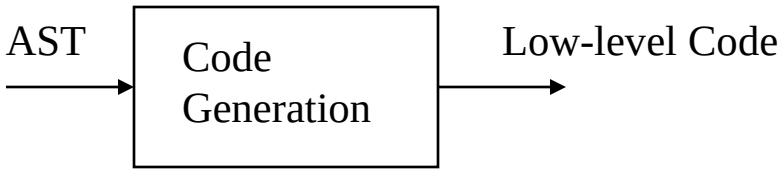
The Runtime Environment

- In the Analysis phase, we were concerned about understanding the source language
- In the Synthesis phase, we are concerned about converting the source into a new format, typically assembly language
  - Thus, the focus shifts from the source language to the target architecture

- The next phase of the compiler we will look at is Code Generation
  - In this phase, we walk the AST, creating a new data structure for low-level code



The Runtime Environment

- Here's what your main() might look like:

```
Parser myParser = new CMinusParser("test.cm");
Program ast = myParser.parse();
ast.print();
ast.semanticAnalysis();
LowLevel ll = ast.codeGen();

// then the rest of the back end
```

The Runtime Environment

- Code Generation is highly tied to the target architecture
  - It needs to put code in a format similar to assembly code

- Thus, we need to address some architectural issues before we look at Code Generation
  - This chapter explores the Runtime Environment
    - This is essentially our roadmap for how we will handle memory and register management so that we can interface smoothly with the hardware environment
    - The next chapter actually looks at Code Generation

- We will look specifically at :
  - How memory will be organized to make our program execute efficiently
  - How function parameter passing is accomplished

The Runtime Environment

- Let's look first at how the memory is organized and used by your program

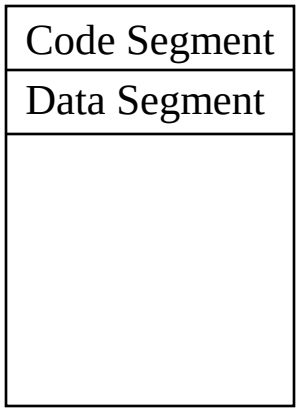
- Consider first the actual object code you generate for functions
  - Is the size of this code fixed after compilation?

  - We know exactly how much room to allocate for the portion of our program which actually contains the executable instructions
  - We call this area of memory the code segment

- What about global variables? Do they require a fixed amount of space in memory? Could we allocate space for them at compilation?

The Runtime Environment

- Memory for global variables is allocated statically, and is stored in the data segment



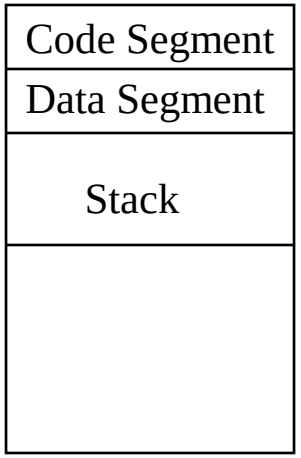
- What about local variables?
  - First, think about static local variables
  - What does it mean when I declare:

    static int count;
  - Static local variables can be allocated like globals, in the data segment
  - But non-static local variables are different
    - When I exit the function, its value is lost
    - If I have a recursive function, I need to have multiple unique copies of the same local variable
  - How do we handle this?

The Runtime Environment

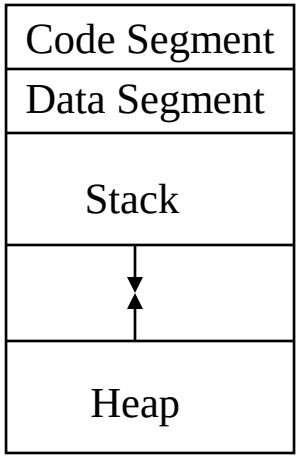
- Local variables are typically stored on the stack
  - In a data structure known as the activation record or stack frame
    - We have one of these for every currently active function

- We also allow dynamically allocated variables or objects
  - In C, we use malloc()
  - In C++ and java, we use new
  - These variables don't necessarily just live during the execution of a function
  - They die when a dispose command (C++) or when no one is pointing to them (java)
  - Where do we store them?



