# Course Notes: CPSC 4100 Spring 2020

## Languages

### C

- originally 1973
- Dennis Ritchie (The R in K&R)
- ANSI (American National Standards Institute) C standard since 1989
- imperative: statements affect program state
- structured: formal control structures / blocks
- procedural: code organized into called procedures (subroutines)
- static typing: data type property assigned at compile time
- weakly typed (`void*`) : implicit type casting under some conditions
- compiles all the way to the hardware (executables not portable)
- allows for raw memory management and manipulation
- modeled naturally on the standard *von Neumann machine* architecture
  - CPU with registers, ALU, control unit
  - memory containing both instructions and data

### hello world in C (the parts of)

```
#include <stdio.h>              /* preprocessor directive */

int main(int argc, char *argv[]) /* program entry point with command line arguments
{
    printf("Hello World\n");    /* a subroutine that does IO -- declared in stdio.h
    return 0;                   /* return code from main -> exit code for program */
}
```

## C data types

```
 void
 char
 int
 float
 double

 // these assume int
 short
 long
 signed
 unsigned

// may be optimized by using read only memory
const

// no implementation-independent semantics
volatile
```

- typedef syntactic renaming of a type

  ```
  typedef unsigned int uint
  typedef const double CONSTANT;
  ```

- arrays

  ```
  int values[10];  -->   int* values
  ```

  array lookups as math problems

```
int values[10];

for (int i=0; i<10; i++)
{
    printf("%d: %d\n", i, values[i]);
}

printf("set values[3]\n");
/* type of values tells it how to interpret the 3 */
*(values + 3) = 999;

/* raw addresses */
printf("set values[8]\n");
*((int*)((void*)values + sizeof(int)*8))=1234;
```

## C++

- Bjarne Stroustrup
- first appeared 1985; standardized in 1998
- adds object oriented features, namespaces, generics, exceptions

### hello world in C++

```
#include <iostream>

int main(int argc, char *argv[])
{
    std::cout << "hello world" << std::endl;
    return 0;
}
```

### Null-terminated strings

String data is stored as an array of characters with the NULL character (ascii 0) indicating the end of the string.

Many useful functions for dealing with data of this type are available in `string.h`.

## Scheme

A lexically scoped dialect of Lisp

Strongly typed / dynamically typed.

### data types

- numbers: `1.0`, `45`, `8+3i`, ...
- characters: `#\x`
- booleans: `#t` `#f` (note this is different than `#\t` and `#\f`)
- symbols: 'foo
- strings: "hello"
- vectors: `#(1 2 3)` – like a fixed-length list

**procedures**

- created with a `lambda` expression

```
(lambda (a b)
  (+ a b))                              ;add a and b
```

- bound to a variable with the `define` special form

```
(define add2 (lambda (a b)
               (+ a b)))
```

**function currying**

concept from the lambda calculus where procedures may only take a single parameter.

f(a,b) = {f'(a)}(b)

For example, if lambdas could only take one argument:

```
(define add (lambda (a) (lambda (b) (+ a b))))
((add 5) 10)
```

**read from user**

- `(read)` -> symbol
- `(use-modules (ice-9 readline)) (readline "enter a string")` -> string

**control structures**

- if/else
  - eq? equal? ...

- when/unless
- loops do exist: while / do
- begin (implied within lambda)

**working with lists/pairs**

- cons
- car / cdr
- car, caar, cddr, cadr, etc.
- list / pair procedures
- memq , assoc

`cons` creates a new list with the first parameter stuck on the beginning of the second.

```
scheme@(guile-user)> (cons 'a '(1 2 3))
$2 = (a 1 2 3)
```

`car` returns the first item in a list. `cdr` returns the rest of the list, after the `car`

```
scheme@(guile-user)> (car '(this that the other))
$3 = this
scheme@(guile-user)> (cdr '(this that the other))
$4 = (that the other)
```

**recursion as iteration**

- factorial
- map, filter

**binding local variables introduce new variables**

- let, let*, letrec, letrec* let* -> nested lets
- nested define -> letrec

#### closures

A function with environment containing free variables bound in some other *environment* that existed when the function was created.

```
(define (make-adder base)
  (lambda (i)
    (+ base i)))
```

