C++ templates and parametric polymorphism

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

March 10, 2016

C++ polymorphism

	Templates	Dynamic polymorphism
When	compile-time	run-time
Typing	Type parameters	Subtyping
Efficiency	+ no runtime overhead - potential code bloat	- indirection via pointers at runtime
Related to	OCAML and Haskell polymorphism Java generics ML functors	Objective C messages Java methods

Over the last two decades, templates have developed from a relatively simple idea to the backbone of most advanced C programming. (Stroustrup 2012, section 25.1)

C++ templates

- templates are important: Standard Template Library
- type-safe collections
- concurrent data structures written by experts
- templates interact/collide with other C++ features
- templates allow compile-time computation $\Rightarrow$ zero overhead
- templates are a different language design dimension from object-orientation
- one way to use them is similar to polymorphism in functional languages
- In this module, we will build on your knowledge of functional programming

Templates and polymorphism

There are two kinds of templates in C++:

1. class templates
2. function templates

These correspond roughly to

1. polymorphic data types
2. polymorphic functions

in functional languages.
But: classes can contain member functions, not just data.

Polymorphism in functional languages

```
# [1; 2; 3];;
- : int list = [1; 2; 3]

type 'a bt = Leaf of 'a
           | Internal of 'a bt * 'a bt;;

# let twice f x = f(f x);;
val twice : ('a -> 'a) -> 'a -> 'a = <fun>
```

Templates: keyword `template`

```
template<typename T>
struct s {
 ... T ... T ...
 };
```

Then instantiating the template with argument `A` in `s<A>` is like

```
struct sA {
 ... A ... A ...
 };
```

Compare: λ calculus.

Templates: type parameter

```
template<typename T>
struct S
{
    // members here may depend on type parameter T
    T data;          // for example a data member
    void f(T);       // or a member function
    using t = T;     // or making t an alias for T
};
```

Class template example

```
template<typename T>
struct Linked
{
    T head;
    Linked<T>* tail;
};
```

Class template - other keywords

```
template<class T>
class Linked
{
public:
    T head;
    Linked<T>* tail;
};
```

Telling a template what to do

- We can pass types to templates
- We may also configure its behaviour
- sometimes called "policy", "callbacks", "algorithm"
- like higher-order functions
- there are many ways of doing this in C++
- classes with static member functions
- function pointers
- function objects, "functors"
- lambda expressions (new in C++11)
- restriction: template parameters must be known at compile time
- general function types using `function` template
- If C++ is not the bestest language, then it is at least the mostest 😋

Template parameters

- template parameters can be "type" or "nontype"
- example: `typename T` vs `int n`
- classes in C++ can be used as types
- but a class is also a namespace for its member functions
- `C::f()`
- hence functions can be passed to a template as static member functions of a class
- this double nature of classes is confusing from a type-theory view

Function template example with class parameter

We pass a type `T` and a class `Ops` that provides two operations.

```
template<typename T, class Ops>
T fold(Linked<T> *p)
{
    T acc = Ops::initial();
    while (p) {
        acc = Ops::bin(acc, p->head);
        p = p->tail;
    }
    return acc;
}
```

Note: we pass the class itself, not an object (instance) of that class

Class as argument of a template

This class provides integer operations:

```
struct IntOps {

    static int initial() { return 0; };

    static int bin(int x, int y) { return x + y; }
};
```

You could call this a “policy class”
In essence, this class is just a pair of functions.

Using the templates on int lists

```
int main(int argc, char *argv[])
{
    auto sumup = fold<int, IntOps>;
    // auto in C++ means type inferred automagically

    Linked<int> x {3, nullptr };
    Linked<int> y {2, &x };

    std::cout << sumup(&y);

    return 0;
}
```

Another class: string operations

This provides string operations:

```
class StrOps {
public:
    static std::string initial() { return ""; };

    static std::string bin (std::string x, std::string y)
    {
        return x + y;
    }
};
```

Using the template on string lists

```
int main(int argc, char *argv[])
{
    Linked<std::string> b = { "bar", nullptr };
    Linked<std::string> a = { "foo ", &b };

    auto sumupstr = fold<std::string, StrOps>;

    std::cout << sumupstr(&a) << "\n";

    return 0;
}
```

Template `std::function`

The template `std::function` gives general function types.
May need `#include<functional>`
Example: type of functions taking two integers and returning a float is

```
function<float(int, int)>
```

Useful for typing function parameters.
The same type can be used for C-style function pointers and C++ lambda expressions.
Ideally, something as basic as function types should have been built into the language from the start. For historical reasons, it wasn't.

