From C function pointers to object-oriented programming

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

February 2015
Objects and C

- C gives us primitive building blocks
- struct, pointers, functions
- What we do with them is up to us
- How far can we push C?
- How about objects? Or something reasonably close?
- We will assume: virtual functions as fundamental for OO
- Early C++ was a preprocessor for C
- Advanced example of pointers in C
- Some idea of how C++ is implemented
The big picture: building objects in C

C++ compiler → C++ → C

C compiler → C → machine code

C++ build and use vtables manually
In C++ we can write:

```cpp
class inCPP {
  int x;
public:
  int get() { return this->x; }
};
```

In C we can write:

```c
struct inC {
  int y;
  int (*cget)(struct inC *thisp);
}

int cf(struct inC *thisp) { return thisp->y; }
```
Simple objects simulated in C

In C++ we can write:

```cpp
class inCPP {
    int x;
public:
    int get() { return this->x; }
};
```

In C we can write:

```c
struct inC {
    int y;
    int (*cget)(struct inC *thisp);
};
```

```c
int cf(struct inC *thisp) { return thisp->y; }
```
In class-based OO languages (like C++), objects can share their member functions in a virtual function table, one per class

```c
struct vtbl {
    void (*f1)(); // member functions
    int (*f2)();
    ...
};
```

```c
struct s {
    struct vtbl *vptr; // pointer to shared vtbl
    int x; // data members
};
```
Physical subtyping in C example

```c
struct s1 {
    struct s1 *p;
    int x;
};

struct s2 {
    struct s2 *q;
    int y;
    struct s2 *q2;
};
```

Code that works on s1 can also work on s2. In that sense, s2 is a physical subtype of s1. A limited form of polymorphism in C due to structure layout.
In C++ we write a virtual function call as

```cpp
left->print();
```

Simulated in C, this becomes:

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

Give each function access to object via “self” or “this” pointer

Call virtual function indirectly through virtual function table
Example class in C++

Canonical example of OO:
parse trees for expressions
virtual functions for processing trees

```cpp
class Expression {
public:
    virtual int eval() = 0;
    virtual void print() = 0;
};
```
Virtual function table in C: types

structure + pointer + function:

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

Base class has pointer to vtbl:

```c
struct Expression00
{
    struct vtbl *vptr;
};
```
Derived class via physical subtyping

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

In memory:

Expression00: Constant:

<table>
<thead>
<tr>
<th>vptr</th>
<th>vptr</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
</tr>
</tbody>
</table>

Position of vptr is the same.
Virtual member functions populate the vtable

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

int evalConstant(struct Constant *thisp)
{
    return thisp->n;
}

Global variable for vtable, containing function pointers

struct vtbl vtblConstan =
{
    &printConstant,
    &evalConstant
};
```
Constructor

malloc and initialize, including vptr

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

    p = malloc(sizeof(struct Constant));
    if(p == NULL) exit(1);
    p->n = n;
    p->vptr = &vtblConstant;
    return p;
}
```
Another derived class, for plus

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

In memory:

Expression00:                        Plus:

    vptr                        vptr
        left
        right
```
Virtual member functions

```c
void printPlus(struct Plus *thisp) {
    thisp->left->vptr->print(thisp->left);
    printf(" + ");
    thisp->right->vptr->print(thisp->right);
}
```

The eval function:

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

```c
struct vtbl vtblPlus =
{
    &printPlus,
    &evalPlus
};
```
void *makePlusOO(struct ExpressionOO *left,  
               struct ExpressionOO *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;
}
Using it

```c

p1 = makeConstant00(1);
p2 = makeConstant00(2);
p3 = makeConstant00(3);
p4 = makeConstant00(4);

p5 = makePlus00(p1, p2);
p6 = makePlus00(p3, p4);

p7 = makePlus00(p5, p6);

printf("\nTesting print 1 + 2 + 3 + 4\n");
p7->vptr->print(p7);
```
In C++ we write a virtual function call as

```c++
left->print();
```

Simulated in C, this becomes:

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

Give each function access to object via “self” or “this” pointer
Call virtual function indirectly through virtual function table
How big are objects in C++

class A {
    void fA() { }
    int *a;
};

class B {
    virtual void fB() {}
    int *b;
};

class C {
    virtual void fC1() {}
    virtual void fC2() {}
    int *c;
};

sizeof(A) = 8, sizeof(B) = 16, sizeof(C) = 16 on typical compiler
How big are objects in C++

class A {
    void fA() { }
    int *a;
};

class B {
    virtual void fB() {}
    int *b;
};

class C {
    virtual void fC1() {}
    virtual void fC2() {}
    int *c;
};

sizeof(A) = 8, sizeof(B) = 16, sizeof(C) = 16 on typical compiler
Inheritance puzzle

class base {
public:
    int x = 1;
    virtual int f() { return x + g(); }
    virtual int g() { return 10; }
};

class derived : public base {
public:
    int x = 100;
    virtual int g() { return x; }
};

What is (new derived())->f()
class base {
public:
    int x = 1;
    virtual int f() { return x + g(); }
    virtual int g() { return 10; }
};

class derived : public base {
public:
    int x = 100;
    virtual int g() { return x; }
};

What is (new derived())->f()
101
Functions use indirection via vtable, whereas variables do not

From the book “Essentials of Programming Languages”, by Wand, Friedman, Haynes, 2nd edition
Conclusions on C++ → C

- C is simple, powerful and flexible
- pointers
- control over memory
- physical subtyping
- function pointers
- static type checking, up to a point
- C type system is not a straightjacket
- C++ objects can be built on top of C quite easily
- Objects become clearer if you know how they are implemented
- Translations (like compiling) are a fundamental technique in programming languages