The Runtime Environment

- Dynamically allocated variables (or objects) are stored on the heap
  - This has nothing to do with the data structure heap we studied in Data Structures
  - It is simply a memory area, supported by special routines which can allocate blocks of a requested size
  - The Heap Manager hides these details
  - The heap suffers from fragmentation



- The CS and DS are fixed size
- The stack and heap are dynamic, growing and shrinking
  - Typically the stack starts at one end of available space, and the heap at the other
  - They grow toward one another, and if they hit, you get a nasty out of memory exception, or a crash

The Runtime Environment

- We mentioned the activation record
  - When a function is called, the compiler needs to create an activation record for this instance of the function
  - The activation record may contain space for
    - Function arguments
    - Local variables
    - Storing return address and other necessary bookkeeping information
    - Scratchpad space required by register allocator – the spill area
    - Space for the return value, if not a scalar
  - Some parts of the activation record require the same space regardless of the function
  - Most vary in size according to the needs of the function
    - But all records for a given function are same size

The Runtime Environment

- How the allocation record is handled varies quite a bit based upon the language
  - In Fortran compilers, it is typically put in data segment
  - In C++ and Pascal, it is put on the stack, and called the stack frame
  - In Lisp, it is put on the heap

The Runtime Environment

- Before we look further at why languages handle activation record differently, want to look briefly at what has to happen to call and return from a function
  1. The caller computes the args, and puts them somewhere the callee can find them (maybe its activation record)
  2. The caller may have to save registers the callee might trash
  3. The caller does a jump-subroutine command
     - Saves return address
  4. The callee must set up the activation record and save any necessary bookkeeping information
  5. The callee may have to save registers it will trash
  6. The callee then executes
  7. The callee restores registers it saved
  8. The callee releases the activation record
  9. The callee returns
  10. The caller restores registers it saved

Caller and Callee Registers

- Notice that both the caller and the callee may be responsible for saving registers (to memory)
  - At compilation, register allocation is typically just done on one function at a time (to save complexity)
  - The caller doesn't have visibility into what registers the callee will use
  - Likewise, the callee doesn't know what registers are "live" in the caller when the call occurred
  - Thus, the callee could trash registers the caller was using
  - Approaches for handling this
    - Caller could save all registers who are "live" across the call
      - Efficient if not many live registers
    - Callee could save all registers it uses
      - Good if it doesn't use too many registers

Caller and Callee Registers

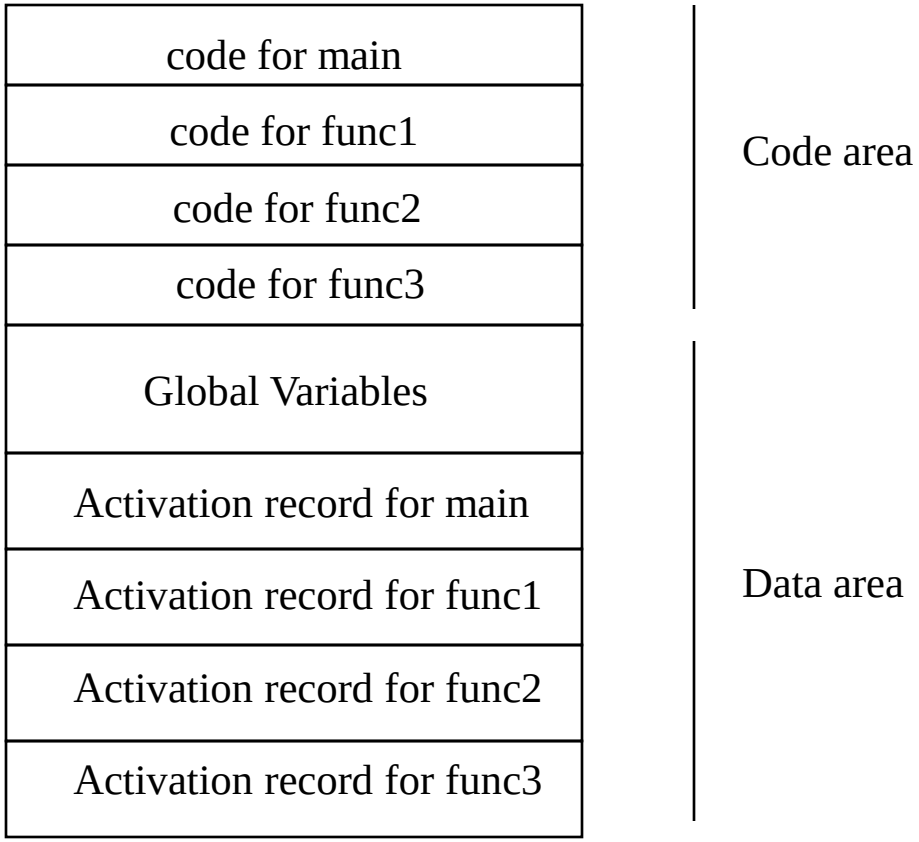
- Most architectures have a defined standard for how registers get saved because of functions
  - Most will use a combination of caller-save and callee-save
    - Makes for the most efficient approach
    - The register allocator tries to put values that span a function call into callee-saved registers – why?
    - The register allocator tries to put other values in caller-saved registers – why?
  - Your compiler must follow the standard called for by the architecture and O/S
    - Why?

