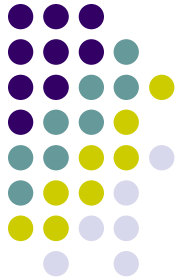
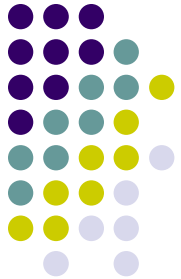
Functions?



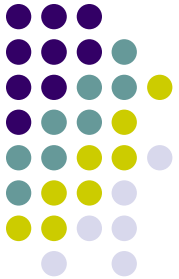
- The main idea is to group several commands into one. Consequence:
  - Simpler, and clearer, code
  - Simpler, and more reliable code reusage
- Declaration – type of return value, parameters, specifiers
- Definition – same as declaration, but includes function body

What does this code do?



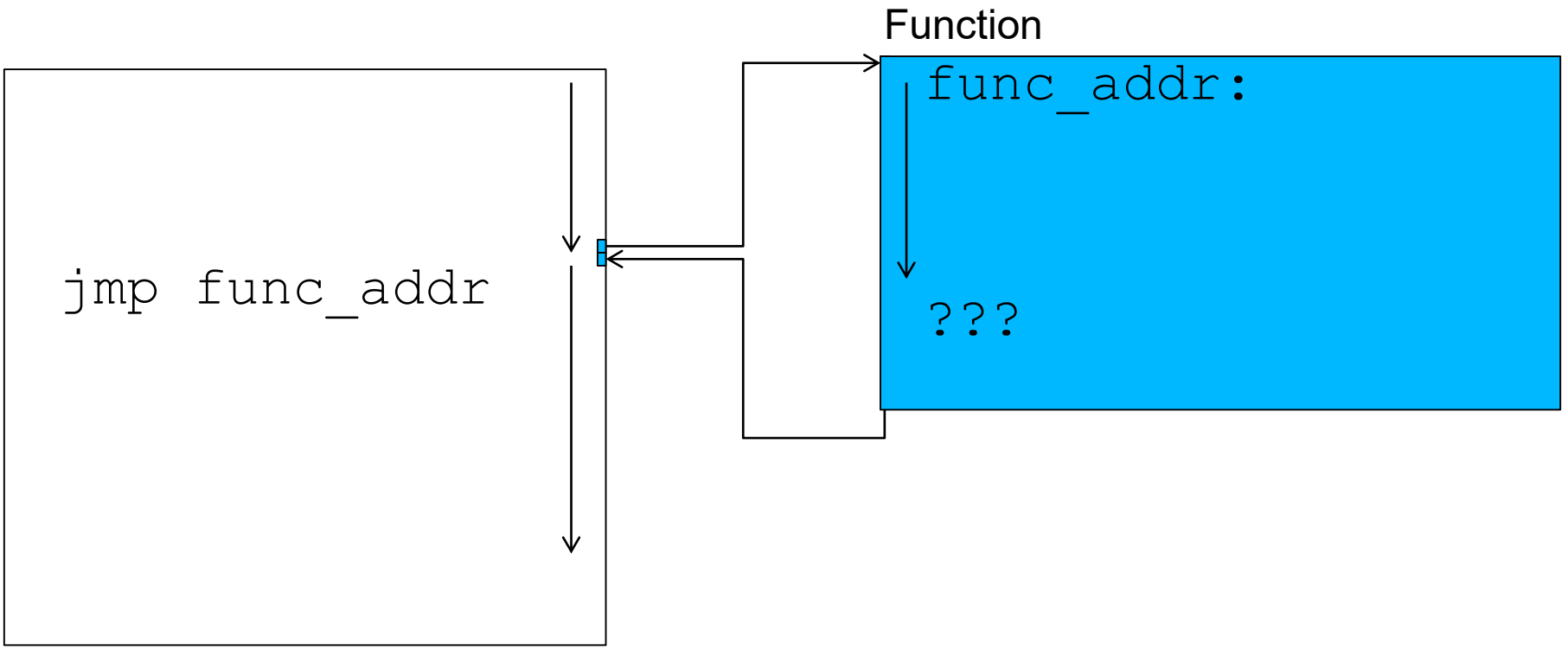
```
void foo(int** mat, int n, int m)
{
  int k;
  for (k = 0; k < m; k += 2)
  {
    int i;
    for (i = 0; i < (n - 1); i++)
    {
      int j;
      for (j = i + 1; j < n; j++)
      {
        if (mat[k][i] < mat[k][j])
        {
          int tmp;
          tmp = mat[k][i];
          mat[k][i] = mat[k][j];
          mat[k][j] = tmp;
        }
      }
    }
  }
  for (k = 1; k < m; k += 2)
  {
    int i;
    for (i = 0; i < n; i++)
    {
      mat[k][i] = 0;
    }
  }
}
```

