Abstract data types

These slides come from the set of slides available for "Programming Languages Design and Implementation" 4th Edition by T. Pratt and M. Zelkowitz, Prentice Hall, Upper Saddle River, NJ, © 2001.

Development of data types

1957-1960 FORTRAN, ALGOL - primitive data that mimics machine hardware

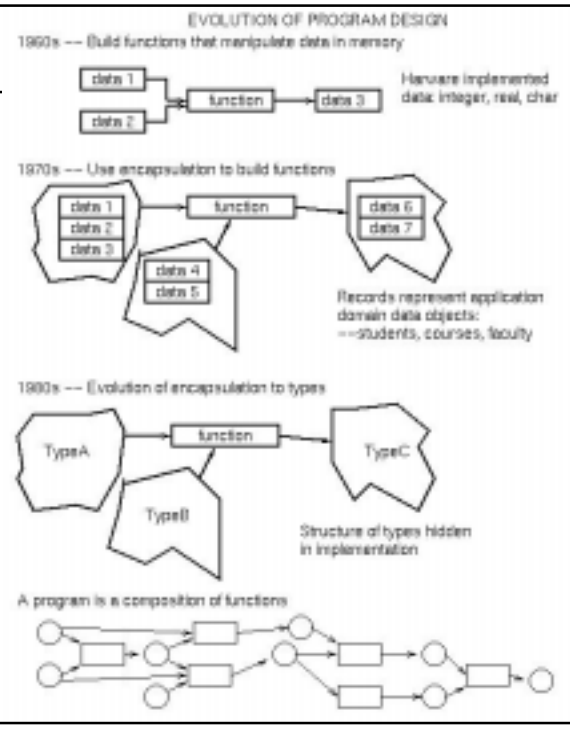
1960-1963 COBOL, PL/I - Extend primitive data with collections of primitive objects, records

1968-1970 Pascal, ALGOL-68 - Extends functionality with new types. Create functions that use those new types.

1972--1976 Age of encapsulation (Parnas)- Information hiding

- Create data type signatures
- Overloading
- Inheritance
- New languages:
  Alphard (Wulf, Shaw, LondonU) Euclid (Toronto)
  CLU (Liskov) Modula (Wirth)
  Smalltalk (Kay) Ada (DoD)
  Eiffel (Meyer)

1980--1985 Object Orientation - Smalltalk-80, C++, Miranda, ML, FORTRAN-90, Ada-95



3

Types

A type (data type) is a set of values that an object can have.

An abstract data type (ADT) is a:

- data type
- Set of functions (operations) that operates on data of that type
- Each function is defined by its signature.
- Can also specify operations formally (see section 4.2.5) (Algebraic data types): f(g(A,S)) = A

4

Example ADT

ADT student:

- Operations:

  SetName: name x student → student

  GetName: student → name

  SetCourse: course x student → student

  GetCourse: student → course *

  GetCredits: student → integer

In this example, Name and Course are undefined ADTs. They are defined by some other abstraction.

Information hiding: We have no idea HOW a student object is stored, nor do we care. All we care about is the *behavior* of the data according to the defined functions.

Information hiding

Information hiding can be built into any language

C example:

```
typedef struct {i:int;  ... } TypeA;
typedef struct { ... } TypeB;
P1 (TypeA X, other data) { ... } - P1:other data → TypeA
P2 (TypeB U, TypeA V) { ... }    - P2:TypeA → TypeB
```

Problems with this structure:

- No enforcement of information hiding. In P2 can write V.i to access component i of of TypeA object instead of calling P1.
- Success depends upon conventions
- But advantage is that it can be done in Pascal or C.

We will look at mechanisms later to enforce information hiding (Smalltalk, C++, Java, Ada). We will call this enforcement encapsulation.

Implementation of subprogram storage

Before looking at ADTs in detail, we first need to understand the usual method for subprograms to create local objects.

