MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Imperative Programming (Part 1)

Daniel Playne
d.p.playne@massey.ac.nz

Imperative Paradigm

In the early development of programming languages, much development was focused on abstracting the details of hardware.

The emphasis moved from the machine to the higher-level algorithm nature of a program. This introduced common notions such as understanding of data-types, flow control etc.

> *The basis of the* ***imperative paradigm*** *is the fully specified and fully controlled manipulation of named data in a step-wise fashion.*
>
> - Bal & Grune, Programming Language Essentials

Imperative Paradigm

One of the successes of the imperative paradigm is the ease with which it fits with human thinking. It has a lot in common with the way we give instructions for many other problems - baking a cake, assembling furniture, changing the oil on a car.

These instructions are given as a series of actions (that are performed in order) and a defined set of objects to work with.

It also maps nicely to the machine domain that executes a series of machine instructions that modify values in memory.

Imperative Paradigm

Writing a program with an imperative language, a programmer needs to define named data and a series of manipulations.

The relatively simple mapping between imperative languages and the machine domain is one of the reasons imperative languages generally outperform other paradigms.

The downside is that it can bog the programmer down in specifying a lot of detail.

Imperative Paradigm

It is worth considering that while the imperative programming language fits well into the human nature of giving a series of instructions, it does come with associated weaknesses.

There are many books, courses, methodologies and degrees dedicated to solutions for making humans work together on a problem.

Similar problems arise in concurrent programming when multiple cores try to work together on the same problem.

Imperative Paradigm

Components of the imperative programming language paradigm:

- ***Data***
- ***State***
- ***Flow of Control***
- ***Program Composition***

Data

All of the data stored within a computer is stored the same way - as a series of 1's and 0's or bits in memory (usually as bytes or words).

For a data item to be manipulated in a meaningful way, a structure and often a name must be associated with this data item.

The association of a structure (type) and a name with a particular data item is a ***data declaration***.

Data Initialisation

Binding a name and a type to a data item allows the data item to be identified, modified and allows a value to be interpreted from the bit pattern in the machine.

The combination of a name, type and value is a ***variable***.

In C and Ada

```
int i, j;                    I, J: Integer;
```

That can be optionally ***initialized***

```
int i = 5, j = 5;           I, J: Integer := 5;
```

Symbol Table

Remember that a variable name only really has any meaning within a compiler. A compiler will usually keep track of the variables declared in a structure in a ***symbol table***.

The symbol table records the set of variable names declared in the program and the address in memory allocated to that variable. Symbol tables usually store other information such as the type of the variable etc.

Figure: Symbol Table

Name	Type	Address
`i`	`int`	`0x004f3e02`
`j`	`int`	`0x004f3e06`

Constant Declarations

Some languages provide support for constant declarations that prevent a variable's value from being modified after they have been initialised.

```
const int i = 24;
I constant Integer := 42;
```

This really an instruction to the compiler not to generate code that modifies the value of the constant variable. Although it can still be changed by accidental memory access.

Symbol Table

A constant declaration can be recorded as a field in the symbol table.

When the compiler encounters an instruction that attempts to change the value of a constant variable, it can check this field and generate an error instead.

Figure: Symbol Table

Name	**Type**	**Address**	**Constant**
`i`	`int`	`0x004f3e02`	**`true`**
`j`	`int`	`0x004f3e06`	`false`

Constant Declarations

Example of how to bypass the constant declaration functionality of C++ (please don't do this):

```
int i;
std::cin >> i;

const int c = i;
int *p = (int*)&c;

std::cout << "c = " << c << std::endl;
*p = 16;
std::cout << "c = " << c << std::endl;
```

Output:

```
c = 7
c = 16
```

Uninitialised Variables

Non-initialising declarations result in uninitialised variables. There are a number of methods for dealing with this problem:

- Don't - it's the programmers responsibility/problem (inelegant).
- Uninitialised variables are declared erroneous, compiler enforces a check (expensive)
- All uninitialised variables are silently initialised to some default value, e.g. zero bit pattern
- Force programmer to initialise all variables.
- Use a non-initialised value to allow dynamic checking of variable initialisation.