#### imperatives

- `set!` (and many other variants)

#### lambda as object

- closures and imperatives together give object-like functionality

## General Concepts

### Binding Times

The act of associating *names* with properties (data type, address, value) is called *binding*, and different properties are bound at different times.

```
#include <math.h>


  void main()
  {

  int i;
  double sum=0;

      for (i=1; i<100; i++)
           sum += sqrt(i);
  }
```

- **language definition time**

  meaning of keywords is bound – all implementations must behave the same way (void, for)

- **language implementation time**

  e.g. the range of values for `int` is implementation dependent. (not the same in java)

- **compile time**

  - data type for `i` is bound here. (static typing)
  - details of `sqrt` interface (declaration in math.h)

- **link time**

  definition of `sqrt`

- **load time**

  memory address for all of these symbols

- **runtime**

  `i` takes on a sequence of values

- early binding : before runtime / late binding == runtime binding
- not all language systems use all times (interpreters are not compiled)

## Parameter Passing Semantics

### Definitions

- formal parameters (specified in subroutine)
- actual parameters (passed to subroutine)
- the call stack

### parameter *correspondence*

- java and C use positional parameters
- other languages may have keyword parameters
- default parameters (C++ has this)

- variable arguments in C processed with system calls

### Call-by-value

- formal parameters are local variables in the stack frame (aka *activation record*) of the called method
- initialized with the value of the corresponding actual parameter
- variables used in calling function cannot be directly modified since only the values are passed (pointers & references complicate this)

### Call-by-reference

The lvalue of the actual parameter is computed before the method executes. Formal parameters are replaced with actual parameter's lvalue. Effectively, the formal parameters become aliases for the actual parameters.

### Call-by-macro-expansion

- formal parameters replaced with text of actual parameters
- macro call replaced with expanded macro

- variable capture For a given code snippet
  - **free variables** have no binding (are not associated with a specific memory location)
  - **bound variables** do
  - Macro expansion can cause free variables to become bound inside the macro expansion – this has undefined semantics and will result in errors (see the SWAP exmaple)

### Call-by-name

Formal parameters are substituted (in a capture-avoiding way) with "text" of actual parameters.

Also: **call-by-need** semantics: this is done in a memoized way (values are cached so it should be faster for certain applications)

## Parameter evaluation

- applicative order: parameters are evaluated before subroutine is called (this is typical for C, Java, scheme...)
- normal order: actual parameters are substituted into subroutine body and evaluated after subroutine call begins
  - see, Call-by-name

### Typing Systems

- rules surrounding the binding of data type to variables and expressions
- statically typed – data types bound at compile time
- type safety: how aggressively does the language apply typing rules to force you to write safe, good code
  - strongly typed <—> weakly typed
    C is (relatively) weakly typed because we can throw away type information by casting to `void*`

### Definitions

#### statements vs. expressions

- a statement is a executable step in the algorithm
  - the building block of an algorithm
- an expression is anything with a value (can be evaluated)

#### lvalues and rvalues

Every expression is either an lvalue or rvalue

- rvalue's are temporary and have the lifetime of the corresponding expression
- lvalue's persist beyond the expression
  - variables (anything with a name)
- C has the ability to convert between the two in a manner

## Social Distancing Lecture Notes

### Lecture 17 - Social Distancing 1 *[2020-03-23 Mon]*

- continuations
  - a black box / abstraction of the current state of a program
  - a continuation is the call stack
  - in guile scheme, continuations are copies of the call stack right now
  - an *escape procedure* – replaces the call stack .. made with call/cc
  - the presence of first-class continuations suggests that programs can act

  on their own state

```
(define kont (list))
(cons 'top (call-with-current-continuation (lambda (c) (set! kont c))))
```

```
(+ (call/cc (lambda (k^) (/ (k^ 5) 0))) 8)
```