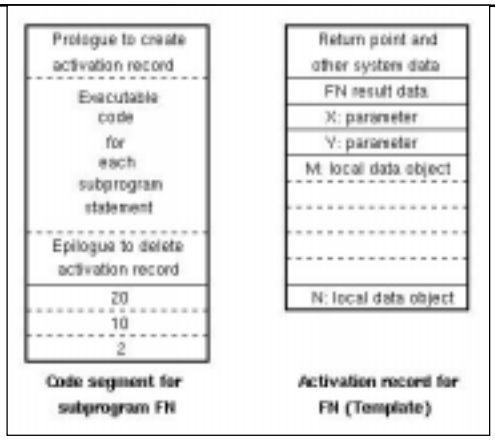
Each subprogram has a block of storage containing such information, called an activation record.

Consider the following C subprogram:

```
float FN( float X, int Y)
    const initval=2;
    #define finalval  10
    float M(10); int N;
    N = initval;
    if(N<finalval){ ... }
    return (20 * X + M(N)); }
```

Information about procedure FN is contained in its activation record.

Activation records



Above left: Code produced by compiler for the execution of procedure FN.

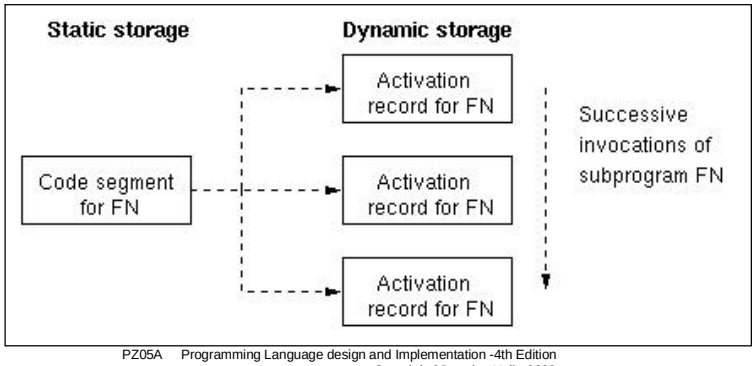
Above right: Data storage needed by procedure FN during execution. Compiler binds each identifier in FN with a storage location in activation record.

Dynamic nature of activation records

Each invocation of FN causes a new activation record to be created.

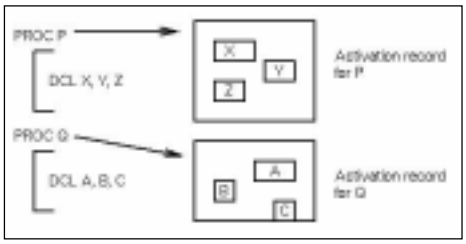
Thus the static code generated by the compiler for FN will be associated with a new activation record, each time FN is called.

As we will see later, a stack structure is used for activation record storage.



Storage management

Local data is stored in an area called an activation record (also called a stack frame).



Address data is a two step process:

1. Get a pointer to the relevant activation record containing the declarations.
2. Find the offset in that activation record for the data item.

Lifetime

IMPORTANT:

1. The activation record pointer is determined at during program execution. It occurs frequently, so it should be an efficient calculation.
2. The offset within an activation record is fixed and determined during compilation. It should require no - or minimal - calculations.

The lifetime of a variable is that period during program execution when the storage for the declared variable exists in some activation record (i.e., from the time the activation record for the declaring procedure is created until that activation record is destroyed).

The scope of a variable is the portion of the source program where the data item can be accessed (i.e., the name of the item is a legal reference at that point in the program).

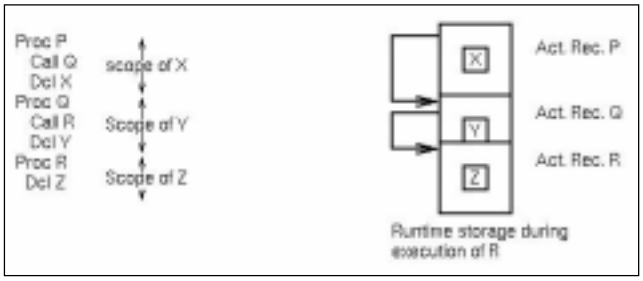
Lifetimes (continued)

Two models of data lifetimes:

Static: Based upon the static structure of a program

Dynamic: Based upon the execution of a program

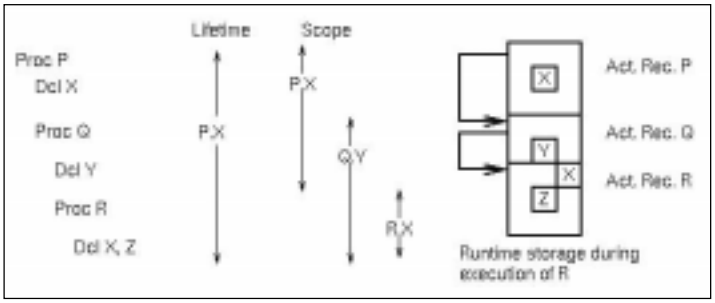
In general, static lifetimes are implemented in stacks; dynamic lifetimes implemented in heaps.



Lifetimes of variables

Lifetime of variable X is the period when the activation record for P exists.

Note that we can change the scope, but the not lifetime of a variable by nesting procedures:



Safe expressions

Type checking: Want languages to only permit safe expressions:

f: x $\rightarrow$ y.

f is a safe expression if f(a) = b implies a ε x. [That is, it is impossible to create an improper function call.]

+: int x int $\rightarrow$ int

2+3 = int result - all arguments are safe

But what about /:float x float $\rightarrow$ float?

3/0?

Strong types

A language is strongly typed if only safe expressions can be expressed in the language (i.e., results must be defined for that data type).

E.g., char a, b, c;

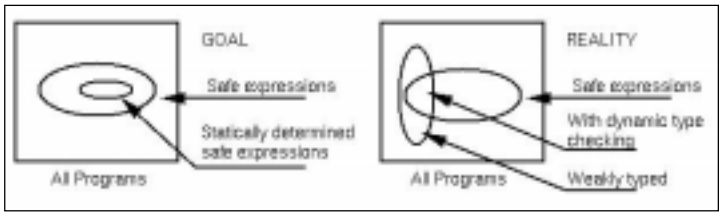
c = a+b; ⇐ Not safe since a+b can be > 255

But long a, b, c;

c = a+b; ⇐ Safe since overflows will halt execution

A[I] ⇐ Safe if dynamic checking of subscripts is performed

A language is weakly typed if it accepts expressions that are not safe.



Equality

Two aspects of equality:

- When do 2 objects have the same value?
- When are two types the same?

We will look at the first of these (equality of values)

Define a new language: assign 2+1 to A

- Is A equal to 3?

This is ambiguous as stated, although true for most languages studied so far.

Need signature for *assign* and *type* for A

- If A is integer, then coersion forces 2+1 to be 3.
- If A is type code, then 2+1 is expression to be evaluated

SNOBOL4 and Prolog operate this way. We will look at Prolog later.

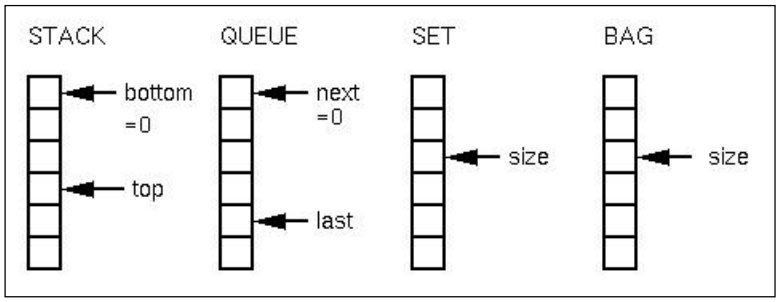
Equality of data objects

```
For primitive data types equality is bit-level equality:
   int x; int y;
   float x; float y;
   x=y if they have same bit values
But look at user types:
   type rational = record
       numerator: real;
       denomenator: real
       end;
   var A, B: rational;
   if A=B then ...          ⇐ When is this true?
True if: A.numerator=B.numerator and
   A.denomenator=B.denomenator
or A.numerator/A.denomenator = B.numerator/B.denomenator
There is no way for compiler to understand that rational
   means the usual mathematical concept of rational
   numbers. Equality must be defined, not coersed.
```