The Runtime Environment

- We said that different languages handle activation records differently
  - Fortran typically stores them in the data segment
  - This is because all variable allocation in Fortran is static, and it uses a fully static runtime environment
- Two interesting features of Fortran
  - All variables are static
    - Local variables to a function retain their values between different calls to the function
  - No recursion allowed
- These two features contribute to the conclusion that we only need a single copy of the activation record for each function
  - We can determine the size at compile time, and thus can put the activation record in the data segment

The Runtime Environment

- A possible memory layout for a fully static runtime environment

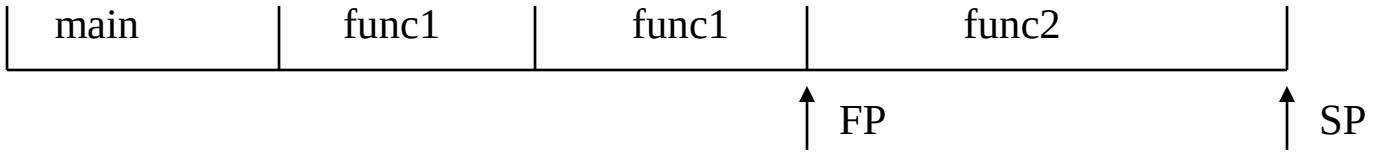


The Runtime Environment

- Most languages allow recursion, and don't use only static variables
  - Each function activation needs its own environment, complete with unique copies of its local vars
  - Thus, we need an activation record for each function invocation
  - Well, we could allocate these activation records from the heap
    - But this doesn't really imply any ordering of the records, while there is a logical ordering to the activations
    - When I return from a subroutine, I need to go back to the correct environment
  - This suggests the use of the stack to keep track of the activation records in a particular order
    - Text calls this a stack-based runtime environment

Stack-Based Runtime Environment

- When the stack is used for the activation records, we call this the runtime stack or the call stack
  - The activation record itself is called a stack frame
- During execution, we dynamically maintain a stack of the stack frames for all currently active functions
  - main() will be at the bottom of the stack
  - Currently executing function will have its stack frame at the top of the stack
  - Two pointers (sometimes dedicated registers) support the runtime stack
    - Stack pointer (SP) and frame pointer (FP)
    - A particular architecture may use one or both



Stack-Based Runtime Environment

- On different architectures, stacks may grow up or down
  - For X86, the stack starts at the highest memory addresses and grows down
    - A Push causes the SP to be decremented
- Regardless of which direction the stack grows, the SP is typically placed at the top of the stack, or end of the current stack frame
- The FP is typically used to mark the beginning of the current stack frame
- When program execution reaches a function call, code must be inserted to update the stack, adding a new stack frame
  - Both the caller and the callee can be involved in setting up the new stack frame
    - Typically the callee will do most of the work

