F28HS2 Hardware-Software Interface

Lecture 2: Programming in C - 2

Function declaration

type name (*type*$_1$ *name*$_1$, . . . , *type*$_N$ *name*$_N$)

{ *declarations*

statements

}

- result *type* optional
  - default is `int`
- *name*$_i$ == formal parameters
- { . . . } == *body*
- *declarations* optional

Function declaration

- formal parameters optional
- *expression* function
  - last statement to be executed should be:

    `return` *expression*`;`
- *statement* function
  - no result *type*
  - may end call with:

    `return;`

Function call

name (exp_1 , . . . , exp_N)

- exp_i == actual parameters

1. evaluate exp_i from left to right
2. push exp_i values onto stack
3. execute body, allocating stack space to any declarations
4. reclaim stack space for parameters & any declarations
5. return result if any

- NB end statement function call with ;

Example: polynomial

poly2.c

```
#include <stdio.h>

int poly(int a,int b,int c,int x)
{  return a*x*x+b*x+c;  }

main(int argc,char ** argv)
{  printf("%d\n",poly(2,4,3,5));
}
```

Changing parameters

- parameters passed by value
- value of actual parameter copied to space for formal parameter
- final value is not copied back from formal to actual
- so changing the formal does not change the actual

Example: swap

```
#include <stdio.h>

swap(int a,int b)
{   int t;
    t = a;
    a = b;
    b = t;
}
```

Example: swap

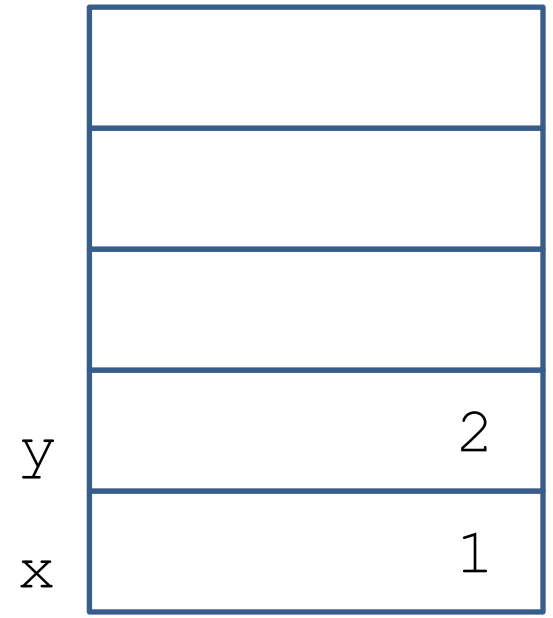
```
main(int argc, char ** argv)
{   int x,y;
    x = 1;
    y = 2;
    swap(x,y);
    printf("x: %d, y: %d\n",x,y);
}
```

➔

```
x: 1, y: 2
```

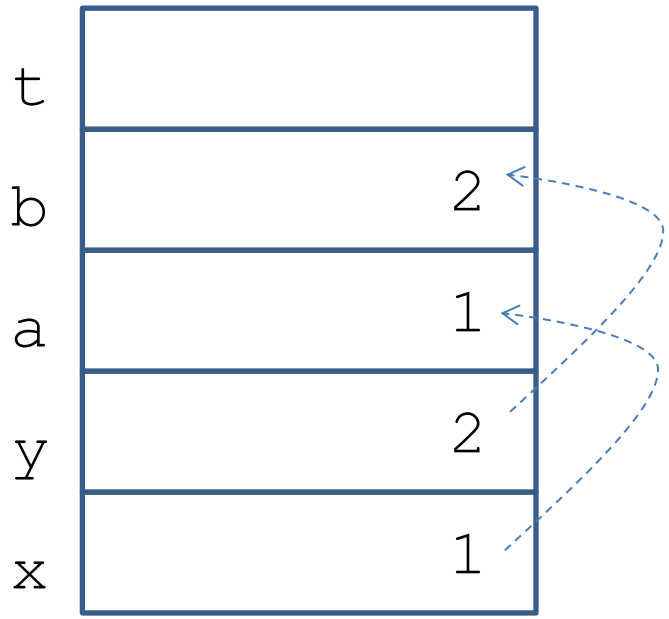
Example: swap

```
swap(int a,int b)
{   int t;
    t = a;
    a = b;
    b = t;   }
...
int x,y;
x = 1;
y = 2;
swap(x,y);
```



