Functions

Procedures and functions in C/C++

- function definition

```
type identifier(parameters)
{
   body
}
```

```
int area(int x, int y)
{
  return x*y;
}
```

– the result is returned by **`return`**

- local variable can be declared in function:

```
int area(int x, int y)
{
    int S;
    S = x*y;
    return S;
}
```

- procedure
  - the function that returns no value

```
- void print(int x)
  {
    printf("%+06d",x);
  }
```

- function that returns **int** (in C)

```
my_function(int x)
{
   …
}
```

- procedure/function without parameters

```
void print_underlines(void)
{
  printf("---------------");
}
```

Differences in C and C++

- the returned type must be always declared in C++
  - `my_function(int x) { … }` causes an error in compilation in C++
- if the programmer forget return in function body, C compiler writes only warning, C++ error
- the function prototype must exist in C++

Function prototype (header)

- declaration

  `type identifier(parameters);`

  without body
- prototypes are typically in header files

```
int area(int x, int y);
void print(int x);
void print_underlines(void);
```

- prototype in C

```
void print_underlines();
```

  means that no information about parameters is declared

- prototype in C

```
void print_underlines(void);
```

  means that function has no parameters

- **both prototypes in C++ means the same: function has no parameters**

- the compilation in C is only with warning (calling function sum without prototype) – passed, but in C++ causes error:

```
void main(void)
{ int x;
  x= sum(3,4);
}
int sum(int a, int b)
{
  return a+b;
}
```

- correctly in C++:

```
int sum(int a, int b)
{
  return a+b;
}
void main(void)
{ int x;
  x= sum(3,4);
}
```

- or

```
int sum(int a, int b); //prototype

void main(void)
{ int x;
  x= sum(3,4);
}

int sum(int a, int b)
{
  return a+b;
}
```

Passing parameters to functions

Question:

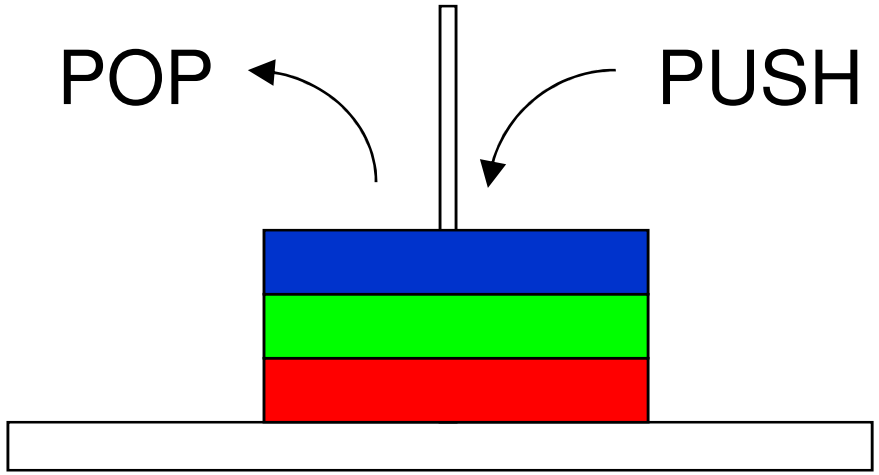
How are real parameters passed to functions?

Answer: Using **stack**

- parameters are passed by values in C (call by value), i.e. copy to stack
- ”output“ parameters of functions are realized by pointers

Stack

- structure called LIFO (last in – first out)
- the last element stored to the stack is read (removed) as the first one
- operations over stack:
  - PUSH (storing element to the top of the stack)
  - POP (removing element from the top of the stack)



- stack is used also to store intermediate data, it is automatically used by calling functions/procedures
- instruction CALL *address*
  - the processor automatically stores return address to the stack and jumps to the *address*
- instruction RET
  - return address is removed from the stack and processor returns to this address

```
int area(int a, int b)
{
  return a*b;
}
```

function area

1234: ...