Renaming & Aliasing

Another type of declaration is to rename or alias an existing data item.

```
int &k = i;
K: Integer renames i;
```

No extra memory is allocated, just another name. Not the same as pointer.

Renaming & Aliasing

One way to implement aliasing is to simply add an extra entry into the symbol table that refers to the same address in memory.

While `i` and `k` appear as different variables, they refer to the same memory location and thus the same value (takes extra memory in the symbol table but not in the compiled program).

Figure: Symbol Table

Name	**Type**	**Address**	**Constant**
`i`	`int`	**`0x004f3e02`**	`false`
`j`	`int`	`0x004f3e06`	`false`
`k`	`int`	**`0x004f3e02`**	`false`

Types and Type Constructors

From the perspective of the machine, a type is a method of interpreting data bits in the machine. From the view of a programming language it is a set of values and operations.

The compiler writer must determine how best to represent the different types in the available hardware. The types within a programming language shouldn't depend on the hardware (or Operating System).

In C, the different size of `long` on Unix and Windows has been the source of many problems.

Types and Type Constructors

Hardware usually supplies the following ***basic*** or ***primitive*** types.

- Characters - `char`
- Integers - `short, int, long, long long`
- Floating-Point - `float, double`

These primitive types are built into a language, these basic types can be extended using ***type constructors***.

Types and Type Constructors

The simplest type constructor is an ***enumeration***. It does not really build a new type but provides a set of names which are the values of this type.

Enumeration definitions in C and Ada

```
typedef enum {
   red, orange, green
} traffic_light_colour;

typedef enum {
   false, true
} boolean;
```

```
Type Traffic_Light_Colour is
(Red, Orange, Green);

Type Boolean is (False, True);
```

Types and Type Constructors

Another very simple type constructor is the ***array*** - a series of a known number of elements (all of the same type).

Each ***element*** in the array can be accessed by its ***index***. Languages may provide operators to determine the first index, last index and number of elements in the array. Languages may also support the initialisation of an array through a ***compound value*** or ***aggregate***.

```
int a[10];
int b[3] = {3, 5, 8};

IA: array (Integer range 1..10) of Integer;
IB: array (Integer range 1..3) of Integer := (3, 5, 8);
```

Types and Type Constructors

In the previous examples, the bounds of the arrays are constant - the bounds of the array can be determined at compile-time. These are ***static*** bounds, as opposed to ***dynamic*** bounds that are not determined until the program is run.

Arrays with dynamic bounds are supported by Ada:

```
IA: array (Integer range M..N-1) of Integer;
```

Types and Type Constructors

Flexible arrays can be resized after they have already been declared. Neither C nor Ada support flexible arrays but some languages such as Algol 68, Orca and Icon do.

In C dynamically bounded and flexible arrays can be simulated by the programmer by using `malloc` and `realloc`.

Types and Type Constructors

The previous arrays have all been one-dimensional arrays. While multi-dimensional arrays can be simulated with one-dimensional arrays, many languages provide support for ***multi-dimensional*** arrays or arrays of arrays.

An n-dimensional array is a rectilinear block of elements that requires n-indices to access an element. Arrays of arrays do not necessarily require all the arrays to be the same length.

Row/Column Major

While not exactly important from a language perspective, for performance it is important to understand how languages treat multi-dimensional arrays. Languages may store a multi-dimensional array in ***row-major*** (C, C++, Java) while Fortran uses ***column-major***.

If the elements of an array are iterated over it is important to do so in the correct order to make best use of data caching.

Array Slicing

Array slicing is similar to indexing but instead of accessing a single elements is used to access a subarray.

Ada allows a subarray to be accessed through `IA(3..6)`

Fortran 90 allows a stride to be specified as well `A(4:10:2)` which access elements `A(4)`, `A(6)`, `A(8)`, `A(10)`.

Associative Arrays

Associative Arrays are arrays that don't just use integers as index values but allow other types to be used (provided there is no ambiguity).

The index value (whatever it is) is associated with a particular element in the array.

One common usage includes associating a string with an element in an array.

Records & Structs

Records or ***structs*** are a way of defining a user-defined type that groups a fixed number of data items (of possibly different types) together. The items in a record are ***fields*** or ***components***.

