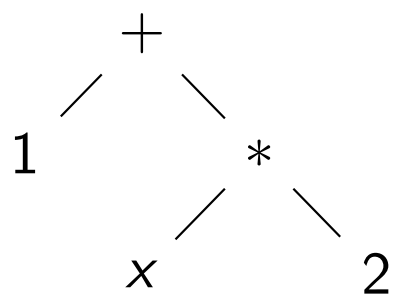
Typed trees and tree walking in C with struct, union, enum, and switch[1]

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

February 16, 2017



[1] and pointers, of course

Introduction to this section of the module

Different kinds of trees in C

union

Struct, union and enum

union, enum and switch

Adding recursion $\Rightarrow$ trees

Extended example: abstract syntax trees as C data structures

C data structures and functional programming

Example: a recursive-descent parser in C

Object orientation and the expression problem

Progression: position of this module in the curriculum

First year	Software Workshop, functional programming, Language and Logic
Second year	C/C++
Final year	Operating systems, compilers, parallel programming

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+tree
   $\Rightarrow$ trees in C
   such as parse trees and abstract syntax trees
3. object-oriented trees in C++
   composite and visitor patterns
4. templates in C++
   parametric polymorphism

An assessed exercise for each.

Trees from struct and pointers

- We have seen n-ary trees built from structures and pointers only ☺
- recursion ends by NULL pointers
- hence `if(p)` and `while(p)` idioms
- only one kind of node
- sufficient for some situations, e.g. much OS code ☺
- But there are more complex trees in computer science
- different kinds of nodes with different numbers and kinds of child nodes
- needs a type system of different nodes
- canonical example: abstract syntax trees
- fundamental ideas in compiling

Struct, union, and enum idioms

- How do we represent typed trees, such as abstract syntax trees or parse trees?
- Composite pattern in OO
- In functional languages: pattern matching
- Based on and inspired by: patterns, expression problem, type theory, compilers
- Pitfall: "pattern" means different things here:
  OO desing patterns vs pattern-matching in OCaml and Haskell
  usually clear from context

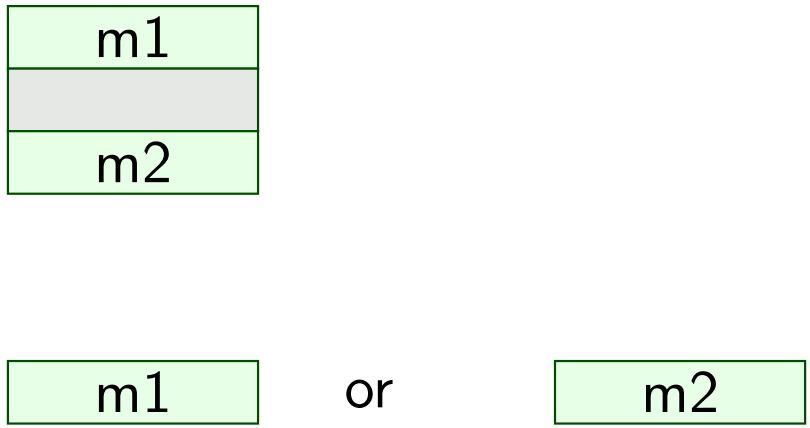
union syntax

The syntax of `union` is like that of `struct`:

```
union u {
    T1 m1;
    T2 m2;
    ...
    Tk mk;
};
```

Structure vs union layout in memory

```
struct s {
    T1 m1;
    T2 m2;
};
union u {
    T1 m1;
    T2 m2;
};
```



C11 draft standard says in section 6.7.2.1 that

> *a structure is a type consisting of a sequence of members, whose storage is allocated in an ordered sequence*

and

> *a union is a type consisting of a sequence of members whose storage overlap.*

unions are not tagged

```
union u {
    T1 m1;
    T2 m2;
};
```

The memory does not know whether it contains data of type `T1` or `T2`.