Stack-Based Runtime Environment

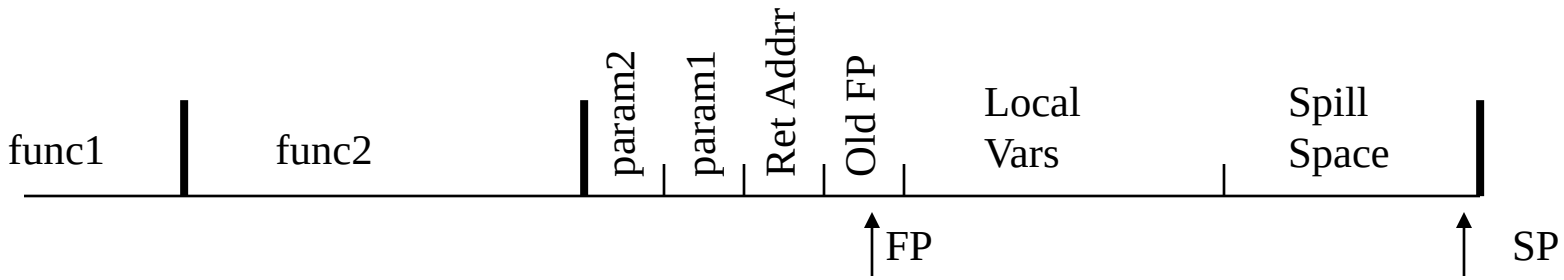
- Let's look again at some of the actions which must happen
  - The caller computes passed arguments, and puts them in standard location
    - For X86, that means pushing them onto the stack after his stack frame
  - The call is executed, saving return address into a register (or possibly straight onto stack)
    - Since the callee may in turn make calls, this register must be saved (to stack, if not done automatically)
  - Next we need to move the stack frame
    - Assume we move both SP and FP
    - We can move FP to where the SP is now
    - Because we know the size of the frame, we can move SP to appropriate spot
    - When exiting function, how will we be able to return the FP back to where it was?

Stack-Based Runtime Environment

- So, before moving the stack frame, we want to store the old FP
  - We simply save its location onto the stack
    - For X86, we do Push EBP
  - Now we can do FP = SP and SP = SP - frameSize
  - Stack frame is all set up and we can use it
- When we get ready to exit the function, we need to restore the old stack frame just like it was
  - We move SP down to FP (SP = FP)
  - Restore FP (Pop EBP)
  - Get the return address off stack and jump there, or just execute a Return, if available
- SP is still pointing after function arguments
  - Caller will have to fix it, either with pops or by adding paramSize to SP

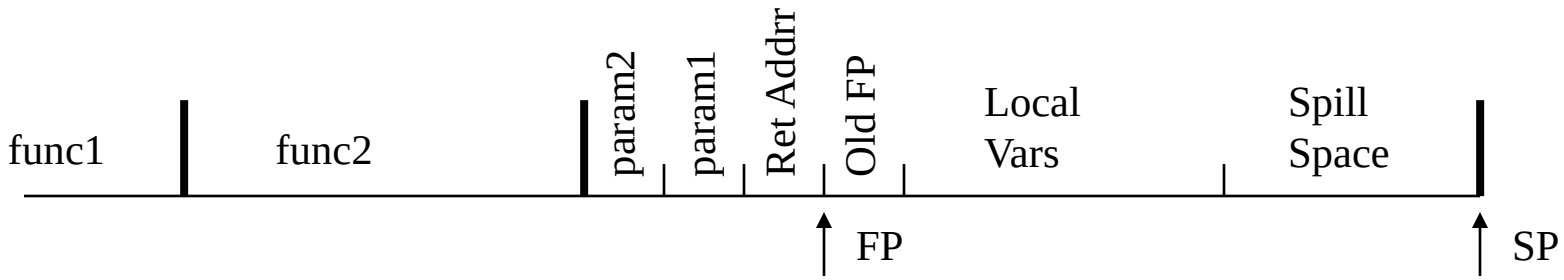
Stack-Based Runtime Environment

- So, that's the basic mechanism for updating a stack frame at function calls
  - For your project, we've abstracted most of this out into just FunctionEntry and FunctionExit
    - Subsequent pass can make architecture-specific
- We needed locations for parameters, return address, and old FP
  - We also need space for local variables, register spill space, object return space, etc.
- Stack frame is combination of all of this
  - I usually view function parameters as between frames