– k^ is an escape procedure. we can build break/resume

```
scheme@(guile-user)> (if (call-with-current-continuation (lambda (x) (set! kont
$38 = false
scheme@(guile-user)> (kont #t)
$39 = true
scheme@(guile-user)> (kont #f)
$40 = false
```

more...

```
(define kont '())
(define (go)
  (let ((cmd (call/cc (lambda (c) (set! kont c) 'none))))
    (cond
     ((eq? cmd 'up) 1)
     ((eq? cmd 'down) -1)
     (else 0))))

(+ 100 (go))
(kont 'up)
(kont 'down)
```

**last continuation example (lays the groundwork for cooperative multitasking)**

```
(define escape)                                 ;top-level
(define RESUME)
(call/cc (lambda (k^) (set! escape k^)))

(define BREAK
  (lambda (msg)
    (call/cc
     (lambda (k^)
       (set! RESUME k^)
       (escape msg)))))

(map
 (lambda (a)
   (BREAK (string-append "mid map: " (number->string a)))
   (+ 5 a))
 '(1 2 3))
```

**exam review**

### Lecture 18 - Social Distancing 2 *[2020-03-25 Wed]*

#### Scripting languages

- often distinguished from *systems programming* languages like C that are ...
  - statically typed
  - compiled to machine code
- scripting often refers to **glue code** for more connecting different pieces
  - dynamic typing facilitates this
  - historically very text-oriented
- Modern scripting languages may better be described as *dynamic languages*
  - dynamic typing
  - first class functions
  - eval

#### Bash

- glue language part 1

- pipelines

```
ls /Users |wc -l
```

- variables and arrays – the environment
- simple logic

```
for i in {0..5}; do echo $i; done
```

**Perl**

- practical extraction and report language
- pathologically eclectic rubbish lister

Initially created by Larry Wall in 1987.

**We are talking about Perl 5**

A great example of a glue code language. It has some interesting capabilities including an unparalleled implementation of regular expressions, and inspired many modern scripting languages.

- closures / first-class functions
- objects
- reflection
- garbage collection

Last time, I introduced you to perl syntax, and it is unusual. I'd like to emphasize a few aspects of this.

1. Larry Wall created this language because he had a very specific idea of how he wanted to be able to write programs, and it involved a melange of ideas.
2. This is primarily a language for the programmer, not for the system.
   - Every programmer should have a go-to scripting language
3. They added capabilities to support bigger projects as the language became more popular (strict, objects, etc.)

While we're here, we want to look at some design decisions that are clearly different than those by other languages: dynamic & weak typing, call by reference, dynamic variable scope. and we want to celebrate the things it's good at – text processing, rapid prototyping.

- weird things
  - variables use leading sigils to distinguish usage
  - everything is global by default
  - default punctuation variables
  - parentheses are often optional
- syntax
  - primary data types

    **Scalars**
    * numbers
    * strings
    * everything that is neither 0 nor empty string "" is true

    **Compound**
    * lists
    * associative arrays (hash tables)
- weak & dynamic typing

  this necessitates something that feels like the opposite of operator or function overloading. we have different versions of things to force the data to be treated as expected: `cmp vs <=>`

```
$z = "3" x 3;
print $z, "\n";
print $z+5, "\n";
```

- imperative, procedural and functional
  - map, grep
- dynamic variable scope

```
use strict;
use vars qw($x);


sub test_variable {
  print "$_[0]: The variable is $x\n";
}



test_variable(1);



$x = "house";
test_variable(2);


{
  my $x="hat trick";
  test_variable(3);
}

test_variable(4);


{
  local $x="submarine";
  test_variable(5);
}



test_variable(6);


my @list=qw(this is a funny way to make a list of strings!);
print "@list\n";

$"="\n";
print "@list\n";
```

- call-by-reference

```
sub change_it {
  #my ($x) = @_;
  $_[0]=100;
  #$x = 100;
}
```

```
my $z = 9;

print "$z\n";
change_it($z);
print "$z\n";
```

- A common -idiom- results in call-by-value semantics
- integration with the shell

```
ls -al /Users | perl -pe 's/.*?([a-z]+)\/?$/uc $1/egi'
```

- regular expressions

  ```
  abc
  .
  *, +, ?
  ()
  ```

  useful for matching and subtstiution.