In C, memory contains bits without type information

If we want a tagged union, we need to build with from struct and enum

Quiz

```
union u {
    char s[10];
    int n;
};

int main()
{
    union u x;
    strncpy(x.s, "gollum", 7);
    printf("%d\n", x.n);
}
```

What does it print?

1. gollum
2. Nothing, type error
3. 1819045735
4. 2987297274
5. Unspecified, could be any number

Does valgrind report errors?

```
union u {
    char s[10];
    int n;
};

int main()
{
    union u x;
    strncpy(x.s, "gollum", 7);
    printf("%d\n", x.n);
}
```

Does valgrind report errors?

```
union u {
    char s[10];
    int n;
};

int main()
{
    union u x;
    strncpy(x.s, "gollum", 7);
    printf("%d\n", x.n);
}
```

No, valgrind is fine with the above
We are not using any bits we shouldn't
The type information is not visible to valgrind
Valgrind works on compiled code, not C source
There are no unions there, only memory accesses

Nesting in C type definitions

```
struct s1 {
  T1 m;
  int j;
};
```

Recursion in the grammar of C types:

$$\texttt{T1} \Rightarrow \texttt{struct s2 \{ int k; ... \}}$$

A struct may contain a type that may itself be a struct

Nesting: struct inside struct

```
struct s1 {
  struct s2 { int k; ... } m;
  int j;
};
```

Recursion in the grammar of C types:

$$\texttt{T1} \Rightarrow \texttt{struct s2 \{ int k; ... \}}$$

A struct may contain a type that may itself be a struct

Nesting struct inside struct lifted out

```
struct s2 { int k; ... };

struct s1 {
  struct s2 m;
  int j;
};
```

Recursion in the grammar of C types:

$$\texttt{T1} \Rightarrow \texttt{struct s2}$$

A struct may contain a type that may itself be a struct

struct and member names

```
struct s1 {
  struct s2 { int k; ... } m;
  int j;
};

struct s1 a;

a.j = 1;
a.m.k = 2;
```

s2 is the name of the type, and could be omitted here

m is the name of the nested struct as a member of the outer one

enum = enumeration type, much as in Java

```
enum dwarf { thorin, oin, gloin, fili, kili };

...

enum dwarf d;
...
switch(d) {
  ...
  case thorin: hack(orcs);
  ...
```

Implementation: small integers, e.g. `thorin` = 0, and so on

Tagged unions idiom

We use an enum for the tags.
Then we package the union in a struct together with the enum

```
enum ABtag { isA, isB };

struct taggedAorB {
  enum ABtag tag;
  union {
    A a;
    B b;
  } AorB;
};
```

It could be an `A` or a `B`
and we know which by looking a the tag.

switch statement and tagged unions

```
struct taggedAorB {
  enum ABtag tag;
  union {
    A a;
    B b;
  } AorB;
};
```

Access the tagged unions with switch:

```
struct taggedAorB x;
...
switch(x.tag) {
  case isA:
    // use x.AorB.a
  case isB:
   // use x.AorB.b
}
```

Disjoint union in set theory

`union` is a bit like union $\cup$ for sets
One can define a disjoint union with injection tags

$$\begin{aligned} & A + B \\ = \; & \{(1, a) \mid a \in A\} \\ \cup \; & \{(2, b) \mid b \in B\} \end{aligned}$$

We can tell if something comes from A or B by looking at the tag, 1 or 2.
Somewhat like a `switch`.
(This won't be in the exam.)

Example for union and switch: geometric shapes

- Consider geometric shapes and a function to compute their area
- A shape could be a rectangle, OR a circle, OR some other shape
- A circle has a radius
- A rectangle has a height AND a width
- OR $\Rightarrow$ tagged `union` idiom
- AND $\Rightarrow$ `struct`

Example: geometric shapes 2

```
enum shape { circle, rectangle };

struct geomobj {
    enum shape shape;
    union {
         struct {
            float height, width;
        } rectangle;
        struct {
            float radius;
        } circle;
    } shapes;
};
```

Example: geometric shapes — constructor-like function

This function is analogous to a constructor in object-oriented languages.
It encapsulates the low-level call to malloc and performs initialisation.

```
struct geomobj *mkrectangle(float w, float h)
{
  struct geomobj *p = malloc(sizeof(struct geomobj));
  if(!p) {
     fprintf(stderr, "malloc failed\n");
     exit(1); // give up :(
  }
  p->shape = rectangle;
  p->rectangle.width = w;
  p->rectangle.height = h;
  return p;
}
```

Note that there is both -> and .