Stack-Based Runtime Environment

- If we want to reference incoming parameters, they would be positive offsets (assuming stack grows down) from FP
  - R1 = load (FP + 8)
- If we want to reference local vars, they are stored at negative offset
  - R3 = load (FP – 12)
- Alternatively, we could reference everything as a positive offset from the SP
  - FP is actually not needed
  - With only 8 regs in X86, not using EBP as FP is big gain



Stack-Based Runtime Environment

- We even had an optimization in the IMPACT compiler to reclaim the SP for general purpose register use in the case of leaf functions
  - Essentially, we declare a static allocation record in the DS for leaf functions
  - We could reference these locations as offsets from a global variable, and didn't need to use a register
  - We would save SP in this static area also, and then use it as a general purpose register throughout execution of this leaf function

Stack-Based Runtime Environment

- Example: A simple X86 assembly language program
  - X86 Assembly Language
  - Note: Leave does ESP = EBP, pop EBP

```
int func1 (int x, int y) {

    return (x+y);
}


int main () {

    int i, j, k;
    i = 1;
    j = 2;
    k = func1 (i, j);
    printf ("%d",k);
}
```

Stack-Based Runtime Environment

- So, that's the basics of building a stack frame
- As always, there are a few complications
  - Variable-length arguments
  - Compound statements introducing new scope and new variables
  - Nested subroutines

- Variable-length arguments
  - Consider printf
    - We can do: printf ("the sum is %d\n",k);
    - Or: printf ("the sum of %d and %d is %d\n",i, j, k);
  - At one call site, it passes 2 args; at another, 4 args
  - How does the callee know how many args got passed?

Stack-Based Runtime Environment

- Variable-length arguments (cont)
  - To help with this problem, C pushes parameters in reverse order
  - Note that in our X86 example, the first parameter was always pushed last, and will thus be located at FP + 8
  - This 1st parameter must give an indication of how many more parameters to expect
  - A related complication is for languages which allow an array to be passed as an argument
    - How do you find beginning of array, as function of SP?
    - C always passes as pointer, so it doesn't have problem
    - One solution is to store array on stack, pushing a pointer to this array as the array parameter
      - Adds additional level of indirection to all array references

Stack-Based Runtime Environment

- Additional declarations inside compound statements
  - In C++, you can declare variables anywhere
  - Complicates scope analysis during Semantic Analysis
  - Not really a big issue during Code Generation
    - All function variables, regardless of scope, treated the same
      - May be stored on stack, or in register
    - Some care needs to be taken during Code Generation to reference the correct version of a variable name which is overloaded

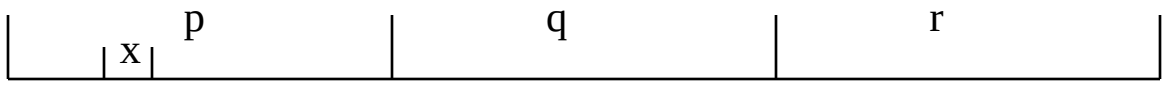
Stack-Based Runtime Environment

- Nested subroutines
  - C doesn't allow this, but Pascal does
  - Inside a function, you are allowed to declare another function, whose name only is valid within outer function's scope
  - Problem: procedure r changes the variable x, but it isn't stored in its own stack frame
    - It isn't even stored in the previous frame
  - "What are we gonna do?"

```
procedure p;
   var x: integer;

   procedure r;
      begin
      x := 2;
      end (* r *);

   procedure q;
      begin
         r;
      end (* q *);

   begin
      q;
   end (* p *);
```