It does this

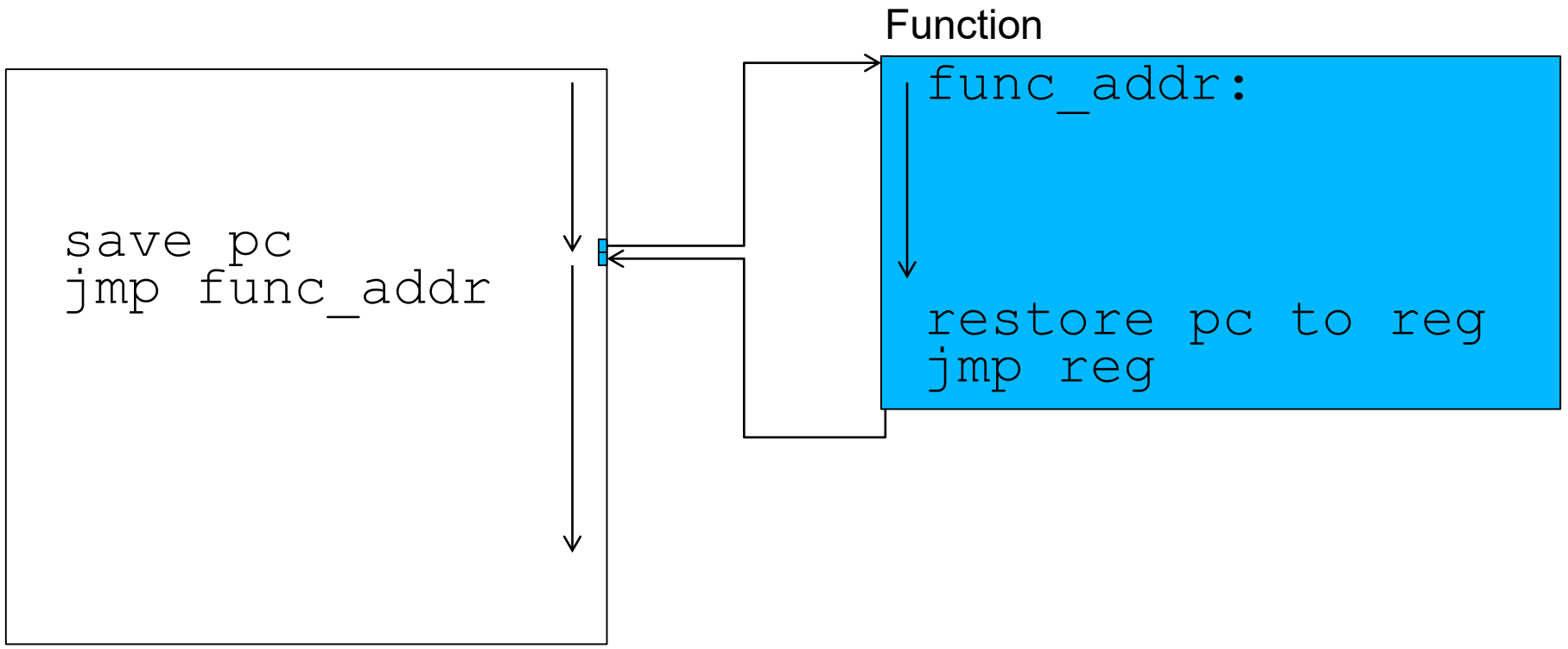


```
void foo(int** mat, int n, int m)
{
  int k;
  for (k = 0; k < m; k += 2)
  {
    sort(mat[k], n);
  }
  for (k = 1; k < m; k += 2)
  {
    zero(mat[k], n);
  }
}
```