Function as template argument

Here we pass a type T, a value of type T and a binary operation on T

```
template<typename T, T init, function<T(T,T)> bin>
T fold2(Linked<T> *p)
{
    T acc = init;
    while (p != nullptr) {
        acc = bin(acc, p->data);
        p = p->next;
    }
    return acc;
}
```

Function as template argument: using it

```
int sum(int x, int y)
{
    return x + y;

}

auto sumup2 = fold2<int, 0, sum>;
```

Member functions of template classes and scope

```
template<typename T>   // scope of T is class declaration
class C {
    T1 m;              // T1 could contain T
  public:
    T2 f(T3);          // T2 or T3 could contain T
};

template<typename T>   // need type parameter
T2 C<T>::f(T3 y)       // T2 or T3 could contain T
{
    ... T ...         // code can refer to T
    ... m ... y ...   // code can refer to m
}
```

Example: AST with parametric value type

$$
\begin{array}{lcll}
E & \rightarrow & c & \text{(constant)} \\
E & \rightarrow & x & \text{(variable)} \\
E & \rightarrow & (\otimes\ L) & \text{(operator application for some operator } \otimes\text{)} \\
E & \rightarrow & (=\ x\ E\ E) & \text{(let binding)} \\
\\
L & \rightarrow & E\ L & \text{(expression list)} \\
L & \rightarrow & &
\end{array}
$$

Expressions and environments

```
template<typename V>
struct env {
  string var;
  V value;
  env<V> *next;
};

template<typename V>
class Exp {
 public:
  virtual V eval(env<V>*) = 0;
  // much polymorphism wow
};
```

Derived classes

```
template<typename V>
class Let : public Exp<V> {
  string bvar;
  Exp<V> *bexp;
  Exp<V> *body;
 public:
  Let(string x, Exp<V> *e, Exp<V> *b)
    {
      bvar = x; bexp = e; body = b;
    }

  V eval(env<V>*);
};
```

Derived classes continued

```
template<typename V>
struct operators {
  std::function<V(V,V)> binop;
  V unit;
};

template<typename V>
class OpApp : public Exp<V> {
  operators<V> ops;
  ExpList<V> *args;
 public:
  OpApp(operators<V> o, ExpList<V> *a)
    {
      ops = o; args = a;
    }

  V eval(env<V>*);
};
```

Member functions of derived classes

```
template<typename V>
V Constant<V>::eval(env<V> *p)
{
  // code to evaluate a Constant
}
```

Void pointer polymorphism in C compared to templates

- C has void pointers as a kind of hacky polymorphism
- any pointer type can be cast to and from void pointer
- at the time (1970s) this was arguably better than what Pascal had
- C++ template are far more advanced than void pointers
- templates are type safe
- templates avoid the indirection of a pointer
  $\Rightarrow$ faster code

Void pointer polymorphism in C: polymorphic quicksort

Quicksort from C library:

```
void qsort (void *base, size_t num, size_t size,
            int (*compar)(void*, void*));
```

(Remember how to read function types with pointers.)

To use qsort, you need to supply a comparison function using void pointers:

```
int comparefloat (void *p, void *q)
{
  if ( *(float*)p <  *(float*)q ) return -1;
  if ( *(float*)p == *(float*)q ) return 0;
  if ( *(float*)p >  *(float*)q ) return 1;
}
```

Void pointer polymorphism example: polymorphic lists

```
struct Linked
{
    void* data; // indirection via void pointer
    struct Linked* next;
};
```

Void pointer polymorphism example: polymorphic fold

We could try something like this:

```
void* fold(struct Linked *p, void *initial,
           void *(*bin)(void *x, void *y))
{
    void *acc = initial;
    while (p) {
        acc = (*bin)(acc, p->data);
        p = p->next;
    }
    return acc;
}
```

Templates do this much more cleanly.

