From C and Java to C++

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

March 3, 2017

Design and evolution of C++

From C and Java to learning C++

Object-orientation in Java and C++

Virtual functions

Object-oriented syntax trees and tree walking

Composite object-oriented design pattern

Memory management in C++ with constructors and destructors

Stack and heap allocation of objects in C++

Object oriented programming in C with function pointers

The C programming language

- Designed by Dennis Ritchie at Bell Labs
- based on earlier B by Ken Thompson
- Evolution: CPL → BCPL → B → C → C++ → 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 being free
- Easy to implement, not easy to use
- Never intended for beginners
- Aimed a users who write their own compiler and operating system

Looking back at C part of this module

By now you know what C is like
We have covered the most important constructs; some thing were deliberately omitted because they are obsolete, e.g. register, inline

Why is C useful for systems programming:

- every feature can be compiled efficiently
- fine control over memory layout
  (pointer + struct + union . . .)
- no slow and unpredictable garbage collection
- you can write your own malloc in C
- you can write a garbage collector in C
- you can write an OS in C
- you can write C that can run without an OS or VM
- can do (nearly) as much as in assembly

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
- For background, see Stroustrup's book "Design and evolution of C++"

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 it's 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
such as the expression problem in OO and type theory

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
+ many feature of varying usefulness

Java

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

Position of C and C++ in this module

- The term C/C++ is ambiguous: how much is C and how much ++?
- You really need to learn C: pointers!
- C is a simple and by now ancient language, early 1970s technology based on research from the late 1960s
- C lacks abstraction mechanisms for structuring programs or "programming in the large"
- C++ adds modern abstraction mechanisms like parametric polymorphism
- NOT: look at C++ features for their own sake
- look at important programming constructs and how they are manifested in C++
- layer on top of C, perhaps "C with templates"
- C is the cake, C++ is the icing on the cake

From C and Java to learning 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("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 << "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
- Note: Java has single implementation inheritance, but a class can implement multiple interfaces
  a sensible compromise
- 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) example

```
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) example

```
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 example

```
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 example

```
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.

Stack-allocated objects and virtual functions

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

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

void peppatest()
{
    Pig peppa;       // no new, no memory leak
    peppa.speak();  // Oink!
    Animal *p = &peppa;
    p->speak();      // Oink!
}
```

Virtual vs non-virtual is independent of stack vs heap

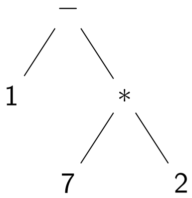
Pure virtual functions

A pure virtual function has no implementation in the base class
Notation:

```
class C {
public:
  virtual T1 f(T2) = 0; // declare f as pure virtual
  ...
};
```

Awful notation: it does not mean anything is equal to 0 ☹
the = 0 is supposed to suggest that something is missing
Derived classes should implement f
Compare: abstract classes, interfaces in Java
For example, we could have said that there is not really any noise that all animals can make

Evaluation function as abstract syntax tree walk



- 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 ✓

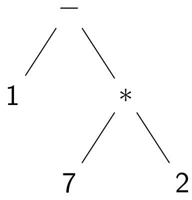
$$E \rightarrow n$$
$$E \rightarrow E - E$$
$$E \rightarrow E * E$$

```
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/2016/c-plus-plus/ParserTree.c`

Evaluation function as abstract syntax tree walk ✓



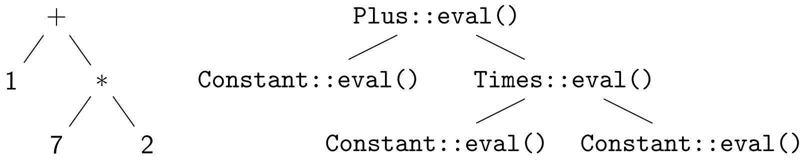
`switch(p->tag) ...`

- 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)
{
    assert(p);
    switch(p->tag) {
        case constant:
            return p->Eunion.constant;
        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 with virtual functions



- 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
- in OO, data stuctures know what to do, so to speak
- a self-walking tree ☺
- no switch statement is needed
- instead: dynamic “polymorphism” via virtual functions

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

$$\begin{array}{rcl} E & \to & n \\ E & \to & E - E \\ E & \to & E * E \end{array}$$

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

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();
}
```

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 functions, one per derived class:

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

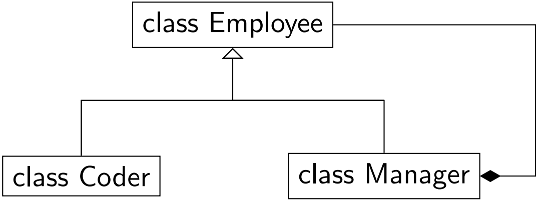
COMPOSITE pattern and OO trees

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

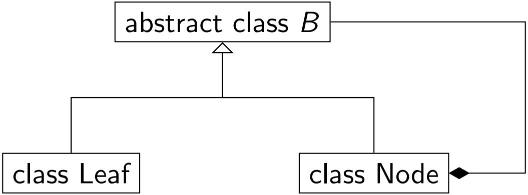
Manager “is-a” Employee
Manager “has-a” Employee



Composite pattern and grammars

Consider the binary tree grammar:

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



abstract class B

class Leaf

class Node

Tree or “hierarchy” as C++ type recursion