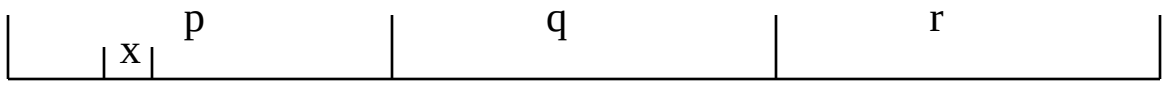
Stack-Based Runtime Environment

- Nested subroutines (cont)
  - We are inside procedure r, and need to write to a variable in some enclosing scope
  - Any variable we can write to must be in our enclosing scope
    - r couldn't write to a variable in q
  - Solution is to maintain an extra pointer, to enclosing scope
    - Called an access link or static link
    - Note that if r is executing, p has to be on the stack frame somewhere below it
    - If r has a pointer to p's FP, then it could access an offset from the FP



Stack-Based Runtime Environment

- Nested subroutines (cont)
  - As part of the call sequence, p must pass it's FP to q as the static link – could just be passed similar to parameters
  - When q calls r, it notes (at compile time) that r is not one of it's local variables, so it passes its parent static link on
  - The situation is a bit more complex if r was nested inside q rather than being siblings
    - q would pass its own FP as static link to r
    - For r to access x, it would have to follow 2 static links
      - No big deal, compiler can determine how many scopes it has to go



Stack-Based Runtime Environment

- Nested subroutines (cont)
  - The situation gets a bit more ugly if we allow procedures to be passed as arguments
    - It may make determining the static link impossible
  - Solution is to pass the static link as an argument along with the procedure
  - When q calls p(r), p has no way of knowing r's static link
    - q passes its FP to be used as the static link
  - When p calls a, it uses this static link

```
procedure p(procedure a);
   begin
      a;
   end (* p *);

procedure q;
   var x: integer;
   procedure r;
      begin
         writeln(x);
      end (* r *);

   begin
      x := 2;
      p(r);
   end (* q *);

begin
   q;
end (* main *);
```

Dynamic Runtime Environment

- Most languages use a stack-based environment
  - However, some languages have features which require an alternate solution
- Consider the following, illegal C

```
int * illegalReturn ( ) {
  int x;
  return &x;
}
```

- If a language requires you to be able to return a local variable, a stack-based approach wouldn't work
  - You need to have the activation record stored somewhere where it would survive beyond the lifetime of the function
  - Implies that the activation record should be stored on the heap – and stay active until it is no longer referenced

Dynamic Runtime Environment

- Languages such as LISP and ML allow constructs such as the previous example
  - All activation records are allocated dynamically when the function is called
  - When the function exits, the activation record lives on until it is no longer referenced and garbage collection can clean up

- It is certainly most important that you understand the stack-based environment, and how stack frames are built and maintained
  - But just be aware that you must be able to adapt the approach you choose to the features of requirements of the language being compiled

Parameter Passing

- The next major topic we will cover is Parameter Passing
  - We have already looked at it briefly
  - Address how values are sent to a subroutine, and how the return value is sent back

- First subject we will look at is order of evaluation of parameters
  - Consider the code below – the order of evaluation is critical
  - We already said we will likely pass parameters right-to-left
  - However, you may find the left-to-right evaluation is required to correctly execute the code
    - In C, the evaluation order isn't specified, and it's up to the compiler writer

```
func1 (++x, x);
```