Templates: basic and more advanced features

- the main use of C++ templates is for polymorphic data types
- example: vectors, stacks, queues, ... of some type `T` and operations on them
- see Standard Template Library (STL)
- analogous to polymorphism in OCaml and Haskell
- replaces (some uses of) void pointers in C
- But there is much more to templates:
- template specialization
- higher-order templates
- compile-time computation
- these advanced features of templates are still somewhat experimental

Template specialization

- we may wish to fine-tune templates for different types
- example: treat pointer types `T*` different from other type
- example: vector of booleans could be implemented as bits
- Template *specialization* is like pattern matching in functional languages
- *specialization* $\neq$ instantiation
- We can pattern-match on types or values
- Templates can be recursive
- One possibility: compute functions at compile-time, e.g. factorial
- More serious: optimize templates for particular type parameters.
- We can write dependent types, like in Agda

Template specialization example: parsing C types

```
template<typename T>
struct NameofType;

template<>
struct NameofType<int> {
    static void p()
    {
        std::cout << "int";
    }
};

template<>
struct NameofType<float> {
    static void p()
    {
        std::cout << "float";
    }
};
```

Template specialization example: parsing C types 2

```
template<typename T>
struct NameofType<T*> {
    static void p()
    {
        std::cout << "pointer to ";
        NameofType<T>::p();
    }
};

template<typename T>
struct NameofType<T[]> {
    static void p()
    {
        std::cout << "array of ";
        NameofType<T>::p();
    }
};
```

Template specialization example: parsing C types 3

```
template<typename T, typename S>
struct NameofType<T(*)(S)> {
    static void p()
    {
        std::cout << "pointer to function returning a ";
        NameofType<T>::p();
        std::cout << " and taking an argument of type ";
        NameofType<S>::p();
    }
};
```

Values and types parameterized on values and types

$\downarrow$ parameterized on $\rightarrow$	Value	Type
Value	Function	Polymorphic function
Type	Dependent type	Polymorphic type

Dependent type example:

```
template<int n>
struct s {
    // structure may depend on int parameter
};
```

Arrays are also dependent types, but they decay into pointers in C/C++.

N-dimensional matrix template example

```
template<typename T, int n>
struct ndimMatrix;

template<typename T>
struct ndimMatrix<T, 0>
{
    T m[];
};

template<typename T, int n>
struct ndimMatrix
{
    ndimMatrix<T,n - 1> m[];
};
```

Template template parameters

```
template<template<typename>class Cont>
struct UseContainer {
    Cont<int> key;
    Cont<float> value;
};
...
UseContainer<vector> uc;
```

Lambda expressions in C++11

Lambda expressions in C++ provide two different new features:

1. anonymous functions, called “lambda expressions”
2. closures (capturing variables from a surrounding context) as the implementation technique for lambda expressions

Many modern language provide first-class functions and closures, not only C++

Lambda expressions in C++11

Lambda from the λ-calculus.
Here is a "lambda expression":

```
[=] (int x) { return x + a; }
```

It is like

```
fun x -> x + a
```

in OCAML or

$$\lambda x.x + a$$

Variable a needs to be in scope.
Variables can be captured by reference:

```
[&] (int x) { return x + a; }
```

Naive view of function call

```
int f(int x) { return x + x; }
```

$$f(2)$$

Naive view of function call

```
int f(int x) { return x + x; }
```

$$\begin{aligned} & f(2) \\ \rightsquigarrow\ & 2+2 \end{aligned}$$

Naive view of function call

```
int f(int x) { return x + x; }
```

$$\begin{aligned} & f(2) \\ \rightsquigarrow\ & 2+2 \\ \rightsquigarrow\ & 4 \end{aligned}$$

Lambda calculus: anonymous functions

Function definition without a name; just an expression

$$f = (\lambda x . x + x)$$

$$(\lambda x . x + x)(2)$$

Lambda calculus: anonymous functions

Function definition without a name; just an expression

$$f = (\lambda x . x + x)$$

$$\begin{aligned} & (\lambda x . x + x)(2) \\ \leadsto \; & 2 + 2 \end{aligned}$$

Lambda calculus: anonymous functions

Function definition without a name; just an expression

$$f = (\lambda x . x + x)$$

$$\begin{aligned} & (\lambda x . x + x)(2) \\ \leadsto \; & 2 + 2 \\ \leadsto \; & 4 \end{aligned}$$

Lambda calculus example: function as parameter

$$(\lambda f.f(3))\ (\lambda x.x + x)$$

Lambda calculus example: function as parameter

$$\begin{array}{rl} & (\lambda f.f(3))\,(\lambda x.x+x) \\ \rightsquigarrow & (\lambda x.x+x)\,(3) \end{array}$$