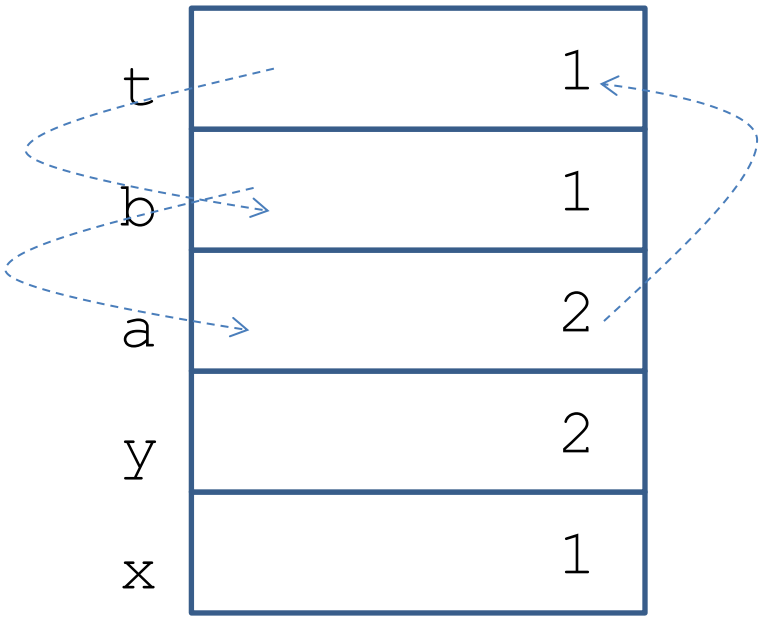
Example: swap

```
swap(int a,int b)
{   int t;
    t = a;
    a = b;
    b = t;   }
...
int x,y;
x = 1;
y = 2;
swap(x,y);
```



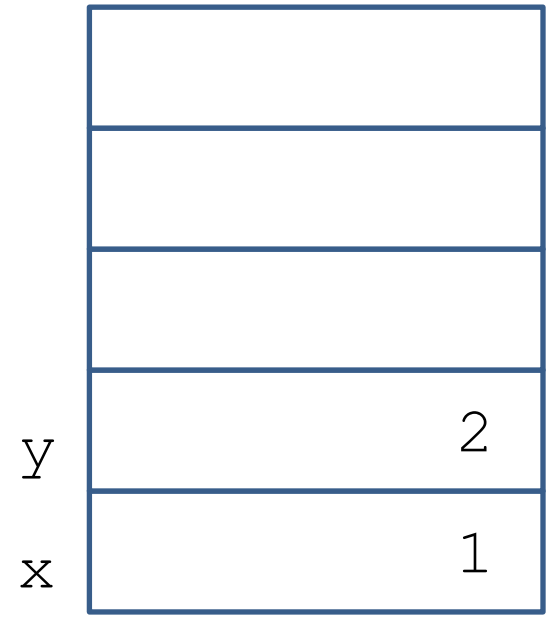
Example: swap

```
swap(int a,int b)
{   int t;
    t = a;
    a = b;
    b = t;   }
...
int x,y;
x = 1;
y = 2;
swap(x,y);
```



Example: swap

```
swap(int a,int b)
{   int t;
    t = a;
    a = b;
    b = t;   }
...
int x,y;
x = 1;
y = 2;
swap(x,y);
```



Pointers

type * *name* ;

- variable *name* holds address for byte sequence for *type*
- allocates stack space for address but *does not create instance of type*
- must allocate space explicitly

`NULL`

- empty pointer

Changing parameters

- to change actual parameter
  - pass address (`&`) of actual to formal
  - change indirect (`*`) on formal
- so formal must be pointer type

Example: swap2

swap2.c

```
#include <stdio.h>

swap(int * a,int * b)- parameters point to  int
{   int t;
    t = *a;         - t is value a points to
    *a = *b;        - value a points to  is value b points to
    *b = t;         - value b points to  is t's value
}
```

Example: swap2

```
main(int argc, char ** argv)
{   int x,y;
    x = 1;
    y = 2;
    swap(&x,&y);
    printf("x: %d, y: %d\n",x,y);
}
```

