Introduction to C

Programming Workshop in C (67316)
Fall 2018
Lecture 2
18.10.2018

C – 32 Keywords only

Review

- **Variable** -name/reference to a stored value (usually in memory)
- **Data type** - determines the size of a variable in memory, what values it can take on, what operations are allowed
- **Operator** - an operation performed using 1-3 variables
- **Expression** - combination of literal values/variables and operators/functions

Review - data types

- Various sizes (`char`, `short`, `long`, `float`, `double`)
- Numeric types - `signed`/`unsigned`
- Implementation - little or big endian
- Careful mixing and converting (casting) types

Review - operators

- Unary (++) , binary (+) , ternary (?:)
- Arithmetic (+), relational (<), binary (&&), assignment (=)
- Order of evaluation (precedence, direction) (++x vs. x++)

Input/Output

Character Input/Output

```c
#include <stdio.h>
int main()
{
    int c;
    while( (c = getchar()) != EOF )
    {
        putchar(c);
    }
    return 0;
}
```

#define macro

```c
#include <stdio.h>
#define NUM_OF_LINES 10
int main()
{
    int n = 0;
    int c;
    while(((c=getchar()) != EOF) &&
          (n < NUM_OF_LINES) )
    {
        putchar(c);
        if( c == '\n' )
            n++;
    }
    return 0;
}
```

AND operator

How many iterations are performed?

General Input/Output

```c
#include <stdio.h>
int main()
{
    int n;
    float q;
    double w;

    printf("Please enter an int, a float
            and a double\n");
    scanf("%d %f %lf", &n, &q, &w);
    printf("I got: n=%d, q=%f, w=%lf", n, q, w);

    return 0;
}
```

scanf problem

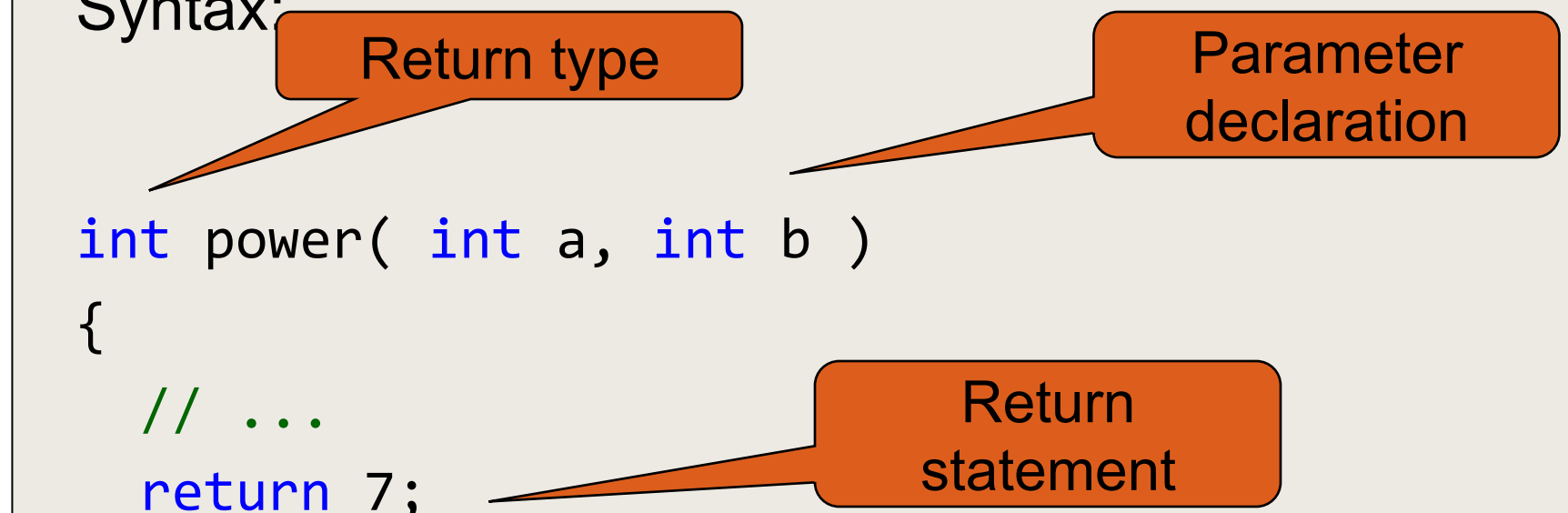
```c
#include <stdio.h>
int main()
{
    int number, result;

    do {
        printf("Give me a number:");
        result = scanf("%d", &number);
        if (result < 1)
            printf("You didn’t type a number!\n");
    } while (result < 1);

    printf("%d is your number", number);
    return 0;
}
```

Functions

Functions

C allows to define functions

Syntax:



Procedures

Functions that return `void`

```
void power( int a, int b )
{
  // ...
  return;
}
```

Example – printing powers

```c
#include <stdio.h>

int power( int base, int n )
{
   int i, p;

   p = 1;
   for( i = 0; i < n; i++ )
   {
      p = p * base;
   }
   return p;
}
```

```c
int main()
{
   int i;

   for( i = 0; i < 10; i++ )
   {
      printf("%d %d %d\n",
         i,
         power(2,i),
         power(-3,i) );
   }
   return 0;
}
```

Functions Declaration

```
void funcA()
{
   ...
}
void funcB()
{
   funcA();
}
void funcC()
{
   funcB();
   funcA();
   funcB();
}
```

```
void funcA()
{
   ...
}
void funcB()
{
   funcC();
}
void funcC()
{
   funcB();
}
```

"Rule 1": A function "knows" only functions which were declared above it.

