From C and Java to C++

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

February 25, 2016

Design and evolution of C++

C++ vs C and Java

Object-orientation in Java and C++

Virtual functions

Object-oriented trees and tree walking

Memory management in C++ with constructors and destructors

Stack and heap allocation of objects in C++

The C programming language

- Designed by Dennis Ritchie at Bell Labs
- based on earlier B by Ken Thompson
- Evolution: CPL $\rightarrow$ BCPL $\rightarrow$ B $\rightarrow$ C $\rightarrow$ C++ $\rightarrow$ Java
- C is typical of late 1960s/early 1970 language design (compare Pascal or Algol-W)
- C is minimalistic, much is reduced to pointers
- C is above all a systems programming language
- Other systems programming languages are largely forgotten
- C took over the world by accident, due to Unix
- Easy to implement, not easy to use
- Never intended for beginners
- Aimed a users who write their own compiler and operating system

The C++ programming language

- Designed by Bjarne Stroustrup and then committees
- C++ aims: as efficient as C, but more structure
- both high-level and low-level
- No Garbage Collection, unlike Java, OCaml, Haskell, Javascript
- C (is essentially) a subset of C++
- C is NOT a subset of Java, not even close
- C++ is the most complicated major language
- C++ has object-orientation but does not force you to use it
- C++ keeps evolving, e.g. templates, lambda

C++ has its critics

I made up the term "object-oriented", and I can tell you I did not have C++ in mind. — Alan Kay [1]

It does a lot of things half well and its just a garbage heap of ideas that are mutually exclusive. — Ken Thompson

Inside C++ is a smaller, cleaner, and even more powerful language struggling to get out. And no, that language is not C, C#, D, Haskell, Java, ML, Lisp, Scala, Smalltalk, or whatever.
— Bjarne Stroustrup

Stroustrup's overview of C++:
`http://w.stroustrup.com/ETAPS-corrected-draft.pdf`
Homework: read this paper.

[1] "Don't believe quotations you read on the interwebs; they could be made up." —Alan Turing

Books

Stroustrup	Bjarne Stroustrup: The C++ Programming Language (2013)
Patterns	Gamma, Helm, Johnson, Vlissides: Design patterns: elements of reusable object-oriented software sometime called "Gang of Four"

Some of this module is also informed by programming language research.

C, C++, and Java

C

core imperative language (assignment, while, functions, recursion)
+ malloc and free; no garbage collector
+ pointers combined with other language features

C++

core imperative language (assignment, while, functions, recursion)
+ new and delete; no garbage collector
+ object orientation
+ templates

Java

core imperative language (assignment, while, functions, recursion)
+ simplified version of C++ object-orientation
+ garbage collector $\Rightarrow$ programmer can be naive about memory

From C and Java to C++

- If you know both C and Java, then C++ is not that hard to learn
- Certainly easier than if you know only one of C or Java
- Most recent parts of C++ are close to functional languages: templates and lambda
- But C++ is still are large and messy language that keeps evolving; many overlapping and deprecated features
- C++ is hampered by the need for backwards compatibility with C and older versions of itself
- C++ is trying to be many things at once
- A modern language designed from scratch could be much cleaner; but there is no serious contender at the moment

Object-orientation in Java

```
class Animal { // superclass
public void speak()
    {
        System.out.println("Generic animal noise");
    }
}

class Pig extends Animal {  // subclass
public void speak()
    {
        System.out.println("Oink!"); // override
    }
}

class Pigtest {
    public static void main(String[] args) {
        Animal peppa = new Pig();
        peppa.speak();
        (new Animal()).speak();
    }
```

Object-orientation in C++

```
class Animal { // base class
public: virtual void speak()
    {
        cout << "Generic animal noise\n";
    }
};

class Pig : public Animal { // derived class
public: void speak()
    {
        cout << "Oink!\n";      // override
    }
};

int main(int argc, char* argv[]) {
    Animal *peppa = new Pig();
    peppa->speak();
    (new Animal())->speak();
}
```