Records in C and Ada

```c
typedef struct {

   brand_type brand;
   unsigned int number_of_doors;
   int price;
} car_type;
```

```ada
type Car_Type is
record
   Brand: Brand_Type;
   Number_Of_Doors: Natural;
   Price: Integer;
end record
```

Records & Structs

Self referencing records or structures can cause problems.

Records in C and Ada

```c
typedef struct {

   bin_tree left, right;
   item_type value;
} bin_tree;
```

```ada
type Bin_Tree is
record
   Left,Right: Bin_Tree;
   Value: Item_Type;
end record
```

Records that reference the same type of record usually require pointers or some equivalent.

Unions

A ***union*** is like record of N fields where one value of the data item can be any of the N fields. The compilers typically use the same overlapping memory for all N fields (from the bad old days of very limited memory).

Records in C and Ada

```c
typedef union {
   car_type car;
   unsigned int num_bicycles;
} garage;
```

```ada
type Contents is
   (Has_Car, Has_Bicycles);
type Garage(Status: Contents) is
 record
    case Status is
      when Has_Car =>
         Car: Car_Type;
      when Has_Bicycles =>
         Num_Bicycles: Natural;
    end case;
  end record;
```

Unions

Unions in languages such as Ada have additional status information (***discriminated unions***) and are able to provide strong type checking.

Languages such as C do not have this status information (***undiscriminated unions***) and the programmer must remember the status of the union. The compiler will not be able to perform type-checking and accessing the wrong field may return invalid bit patterns (e.g. invalid floats).

Pointers

A ***pointer*** is simple a memory address that can be used to access another data item. A pointer must be ***dereferenced*** to access the data that it *points* to.

Pointers in C and Ada

```
data_type *ptr;                     Ptr: access Data_Type;
(*ptr)                              Ptr.All
(*ptr).field                        Ptr.All.Field
ptr->field                          Ptr.Field
```

The data types the pointers are going to *point* to allow the compiler to perform some type-checking.

Pointers

Pointers are used heavily and languages such as C and C++ and are an analogue of raw memory addresses in Assember and Machine Code. They are powerful and efficient but also the source of many errors in program.

They fit well with the model of computation for imperative programs that deal directly with data values stored in memory addresses.

Dynamic Memory Allocation

Language with pointers can also support dynamic memory allocation where the program requests memory storage during its execution.

The program is responsible for requesting the appropriate amount of memory and returning or freeing the memory once it is finished with it.

Pointers

Languages with pointers provide a ***Null*** or equivalent value that represents a *no-address* value.

In general, pointers should always be initialised to either the address of a data item or this value (may be done automatically or by the programmer).

It is good practice to *nullify* a pointer after the memory it points to has been freed, pointers that are not set to null are called ***dangling pointers***.

Pointers

Pointers can make working with data significantly more efficient by passing pointers to data structures rather than copying the entire structure. More complex structures can be build by creating pointers to pointers to pointers...

In some languages (C), pointers are used as a substitute for references/aliases (they are only a substitute though).

Some languages allow arithmetic to be performed on pointer values while other may restrict them to assigning an address and dereferencing the pointer.

No Pointers

There are some languages such as SETL, ABC and Orca that do not support pointers and instead provide inbuilt implementations for common data structure (lists, tables, sets etc).

These languages generally do not emphasise efficiency or require optimising compilers.

Routines as Data Types

Some languages provide functionality for routines to be used as ordinary data types (with some restrictions). C and modern Ada both provide this through ***function pointers*** which can be used to dynamically chose which function to execute.

Function Pointers in C and Ada

```
int (*to_int)(float);              type R2I_Access_Type is
                                         access function (F: float)
                                         return Integer;
                                   To_Int: R2I_Access_Type;

i = to_int(3.14);                  I = To_Int(3.14);
i = (*to_int)(3.14);               I = To_Int.all(3.14);
```

Language Functionality

Languages that do not directly support these functions and data structures can often implement the same functionality and may be available as a library.

Is it better to have a larger language with more support for this functionality or a smaller language that implements them as a library?

Summary

- Data & Data Declarations
- Constant Declarations
- Uninitialised Variables
- Renaming & Aliasing
- Types and Type Constructors
- Enumerations, Arrays, Records
- Pointers