Example: geometric shapes — switch

```
float area(struct geomobj x)
{
    switch(x.shape) {
        case rectangle:
            return x.shapes.rectangle.height
                * x.shapes.rectangle.width;
            // and so on
    }
}
```

XCode warns about missing case, analogous to non-exhaustive patterns in OCaml

Example: geometric shapes — enum and switch

Type definition:

```
struct geomobj {
    enum shape shape;
    union {
        struct {
            float height, width;
        } rectangle;
      // more shapes
    } shapes;
};
```

Code that operates on the type:

```
    switch(x.shape) {
        case rectangle:
            return x.shapes.rectangle.height
                * x.shapes.rectangle.width;
        // more cases and formulas for areas
```

Inconsistent use of tagged union idiom ☹

Warning: you can make mistakes like this

```
switch(x.shape) {
    case circle:
        return x.shapes.rectangle.height
            * x.shapes.rectangle.width;
    // ...
}
```

Does valgrind detect this kind of bug?

Inconsistent use of tagged union idiom ☹

Warning: you can make mistakes like this

```
switch(x.shape) {
    case circle:
        return x.shapes.rectangle.height
            * x.shapes.rectangle.width;
    // ...
}
```

Does valgrind detect this kind of bug?
Sometimes, but not always
Valgrind cares about bits, not about types

unions compared to Java inheritance

In C we have unions:

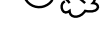
```
union AorB {
  A a;
  B b;
};
```

In Java, we could build a hierarchy of classes:

```
class AorB { ... };

class A extends AorB { ... };

class B extends AorB { ... };
```

Now `AorB` can be used somewhat like a union of `A` and `B`.
This is closer to a tagged union than a union by itself.
Could use `instanceof`, but bad OO 

Naming struct and union members

- Having to invent names for struct, union and enum members is tedious
- You may wish to make a systematic naming scheme for a given situation and stick to it
- matter of taste
- Since C11, anonymous structs and unions require fewer names
- flatten the tree-like name space of nested struct and union
- less verbose when nesting structs or unions
- also needs fewer dot operators to access
- clang supports them ☺

anonymous structure examples

Inner struct is not anonymous ☺

```
struct s1 {
    struct t1 { int n; ... } b;
    int n;
};
```

Inner struct is anonymous, lacking a member name, but compiler still knows what q is ☺

```
struct s2 {
    struct t2 { int q; ... } ;
    int n;
};
```

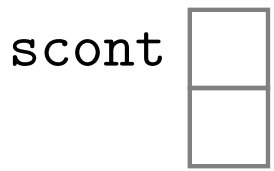
Not allowed: confusion about what member n is ☹

```
struct s3 {
    struct t3 { int n; ... };
    int n;
};
```

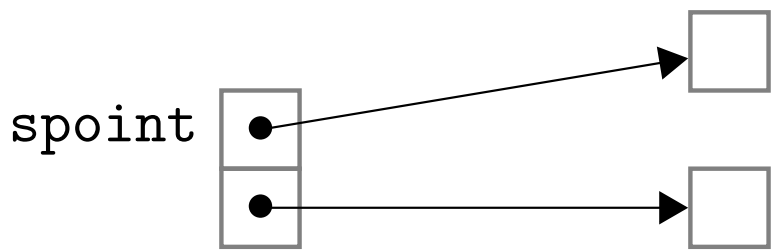
Structures containing structures or pointers to them

Anonymous structures are about using fewer names (and dots).
This is not the same as accessing via pointers or not.