C++ compared to C

- C++ is more modern and high-level than C
- C++ has abstraction mechanisms: OO and templates
- In C++, classes are like structs.
- Fundamental design decision: OO is spatchcocked into C.
- By contrast, in Objective-C, objects are a layer on top of C separate from struct.
- Arguably, Objective-C is more object-oriented than C++ and Java
- C is a simple language, C++ is extremely complicated

C++ compared to Java

- Java does not contain C, C++ does[2]
- C++ is more fine-grained than Java.
- Java `.` vs C++ `.`, `->`,`::`
- Java inheritance: methods = `virtual` functions
  and `public` inheritance, not implementation-only interitance
- Java `new` is garbage-collected
- C++ `new` is like a typed `malloc`, must use `delete`
- Constructors and destructors in C++

[2]modulo minor tweaks

We will only use a Java-like subset of C++ OO system

- Only single inheritance
- No multiple or private inheritance
- If you don't understand some part of C++, don't use it
- We use a subset similar to Java type system
- Even so: need memory management: destructors and `delete`
- If you want to see more C++ features, read Stroustrup's The C++ Programming language, which covers the whole language (1368 pages)

Virtual functions

non-virtual	the compiler determines from the type what function to call at compile time
virtual	what function to call is determined at run-time from the object and its run-time class

Stroustrup's overview of C++:

`http://w.stroustrup.com/ETAPS-corrected-draft.pdf`

> *Only when we add virtual functions (C++s variant of run-time dispatch supplying run-time polymorphism), do we need to add supporting data structures, and those are just tables of functions.*

At run-time, each object of a class with virtual functions contains an additional pointer to the virtual function table (vtable).

Non-virtual function (by default)

```
class Animal {
public: void speak() { std::cout << "Noise\n"; } // not virtual
};

class Pig : public Animal {
public: void speak() { std::cout << "Oink!\n"; }
};

int main(int argc, char *argv[]) {
    Animal *peppa = new Pig();
    peppa->speak();
    // the type of peppa is Animal
}
```

Output:

Non-virtual function (by default)

```
class Animal {
public: void speak() { std::cout << "Noise\n"; } // not virtual
};

class Pig : public Animal {
public: void speak() { std::cout << "Oink!\n"; }
};

int main(int argc, char *argv[]) {
    Animal *peppa = new Pig();
    peppa->speak();
    // the type of peppa is Animal
}
```

Output:

```
Noise
```

Virtual function

```
class Animal {
public: virtual void speak() { std::cout << "Noise\n";  }
};

class Pig : public Animal {
public: void speak() { std::cout << "Oink!\n"; }
};

int main(int argc, char *argv[]) {
    Animal *peppa = new Pig();
    peppa->speak();
    // peppa points to an object of class Pig
}
```

Output:

Virtual function

```
class Animal {
public: virtual void speak() { std::cout << "Noise\n";  }
};

class Pig : public Animal {
public: void speak() { std::cout << "Oink!\n"; }
};

int main(int argc, char *argv[]) {
    Animal *peppa = new Pig();
    peppa->speak();
    // peppa points to an object of class Pig
}
```

Output:

```
Oink!
```

Virtual function and compile vs run time

```
class Animal {
public: virtual void speak() { ... }
};

class Baboon : public Animal { ... };
class Weasel : public Animal { ... };

Animal *ap;
if(...) {
    ap = new Baboon();
else
    ap = new Weasel();
ap->speak();
```

The compiler knows the type of ap.
Whether it points to a Baboon or Weasel is known only when the code runs.