Parameter Passing

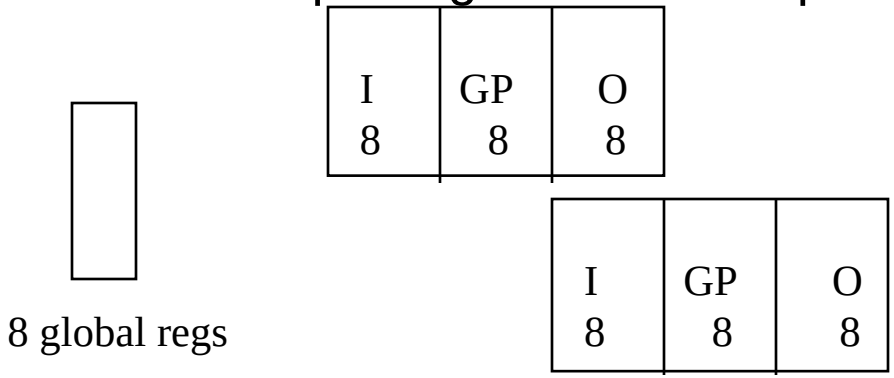
- So, how do we pass parameters to a subroutine?
- Several mechanisms
  - Pass via memory
    - This was the common technique in early processors which had few registers
    - All parameters are passed via the stack, just as we saw in the X86 example
    - Parameters frequently passed right-to-left
    - Architectures frequently support push and pop operations for moving parameters onto/off of stack
    - Parameters can be accessed as offsets from FP or SP
    - For larger size data (doubles, structs), must ensure that data maintains proper alignment
      - e.g., doubles may have to be on an 8-byte boundary
      - Both caller and callee must understand these rules

Parameter Passing

- Parameter passing mechanisms (cont)
  - Pass via registers
    - Processors which have a large number of registers frequently pass parameters in registers
    - More efficient, since it avoids expensive loads/stores
    - 1st parameter always in a particular register
      - If separate integer and floating point register sets, first integer parameter always in particular register and first floating point parameter in particular register
    - Easy to access data
    - Don't have to worry about order of passing the parameters – can set registers in any order

Parameter Passing

- Parameter passing mechanisms (cont)
  - Pass via registers (cont)
    - Can't pass everything via register
      - Usually only a certain # of regs designated
        - Remainder of parameters passed on stack
      - Some parameters too big – structs
  - Register Windows
    - Used by Sun Sparc – have up to 520 total regs
    - Input registers and output registers overlap



Parameter Passing

- You are (or should be) already familiar with the concepts of pass by value and pass by reference
  - There are some minor variations on these two themes which the text discusses, but they aren't too common

- Pass by value
  - When a pass by value call is made, the variables are copied and placed in appropriate location for passing
  - These copies can then be used by the callee as local variables (Bad coding practice - I don't recommend this!!!)
    - Can overwrite them
  - Upon exit, these copies just get forgotten about, just like any other local variable
  - Originals in caller unchanged
  - Everything in C is pass by value
    - However, if a pointer is passed, then you can side-effect

Parameter Passing

- Pass by Reference
  - In some languages, this is the only option (Fortran)
  - In C++ you can create reference parameters by using the &
    - int func1 (int &a);
  - Pascal uses the word var
    - procedure func1 (var a : integer) : integer;
  - In practice, the compiler will simply take the address of the parameter, and pass that. The callee will then deference the address to access the actual variable
  - Variable being passed must have a memory location to pass
    - Can't just be allocated to a register for its entire life

Parameter Passing

- Pass by Reference (cont)
  - How would the code generated to support these two functions differ?

```
void func1 (int &a) {
   a++;
}

…

func1 (x);
```

```
void func1 (int *a) {
   (*a)++;
}

…

func1 (&x);
```

  - View example

Return Values

- OK, so that's how we pass values
  - How do we return values from a function?

- If the value is a scalar (int, float, double, pointer), it is typically returned via register
  - X86 returns via EAX register

- If the variable to be returned is something like a structure, array, or object, then you have a bit of a problem
  - Java avoids this by making all objects reference
  - In C/C++ must be able to return objects
    - Callee shouldn't allocate space within it's stack frame
      - It will be de-allocated
  - How could you handle this problem?

Return Values

- A couple of solutions
  - Create space in data segment
    - Would have to be a large as biggest return value in entire program
    - Alternatively, could create a return space in data segment for each function which needs it
  - Create space in the caller's stack frame
    - Caller knows size
    - Passes pointer to where he wants the return value stored, as a parameter to callee
    - Callee simply writes info straight into caller stack frame
  - Regardless, caller is required to move the data from the return location immediately upon return from the subroutine