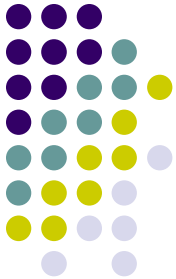
Functions



Functions


Function
func_addr:
save pc
jmp func_addr
restore pc to reg
jmp reg

Definition and declaration of functions



```
type-specifier return-type function-name(parameters)
{
  declarations
  statements
  return value;
}
```

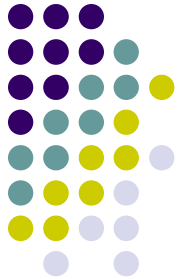
- **type-specifier** – determines visibility of functions (`static` or nothing)
- **return-type** – return value type; `void` if there is no return value
- **function-name** – unique function name (function and variable of the same scope can not have the same name)
- **parameters** – comma separated list of declaration of variables that represent function parameters
- **return value;** - value which will be returned by function (not needed if `void`)

```
return-type function-name(parameter-types)
```

- **parameter-types** – comma separated list of types; here you do not have to give names of the parameters (but, you should do it anyway).

```
int foo(float, int, const long*);
```

Parameter passing

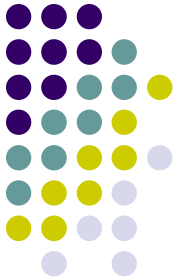


- When function is called formal parameters are replaced with real parameters (also called “arguments”).

```
void foo(float x, int y);
foo(15, 6);
foo(40, a);
```

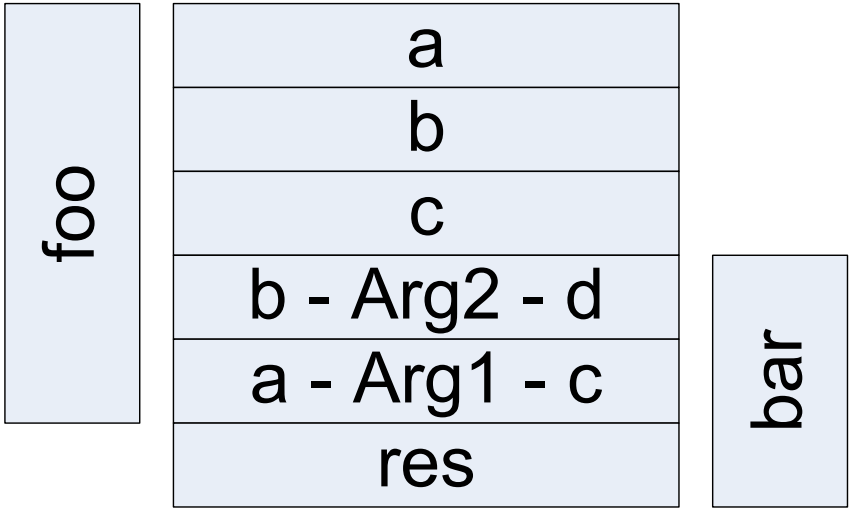
- There has to mechanism which passes values, i.e. parameters, to function, when it is called.
- Conceptually there are two kinds of parameter passing:
  - By value – function receives copy of real parameters, i.e. values that real parameters have at the calling point. Consequence: changing formal parameter from within the function will not change real parameter.
  - By reference – function receives real parameter directly. i.e. information about where real parameter is. Consequence: changing formal parameter will reflect on real parameters.