Why a need for separate definition of equality

```
Consider the following 4 data types and their
   representations:

All can be defined as:
   struct {int size;
           int storage[100];}
```



Meaning of A=B

```
What does A=B mean for each data type?
stack: A.size = B.size
   For all i, A.storage[i]=B.storage[i]
queue: A.size = B.size
   For all i, A.storage[i]=B.storage[i]
set: A.size = B.size
   All A.storage[i] differ
   For all i, some j, A.storage[i]=B.storage[j]
   (i.e., B.storage is a permutation of A.storage)
bag: A.size = B.size
   B.storage is a permutation of A.storage

So equality of user types is a complex issue that
   cannot be built into the operator by the language
   definition (effectively).
```


Type equivalence

```
Are X and Y the same type?
   type A = ...;  type B = ...;   A X;  B Y;
Name equivalence: X and Y are name equivalent if they
   are defined using the same name identifier:
In Pascal: var X: array [1..10] of integer;
              Y: array [1..10] of integer;
are not the same type, but:
   type arr = array [1..10] of integer;
   var U: arr; V: arr;
are the same type.
Structural equivalence: X and Y have the same storage
representation:
   x: record
        a: real; b: integer end
   y: record
        m: real; n: integer end
x and y have structural equivalence
```

C type equivalence

C uses name equivalence

Explain the following:

```
#define A struct {int M}  typedef struct {int M} A;
A X, Y;                   A X, Y;
A Z;                      A Z;
X=Y is legal              X=Y is legal
X=Z is an error           X=Z is legal
```

Although many C compilers will accept both definitions

Encapsulated data types

Remember from previously: An ADT with:

- Type with set of values
- Set of operations with signatures
- Representation - component structure of data type

Only the type name and operations are visible outside of the defining object.

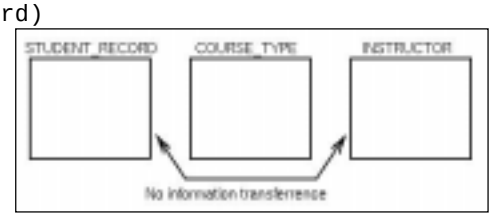
Example: StudentRecord is type

Externally visible:

```
 void SetName(StudentRecord, Name)
 name GetName(StudentRecord)
```

Internal to module:

```
 char Name[20];
 float GPA;
 char Address[50];
 CourseType Schedule[10];
```



Implementation of encapsulation

```
Usual Implementation in Ada:
package RationalNumber is
type rational is record  -- User defined type
   num, den: integer
   end record;
procedure mult(x in rational; -- Abstract operation
   y in rational; z out rational);
end package;
package body RationalNumber is -- Encapsulation
procedure mult(x in rational;
   y in rational; z out rational)
begin
   z.num := x.num * y.num;
   z.den := x.den * y.den;
   end;
end package;
```


Use of encapsulated data

```
Usual use of encapsulated RationalNumber:

Example: In main procedure do:
   var A, B, C: rational;
   A.num := 7;             ⇐ Should be illegal
   A.den := 1;
   Mult(A, B, C);

No enforcement of encapsulation

Any procedure has access to components of type
   rational. Can manipulate A.num and A.den without
   using procedures in package RationalNumber.

Let's look at alternative model to enforce
   encapsulation.
```

Private types

```
package RationalNumber is
type rational is private;  -- User defined type
procedure mult(x in rational; -- Abstract operation
   y in rational; z out rational);
private
type rational is record  -- User defined type
   num, den: integer
   end record;
end package;
package body RationalNumber is -- Same as before
procedure mult(x in rational;
   y in rational; z out rational)
begin
   z.num := x.num * y.num;
   z.den := x.den * y.den;
end;
end package;
```


Private types add protection

But now:

```
var A: rational;
A.num := 7; -- Now illegal. Private blocks use of num
   and den outside of package RationalNumber.
```

What is role of private?