Lambda calculus example: function as parameter

$$\begin{array}{ll} & (\lambda f.f(3))\,(\lambda x.x+x) \\ \leadsto & (\lambda x.x+x)\,(3) \\ \leadsto & 3+3 \end{array}$$

Lambda calculus example: function as parameter

$$\begin{array}{ll} & (\lambda f.f(3))\,(\lambda x.x+x) \\ \rightsquigarrow & (\lambda x.x+x)\,(3) \\ \rightsquigarrow & 3+3 \\ \rightsquigarrow & 6 \end{array}$$

Lambda calculus example: function as result

$$(\lambda x.(\lambda y.x + y))\,(2)\,(3)$$

Lambda calculus example: function as result

$$\begin{array}{rl} & (\lambda x.(\lambda y.x + y))\,(2)\,(3) \\ \rightsquigarrow & (\lambda y.2 + y)\,(3) \end{array}$$

Lambda calculus example: function as result

$$\begin{array}{ll} & (\lambda x.(\lambda y.x + y))\,(2)\,(3) \\ \leadsto & (\lambda y.2 + y)\,(3) \\ \leadsto & 2 + 3 \end{array}$$

Lambda calculus example: function as result

$$\begin{array}{ll} & (\lambda x.(\lambda y.x + y))\,(2)\,(3) \\ \leadsto & (\lambda y.2 + y)\,(3) \\ \leadsto & 2 + 3 \\ \leadsto & 5 \end{array}$$

Lambda calculus example: function as result

$$
\begin{array}{rl}
 & (\lambda x.(\lambda y.x + y))\,(2)\,(3) \\
\leadsto & (\lambda y.2 + y)\,(3) \\
\leadsto & 2 + 3 \\
\leadsto & 5
\end{array}
$$

Real programming languages do not actually copy parameters into the function.
Closures are used to implement lambda expressions.
In C++, we need to be aware of what happens in memory.

Lambda expressions are implemented by closures

```
[=] (int x) { return x + x; }
```

Lambda expressions are implemented by closures

```
[=] (int x) { return x + x; }
```

Closed function: no free variables.
Easy to implement, like a function pointer in C.

Lambda expressions are implemented by closures

```
[=] (int x) { return x + x; }
```

Closed function: no free variables.
Easy to implement, like a function pointer in C.

```
[=] (int x) { return x + a; }
```

Lambda expressions are implemented by closures

```
[=] (int x) { return x + x; }
```

Closed function: no free variables.
Easy to implement, like a function pointer in C.

```
[=] (int x) { return x + a; }
```

Not closed due to a: must build a closure containing a

Lambda expressions are implemented by closures

```
[=] (int x) { return x + x; }
```

Closed function: no free variables.
Easy to implement, like a function pointer in C.

```
[=] (int x) { return x + a; }
```

Not closed due to a: must build a closure containing a

```
[&] (int x) { return x + a; }
```

Lambda expressions are implemented by closures

```
[=] (int x) { return x + x; }
```

Closed function: no free variables.
Easy to implement, like a function pointer in C.

```
[=] (int x) { return x + a; }
```

Not closed due to a: must build a closure containing a

```
[&] (int x) { return x + a; }
```

Closure with reference (implemented as pointer) to a

Lambdas as function parameters

```
int twice(function<int(int)> g, int n)
{
    return g(g(n));
}
```

What does this print?

```
cout << twice([] (int m) { return m + 1; }, 10) << endl;
```

Lambdas as function parameters

```
int twice(function<int(int)> g, int n)
{
    return g(g(n));
}
```

What does this print?

```
cout << twice([] (int m) { return m + 1; }, 10) << endl;
```

It prints 12.

Lambdas as function results

```
function<int(int)> f(int x)
{
    return [=] (int y) { return x + y; };
}

int main(int argc, const char * argv[])
{

    auto g = f(2);
    cout << g(3) << endl ;
    cout << g(4) << endl ;
}
```

What does this print?

Lambdas as function results

```
function<int(int)> f(int x)
{
    return [=] (int y) { return x + y; };
}

int main(int argc, const char * argv[])
{

    auto g = f(2);
    cout << g(3) << endl ;
    cout << g(4) << endl ;
}
```

What does this print?
It prints 5 and 6.