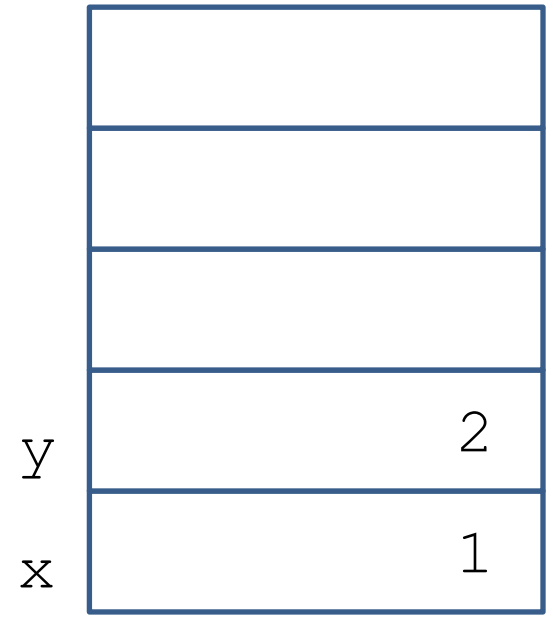
- pass pointers to `a` and `b`

➔

```
x: 2, y: 1
```

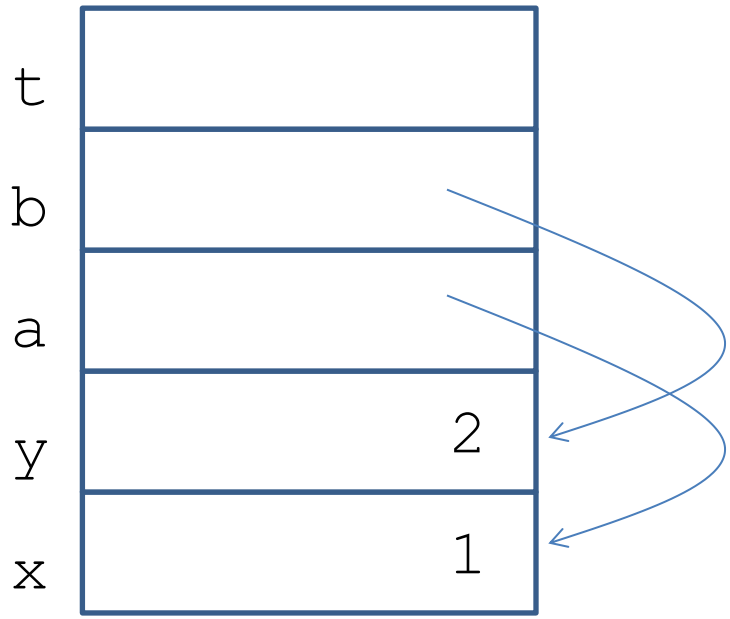
Example: swap2

```
swap(int * a,int * b)
{   int t;
    t = *a;
    *a = *b;
    *b = t;   }
...
int x,y;
x = 1;
y = 2;
swap(&x,&y);
```



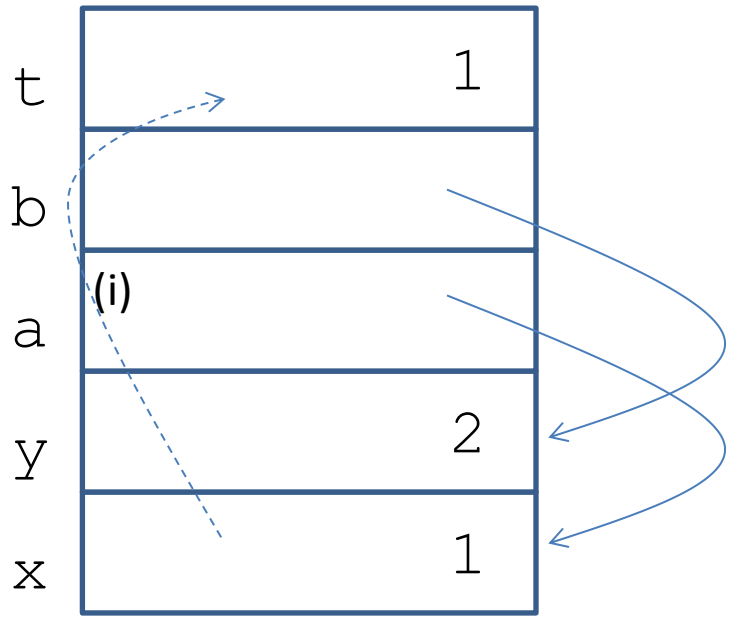
Example: swap2

```
swap(int * a,int * b)
{   int t;
    t = *a;
    *a = *b;
    *b = t;   }
...
int x,y;
x = 1;
y = 2;
swap(&x,&y);
```