- Any declarations in private part is not visible outside of package

What is difference in semantics of rational?

What is difference in implementation of rational?

This solution encapsulates and hides implementation details of rational.

C++ RationalNumber example

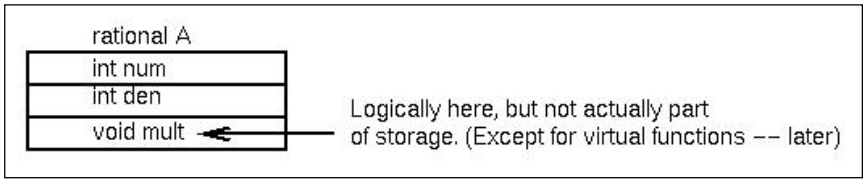
C++ creates objects of a user defined class.

- Data storage
- Set of operations

Type rational can be specified in C++ as:

```
class rational{
public: void mult( rational x; rational y)
   { num = x.num * y.num;
   den = x.den * y.den;}
protected: int num; int den }
rational A, B, C;
A.mult(B,C)           ⇐ invoke encapsulated function
A.num = B.num * C.num ⇐ Illegal. No access to num and den
```


Storage for C++ classes



Visibility of objects:

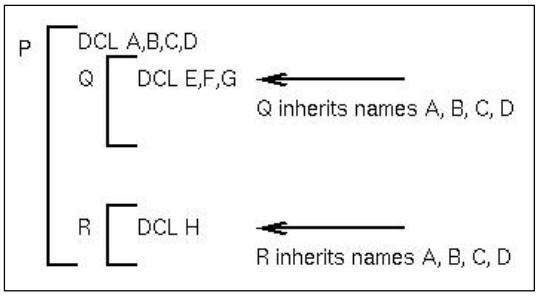
- public: globally known
- private: locally known only
- protected -- provides for inheritance

Inheritance

Inheritance provides for passing information from one data object to another automatically

It provides a form of data scope similar to syntactic scope.

Inheritance through data in object oriented languages is explicit through derived types.



Static scope (above) - Names are known implicitly through nested procedure names

C++ derived classes

Consider C++ class rational discussed earlier:

```
class complex: rational {
public: void mult( complex x; complex y);
{ realpt.mult(x.realpt,y.realpt)-
   realpt.mult(x.imagpt,y.imagpt) ...
void initial(complex x) {x.realpt.num = 0;
      x.realpt.den = 1 }
// complex inherits rational components.
   private: rational realpt;
   rational imagpt }
   . . .
   complex M, N, P;
   M.mult(N,P)
```

Power of inheritance

```
class rational {
   public: mult( ...) { ... }
   protected: error( ...) { ... } ...
   private: ... }
class complex:rational {
   public: mult( ...) { ... }
   private: ... }
complex X;
```

Function *error* is passed (inherited) to class complex, so X.error is a valid function call. Any derived class can invoke error and a legal function will be executed.

But what if we want error to print out the type of its argument? (i.e., want to know if error occurred in a rational or complex data?)

Power of inheritance (continued)

Inheritance is normally a static property:

Function *error* in class complex is known by compiler to be within class rational.

x.error ⇐ compiler knows where the error function is.

So how can rational::error know where it was invoked, as either rational::error or complex::error?

One way - Use function argument: error('rational') or error('complex')

Alternative: Use of virtual functions

Virtual functions

```
Base class:
class rational {
error() { cout << name() << endl; }
string name() { return "Rational";} ... }
Derived class:
class complex: rational {
string name() { return "Complex";} ... }
```

But if *error* is called, Rational is always printed since the call rational::name is compiled into class rational for the call in the error function.

