Issues about Memory & C

- **Bindings and lifetimes**
- **Static, stack, heap storage**
  - **Run time stack**
  - **Problems with heap storage**
  - **Garbage collection**
- **C**
  - **RAM architecture**
  - **Comparison to Java**
  - **Control-flow statements**
  - **Running C programs**
  - **Sample program**

Names, Bindings, Memory

- **Binding - an association between two items (e.g., a name and a memory location)**
- **Compile time - often layout of data in memory is chosen at this time,** ***static***
- **Link time - separately complied modules of a program are joined together by linker (e.g., adding in standard library routines for I/O)**
- **Load time - time when program is actually loaded into memory to run.**
  - **Virtual addresses versus physical addresses**
- **Run time - when program executes,** ***dynamic***

Binding Times - Choices

- **Earlier binding times -- more efficient**
- **Later binding times -- more flexibility in PL**
- **Examples of static binding:**
  - **Function signature types in C**
  - **Declared types in Pascal or Algol**
- **Examples of dynamic binding:**
  - **Methods called in Java or virtual calls in C++**
  - **Actual types of objects pointed to by a Java reference variable**

Lifetimes

- **Binding lifetime:**
  - **E.g., by reference variable passed to a Fortran subroutine**
    - **Variable exists before and after the subroutine call and the parameter/argument binding lifetime**
  - **E.g., object created by a new() in a Java program**
    - **Object exists after the return from the method that creates it. But if all its references were local to the called method, there is no way to reach the object -- GARBAGE. Here lifetime of object is longer than lifetime of reference binding.**
  - **E.g., recursive data structure created dynamically in a C++ function and then *dispose*d without resetting the pointer which referred to it.**
    - **Data structure doesn't exist and so pointer is pointing to garbage - DANGLING POINTER Here lifetime of data structure is shorter than lifetime of pointer binding to its address as a value.**

Kinds of Data Storage

- **Static data - given absolute address which is the same throughout execution**
- **Stack data - local storage allocated on a run time stack for use in a method or function; lifetime of stack variables is the time the method call takes to complete.**
  - **Needs a stack management algorithm during execution to manage storage for method calls**
- **Heap data - long-lived storage which is allocated and deallocated at arbitrary times during execution.**
  - **Needs the most complex storage management algorithm**

Examples of Static Data

- **Numeric constants**
- **String constants**
- **Tables of types in the program (e.g., Java inheritance structure) and other run time tables the compiler produces**

Examples of Stack Data

- **Parameters**
- **Local variables**
- **Compiler-generated temporaries (I.e., for expression evaluation**
- **Stack management information**

Runtime Stack

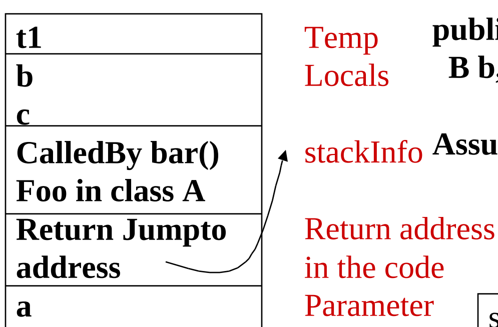
- ***Idea:*** **when method is entered, its frame is placed on the stack and *currentFramePointer* is updated. All accesses for local variables use this frame. When method is exited, frame is removed from the stack and *currentFramePointer* is updated.**
- **Stack contains *frames* of all methods which have been entered and not yet exited from**
- **Frame contains all information necessary to update stack when method is exited**
- **Addresses for local variables encoded as stack frame plus offset**

Frame Details

- **Fixed length portion (per procedure)**
  - **Return pointer into stack frame of caller**
  - **Return address (to code within caller)**
  - **Saved state (register values of caller)**
  - **Address accessing mechanism for nonlocal variables**
- **Variable length portion**
  - **Local variable storage (including parameters)**
  - **Compiler-generated temporary storage for subexpressions**

Frame Example

Scott, p110



public void method foo(A a){
B b,c; … a = new B();…return;}

Assume foo() is called by bar().

stackInfo contains pointer to bar’s frame, type information about foo, base address for foo’s frame.

foo’s frame on the runtime stack

Heap Storage

- **Heap allows allocation and deallocation of indeterminate sized blocks of storage during execution**
  - **E.g., objects, variable length Strings in Java, recursive data structures like lists and trees in C and Pascal**
- **Fragmentation of the heap - use of many small areas in the heap sometimes makes it impossible to allocate, even when the sum of the free space is enough.**

  Free list of heap blocks not in use. How to pick best block to allocate from?