1240: MUL EAX,EBX

1242: RET

```
void main(void)
{
  x = area(3,4);
}
```

main program

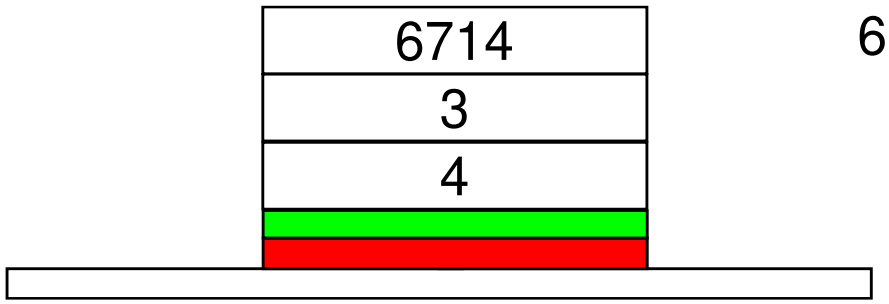
6709: push 4

6710: push 3

6711: call 1234

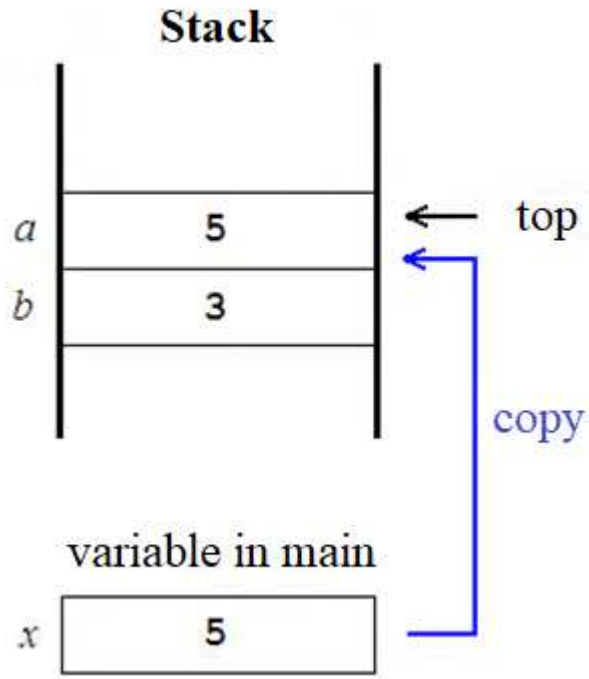
6714: pop

6716: pop

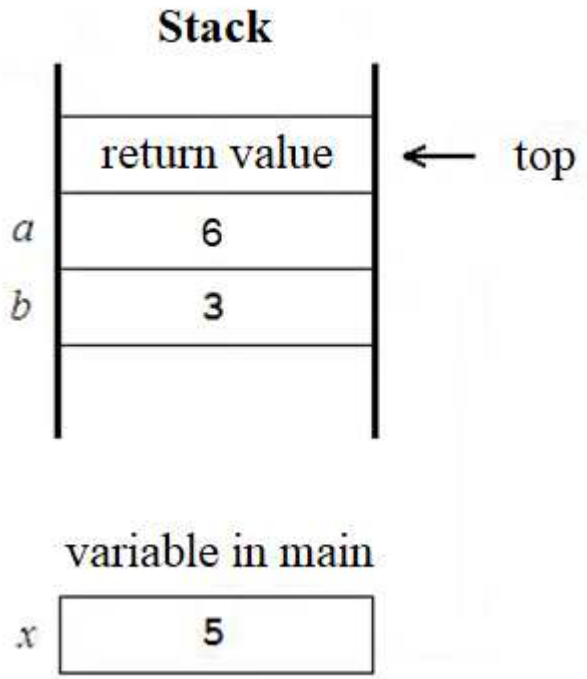


```
void print(int a, int b)
{  a++;
   printf("%d\t%d",a,b);
 }
void main(void)
{  int x = 5;
   print(x,3);
}
```

Before calling print



Before return to main



We write two functions computing area and circumference of rectangle:

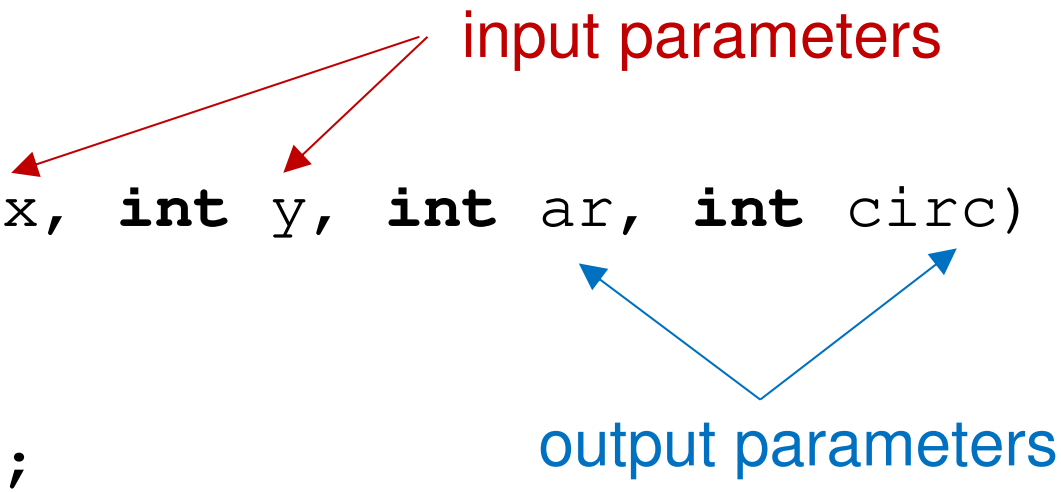
```
int area(int x, int y)
{
  return x*y;
}

int circumference(int x, int y)
{
  return 2*(x+y);
}
```

Now we want to write only one subprogram (subroutine – function or procedure) which computes both results

- but function returns only one result

- **the solution:** output parameters of procedures




```
void calcul(int x, int y, int ar, int circ)
{
  ar = x*y;
  circ = 2*(x+y);
}
```

Is it correct? No!

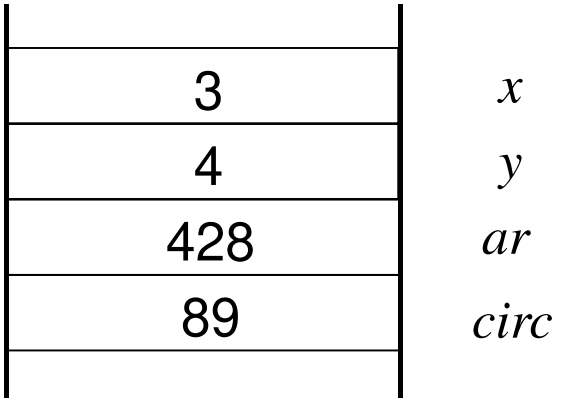
```
void calcul(int x, int y, int ar, int
  circ)
{
   ar = x*y;
   circ = 2*(x+y);
}

void main(void)
{
   int area, circumference;
   calcul(3,4,area, circumference);
}
```

```
void calcul(int x, int y,
        int ar, int circ)
{
  ar = x*y;
  circ = 2*(x+y);
}
```

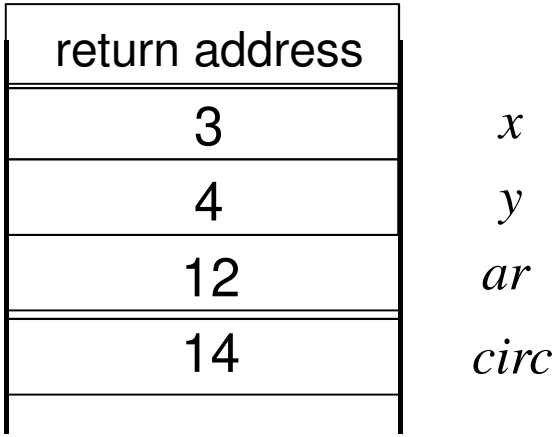
```
void main(void)
{
 int area,circumference;
 calcul(3,4,area,circumference);
}
```