Currying in C++

In OCaml:

```
let curry f x y = f(x, y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

Using C++ templates and lambda, the above becomes:

```
template<typename A, typename B, typename C>
function<function<C(B)>(A)>
curry(function<C(A,B)> f)
{
    return ([=] (A x) { return ([=] (B y)
        { return f(x, y); }); });
}
```

Currying in C++

```
template<typename A, typename B, typename C>
function<function<C(B)>(A)>
curry(function<C(A,B)> f)
{
    return ([=] (A x) { return ([=] (B y)
        { return f(x, y); }); });
}
```

What does this print:

```
    auto cadd = curry<int,int,int>(add);
    cout << cadd(100)(10) << endl;
```

Currying in C++

```
template<typename A, typename B, typename C>
function<function<C(B)>(A)>
curry(function<C(A,B)> f)
{
    return ([=] (A x) { return ([=] (B y)
        { return f(x, y); }); });
}
```

What does this print:

```
    auto cadd = curry<int,int,int>(add);
    cout << cadd(100)(10) << endl;
```

It prints 110.

Lambdas in C++

- Lambda expressions bring some more functional programming into C++
- Lambda are usefully combined with templates and type inference
  But C is lurking underneath
- Using C++ for functional programming is like using a hammer to hammer in screws
  ("Birmingham screwdriver")
- but there some useful idioms that make lightweight use of lambdas
- C++ references are implemented much like C pointers.
- Capture by reference

  ```
  [&] (...) { ... };
  ```

  requires understanding of object lifetimes, e.g. stack

Internal iterators and lambdas

- common problem: need to iterate through data structure (list, tree, ...)
- many languages provide external iterators, e.g. Java
- an internal iterator is a block of code that gets applied to each data item in turn
- the code to be iterated could be some kind of function
- lambdas are a good way to turn some snippet of code into a first class function

Example: trees with internal iterator

```
template<typename T>
class bintree {
public:
    virtual void employ(std::function<void(T)>) = 0;
};

template<typename T>
class leaf : public bintree<T> {
    T data;
public :
    leaf(T x) { data = x; }

    void employ(std::function<void(T)> f)
    {
        f(data);
    }
};
```

Example: trees with internal iterator, internal nodes

```
template<typename T>
class internal : public bintree<T> {
    class bintree<T> *left, *right;
public:
    internal(bintree<T> *p1, bintree<T> *p2)
    {
        left = p1; right = p2;
    }
    void employ(std::function<void(T)> f)
    {
        left->employ(f);
        right->employ(f);

    }
};
```

Example: functions and lambda as internal iterators

```
int sum1;
void sumfun(int n) { sum1 += n; }

int main(int argc, const char *argv[])
{
    int sum2;
    class bintree<int> *p = new internal<int>(new leaf<int

    sum1 = 0;
    p->employ(sumfun);   // employ a C function
    std::cout << sum1 << std::endl;

    sum2 = 0;
    p->employ([&] (int x) { sum2 += x; });
    // employ a C++ lambda
    std::cout << sum2 << std::endl;
}
```

Lambda expression as internal iterators, summary

The above is a good use of lambda expressions.

```
sum2 = 0;
p->employ([&] (int x) { sum2 += x; });
```

A small piece of code made on the fly.
The code only works because the closure contains a reference to the variable
Before lambda in C++11, this wouuld have required the "function object" pattern
Similar cases: "delegates" and listeners

Lambdas and automatic variables

```
function<int()> seta()
{
    int a = 11111 ;
    return [=] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```
    cout << geta(seta()) << endl;
```

Lambdas and automatic variables

```
function<int()> seta()
{
    int a = 11111 ;
    return [=] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```
    cout << geta(seta()) << endl;
```

It prints 11111.

Lambdas and automatic variables, by reference

```
function<int()> seta()
{
    int a = 11111 ;
    return [&] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```
    cout << geta(seta()) << endl;
```

Lambdas and automatic variables, by reference

```
function<int()> seta()
{
    int a = 11111 ;
    return [&] () { return a; };
}

int geta(function<int()> f)
{
    int b = 22222;
    return f();
};
```

What does this print:

```
    cout << geta(seta()) << endl;
```

It prints 22222 when I tried it. Undefined behaviour.

Templates and lambda calculus

$$(\lambda x.(\lambda y.x + y))\,(2)\,(3)$$

Templates and lambda calculus

$$\begin{array}{rl} & (\lambda x.(\lambda y.x + y))\,(2)\,(3) \\ \rightsquigarrow & (\lambda y.2 + y)\,(3) \end{array}$$

Templates and lambda calculus

$$
\begin{array}{rl}
 & (\lambda x.(\lambda y.x + y))\,(2)\,(3) \\
\leadsto & (\lambda y.2 + y)\,(3) \\
\leadsto & 2 + 3 \\
\leadsto & 5
\end{array}
$$

Templates and lambda calculus

$$\begin{aligned} & (\lambda x.(\lambda y.x + y))\,(2)\,(3) \\ \leadsto\ & (\lambda y.2 + y)\,(3) \\ \leadsto\ & 2 + 3 \\ \leadsto\ & 5 \end{aligned}$$