Passing parameters by value



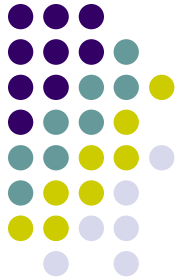
```
int bar(int c, int d)
{
    int res = c + d;
    c = 3;
    d = 7;
    return res;
}

int foo()
{
    int a = 5, b = 9, c;
    c = bar(a, b);
    c = c + a + b;
    return c;
}
```



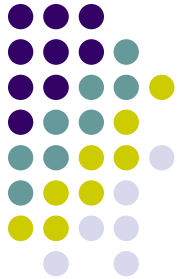
- **bar can not change a and b from function foo.**
  c and d are totally different variables, which are at function call set to values that a and b have at that point.
- **Return value from foo: 28**

Passing parameters by reference



- **Doesn't exist in C!**
- In C all parameters are passed by value.
- Question: Then, how function can influence the "outside world", i.e. how does it change it?
  1. By return value – caller function can do something with returned value, but it can ignore it too.
  2. By global variables – every modification of a global variable will influence the world outside the function. This is called "side effect". Many mathematicians consider this to be a great sin, but their teachings are not dogma in the engineering church.
- However, this is not enough. Through return value you can influence only one thing, and global variable which the function modifies can not be easily changed from call to call.

Passing parameters by reference



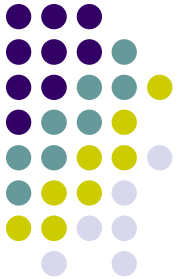
- Solutions:
  - Structures as return values
  - **Using pointers**
- When you pass pointers in order to access objects outside function, it is called “passing by reference”, because the effect is very similar.

```
void foo(int* x)
{
  *x = 5;
}
```

```
int bar(int* x)
{
  return *x + 5;
}
```

```
int fiz(int* x)
{
  return x[3] + x[6];
}
```

Passing parameters by reference



- Solutions:
  - Structures as return values
  - **Using pointers**
- When you pass pointers in order to access objects outside function, it is called “passing by reference”, because the effect is vey similar.

```
void foo(int* x)
{
  *x = 5;
}
```

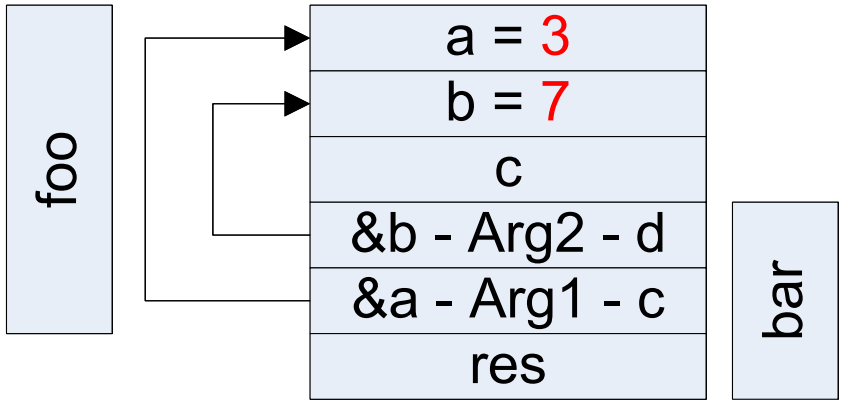
```
int bar(const int* x)
{
  return *x + 5;
}
```

```
int fiz(const int* x)
{
  return x[3] + x[6];
}
```

Passing parameters by reference

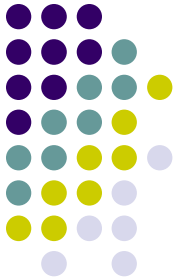
```
int bar(int* c, int* d)
{
    int res = *c + *d;
    *c = 3;
    *d = 7;
    return res;
}

int foo()
{
    int a = 5, b = 9, c;
    c = bar(&a, &b);
    c = c + a + b;
    return c;
}
```



- **Second and third line of function bar have effect to the outside world**
- **Return value from foo: 24**