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




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



On to trees

- We have seen struct+enum+union to get tagged union idiom
- can be processed with switch
- all well and good, but ...
- this needs more pointers
- and recursion
- when we add pointers and recursion, we get typed trees
- from trees and pointers we also get graphs

Trees with values only at the leaves

```
enum treetag { isLeaf, isInternal } tag;

struct intbt
{
    enum treetag tag;
    union {
        // if tag == isLeaf use this:
        int Leaf; // no recursion
        // if tag == isInternal use this:
        struct {
            struct intbt *left;    // recursion
            struct intbt *right;   // recursion
        } Internal;
    } LeafOrInternal;
};
```

Not the same as the trees with struct+pointer

```
struct twoptrs {
  struct twoptrs *ptrone, *ptrtwo;
  // recursion, unless NULL
  int data;
  // at all nodes, not just leaf nodes
}
```

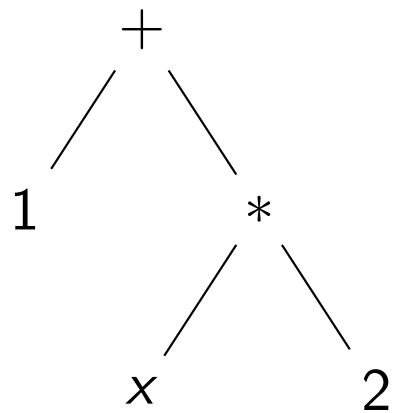
Needs NULL pointers to terminate.
NULL pointers are simple and efficient, but a bit of a hack
C has NULL pointers, but OCaml and C++ references can never be null

Trees and syntax

- parsing happens every time you run clang
- parsing happens every time you look at a web page
- fundamental idea: syntax is about trees
- trees can be represented in various ways in modern languages
- good example for C/C++ module
- there are two fundamentally different ways of representing trees in C++
- once you have the tree, meaning (of programs) is tree walking
- We will use abstract syntax trees as a worked example
- similar to parse trees
- struct + union + enum

Abstract syntax tree (AST)

In principle, a parser could build the whole parser tree:
In practice, parsers build more compact abstract syntax trees.
Just enough structure for the semantics (=meaning) of the language. For example:



In C, we use the for ASTs:
struct + union + enum + pointers + recursion + switch

AST for expressions in infix

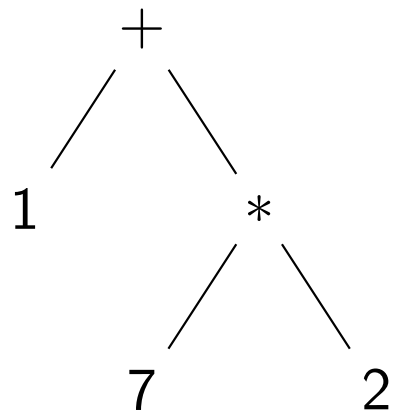
$$
\begin{aligned}
E &\rightarrow n \\
E &\rightarrow E - E \\
E &\rightarrow E * E
\end{aligned}
$$

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

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 $+$

Null pointers and assertions

- Our syntax trees are not supposed to contain null pointers at all
- Part of the invariant of the data structure
- the recursion ends by one of the cases of the union
- we can use the `assert` macro from `assert.h`
- `assert(p)`
  means we are not expecting a null pointer here
  give an error if the assertion fails
- assertions can be used for debugging and switched off for production code
- the idea of a precondition of code is generally useful, not just in C

eval as abstract syntax tree walk

```
int eval(struct E *p)
{
    assert(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);
    }
}
```

Exercise

Write a pretty-printing function that outputs an expression tree in (some form of) properly indented XML. For example, 1+2 should be printed as

```
<plus>
    <constant>
        1
    </constant>
    <constant>
        2
    </constant>
</plus>
```

Hint: the pretty printing function should take an integer parameter representing the current level of indentation.