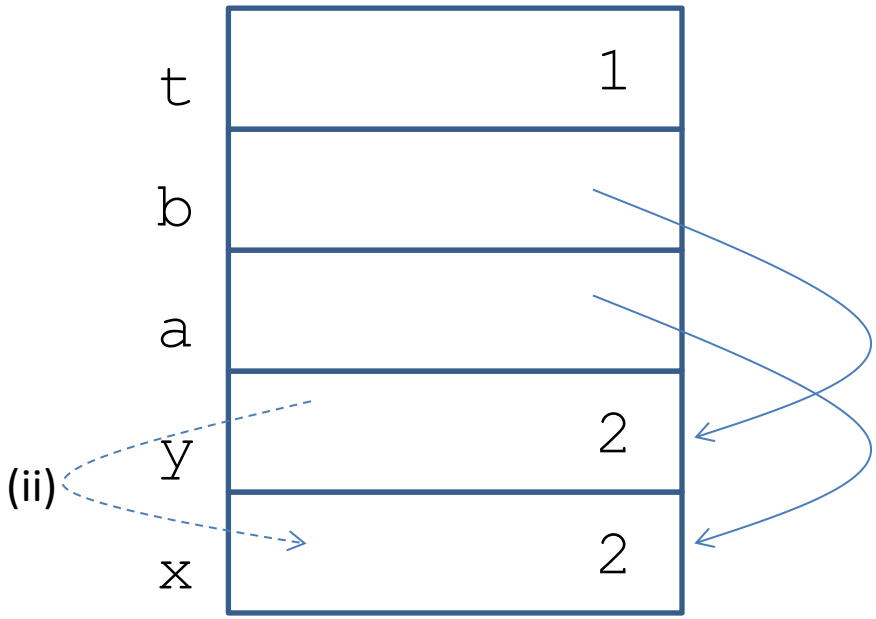
Example: swap2

```
swap(int * a,int * b)
{   int t;
    t = *a;        (i)
    *a = *b;
    *b = t;   }
...
int x,y;
x = 1;
y = 2;
swap(&x,&y);
```



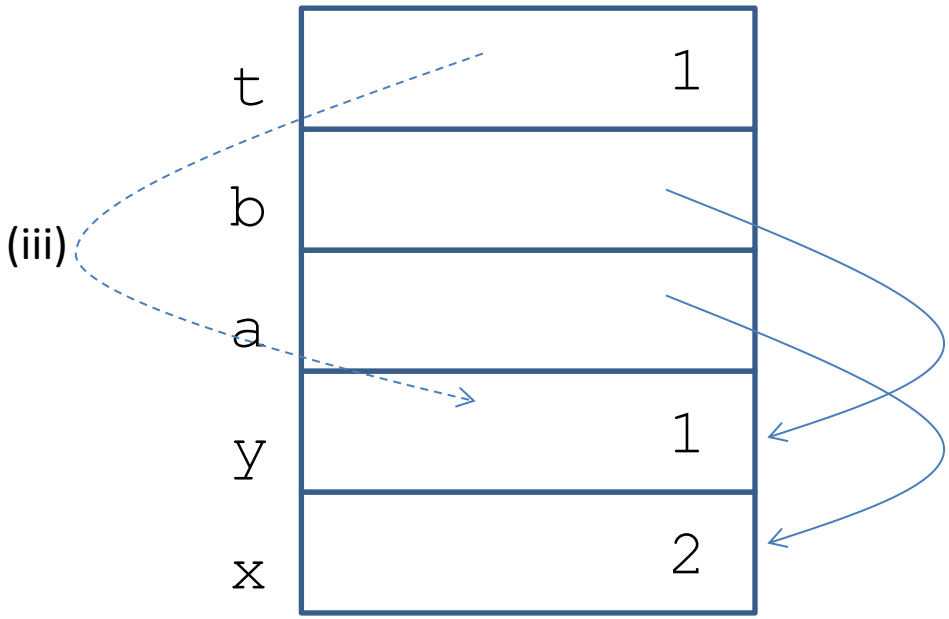
Example: swap2

```
swap(int * a,int * b)
{   int t;
    t = *a;
    *a = *b;      (ii)
    *b = t;   }
...
int x,y;
x = 1;
y = 2;
swap(&x,&y);
```