```
class Manager : public Employee {
public:
    Employee **underlings;
    int numunderlings;
    long long double bonus = 1234567.89;
    void downsize(); // recursively deallocate underlings
};
```

I CAN HAS-A BONUZ

Object orientation, C/C++ and the expression problem

- Representing abstract syntax trees has generated a lot of research
- sometimes know as “the expression problem” in the literature

The problem:

1. We may wish to add more cases to the grammar, say for a division operator: easy to do in a class hierarchy, hard to do with struct+union
2. We may wish to add more operations to the expression trees, say pretty printing or compilation to machine code
   easy to do with struct and union, hard to do with class hierarchy

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
- some proposals make heavy use of templates

`new` and `delete` in C++

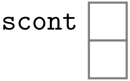
- C has `malloc` and `free` for heap allocation and deallocation
- Java has objects and `new` (but also garbage collection)
- C++ has `new` and `delete` for heap allocation and deallocation
- do not mix new/delete and malloc/free
- Constructors and destructors
- Note: destructors, not deconstructors
  — deconstruction is a buzzword in Arts subjects;
  sounds very strange and pretentious in C++ ☹
- also not to be confused with pattern matching in functional languages, sometimes also called destruction or deconstruction 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, destructor cleans up
- do not call destructors directly, let `delete` do it implicitly
- the clean-up in a destructor may involve deleting other objects “owned” by the object to be deleted
- Example: `delete` the root of a tree ⇒ recursively deallocate child nodes
  if that is what is approporiate
- destructors could perform other resource management, like closing files

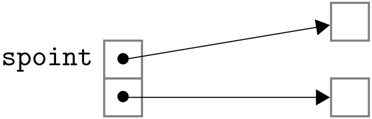
Destructors do not automagically follow pointers

```
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
However, 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(); // only type, no implementation
};

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

E::~E(){ }
```

`http://www.cs.bham.ac.uk/~hxt/2016/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
}
```

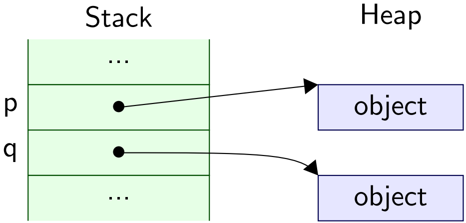
Destructors for the abstract syntax tree

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;
}
```

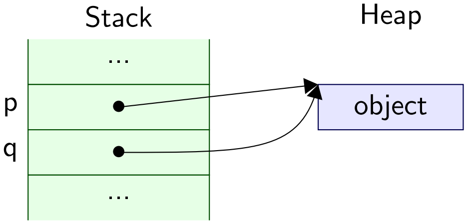
Allocate two different objects

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



Allocate one object and alias it

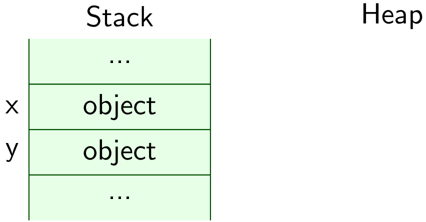
```
void f()
{
  C *p, *q;
  p = new C();
  q = p;
  ...
}
```



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/2016/c-plus-plus/Destructors.cpp`

`http://www.cs.bham.ac.uk/~hxt/2016/c-plus-plus/ParserTreeOO.cpp`

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

Destructors in C++ summary

new and delete in C++ combine ideas you have already seen:

1. memory allocation and deallocation, as in C
2. pointers, as in C
3. extra housekeeping code you can write for constructors, as in Java
4. interaction with inheritance, as in Java

Also relevant: smart pointers in Standard Template library: a limited form of automatic memory management
There are lots of extra details and complications, but you should be able to pick them up when needed
e.g. read Stroustrup's book and/or C++ standard
(both > 1300 pages ☹)

Object oriented programming in C with function pointers 😎

- C gives us primitive building blocks
- struct, pointers, functions
- including function pointers inside a struct! 😎
- 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
- Advanced example of pointers in C
- Some idea of how C++ is implemented
- Early C++ was a preprocessor for C
- internally, clang breaks OO into pointers

Object oriented programming in C with function pointers

```
struct animal {
    int age;
    void (*tellAge)(struct animal *p);
    // p works like the this pointer in C++
};

void pigTellAge(struct animal *p)
{
    printf("Oink! My age is %d.\n", p->age);
}

struct animal *Pig(int n)
{
    struct animal *p = malloc(sizeof(struct animal));
    p->age = n;
    p->tellAge = pigTellAge;
    return p;
}
```

Another derived class and constructor

```
void dogTellAge(struct animal *p)
{
    printf("Woof! My age is %d.\n", p->age);
}

struct animal *Dog(int n)
{
    struct animal *p = malloc(sizeof(struct animal));
    p->age = n;
    p->tellAge = dogTellAge;
    return p;
}
```

We get dynamic polymorphism via function pointers

```
struct animal {
    int age;
    void (*tellAge)(struct animal *p);
};

void animalTest()
{
    struct animal *peppa = Pig(4);
    struct animal *brian = Dog(8);
    peppa->tellAge(peppa);   // pass peppa pointer
    brian->tellAge(brian);
}
```

In C++, the compiler automagically passes the `this` pointer, so you only need to write

```
    peppa->tellAge(); // C++ passes pointer to object
```

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/`

Outline of the module (provisional)

I am aiming for these blocks of material:

1. pointers+struct+malloc+free
   ⇒ dynamic data structures in C as used in OS ✓
2. pointers+struct+union
   ⇒ 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.