Grammar for Lisp-style expressions

E	$\rightarrow$	n	(constant)
E	$\rightarrow$	x	(variable)
E	$\rightarrow$	$(+\ L)$	(operator application for addition)
E	$\rightarrow$	$(*\ L)$	(operator application for multiplication)
E	$\rightarrow$	$(=\ x\ E\ E)$	(let binding)
L	$\rightarrow$	$E\ L$	(expression list)
L	$\rightarrow$		

Lisp syntax is easy to parse.

AST for Lisp-style expressions

$$
\begin{array}{lcl}
E & \rightarrow & n \\
E & \rightarrow & x \\
E & \rightarrow & (+\ L) \\
E & \rightarrow & (*\ L) \\
E & \rightarrow & (=\ x\ E\ E)
\end{array}
$$

```
enum op { isplus, ismult };
enum exptag { islet, isconstant,
  isvar, isopapp };

struct exp {
  enum exptag tag;
  union {
    int constant;
    char var[8];
    struct {
      enum op op;
      struct explist *exps;
    } ; // anonymous struct
    struct {
      char bvar[8];
      struct exp *bexp;
      struct exp *body;
    }; // anonymous
  }; // anonymous
};
```

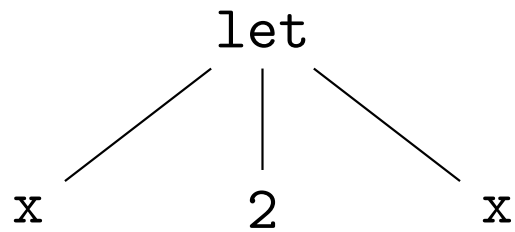
AST for Lisp-style expressions

$$L \rightarrow E\ L$$
$$L \rightarrow$$

```
struct explist {
  struct exp *head;
  struct explist *tail;
};
```

Optimization: instead of a union, we use the standard list idiom

Environments are passed down into the tree



- The second `x` is evaluated with the environment in which `x` is bound to 2.
- The environment is a parameter to the evaluation function

Constructing abstract syntax trees

- C structs are like classes that contain only data members
- structs do not have constructors
- but we can write functions that do the same job
- allocate memory and initialize is to given arguments

```
struct exp *mkopapp(enum op op, struct explist *el)
{
  struct exp *ep = malloc(sizeof(struct exp));
  if(!ep) {
     fprintf(stderr, "malloc failed\n");
     exit(1); // give up :(
  }
  ep->tag = isopapp;
  ep->op =  op;
  ep->exps =  el;
  return ep;
}
```

C tree vs functional programming

```
type intbt = Leaf of int
           | Internal of intbt *  intbt;;
```

The analogue in C using unions and structs is the following:

```
struct intbt
{
    enum { isLeaf, isInternal } tag;
    union {
        // if tag == isLeaf use this:
        int Leaf;
        // if tag == isInternal use this:
        struct {
            struct intbt *left;
            struct intbt *right;
        } Internal;
    } LeafOrInternal;
};
```

C tree vs functional programming

```
type intbt = Leaf of int
           | Internal of intbt *  intbt;;
```

C version more compact with anonymous struct and union:

```
struct intbt
{
    enum { isLeaf, isInternal } tag;
    union {
        int Leaf;
        struct {
            struct intbt *left, *right;
        } ;
    } ;
};
```

Learning C and functional languages

- The way data structures are built and used in C is closer to functional languages than to object-oriented ones
- OCaml or Haskell program
  = data structures (with `*` and `|`)
  and functions on them with pattern matching
- C program
  = data structures (with `struct` and `union`)
  and functions on them with `switch`, `->`, `.`
- object-oriented program in Java or C++
  = data and functions glommed together in classes
- But: no garbage collector in C, no type safety

Comparison of struct + union + pointer to OO

Manager "is-a" employee; manager "has-a" underlings
sounds like LOLcat:
I can haz cheezburger; I iz in your computer

```
struct employee {
    enum job { coder, manager } job;
    union {
        struct {
            char *proglanguage;
            int linesofcode;
        } coder;
        struct {
            struct employee **underlings; // recursion
            // array of pointers to managed employees
            int numunderlings; // size of array
            float bonus;
        } manager;
    };
};
```