```
template<int x>
int templatecurry (int y) {return x + y; }
...
    cout << templatecurry<2>(3) << endl;
```

Functors/function objects in C++

- A function object is an object that can be used like a function.
- One of its member functions overloads the function call syntax ()
- Such an object can have its own data members.
- We can create function object dynamically, unlike C functions.
- In functional programming terms, function objects simulate *closures*.
- Function objects are often used with templates.
- Objects *cannot* be template parameters, only function parameters.

Function object example

```
// class for curried addition function
class cadd {
private:
    int n;
public:
    cadd(int n) { this->n = n; }

    int operator() (int m) { return n + m; }
};

int main(int argc, const char * argv[])
{
    // make a new function object in local var
    cadd addfive(5);

    cout << addfive(7);
}
```

Function objects and templates

```
template <typename T, typename Op>
T twice(T x, Op f)
{
    return f(f(x));
}

int main(int argc, const char * argv[])
{
    cadd addfive(5); // create function object

    cout << twice<int, cadd>(10, addfive)
         << endl;
    cout << twice(10, addfive)
         << endl;
}
```

This prints 20 twice.

Virtual member function templates?

Suppose we want a worker that computes XML from a tree of ints.

```
template<typename T>
class polyworker {
public:
    virtual T workplus(T, T);
    virtual T workconstant(int);
};
```

Virtual member function templates?

Suppose we want a worker that computes XML from a tree of ints.

```
template<typename T>
class polyworker {
public:
    virtual T workplus(T, T);
    virtual T workconstant(int);
};


class Base {
public:
    template<typename T>
    virtual T employ(polyworker<T>*) = 0;
    // compilation error
};
```

See our paper on
A Type-theoretic Reconstruction of the Visitor Pattern

Visitor pattern

- The Visitor pattern is one of the classic patterns from the "Gang of Four" OO Patterns book
  Design patterns : elements of reusable object-oriented software
- Related patterns are Composite and Interpreter
- worker and employ are like visit visitor and accept in GoF
- GoF visitors use local state in the object rather than return types; they have void return types
- The GoF book is from 1995
- There is a lot of emphasis on inheritance
- Since them, C++ has taken on more ideas from functional programming (e.g., lambda, auto)

Some object-oriented patterns

Behavioural patterns are related

- Composite = object-oriented idiom to define trees
- Visitor = tree walker, internal vs external visitor stateful vs functional
- Interpreter, special case of Composite for a grammar of a language
- Iterator: internal visitors from lambda expression with reference

Visitor pattern as per Gang of Four 1

```
class gofvisitor {
public:
  virtual void visitplus(class plus*) = 0;
  virtual void visitconstant(class constant*) = 0 ;
};

class plus : public gofbase {
    class gofbase *p1, *p2;
public:
    plus(gofbase *p1, gofbase *p2)
    {
        this->p1 = p1;
        this->p2 = p2;
    }
    void accept(gofvisitor *v) {
        p1->accept(v);
        p2->accept(v);
        v->visitplus(this);
```

Visitor pattern as per Gang of Four 2

```
class plus : public gofbase {
    gofbase *p1, *p2;
public:
    plus(gofbase *p1, gofbase *p2)
    {
        this->p1 = p1;
        this->p2 = p2;
    }
    virtual void accept(gofvisitor *v)
    {
        p1->accept(v);
        p2->accept(v);
        v->visitplus(this);
    }
};
```

Visitor pattern as per Gang of Four 3

Because the return type is `void`, the visitor must use internal state to accumulate its result:

```
class countplusvisitor : public gofvisitor {
    int count;
public:
    void visitconstant(class constant *p) {}
    void visitplus(class plus *p) { count++; }
    int getcount() { return count; }
    countplusvisitor() { count = 0; }
};
```

Internal iterator and lambda expressions

Now suppose we want to do some work only at the leaf nodes on the data, not the internal nodes that determine the shape of the data structure.
We do not need a whole class.
A function to be called on each leaf.