Efficiency of passing by value



- Real parameters must be copied on some other place in memory, or some appropriate register.
- In any case, instructions are spent on it, not to mention memory.
- Becomes really noticeable when variables are big.
- Example for MIPS architecture:

```
struct s
{
  int array[7];
};

int func(int a, struct s p, int e)
{
  return e;
}
```

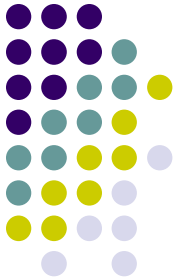
```
struct s
{
  int array[70000];
};

int func(int a, struct s p1, int e)
{
  return e;
}
```

```
    lw      $2,32($sp)
```

```
    li      $2,262144
    addu    $2,$sp,$2
    lw      $2,17860($2)
```

Return value



- Similar to passing parameters, only opposite direction.
- Every function, except `void`, must have `return` command.
- Can we pass return value „by reference”?

```
struct S
{
  int val1;
  float val2;
};

struct S func()
{
  struct S res;
  res.val1 = 1;
  res.val2 = 2.0;
  return res;
}
```

```
struct S
{
  int val1;
  float val2;
};

struct S* func()
{
  struct S res;
  res.val1 = 1;
  res.val2 = 2.0;
  return &res;
}
```

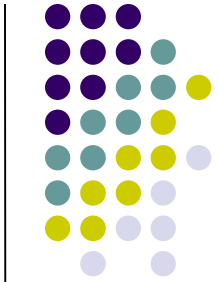
```
struct S
{
  int val1;
  float val2;
};

struct S* func(struct S* res)
{
  res->val1 = 1;
  res->val2 = 2.0;
  return res;
}
```

```
struct S* func()
{
  static struct S res;
  res.val1 = 1;
  res.val2 = 2.0;
  return &res;
}
```

```
struct S* func()
{
  struct S* res;
  res = (struct S*)malloc(
    sizeof(struct S));
  res->val1 = 1;
  res->val2 = 2.0;
  return res;
}
```

Moral of the story



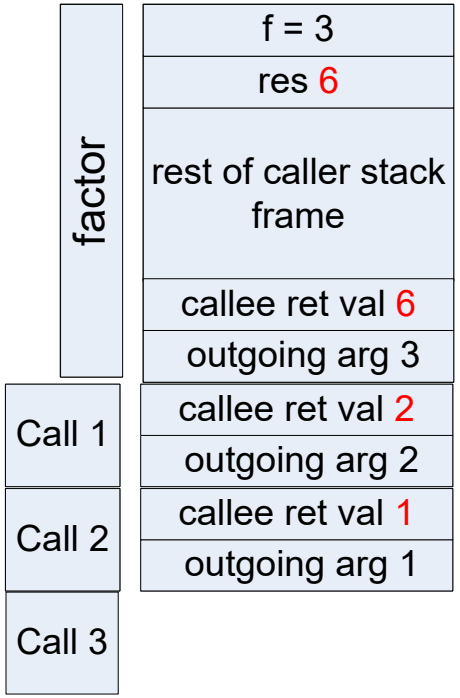
- Think good about what is the most efficient way to pass parameters and return values.
- For correct decision it is necessary to know the target architecture and the calling convention.
- General rule for embedded systems, due to their limited resources, is that you should not overburden a function with parameters, and you should pass bigger objects either “by reference” or sometimes even by global variables.

Recursion

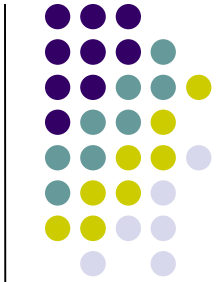
- When function calls itself.
- Can be direct and indirect.
- One of the two reasons why program stack is used (what is the other?)

```
void caller()
{
  int f = 3, res;
  res = factor(f);
  printf("factorial %d = %d\n", f, res);
}


long factor(long n)
{
  if (n <= 1) /* terminal condition*/
    return 1;
  else
    return(n * factor(n - 1));
}
```