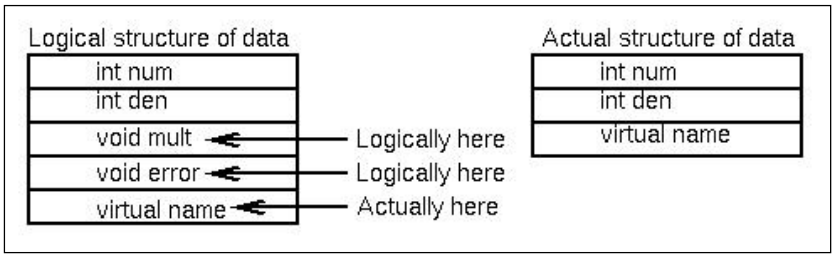
But if *name* is defined as:

```
virtual string name() { return "Rational";}
```

then name() is defined as a virtual function and the function *name* in the current object is invoked when name() is called in rational::error.

Implementing virtual functions

Virtual functions imply a runtime descriptor with a location of object




```
rational A;
complex B;
A.error() ⇐ error will call name() in rational
B.error() ⇐ error will call name() in complex
```

Example of Dynamic Binding

```
class shape {
   public: virtual void area() = 0;
   …}

class triangle: shape{area() {…}}
class square: shape{area() {…}}
class circle: shape{area() {…}}

void foo(shape polygon){
   …
   data = polygon.area;
   …}
```

What implementation of area is actually called?

Additional C++ inheritance attributes

Problem: Want to access data from the class object.

That is, in rational::error, print values for num and den of each component of class object

- Can use additional virtual functions
- this pointers: *this.counter

Accesses counter object in actual object passed to error.

- A.error(string X) compiled as: rational::error(&A, X)

That is, passes class object A as well as string X.

Friend classes

Friend classes: Strict inheritance sometimes difficult to do (i.e., Too hard to do it right so find a way around it!)

```
class thing { ...
MyFcn(complex A) { ... A.realpt }
```

But A.realpt is private data

Fudge solution by adding:

```
    friend class thing
```

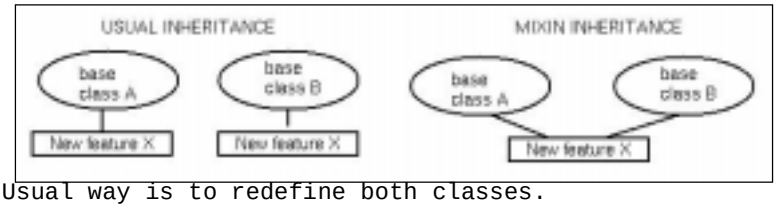
in complex class

Allows thing access to hidden components and avoid strict inheritance hierarchy of C++

Studies have shown this is most error prone feature of C++; redesign class hierarchy instead.

Mixin inheritance

Assume want to add feature X to both class A and B:



Usual way is to redefine both classes.

Mixin inheritance: Have definition which is addition to base class (Not part of C++)

For example, the following is possible syntax:

```
featureX mixin {int valcounter} ⇐Add field to object
newclassA class A mod featureX;
newclassB class B mod featureX;
```

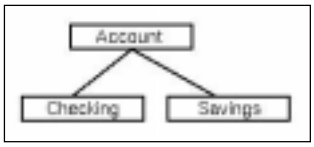
Can get similar effect with multiple inheritance:

```
class newclassA:A,featureX { ... }
class newclassB:B,featureX { ... }
```

Inheritance principles

1. Specialization: Usual form of inheritance: Checking inherits properties of Account.

Opposite is generalization:
Account is more general
than Checking.



2. Decomposition: Breaking an encapsulated object into parts. A rational object is a num and a den.

   Opposite concept is aggregation.

3. Instantiation: Creation of instances of an object:

rational A, B, C; ⇐ Represents 3 instantiations of object rational.

4. Individualization: Related to specialization.

Separate objects by function, not structure. A stack and a set can both be an array and and an index pointer, but functionality different.

   Opposite is grouping.

Overloading

Two operators with the same name but different signatures are said to be overloaded.

Older languages allowed overloading on built-in functions, e.g., real+real is different from integer+integer is different from integer+real is different from real+integer

C++ overloading;

cout << 123 - the << operator is given arguments cout and 123.

cout << ``abc'' - the << operator is given arguments cout and ``abc''.

Overloading (continued)

Non-primitive overloading:

```
myfunction (int a, int b) { ... }
myfunction (int a) { ... }
```

Each signature differs, so a different function is called.

