The Runtime Environment

- Remember basic architecture of compiler
  - Front end – scanning, parsing, semantic analysis
    - Independent of target machine
  - Back end – optimizing, code generation
    - Depends on target machine
- However, target environments share many characteristics
  - Possible to describe code execution environment independent of a specific target machine
  - This is called the runtime environment

Memory Management

- The runtime environment is all about use of memory
  - Memory for variables
  - Memory for function arguments
  - Memory for temporaries
- Memory needs depend on language characteristics
  - Scope and lifetime of variables (dynamic vs static scope)
  - Literal values, constants
  - Functions
    - Are they recursive? Can they be nested? Parameters?
    - . . .
Kinds of Runtime Environments

- Fully Static
  - Variables have fixed locations, statically allocated
- Stack Based
  - Variables put on processor stack during execution
- Fully Dynamic (Heap)
  - Variables dynamically allocated in memory during program execution under program control

Runtime Environment Examples

- Fortran, TINY
  - Fully static – all variables global in TINY
- Scheme
  - Fully Dynamic – all variables are pointers to heap
- Java
  - All objects dynamic, but primitives on stack
- C/C++
  - Uses all three types: global variables, local variables on stack, new'd variables on heap
Typical Program Layout

```c
int x = 1;  // global
g() {
    int * x = new int[10];  // heap
}
f(int x) {  // stack
    static int x;  // static
    {  // stack
        int x;  // stack
        g();
    }
}
main() {
    int x;  // stack
    f(x);
}
```

C Minus

- Global variables
  - Use static storage
- Local variables
  - Use stack storage (could we use static?)
- No pointers, no malloc, no new
  - No need for heap
Functions

- Independent of language, functions need:
  - Space for parameter values
  - Space for bookkeeping such as return address
  - Space for local variables
  - Space for temporaries
- Kept in an activation record
  - Record created for each call to function
  - Sometimes called a stack frame

Activation Record

- With each call, push record on stack
- Pop record at return
- Frame pointer (fp) - current record
- Stack pointer (sp) - top of stack
- Program counter (pc) - current instruction
Stack frames for gcd with 15,10

```
int x, y;

int gcd(int u, int v) {
  if (v == 0)
    return u;
  else
    return gcd(v, u % v);
}

int main() {
  cin >> x >> y;
  cout << gcd(x, y);
  return 0;
}
```

Calling Sequence

- **Caller**
  - Evaluates and fills in parameter values (push on stack)
  - Push return address on stack
  - Jumps to the function
  - On return, copies return value, pops parameters
- **Callee**
  - Push control link (current fp) on stack – this creates new record
  - Change fp to new record
  - Sees formal parameters in its scope
  - Pushes local variables, temps on stack
  - On exit, pop locals, restore fp from control link
- Note that the parameters could be considered part of old frame as well as new frame
- Frame pointer typically points between parameters and local variables
### Accessing Parameters and Locals

For a typical processor managed stack which grows from higher to lower memory addresses…

- **Access** is by **fixed offset** from frame pointer
  - We know the offsets at compile time, but we don't know the frame pointer at compile time
- **Positive offset for parameters**
  - They are above the fp
  - They must be pushed by caller in reverse order
- **Negative offset for locals, temps**
  - They are below the fp
  - Pushed by the code of the function
Accessing parameters and locals

int f(int x, int y) {
    int z = 42;
    return x + y + z;
}

_f:

pushl %ebp  \{ push fp, make stack new fp 
movl %esp, %ebp  \"allocate\" z on stack
subl $4, %esp   \ init z with 42
movl $42, -4(%ebp)  \ save y as return
movl 12(%ebp), %eax  \ add x
addl 8(%ebp), %eax  \ add z
addl -4(%ebp), %eax
movl %ebp, %esp \ restore stack, restore fp
popl %ebp
ret

Picture (32-bit architecture):

```
    16          12          8          4          0  -4
    +--------+-------+------+
    |       |       |     |     |     |
    +--------+-------+------+
    yOffset = 12
    +--------+-------+------+
    |       |       |     |     |     |
    +--------+-------+------+
xOffset = 8
    +--------+-------+------+
    |       |       |     |     |     |
    +--------+-------+------+
    xOffset = 4
    +--------+-------+------+
    |       |       |     |     |     |
    +--------+-------+------+
zOffset = -4
    +--------+-------+------+
    |       |       |     |     |     |
    +--------+-------+------+
```

fp(ebp)  \ control link

sp  \ return address

Spring 2012  
CIS 461/561- Runtime Environment
Questions

- Why not allocate entire activation record in caller instead of just the argument part?
  - Caller shouldn’t need to know about function's local data
  - Function may grow locals according to execution
  - Provides good separation – caller only needs to know about parameters (which it has in call)

- What about return values?
  - Register used (could treat as a parameter value)
  - Register restricts to scalar types unless pointers are used

- What about return address?
  - Pushed by 'call' instruction, popped by 'ret' instruction
  - Sits between parameters and control link

---

Array allocation on stack

```c
int f() { 
    int z[4];
    return z[0] + z[2];
}
```

```
pushl %ebp
movl %esp, %ebp
subl $24, %esp
    // space for array
movl -16(%ebp), %eax       // load z[2]
addl -24(%ebp), %eax       // add z[0]
movl %ebp, %esp
popl %ebp
ret
```

- size must be known at compile time
Array Location

Computed array subscripts

```c
int f(int i) {
    int z[4];
    return z[i];
}
```

```asm
 ----------
_f:       pushl %ebp
    movl %esp, %ebp
    subl $24, %esp
    movl 8(%ebp), %eax   // move i to eax
    leal 0(%eax,4), %edx  // mult by 4
    leal -24(%ebp), %eax  // load z addr
    movl (%edx,%eax), %eax // *(z+4*i) -> eax
    movl %ebp, %esp
    popl %ebp
    ret
```
Nested scopes

```c
void f() {
    int x;
    {
        int y = 2;
        {
            int z = 3;
            x = y + z;
        }
        {
            int w = 4;
            int v = 5;
            x = v + w;
        }
    }
}
```

Call by Reference

- Instead of copying values as parameters on stack, copy addresses
  - Callee requires extra level of indirection to get to value
  - But stack is smaller and callee can modify caller's data
- Arrays in C/C++ are passed by reference
  - But arrays in C/C++ are treated as constant pointers
- Objects in Java are passed by reference
  - Well, not quite since assignment in the callee is allowed, but has no effect on the caller's data
- C++ has syntax for true call by reference
  - Probably implemented with pointers, but compiler takes care of making it all work
Other Scenarios

- Nested functions
  - Inner function may use local variables of enclosing function
  - Need way to get to local variables of another function
  - Solution: provide access link to point to stack frame of enclosing function (set at time of call)
  - To access variables, follow links to correct stack frame
  - May require several link follows (chaining)

- Passing functions
  - Needs closures to capture environment

- Dynamic scope
  - No fixed offset, requires searching at runtime