Strings in C++

- Recall that C uses 0-terminated character arrays as strings.
- C strings are full of pitfalls, like buffer overflow and off-by-one bugs
- C++ has a dedicated string class
- See Stroustrup section 4.2 and Chapter 36
- For looking up C++ libraries, this looks like a good site:
- `http://www.cplusplus.com/`
- `http://www.cplusplus.com/reference/string/string/`

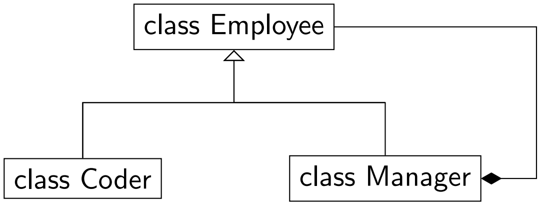
COMPOSITE pattern and OO trees

Composite pattern as defined in Gamma et. al.:
"*Compose objects into tree structures to represent part-whole hierarchies.*"
(From Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.)
Examples include

- managers and underlings
- abstract syntax trees

Composite pattern example in UML notation

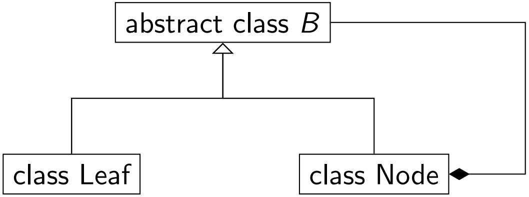
Manager "is-a" Employee
Manager "has-a" Employee



Composite pattern and grammars

Consider the binary tree grammar:

$$B \to B\ B \mid 1 \mid 2 \mid \ldots$$



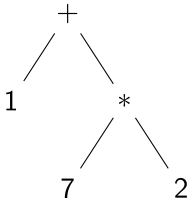
Tree or "hierarchy" as C++ type recursion

```
class Manager : public Employee {
    Employee **underlings;
    int numunderlings;
public: void downsize();
};
```

I CAN HAZ UNDERLINGZ

Evaluation function as abstract syntax tree walk

If you don't think ASTs are interesting then what are you doing in CS, friendo.



- each of the nodes is a struct with pointers to the child nodes (if any)
- recursive calls on subtrees
- combine result of recursive calls depending on node type, such as $+$

AST for expressions in C ✓

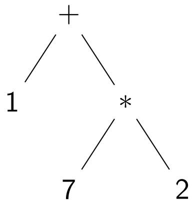
$$\begin{aligned} E &\rightarrow n \\ E &\rightarrow E - E \\ E &\rightarrow E * E \end{aligned}$$

```
enum Etag {
    constant, minus, times
};

struct E {
    enum Etag tag;
    union {
        int constant;
        struct {
            struct E *e1;
            struct E *e2;
        } minus;
        struct {
            struct E *e1;
            struct E *e2;
        } times;
    } Eunion;
};
```

http://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/ParserTree.c

Evaluation function as abstract syntax tree walk ✓



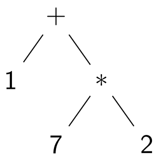
```
switch(p->tag)  ...
```

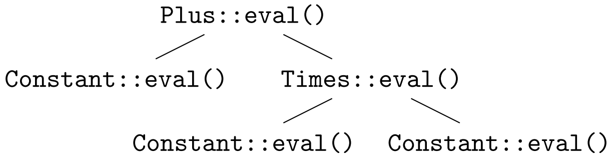
- each of the nodes is (an instance of) a struct with pointers to the child nodes (if any)
- recursive calls on subtrees
- combine result of recursive calls depending on node type, such as $+$

eval as abstract syntax tree walk in C ✓

```
int eval(struct E *p)
{
    switch(p->tag) {
        case constant:
            return p->Eunion.constant;
        case plus:
            return eval(p->Eunion.plus.e1)
                 + eval(p->Eunion.plus.e2);
        case minus:
            return eval(p->Eunion.minus.e1)
                 - eval(p->Eunion.minus.e2);
        case times:
            return eval(p->Eunion.times.e1)
                *  eval(p->Eunion.times.e2);
        default:
            fprintf(stderr, "Invalid tag for struct E.\n\n");
            exit(1);
    }
}
```