Example: swap2

```
swap(int * a,int * b)
{   int t;
    t = *a;
    *a = *b;
    *b = t;   }   (iii)
...
int x,y;
x = 1;
y = 2;
swap(&x,&y);
```



Characters

‘*character*’

- English character set

‘\n’ – newline

‘\t’ – tab

- one byte

Files

`FILE`

- type for system representation of file
- all files are treated as text
  - i.e. character sequences

`FILE *` *`name;`*

- *name* is the address of a `FILE` value

Files

`FILE * fopen(`*path*`,`*mode*`)`

- open file with path *string*
- return
  - `FILE` address
  - `NULL` if no file
- *mode*
  - "`r`" – read
  - "`w`" – write – create new empty file – delete old version!
  - "`a`" – append i.e. write at end of existing file

Files

`fclose(FILE *)`

- close file
- system may not do this for you when program stops

- for I/O, C handles characters as `int`s

- `EOF` == system end of file constant

File character I/O

`int getc(FILE *)`

- return next byte from file address
  - as `int`
- return `EOF` if at end of file

`int putc(`*`int`*`,FILE *)`

- puts *`int`* to file address
  - as byte
- return `EOF` if failure

File character I/O

`int getc(FILE *)`

- return next byte from file address
  - as `int`
- return `EOF` if at end of file

`int putc(`*`int`*`,FILE *)`

- puts *`int`* to file address
  - as byte
- return `EOF` if failure

Command line

`main(int argc, char ** argv)`

- `argc` == number of command line arguments
  - *including call to executable*
- `argv` == pointer to pointer to `char`
  - i.e. pointer to sequence of `char`
    - i.e. array of strings
  - `argv[0]` == name of executable
  - `argv[1]` == 1st argument
  - `argv[2]` == 2nd argument
  - …

`exit`

`exit(int)`

- end program
- return value of `int` to outer system
- in `stdlib` library

Example: file copy

`$ copy` *path*$_1$ *path*$_2$

1. check correct number of arguments
2. open *file*$_1$
   - check *file*$_1$ exists
3. open *file*$_2$
   - check *file*$_2$ exists
4. copy characters from *file*$_1$ to *file*$_2$ until `EOF`
5. close files

Example: file copy

```
#include <stdio.h>
#include <stdlib.h>

copy(FILE * fin, FILE * fout)
{  int ch;
   ch = getc(fin);
   while(ch!=EOF)
   {  putc(ch,fout);
      ch = getc(fin);
   }
}
```

Example: file copy

```
   main(int argc, char ** argv)
{  FILE * fin, * fout;

   if(argc!=3)
   {  printf("copy: wrong number of arguments\n");
      exit(0);
   }
```

Example: file copy

```
    fin = fopen(argv[1],"r");
    if(fin==NULL)
    {  printf("copy: can't open %s\n",argv[1]);
       exit(0);
    }
    fout = fopen(argv[2],"w");
    if(fout==NULL)
    {  printf("copy: can't open %s\n",argv[2]);
       exit(0);
    }
    copy(fin,fout);
    fclose(fin);
    fclose(fout);
}
```

Formatted file I/O

- formatted input

`int fscanf(FILE *,`*"format",exp$_1$,exp$_2$*...*`)`

  - *exp$_i$* == lvalue

- formatted output

`int fprintf(FILE *,`*"format",exp$_1$,exp$_2$*...*`)`

  - *exp$_i$* == rvalue

Formatted numbers

- can precede format character with width/precision info

e.g. `%3d`

  - integer
  - at least 3 characters wide

- e.g. `%4.2f`
  - double
  - at least 4 chars wide
  - 2 chars after decimal point

Keyboard/display `char` I/O

- for keyboard input

`int getchar()` ➔

`getc(stdin)`

- for screen output

`int putchar(`*`int`*`)` ➔

`putc(`*`int`*`,stdout)`

Character values

- characters have integer values
- ASCII (American Standard Code for Information Interchange) representation is international standard

char	int	hex	char	int	hex	char	int	hex
‘0’	48	30	‘A’	65	41	‘a’	97	61
‘1’	49	31	‘B’	66	42	‘b’	98	62
…			…			…		
‘9’	57	39	‘Z’	90	5A	‘z’	122	7A