Recursion



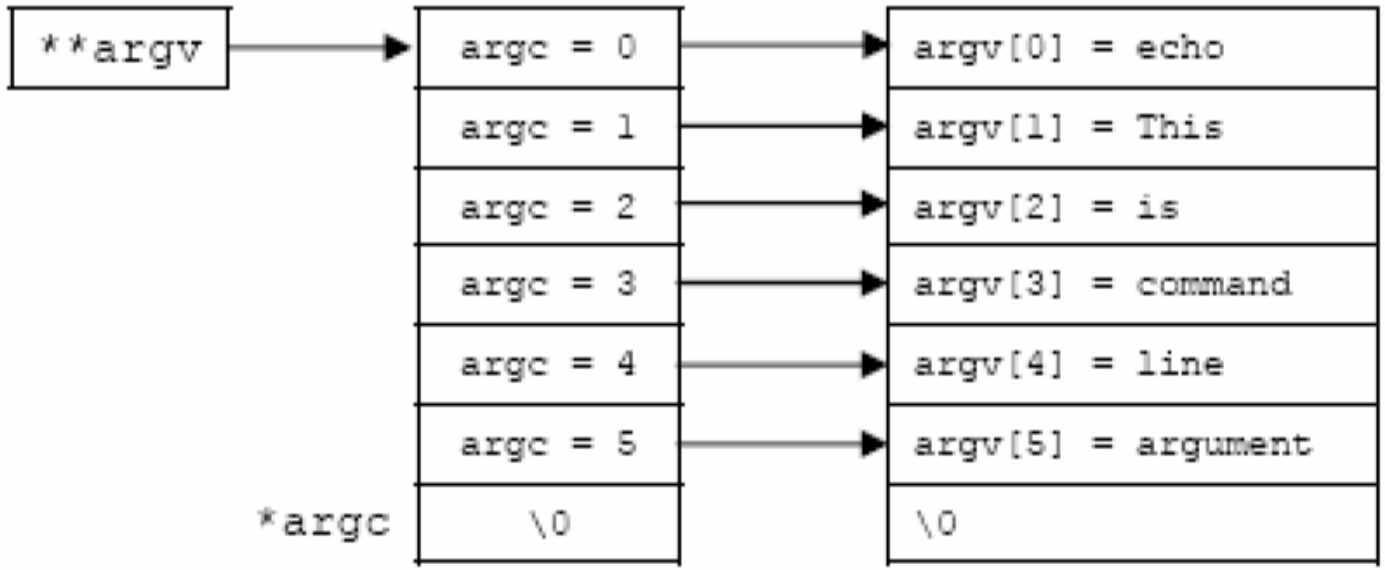
- Recursion is tightly related to stack, and efficiency of working with stack is not good on some embedded platforms.
- Also, some systems have hardware support for function calls and then the depth of call stack is limited by that.
- And finally, recursion is rarely needed in general, and especially in embedded systems. In case that it is needed, it is better, and safer, to manually construct stack and control it directly from the code.

main function

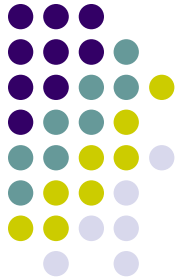
- Starting function
- Declaration: int main(int argc, char** argv)

void main()
void main(int argc, char** argv)
float main(long djura) ?!?!?!?!?!?
...

```
C:\>echo This is command line argument
This is command line argument
```

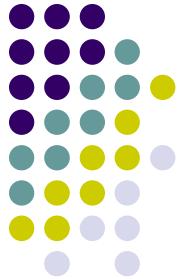


Calling convention 1/2



- Calling convention determines relationship between caller and called function.
- It defines two things:
  - Mechanism of parameter and return value passing
    - Which resources are used for that purpose
      - If registers: relation between order of parameters and concrete registers
      - If stack (memory): relation between order of parameters and order on stack (or address in memory)
    - Resource attached to parameter depends on parameter's position in parameter list and its type
  - Which resources can be clobbered by which function (caller or called)
    - Which registers will be guaranteed to have the same value before and after a function call
    - Who will allocate and free stack for parameters and return values
- Calling conversion is part of ABI (Application Binary Interface)
- ABI defines: type sizes and memory alignment, calling convention, things related to system calls, binary format of object file...