Before calling procedure



area: 428 circumference: 89

Before return to main



area: 428 circumference: 89

- the procedure must work with original variables *area, circumference,* not with copies of values on stack
  - addresses of real parameters *area, circumference* must be pushed to the stack
  - some programming languages (Pascal) have special output parameters (*var* parameters in Pascal – reference parameters), but not C
  - output parameters must be defined as pointers in C

```
void calc(int x, int y, int *ar, int *circ)
{
  *ar = x*y;
  *circ = 2*(x+y);
}

void main(void)
{
  int area, circumference;
  calcul(3,4,&area,&circumference);
}
```

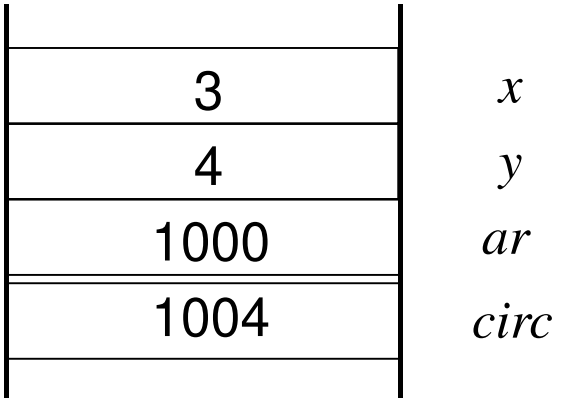
```
void calcul(int x, int y,
        int *ar, int *circ)
{
   *ar = x*y;
   *circ = 2*(x+y);
}
```

```
void main(void)
{
 int area,circumference;

calcul(3,4,&area,&circumference);
}
```

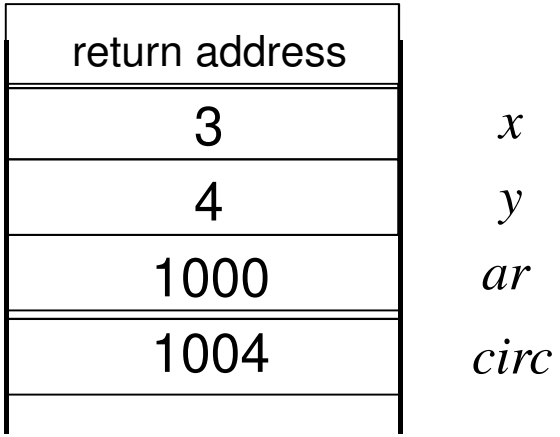
1000 and 1004 are addresses of variables area, circumference in computer memory

Before calling procedure



1000: area: 428 1004: circumference: 89

Before return to main



1000: area: 12 1004: circumference: 14

Note

- local variables are also allocated on stack at the moment of the entrance into functions

- new type **reference** is in C++ which allows to write output parameters more simply

Reference type

- the variable which references to *type T* is the synonymum for another variable

```
int  i = 0;
int& ir = i;          // ir ~ i
ir = 2;               // i = 2
```

- the variable must be initialized within the declaration – by variable of the *type T*

```
int &ir;  // error, ir is not initialized
```

```
float f;
const int ci = 10;

int &ir1 = f;   // error, f is not int
int &ir2 = ci;  // error, ci is a constant
```

- the reference variable references to the same variable all the time, it cannot be changed

Reference constant

- reference constant to the type *T* ~ a synonymum for unchanging value of the type *T*
- reference constant can by initialized by the constant of the type *T* or by the variable of the type *T* or *T1*, if the type *T1* assignment compatible with the type *T*
  - the temporary object with converted value is created in the second case

```
const int max = 100;
float f = 3.14;
const int& rmax = max;
const int& rf = f;
// temporary object with value 3 is referenced

rmax = 10;    // error, rmax is a constant
rf = 5;       // error, rf is a constant

int& rmax1 = max;
int& rf1 = f;       // error, rf1 is not a constant
```

