++ programmers. It is very helpful to draw diagrams for the vptr and vtbl; see Stroustrup’s overview of the foundations of C++ [Str12].

An object-oriented function call (where p points to some object)

\[
p->f(a)
\]

is a shorthand for a plain C function call in which the pointer p is passed as an addition argument:

\[
p->f(p, a)
\]

The additional parameter is accessible as *this* in C++ and Java. If you know Java, you are of course familiar with using *this*; but it may be surprising to think of it as an invisible parameter to all methods that is silently added by the compiler behind the scenes, at it were.

2 Dynamic dispatch in C++ (and Java)

We briefly recall dynamic dispatch\(^2\). Anyone who has been taught object-oriented programming has probably seen one of the stock examples of method overriding, more or less like the following:

```
class animal
{
  virtual void speak() { printf("Grunt."); }
};
class weasel : public animal
{
  void speak() { printf("I M Weasel!"};
};
class baboon : public animal
{
  void speak() { printf("I R Baboon!"};
};
```

Listing 1: Dynamic dispatch example

\(^2\)Sometimes called dynamic “binding”, but that term should be avoided as it clashes with dynamic vs static binding in functional programming.
The idea is that all animals can grunt (as defined in the base class), but for both weasels and baboons this default behaviour can be overridden with a more specific one (in the derived classes).

A little more technically, the code that runs when a method is called is determined at run time, as opposed to being hardwired at compile time.

We will use the following example. We have a grammar for expressions like $1 + 2 + 3$.

$$E ::= n \mid E + E$$

We want to be able to evaluate such expressions to an integer and to print them out as text, using functions $\text{print}$ and $\text{eval}$. The grammar and the functions have been simplified as much as possible. Nonetheless, the example scales up, and a whole compiler could be built this way.

In object-oriented programming, there is a standard solution for the above, called the Composite Pattern [GHJV95]. In our example, the pattern amounts to the following ingredients:

- A class (or interface) for expressions.
- Two derived classes, one for constants and one for plus.
- Functions in the base class for $\text{print}$ and $\text{eval}$.
- Overriding of the functions in the derived classes so as to implement printing and evaluating for constants and plus expressions.

The Composite Pattern is straightforward to implement in C++ or Java. It works due to dynamic dispatch.

## 3 Building objects from structures and function pointers in C

The code is available here all in one file:

http://www.cs.bham.ac.uk/~hxt/2014/c-plus-plus/objects-in-C.c

Here is the declaration for the vtbl, containing two function pointers:

```c
struct vtbl
{
    void (*print)();
    int (*eval)();
};
```

So far, this only declares the type of the vtbl. To create actual virtual function tables in memory, we will need functions for the function pointers to point at. Note that we only give the return types of the functions, and not the types of their parameters, e.g. the empty () in int (*eval)(). The C type system is more permissive in this regard than the C++ one.

The base class for expressions contains nothing other than the vtbl pointer:
The derived class for expressions contains the vtbl pointer and in addition an integer:

```c
struct Constant
{
    struct vtbl *vptr;
    int n;
};
```

Here the vtbl pointer is in the same position in memory as it is in the base class declaration; the fact that it is followed by more data (int n;) does not affect accessing the pointer and following it. We have not declared any relationship between the two structures, but we can nonetheless use one in place of the other based on how the C compiler is guaranteed to lay them out in memory. This situation is sometimes called \textit{physical subtyping}.

The print function for constant expressions just prints the integer:

```c
void printConstant (struct Constant *p)
{
    printf("%d", p->n);
}
```

The eval function for constant expressions returns the integer:

```c
int evalConstant (struct Constant *p)
{
    return p->n;
}
```

Now that we have the two functions, we enter them into a vtbl structure, which is a global variable:

```c
struct vtbl vtblConstant =
{
    &printConstant, 
    &evalConstant
};
```

In our setting, a constructor is a function that allocates an object and initialized it. Note that the vtbl pointer is set to point at the vtbl constructed above.

```c
void *makeConstantOO (int n)
{
    struct Constant *p;

    p = malloc(sizeof(struct Constant));
    if(p == NULL) exit(1);
    p->n = n;
```
Now for the plus expressions. An object contains the vtbl pointer, and in addition two pointer to the left and right subexpression.

```c
struct Plus
{
    struct vtbl *vptr;
    struct Expression00 *left;
    struct Expression00 *right;
};
```

The print function for plus prints the left and right subexpressions, with a + sign between them. Note in particular how the pointer chain (using the operator ->) makes explicit how the virtual function table is accessed. In Java, where pointers are not visible, one would write just thisp.left.print(). By contrast, here we need to spell out following the vptr and providing the additional argument thisp->left to the function call.

```c
thisp->left->vptr->print(thisp->left)
```

The eval function:

```c
int evalPlus(struct Plus *thisp)
{
    return thisp->left->vptr->eval(thisp->left)
        + thisp->right->vptr->eval(thisp->right);
}
```

Setting up the table of virtual functions for plus expressions in a global variable:

```c
struct vtbl vtblPlus =
{
    &printPlus,
    &evalPlus
};
```

Constructor for plus expressions:

```c
void *makePlusOO(struct Expression00 *left,
    struct Expression00 *right)
{
    struct Plus *p;
```
p = malloc(sizeof(struct Plus));
if(p == NULL) exit(1);
p->vptr = &vtblPlus;
p->left = left;
p->right = right;
return p;
}