Capstone example: recursive-descent parser in C

- this example illustrates all the main C constructs we have covered:
  `http://www.cs.bham.ac.uk/~hxt/2016/c-plus-plus/ParserTree.c`
- not trivial, but only 300 lines of code
- classic example; Kernighan&Ritchie and Stroustrup's book also use parsers as examples
- comes from compiling literature
- recursive descent: one C function for each non-terminal symbol of the grammar
- use lookahead into input and switch on it
- only complication is the need to eliminate left recursion
- clang uses a recursive-descent parser
  `http://clang.llvm.org/features.html#unifiedparser`

Scanning input using pointer arithmetic

```
char input[100];

char *pos = input;

char lookahead()
{
    while(isspace(*pos))
        pos++;
    return *pos;
}

void match(char c)
{
    if(lookahead() == c)
        pos++;
    else
        syntaxerror();
}
```

Allocating memory using pointer arithmetic

```
struct E myheap[1000];
struct E *freeptr = myheap;

struct E *myalloc()
{
    if(freeptr + 1 >= myheap + heapsize) {
        fprintf(stderr, "Heap overflow.\n");
        exit(1);
    }
    return freeptr++;
}

void reset()
{
    freeptr = myheap;
    pos = input;
}
```

Optimizing memory managment in C

- we can write our own memory management in C/C++
- as an alternative or on top of the library
- the more we know about our allocation pattern, the more efficient we can make it
- standard malloc is general-purpose and not the most efficient in all situations
- may avoid malloc entirely and use only global vars (what the C standard calls static storage)
- we can write C code for situations where there is no OS and no malloc available
- Can you do that in Java? Build a tree by putting all the nodes into an array and not use any heap?

Grammar for parsing

This grammar has been made suitable for parsing by left-recursion elimination.

$$
\begin{array}{lcl}
P & \to & 1 \mid \ldots \mid 9 \\
P & \to & (\ E\ ) \\
E & \to & F\ E' \\
E' & \to & +\ E\ E' \\
E' & \to & \\
F & \to & P\ F' \\
F' & \to & *\ P\ F' \\
F' & \to &
\end{array}
$$

Parsing function for nonterminal P

```
struct E *parseP() // primary expression
{
    char c;
    struct E *result = 0;

    switch(c = lookahead()) {
        case '0' ... '9':
            match(c);
            result = makeconstant(c - '0');
            break;
        case '(':
            match('(');
            result = parseE();
            match(')');
            break;
        default:
            syntaxerror();
    }
    return result;
}
```

Parsing function for nonterminal E

```
struct E *parseE()
{
    struct E *resultOfF;
    struct E *result = NULL;

    switch(lookahead()) {
        case '0' ... '9':
        case '(':
            resultOfF = parseF();
            result = parseEprime(resultOfF);
            break;
        default:
            syntaxerror();
    }
    return result;
}
```

Stretch exercise: recursive descent parser in C

Write a parser for the Lisp-style expression syntax
Write one parsing function for each non-terminal
Use lookahead and match functions to guide the recursive parsing functions
Construct the abstract syntax tree
Rather than using malloc, construct the node in an array

Exercise: grammar $\rightarrow$ structs

Take some grammars and translate them to the struct/union/enum/pointer idiom.
For example, suppose we have two binary operators $\otimes$ and $\oplus$ and a unary operator $\diamond$, as follows:

$$
\begin{array}{lcl}
A & \rightarrow & B \oplus B \\
A & \rightarrow & \diamond A \\
B & \rightarrow & A \otimes A \\
B & \rightarrow & n \text{ where } n \text{ is an integer}
\end{array}
$$

Object orientation and the expression 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

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 and visitor patterns
4. templates in C++
   parametric polymorphism

An assessed exercise for each.

C

C++