Object-oriented abstract syntax tree walk





- each of the nodes is an object (instance of a class) with pointers to the child nodes (if any)
- each node uses the evaluation member function of its class
- a self-walking tree, so to speak 😋
- in OO, data stuctures know what to do, so to speak

Object-oriented expression trees

- base class for expressions
- derived classes for the different kinds of expressions (and not a union as in C)
- type recursion via the base class
- pure virtual member function for the evaluation function
- overriden by each of the derived classes
- recursion inside member functions

Expression tree base class

```
class E {
public:
    virtual int eval() = 0;
};
```

Constant as a derived class

```
class constant : public E {
    int n;
public:
    constant(int n) { this->n = n; }
    int eval();
};
```

Plus expressions as a derived class

```
class plus : public E
    class E *e1;
    class E *e2;
public:
    plus(class E *e1, class E * e2)
    {
        this->e1 = e1;
        this->e2 = e2;
    }

    int eval();
};
```

Plus expression evaluation function

```
class plus : public E {
    class E *e1;
    class E *e2;
public:
    plus(class E *e1, class E * e2)
    {
        this->e1 = e1;
        this->e2 = e2;
    }

    int eval();
};

int plus::eval()
{
    return e1->eval() + e2->eval();
}
```

Two styles of trees in C++

C style trees	Composite pattern trees in C++
tagged union	common base class
members of tagged union	derived classes
switch statements	virtual functions
branches of the switch statement	member functions
easy to add new functions without changing struct	easy to add new derived classes without changing base class

- can we get the best of both worlds?
- both new functions and new cases easily defined later on, without changing existing code?
- that is called the "expression problem" in the research literature
- at least in C++, it does not have a simple solution

Lisp style expressions

```
struct env {
  string var;
  int value;
  env *next;
};

class Exp {
 public:
  virtual int eval(env*) = 0;
};

class Var : public Exp {
  string name;
 public:
  Var(string s) { this->name = s; }
  int eval(env*);
};
```

Lisp style expressions

```
class Let : public Exp {
  string bvar;
  Exp *bexp;
  Exp *body;
 public:
  Let(string v, Exp *e, Exp *b)
    {
      bvar = v; bexp = e; body = b;
    }
  int eval(env*);
};

class ExpList { // plain old data
 public:
  Exp *head;
  ExpList *tail;
  ExpList(Exp *h, ExpList *t) { head = h; tail = t; }
};
```

Lisp style expressions

```
enum op { plusop, timesop };

class OpApp : public Exp {
  op op;
  ExpList *args;
 public:
  OpApp(enum op o, ExpList *a) { op = o; args = a; }
  int eval(env*);
};

class Constant : public Exp {
  int n;
 public:
  Constant(int n) {this->n = n; }
  int eval(env*);
};
```

Lisp style expressions

The evaluator is a collection of member functiosn, one per derived class:

```
int Constant::eval(env *p)
{
    // implement me
}
int Var::eval(env *p)
{
    // implement me
}
...
```

`new` and `delete` in C++

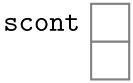
- C has `malloc` and `free` for heap allocation and deallocation
- Java has objects and `new` (but also garbage collection)
- C++ has `new` and `delete` for heap allocation and deallocation
- do not mix new/delete and malloc/free
- Constructors and destructors
- Note: destructors, not deconstructors (those are French philosophers).
- also not to be confused with pattern matching in functional languages, sometimes also called destruction as invense of construction

Writing destructors in C++

- C++ destructors for some class `A` are called `~A`, like "not A"
- compare:
  `new` allocated heap memory, constructor initializes
  `delete` deallocates heap memory, constructor cleans up
- do not call desctructors directly, let `delete` do it
- the clean-up in a destructor may involved deleting other objects "owned" by the object to be deleted
- Example: `delete` the root of a tree $\Rightarrow$ recursively deallocate child nodes
  if that is what is approporiate

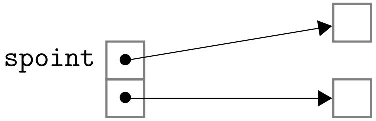
Destructors do not follow pointers

```
struct scont {
  A a;
  B b;
};
```



Deleting an scont object deletes both the contained objects.

```
struct spoint {
  A *ap;
  B *bp;
};
```



Deleting an spoint object does not affect the objects pointed at.
We could write a destructor that calls delete on the pointers.