- characters in usual sequences have sequential values

Character values

- very useful for character processing

e.g. `char ch` is:

- lower case if: `'a' <= ch && ch <= 'z'`
- upper case if: `'A' <= ch && ch <= 'Z'`
- digit if: `'0' <= ch && ch <= '9'`
- if `ch` is a digit then it represents value: `ch-'0'`
- `e.g. '7'-'0'` ➔ `55-48` ➔ `7`

Example: input binary number

- initial value is 0
- for each binary digit
  - multiply value by 2
  - add value of binary digit

input	value
11011	0
1011	2*0+1 → 1 – 1
011	2*1+1 → 3 – 11
11	2*3+0 → 6 – 110
1	2*6+1 → 13 – 1101
	2*13+1 → 27 – 11011

Example: input binary number

- function to convert text from file to value
- parameters for:
  - file
  - next character
- input each character from file into a non-local variable
  - pass address of character variable
  - access character indirect on address

Example: input binary number

binary.c

```
#include <stdio.h>
#include <stdlib.h>

int getBinary(FILE * fin,int * ch)
{  int val;
   val = 0;
   while (*ch=='0' || *ch=='1')
   {  val = 2*val+(*ch)-'0';
      *ch = getc(fin);
   }
   return val;
}
```

Example: input binary number

```
main(int argc,char ** argv)
{  FILE * fin;
   int ch;
   int val;

   if(argc!=2)
   {  printf("getBinary: wrong number of arguments\n");
      exit(0);
   }
   if((fin=fopen(argv[1],"r"))==NULL)
   {  printf("getBinary: can't open %s\n",argv[1]);
      exit(0);
   }
```

Example: input binary number

```
    ch = getc(fin);
    while(1)
    {   while(ch!='0' && ch !='1' && ch!=EOF)
         ch=getc(fin);
        if(ch==EOF)
         break;
        val = getBinary(fin,&ch);
        printf("%d\n",val);
    }
    fclose(fin);
}
```

– skip to binary digit

Example: input binary number

```
bdata.txt
0
1
10
11
100
101
...
1111
```

```
$ binary bdata.txt
0
1
2
3
4
5
...
15
```

Arrays

- finite sequence of elements of same type
- elements accessed by an integer index
- declaration:

type name `[`*`int`*`];`

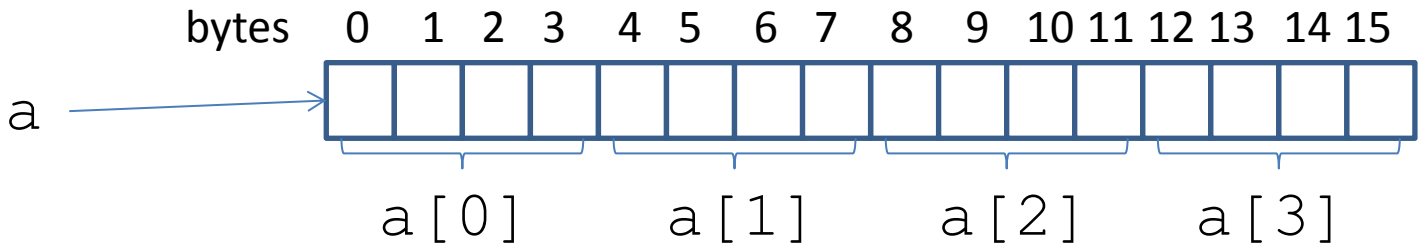
- allocate *`int`* * size for *type* on stack
- elements numbered from `0` to *`int`*`-1`
- e.g. `int a[6];` - allocates 6 * 4 == 24 bytes
- e.g. `char b[6];` - allocates 6 * 1 == 6 bytes
- e.g. `double c[6];` - allocates 6 * 8 == 48 bytes

Array size

- in *type name*`[int];`
- size `int` must be a constant
- cannot:
  - decide size at run-time
  - then declare correct size array
- must declare “too big” array
- during use must check not exceeding amount allocated