Heap Storage

- **Need to compact storage to ensure have enough memory**
- **Need to deallocate storage after program is finished using it**
- **Some PLs have explicit deallocation commands, but all up to the programmer then (e.g., C, Pascal)**

Problems with Explicit Control of Heap

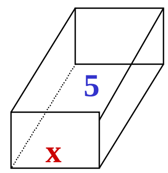
- **Dangling reference**
  - Storage pointed to is freed, but pointer(or reference) is not set to NULL
  - Then you are able to access storage whose values are not meaningful
- **Garbage**
  - Pointer(or reference) itself is freed (perhaps by execution going out of its declaring scope), but heap locations pointed to are not freed
  - Then, there is no way to access this heap storage
- **Memory leaks**
  - Failure to release storage builds up over time

Garbage Collection (GC)

- Implicit allocation and deallocation
- Overhead at run time to have a separate process run at same time as program
  - Analyzes usage of heap and recovers pieces of storage no longer reachable from user pointers
  - Execution time cost traded for easier job for user
  - Many, many kinds of GC algorithms - active CS research area today
  - E.g., functional languages such as Scheme, Java, Ada, Modula

C, An Imperative PL

- **Assignment as main operation**
  - **Names ⇔ Locations ⇔ Values**
  - **L-value: name labeling memory location**
  - **R-value: contents of memory location**
- **State of a computation**
  - **M: Locations -> Values**
  - **Remaining input**
  - **Output so far**



RAM: Random Access Machine

- **Normal control flows from one instruction to the next**
  - ***Thread of computation*: sequence of program points reached as execution flows through the program**
- **Control flow directs the thread without changing state**
- **Data flow (through assignment) affects the state without directly affecting the thread**
- **Imperative PLs have primitives close to the machine instructions (e.g., assignment, branch)**

Bird's Eye View: C versus *Java*

C

- **Types: int, double, char**
- **Pointer (to a value)**
- **Aggregates: array, struct**
- **Control flow: if-else, switch, while, break, continue, for, return, goto**
- **Logic operators: || && !**
- **Logical comparisons: == !=**
- **Numeric comparisons: < > <= >=**
- **string as char * array**

Java

- ***Primitive types: int, double, char, boolean***
- ***Reference (to objects)***
- ***Aggregates: array, object***
- ***Control flow: if-else, switch, while, break, continue, for, return***
- ***Logic operators: || && !***
- ***Logical comparisons: == !=***
- ***Numeric comparisons: < > <= >=***
- ***String* as an object**

C Control-Flow Stmts

- **if, with and without else clause**

  if (c == eof) break

  if (x != 5) continue else x = 10;
- **looping statements**
  - **while (Expr) Stmt**
  - **do Stmt while (Expr)**
  - **for (Expr; Expr; Expr) Stmt**
  - **break, continue**

C Control-Flow Statements

while ((c = getchar()) != eof) putchar(c);

Embedded side effects in expressions allowed!

for (j=0; s[j] == ' '; j++) ; empty statement is body of for

for (k=0; k<n; k++)

{ if (a[k]<0) continue; … }

- **switch, a form of case**
- **goto**

goto start

Structured Programming

- **Controversy in 1970's on PL design**
  - **Goto-less programming**
  - **Dijkstra versus Knuth**
- **Main idea: control flow should be obvious from syntactic structure of program**
  - **single-entry, single-exit loops**
  - **Fortran and C both had goto statements**
- ***Boehm-Jacopini Theory: any control structure can be expressed as a combination of composition, conditionals and while loops!***

C Example, hello.c

```
/*sample program to do hello world
  and print the numbers from 1 to 10*/
#include<stdio.h>
main (void)
{    int j, n;

     printf(" hello world\n");
     n = 10;
     for (j = 0; j <= n; ++j)
       printf(" %d",j);
     printf("\n the even numbers are: ");
     for (j = 0; j < 11; ++j)
       if ((j%2) == 0) printf(" %d",j);

     printf("\n");
}
```


C Example Output

```
/*when hello.c is run it looks like this:
scherzo!c> gcc hello.c
scherzo!c> a.out
 hello world
 0 1 2 3 4 5 6 7 8 9 10
 the even numbers are:  0 2 4 6 8 10
scherzo!c>

*/
```

Running a C program

- ***gcc <filename>*** **calls the GNU C compiler to compile the file.**
  - **This produces an executable *a.out* in the same directory**
  - **Type *a.out* at the prompt to run the C program**
- **When you have several .c files to compile, link and run you often use the -o option to give the executable a name**

  >gcc -o pgm test.c des.c

  puts the executable translation of the program in the files *test.c* and *des.c* in the file *pgm*

  >pgm

  runs the program