Destructors for the abstract syntax tree

```
// abstract base class E
class E {
public:
    // pure virtual member function
    // = 0  means no implementation, only type
    virtual int eval() = 0;
    // virtual destructor
    virtual ~E();
};

// base class destructor must be implemented
// because derived classes call it automagically

E::~E(){ }

http://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/
ParserTreeOO.cpp
```

Destructors for the abstract syntax tree

```
constant::~constant()
{
    printf("constant %d deleted\n", n);
    // only if you want to observe the deallocation
}
```

Destructors for the abstract syntax tree

Suppose we want deletion to delete all subtrees recursively.

```
plus::~plus()
{
    printf("deleting a plus node\n");
    // only if you want to observe the deallocation
    delete e1; // deallocate recursively
    delete e2;
}
```

Stack-allocated objects

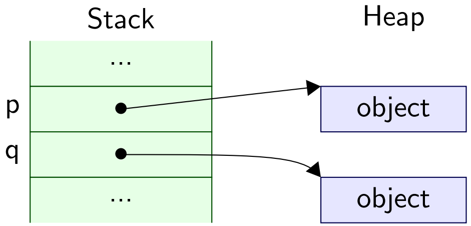
```
class Animal {
public: void speak() {
    std::cout << "Generic animal noise\n";
    }
};

class Pig : public Animal {
public: void speak() {
    std::cout << "Grunt!\n";
    }
};


int main(int argc, char* argv[]) {
    Pig peppa;      // now new, no memory leak
    peppa.speak();
    Animal a;
    a.speak();
}
```

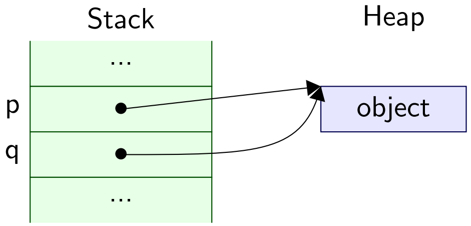
Allocate two different objects

```
void f()
{
  C *p, *q;
  p = new C();
  q = new C();
  ...
}
```



Allocate one object and alias it

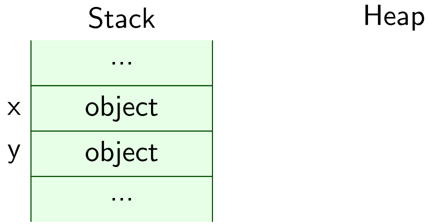
```
void f()
{
  C *p, *q;
  p = new C();
  q = p;
  ...
}
```



Allocate two objects on the stack

Look Ma, no heap. It is not possible to allocate objects on the stack in Java.
Destructors are called automagically at the end of the function.

```
void f()
{
  C x, y; // allocate objects on the stack
  ...
} // destructors called for stack allocated objects
```



More example code for destructors

`http://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/Destructors.cpp`

`http://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/ParserTreeOO.cpp`

Why not do your own experiments with various destructors and see what valgrind reports

Outline of the module (provisional)

I am aiming for these blocks of material:

1. pointers+struct+malloc+free
   $\Rightarrow$ dynamic data structures in C as used in OS ✓
2. pointers+struct+union
   $\Rightarrow$ typed trees in C
   such as abstract syntax trees ✓
3. object-oriented trees in C++
   composite pattern ✓
4. templates in C++
   parametric polymorphism

An assessed exercise for each.