Array access

- **e.g.** `int a[4];`



- `a[0] == a+0*4` ➔ `a`
- `a[1] == a+1*4` ➔ `a+4`
- `a[2] == a+2*4` ➔ `a+8`
- `a[3] == a+3*4` ➔ `a+12`

Array access: expression

name = . . . exp_1 [exp_2] . . . ;

1. evaluate exp_1 to lvalue
2. evaluate exp_2 to integer
3. return *(exp_1+exp_2 * size for *type*)
   - i.e. contents of offset for exp_2 elements of *type* from address of 1st byte

- exp_1 is usually the name of an array

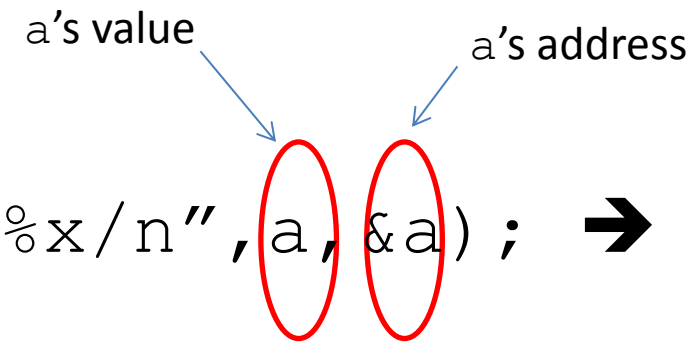
Array access: assignment

$exp_1[exp_2] = expression;$

1. evaluate exp_1 to lvalue
2. evaluate exp_2 to integer
3. evaluate *expression* to value
4. set exp_1+exp_2 * size for *type* to value
   - i.e. address of exp_2th element of *type* from address of 1st byte

Array name and address

- *name* is *alias* for address of 1st byte
- *name* is not a variable
- *i.e name* == &*name*




```
int a[3];
printf("a: %x; &a: %x/n",a,&a);
```

➔

```
a: 80497fc; &a: 80497fc
```

Array bounds

- no array bound checking
- if try to access array outside bounds
  - may get weird values from bytes outside array
- or
  - program may crash

Array type for formal parameter

type name [ ]

- size not specified

- same as:

type * *name*

Constant

`#define` *name text*

- pre-processor replaces all occurrences of *name* in program with *text*
- before compilation

- use to define constants once at start of program

e.g. `#define SIZE 127`

Example: scalar product

- calculate $V_0[0]*V_1[0]+...+V_0[N-1]*V_1[N-1]$

```
$ scalar vecs.txt
 1  2  3  4  5  6
 7  8  9 10 11 12
scalar product: 217
```

Example: scalar product

- file contains:
  - length of vector - N
  - 1st vector – $V_0[0]...V_0[N-1]$
  - 2nd vector – $V_1[0]...V_1[N-1]$
- functions to:
  - read vector
  - print vector
  - calculate scalar product

Example: scalar product

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

getVec(FILE * fin,int v[],int n)
{  int i;
   i = 0;
   while(i<n)
   {  fscanf(fin,"%d",&(v[i]));
      i = i+1;
   }
}
```

Example: scalar product

```
printVec(int v[],int n)
{   int i;
    i = 0;
    while(i<n)
    {   printf("%2d ",v[i]);
        i = i+1;
    }
    printf("\n");
}
```

Example: scalar product

```
int scalar(int v0[],int v1[],int n)
{   int s;
    int i;
    s = 0;
    i=0;
    while(i<n)
    {   s = s+v0[i]*v1[i];
        i = i+1;
    }
    return s;
}
```

Example: scalar product

```
main(int argc,char ** argv)
{  FILE * fin;
   int v0[MAX],v1[MAX];
   int n;
   if(argc!=2)
   {  printf("scalar: wrong number of arguments\n");
      exit(0);
   }
   if((fin=fopen(argv[1],"r"))==NULL)
   {  printf("scalar: can't open %s\n",argv[1]);
      exit(0);
   }
```

Example: scalar product

```
    fscanf(fin,"%d",&n);
    if(n>=MAX)
    {  printf("scalar: %d vector bigger than %d\n",n,MAX);
       fclose(fin);
       exit(0);
    }
    getVec(fin,v0,n);
    printVec(v0,n);
    getVec(fin,v1,n);
    printVec(v1,n);
    fclose(fin);
    printf("scalar product: %d\n",scalar(v0,v1,n));
}
```