Here is a main function for testing:

```c
void ExpressionOOTest()
{

    p1 = makeConstantOO(1);
p2 = makeConstantOO(2);
p3 = makeConstantOO(3);
p4 = makeConstantOO(4);

    p5 = makePlusOO(p1, p2);
p6 = makePlusOO(p3, p4);

    p7 = makePlusOO(p5, p6);
    printf("OO version of expression evaluation\n\n");
    printf("\nTesting print 1 + 2 + 3 + 4\n");
    p7->vptr->print(p7);
    printf("\nTesting eval 1 + 2 + 3 + 4\n");
    printf("Result = %d\n", p7->vptr->eval(p7));
}
int main(int argc, char * argv[])
{
    ExpressionOOTest();
    return 0;
}
```

**Exercise 3.1 (Easy)** Draw the heap with all the structures and pointers, and the virtual function table in particular.

**Exercise 3.2 (Easy)** Trace how the code performs the evaluation of 3+4 by following all the pointers.

**Exercise 3.3 (Easy)** Extend the code to handle times (the product of two expressions), analogously to plus.
Exercise 3.4 (Easy) Translate Listing to C pointers and structures.

Exercise 3.5 (Hard) Explain how inheritance could be supported. (Hint: use physical subtyping in the layout of objects and virtual function tables.)

4 The same classes written in C++

In C++, unlike C, objects and classes are supported by the language. The virtual function table is still there, but so to speak behind the scenes, automatically created by the compiler. Moreover, the compiler only creates it if it is required due to the use if virtual functions. Hence there is no overhead in structures or classes that consist exclusively of “plain old data”, as it is sometimes called, or in classes that are just collections of (non-virtual) functions.

The code is available here all in one file:


The base class provides the interface to the functions:

```cpp
class Expression
{
public :
    virtual int eval() = 0;
    virtual void print() = 0;
};
```

The derived class for constants overrides the pure virtual functions from the base class:

```cpp
class Constant : public Expression
{
private :
    int n;

public :
    int eval();
    void print();
    Constant(int);
};

Constant::Constant(int n)
{
    this->n = n;
}

int Constant::eval()
{
    return this->n;
}

void Constant::print()
{
    std::cout << this->n;
}
```
The derived class for plus overrides the pure virtual functions from the base class:

```cpp
class Plus : public Expression
{
private:
    Expression* left;
    Expression* right;
public:
    Plus(Expression *, Expression *);
    int eval();
    void print();
};

int Plus::eval()
{
    return left->eval()
        + right->eval();
}

void Plus::print()
{
    this->left->print();
    printf(" + ");
    this->right->print();
}

Plus::Plus(Expression *left, Expression *right)
{
    this->left = left;
    this->right = right;
}
```

Most of the time, we need not worry too much about the vptr and vtbl that the C++ compiler automagically manages for us. If efficiency is critical, however, the implementation of objects becomes important.

**Exercise 4.1 (Hard)** Find examples of type errors that the C++ compiler would find if they were made in the code above, but which would not be found by the C compiler for the code in Section 3 (Hint: try applying functions to wrong arguments.)

**Exercise 4.2 (Easy)** Using your C++ compiler of choice, experiment with structures, especially those containing function pointers or virtual functions, by computing their size. Then explain the sizes based on what the compiler needs to generate for each structure. For example, in the following code, you should get sizes such that

```
sizeof(int) = sizeof(s1) ≤ sizeof(s2) ≤ sizeof(s3) < sizeof(s4)
```

Why is \( \text{sizeof(s3)} \) smaller than \( \text{sizeof(s4)} \) despite the fact that its declaration is longer?
struct s1 {
    int n;
};

struct s2 {
    int *p;
};

struct s3 {
    virtual int f();
    virtual int g();
    virtual int h();
};

struct s4 {
    int (*f)();
    int (*g)();
};

int main (int argc, const char * argv[])
{
    printf("%lu\n", sizeof(s1));
    printf("%lu\n", sizeof(s2));
    printf("%lu\n", sizeof(s3));
    printf("%lu\n", sizeof(s4));
}
type errors and the trade-off between flexibility and efficiency. For details, see the extensive documentation on Apple’s developer pages [App12].

**Exercise 5.1 (Optional)** Rewrite the code in Listing 1 in Objective C, without using a common base class.

6 Conclusions

We have reduced the C++ object model, in a simple example, to more fundamental C constructs. Writing objects in C by hand is not something one would typically do in practice. C++ compilers do so much more conveniently and with more accurate type checking. On the other hand, it shows how far we can push C and its type system. Furthermore, we can see that programming with objects is by no means restricted to “object-oriented” languages (Convenience is a different matter.) The use of pointers to structures that may contain pointers to further structures (and so on) is very typical of C programming: see for example the data structures for file or process descriptors in operating systems [BC00]. In operating systems, interrupts and signals rely on function pointers to interrupt or signal handlers.

Further reading

Somewhat confusingly, it is sometimes claimed that C++ objects are instances of abstract data types, or that classes “are” types. For a discussion of what object-oriented programming means, see Cook’s paper on objects versus abstract data types [Coo09]. For more details on the C++ object implementation, see [Lip96] [RDRL11]. For attacks on software that corrupt C++ objects and pointers, see [Sea05].

References


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, Massachusetts, 1995.