Calling convention 2/2



- Calling convention is related to particular platform (combination of processor, operating system, and partly compiler)
- On the same platform you can have several different calling conventions.
- Reasons:
  - different programming languages
  - different compilers
  - compiler optimizations
  - code purpose, etc.

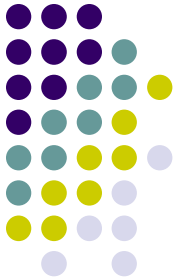
- Different calling conventions lead to problems in case you want to combine code created in different ways.
- To properly combine code ABI has to be satisfied, and the calling convention is the most important part.
- The most common case of code combination is when you use libraries, or combination of C and assembly code (function written in assembly is called from C code, or vice versa)

Mixing C and assembly

- Reasons for mixing:
  - Something is already written in assembly
  - For some things compiler is not efficient enough
  - Access to certain hardware features is not possible through C

- To kinds of mixing:
  - Writing assembly in a separate file
    - Interface is only on function call level
    - Calling convention has to be satisfied
  - Using in-line assembly
    - Assembly code in the same file with C code
    - Has to be supported by compiler
    - Many different approaches, since it is not part of the standard
    - Assembly can be used only within function body

Calling C function from assembly



- Assembly code has to satisfy calling convention
- Besides, we have to know how C compiler decorated names
  Usually it is by adding _ at the name beginning
- Example of calling printf from assembly

```
    global   _main
    extern   _printf

    section .data
    text     db       "Hello World!", 10, 0
    strformat db      "%s", 0

    section .code
    _main
      push     dword text
      push     dword strformat
      call     _printf
      add      esp, 8
      ret
```

- _main symbol has to be declared as public (“global”)
- _printf symbol has to be declared as sextern

Calling assembly function from C

- Calling convention has to be satisfied.
- For assembly, it is programmers responsibility, for C code compiler will take care of things
- Example:

```
int sum(int a1, int a2);

int a1, a2, x;
x = sum(a1, a2);
```

```
_sum
  push ebp            ;save bp
  mov ebp, esp        ;new frame
  mov eax, [ebp+8]    ;take 1. arg
  mov ecx, [ebp+12]   ;take 2. arg
  add eax, ecx        ;
  pop ebp             ;restore bp
  ret                 ;return
```

- By calling convention, return value should be in EAX register

In-line assembly 1/2

- GCC offers mechanism of “asm statement” for using assembly code directly in C functions.
- There are some other mechanism in some other compiler, but GCC’s approach is probably most widely spread.
- Syntax of GCC’s asm statement:

```
asm(assembler template
    : output operands                   /* optional */
    : input operands                    /* optional */
    : list of clobbered registers       /* optional */
    );
```

- assembler template – character string with assembly code
- output operands – C operands (expressions) that will be changed by the assembly code
- input operands – C operands (expressions) whose values will be used by the assembly code
- clobbered registers – registers whose values will explicitly be changed by the assembly code

In-line assembly 2/2

```
int func(int in)
{
int res;
asm ("movl %1, %%eax; \
      movl %%eax, %0;"
     :"=g"(res)          /* output */
     :"r"(in)            /* input */
     :"%eax"             /* clobbered register */
     );
return res;
}
```

- %n – reference on n-th operand in the list of output and input operand (first index is 0)
- %% – really write %
- “xy”(c_expression) – connecting C variable with value used in inline assembler

a,b,c,d	eax, ebx, ecx, edx respectively
S, D	esi and edi respectively
I	constant value (0 to 31)
q	dynamically allocated register: eax, ebx, ecx, edx
r	dynamically allocated register: eax, ebx, ecx, edx, esi, edi
g	eax, ebx, ecx, edx or variable in memory
m	memory
=	will be used for storing data