Reference parameter

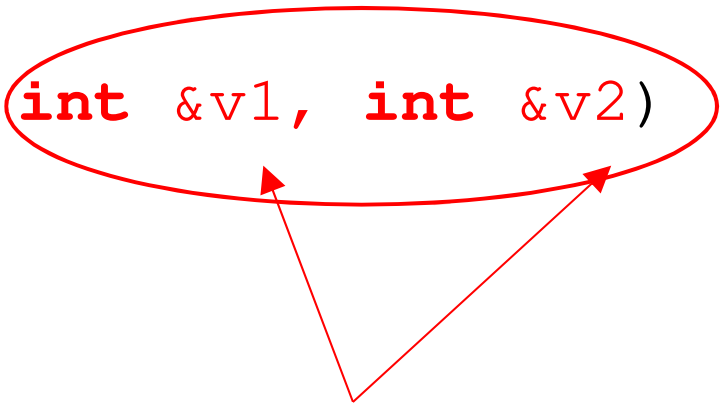
- reference parameter represents the synonym of the *real* parameter and it allows to change the value of the real parameter

- C style using pointers

```
void calcul1(int a, int b, int *v1, int *v2)
{
   *v1 = a*b;
   *v2 = 2*(a+b);
}
```

- C++ style using references

```
void calcul2(int a, int b, int &v1, int &v2)
{
   v1 = a*b;
   v2 = 2*(a+b);
}
```



```
void main(void)
{
  int area, circ;

  calcul1(3,4,&area,&circ);
  calcul2(3,4,area,cicr);
}
```

Array as parameter

- arrays are passed using pointers to the beginning
  - the count of elements must be passed as extra argument
- two ways of definition:

```
int maxarray(int n, int *arr)
{  int i;
   max = arr[0];
   for(i=1;i<n;i++)
    if (A[i]>max) max = arr[i];
   return max;
}
```

```
int maxarray(int n, int arr[])
{  int i;
   max = arr[0];
   for(i=1;i<n;i++)
    if (A[i]>max) max = arr[i];
   return max;
}

void main(void)
{  int maximum;
   int p[10]={5,4,8,7,15,5,9,8,7,10};
   maximum = maxarray(10,p);
   }
```

Two dimensional array

- static:

```
int maxarray(int n, int m, int arr[][20])
{
   …
}
```

- dynamic:

```
int maxarray(int n, int m, int **arr)
{
   …
}
```

Function overloading

- functions are distinguished only by identifiers (names) in classical C
  - it is not possible to define two functions having the same name and different parameters
  - disadvantage – several similar functions with different identifiers, for example:

```
int abs(int x);
double fabs(double x);
```

- the mechanism of **function overloading** exist in C++ (generally in object oriented languages)
  - functions are distinguished by count of parameters or data types
  - i.e. it is possible to define more functions with the same identifier having different count of parameters or/and different data types of parameters

```
int abs(int x);

double abs(double x);
```

- it is not possible to *overload function* only by the type of returned value – why?

```
void print(int x)
{
  printf("%d“,x);
}

void print(char x)
{
  printf("%c“,x);
}
```

- the compiler calls such function from overloaded ones where matching real parameters is the best
- matching parameters
  1. exact match
  2. match after a promotion

     char $\rightarrow$ int
     short $\rightarrow$ int
     enum $\rightarrow$ int
     float $\rightarrow$ double

3. match after a standard conversion

   int $\rightarrow$ float
   float $\rightarrow$ int
   int $\rightarrow$ unsigned
   …
   int $\rightarrow$ long

4. match after a user-defined standard conversion (typecasting)

- if best matching has more than one function the compile error occurs

Examples:

```
void f(float, int);
void f(int, float);
f(1,1);                          // error
f(1.2,1.3);                      // error

void f(long);
void f(float);
f(1.1);                          // error
f(1.1F);

void f(unsigned);
void f(int);
void f(char);
unsigned char uc;
f(uc);                           //f(int)
```