From C structures and function pointers to object-oriented programming

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

March 24, 2016

A puzzle about virtual functions and data members

Translating C++ OO into C

Physical subtyping in C

Objects simulated in C

Virtual function tables and pointers

A taste of compiling

A puzzle about virtual functions and data members

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

class derived : public base {
    int x = 200;
public:
    int g() { return x; }
};
```

What is `(new derived())->f()`

A puzzle about virtual functions and data members

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

class derived : public base {
    int x = 200;
public:
    int g() { return x; }
};
```

What is `(new derived())->f()`

201

Functions use indirection via vtable, whereas variables do not

From the book “Essentials of Programming Languages”, by Wand, Friedman, Haynes, 2nd edition

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

$$
\begin{tikzcd}
C{+}{+} \arrow[rr, "\text{build and use vtables manually}"] \arrow[dr, "\text{C++ compiler}"'] & & C \arrow[dl, "\text{C compiler}"] \\
& \text{machine code} &
\end{tikzcd}
$$

Extending structs

```
struct SinglyLinked {
  struct SinglyLinked next;
  int data;
};

struct DoublyLinked {
  struct DoublyLinked *n;
  int data;
  struct DoublyLinked *p;
};
```

Can we use DoublyLinked where SinglyLinked is expected?
Suppose we have a sum function. What about:

```
struct DoublyLinked {
  int data;
  struct DoublyLinked *next;
  struct DoublyLinked *prev;
};
```

Subtyping

T1 is a subtype of T2 if T1 can be used in any place where a T2 is expected.

Class-based languages like C++ and Java use subtyping

If class C1 extends/publicly inherits from class C2, the object created from C1 can be used wherever those created from C2 are expected.

This always works in Java, but not in C++

C has a form of subtyping of structures

S1 is a physical subtype of S2 if the sequence of types of members in S2 is a prefix of that in S1

S1 is like S2, but may add more at the end

Physical subtyping in C example

```
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

Physical subtyping of struct and flexible array members

```
struct pst1 { // OK because array is allocated at the end
    int x;
    char a[];
};

struct pst2 { // OK because array is of fixed size
    char a[10];
    int x;
};

struct pst3 { // not OK
    char a[]; // incomplete type
    int x;
};

struct pst4 { // OK because array is outside the struct
    char *p;
    int x;
};
```

Simple objects simulated in C

In C++ we can write:

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

Simple objects simulated in C

In C++ we can write:

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

In C we can write:

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

int cf(struct inC *thisp) { return thisp->y; }
```

Classes simulated in C

In class-based OO languages (like C++), objects can share their member functions in a virtual function table, one per class

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

struct s {
   struct vtbl *vptr;  // pointer to shared vtbl
   int x;              // data members
};
```

OO in C: two key pointers

In C++ we write a virtual function call as

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

Simulated in C, this becomes:

```
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 virtual functions:
abstract syntax trees for expressions
virtual functions for processing trees

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

Virtual function table in C: types

structure + pointer + function:

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

Base class has pointer to vtbl:

```
struct ExpressionOO
{
    struct vtbl *vptr;
};
```

Derived class via physical subtyping

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

In memory:

```
ExpressionOO:       Constant:

vptr                vptr
                    n
```

Position of `vptr` in memory is the same

Virtual member functions populate the vtable

```
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 vtblConstant =
{
    &printConstant,
    &evalConstant
};
```

Constructor

malloc and intialize, including vptr

```
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

```
struct Plus
{
    struct vtbl *vptr;
    struct ExpressionOO *left;
    struct ExpressionOO *right;
};
```

In memory:

```
ExpressionOO:          Plus:

vptr                   vptr
                       left
                       right
```

Virtual member functions

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

The eval function:

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

Virtual function table for plus

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

Constructor for plus

```
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

```
struct ExpressionOO *p1, *p2, *p3, *p4, *p5, *p6, *p7;

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("\nTesting print 1 + 2 + 3 + 4\n");

p7->vptr->print(p7);
```

OO in C: two key pointers

In C++ we write a virtual function call as

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

Simulated in C, this becomes:

```
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;
};
```

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

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

Progression: position of this module in the curriculum

First year	Software Workshop, functional programming, Language and Logic
Second year	C/C++ $\leftrightarrow$ Comp Sys Arch, Intro Comp Sec
Final year	Operating systems, compilers, parallel programming

Vector add in CUDA

```
__global__
void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

http://docs.nvidia.com/cuda/cuda-c-programming-guide/

Clang stack frame example

```
long f(long x, long y) // put y at -8 and x at -16
{
    long a;    // put a at -24
    long b;    // put b at -32
    ...
}
```

return addr	
old bp	← base pointer
x	← bp - 8
y	← bp - 16
a	← bp - 24
b	← bp - 32

Compiled with clang -S

```
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

$$\begin{array}{lcl} \mathtt{x} & \mapsto & \mathtt{rdi} \\ \mathtt{y} & \mapsto & \mathtt{rsi} \\ \mathtt{x} & \mapsto & \mathtt{rbp} - 8 \\ \mathtt{y} & \mapsto & \mathtt{rbp} - 16 \\ \mathtt{a} & \mapsto & \mathtt{rbp} - 24 \\ \mathtt{b} & \mapsto & \mathtt{rbp} - 32 \end{array}$$

```
f:
pushq %rbp
movq %rsp, %rbp
movq %rdi, -8(%rbp)
movq %rsi, -16(%rbp)
movq -8(%rbp), %rsi
addq $42, %rsi
movq %rsi, -24(%rbp)
movq -16(%rbp), %rsi
addq $23, %rsi
movq %rsi, -32(%rbp)
movq -24(%rbp), %rsi
imulq -32(%rbp), %rsi
movq %rsi, %rax
popq %rbp
ret
```

Optimization: compiled with clang -S -O3

```c
long f(long x, long y)
{
  long a, b;
  a = x + 42;
  b = y + 23;
  return a * b;
}
```

```
f:
addq $42, %rdi
leaq 23(%rsi), %rax
imulq %rdi, %rax
ret
```