Note that this is a static property determined by the compiler.

What about the following:

```
myfunction (int a, int b=7) { ... }
myfunction (char a) { ... }
```

- myfunction(3)?
- myfunction('a')?

Overload resolution

In C++:

Given X(args) {...}

Given X(args) {...}

1. If signatures match exactly, second is a redeclaration of the first.
2. If the arguments match exactly, but the return types do not, the second is an error.
3. If the arguments do not match exactly, the second is an overloaded definition.

Related concepts:

Coercion: Often implemented as part of built-in overloaded operators (already discussed)

Polymorphism: Types are defined by a parameterized argument. (e.g., type mystring = string(N: integer);)

```
proc MyFunction(X: mystring(N)) {...}
call Myfunction(Mydata(17));
```

We will discuss polymorphism in more detail later.

Polymorphism

The use of parameters to subprograms is one of the oldest characteristics of programming languages

But they have an l-value. That is, they are a data object that requires storage.

Polymorphism means a single operator or subprogram name can to refer to any of a number of function definitions depending on the data types of the arguments and results.

- Similar to overloading
- It is generally applied to functions where a type is one of the arguments.

Type inference examples

ML does not allow automatic coersions between int and real, so what are signatures (if any) to the following?

```
fun add(x,y) = real(x) + y;
fun thing(x::y) = x+ hd(y);
fun thing(x::y) = truncate(x)+ hd(y);
fun thing(x::y:int list) = x+ hd(y);
```

Polymorphism in arrays

Note: Implementation of arrays differs in most languages:

Pascal - arrays are explicit and bounds are part of type. Not polymorphic.

```
type matrix10 = array[1..10] of integer;
var x: matrix10;
```

Ada - polymorphic array bounds

```
type matrix is array (integer range <>) of integer;
y: matrix(1..30);
```

C arrays don't really exist - shorthand for pointer variables.

```
int C[10];
C[5] means *C+5
```


Create polymorphism

Can often build parameterized objects for true polymorphism, e.g., stack of objects:

```
Y: stack(int, 10)  ⇐ up to 10 ints
Z: stack(float, 20) ⇐ up to 20 reals
```

Can do this in ML and Ada for any data types that don't involve calculations (i.e., only for operations that use pointer manipulation (stacking and unstacking)

- Why?

Macros can be used to simulate this for languages without polymorphism, e.g. In Pascal:

sum(int,A,B) ⇐ can be implemented as a macro with 4 sequences depending upon types of arguments:

```
trunc(A)+B
A+trunc(B)
A+B
trunc(A)+trunc(B)
```

Implementation

For statically typed languages (ML, C++), polymorphism is easy: Keep track of function argument types in symbol table of compiler.

For dynamically typed languages:

Two forms of arguments can be passed to a polymorphic function:

1. An immediate descriptor - when the value to a function is smaller than the size of the fixed field. For example, passing a Boolean, character, or small integer to a function uses less space than the fixed-size object permits. The actual value is placed in the argument location, and the extra bits in the field are used to tell the function what the type actually is.
2. A boxed descriptor occurs in all other cases.

Boxed descriptors

The argument field will contain a type indicator stating the argument is boxed.

- The rest of the field will be the address of the actual object (which will be elsewhere, such as in heap storage).
- At this address, the complete type information will be given, such as giving the entire structure of the composite data object.

Example of dynamic polymorphism

Assume argument descriptor field is 5 bytes.

Byte 1 is a data type indicator and Bytes 2 to 5 are the data.

The following arguments can be passed to a polymorphic function:

1. 32-bit Integer data. Byte 1=0 signifying integer, bytes 2 to 5 are the 32-bit integer value.
2. 8-bit character data. Byte 1=1 signifying character, byte 2= actual character argument, bytes 3 to 5 unused.
3. 1-bit Boolean data. Byte 1=2 and byte 2= 0 or 1.
4. Complex record structure. Byte 1=3 and bytes 2 to 5 are pointer to structure. The r-value at this pointer address contains more information about the actual argument.