Error:
funcC is not known yet.

Forward Declaration

Amendment to “Rule 1” : use **forward declarations**

```
void funcC(int param);
void funcA()
{
}
void funcB()
{
   funcC(7);
}
void funcC(int param)
{
}
```

Functions declaration

Declaration tells the compiler function **name and return type**

```
// the following 3 declarations are legit:
int foo(int a); // return int accepts int
int foo(int);   // return int accepts int
int foo();      // return int accepts unspecified
                //                   parameters
int main() {
   foo(5);
   return 0;
}
int foo(int a) { // actual definition of `foo`
   return a;
}
```

Functions declaration

```
int foo(int);   // return int accepts int
void foo(int);  // return void accepts int
```

error: conflicting types for 'foo'

```
int main() {
    foo(5);
    return 0;
}
int foo(int a) { // actual definition of `foo`
    return a;
}
```

Functions declaration

```
int foo(int);       // return int accepts int
int foo(int, int); // return int accepts int, int
```

error: conflicting types for 'foo'

```
int main() {
    foo(5);
    return 0;
}
int foo(int a) { // actual definition of `foo`
    return a;
}
```

NO function overloading

A function may have several declarations,
but only one definition

→ The following code **will not compile**

```
int foo(int a) {return a;}
int foo(int a) {return a;}
error: redefinition of 'foo'

int main() {
  foo(5);
  return 0;
}
```

Functions declaration

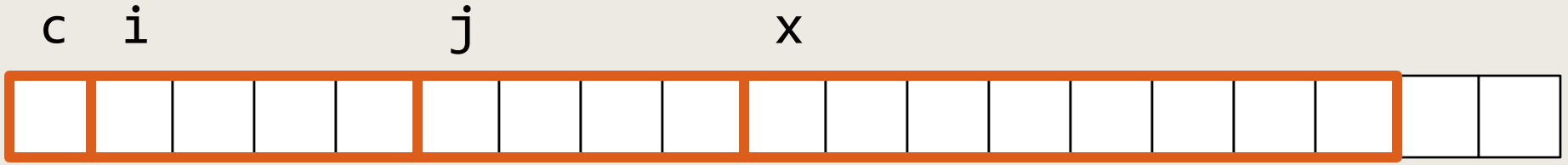
```
int foo(); // return int accepts unspecified
           //                    parameters


int main() {
    foo(5, 6, 7); // strange, but this is OK
    return 0;
}
int foo(int a) { // actual definition of `foo`
    return a;
}
```

Memory and Arrays

For now, we will only discuss static arrays

Memory

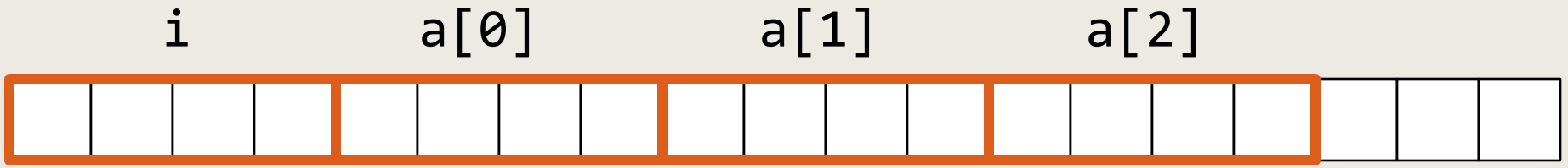
```
int main()
{
    char c;
    int i,j;
    double x;
```



Arrays

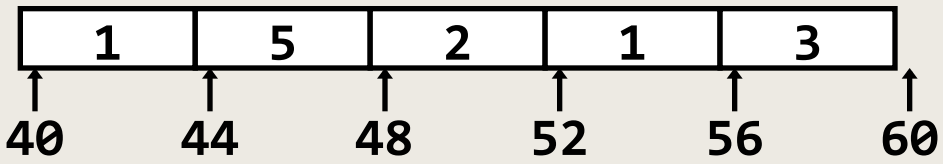
Defines a block of consecutive cells

```
int main()
{
    int i;
    int a[3];
```



Arrays - the [ ] operator

```
int arr[5] = { 1, 5, 2, 1 ,3  };
/*arr begins at address 40*/
```



Address Computation Examples:

1. `arr[0]` `40+0*sizeof(int) = 40`
2. `arr[3]` `40+3*sizeof(int) = 52`
3. `arr[i]` `40+i*sizeof(int) = 40 + 4*i`
4. `arr[-1]` `40+(-1)*sizeof(int) = 36  // can be the code`
   `// segment or other variables`

Arrays

C does not provide any run time checks:

```
int a[4];
a[-1] = 0;
a[4] = 0;
```

This will **compile and run**…

But can lead to unpredictable results/crash.

It is the programmer's responsibility to check whether the index is out of bound.

Arrays

C does not provide array operations:

```
int a[4];
int b[4];

a = b; // illegal

// and how about:
if( a == b ) // legal, address comparison
```

Array Initialization

```
int arr[3] = {3, 4, 5}; // Good
int arr[]  = {3, 4, 5}; // Good: the same
int arr[3] = {0};       // Init all items to 0, takes O(n)
int arr[4] = {3, 4, 5}; // Bad style - The last is 0
int arr[2] = {3, 4, 5}; // Bad
int arr[2][3] = {{2,5,7},{4,6,7}}; // Good
int arr[2][3] = {2,5,7,4,6,7};     // Good: the same
int arr[3][2] = {{2,5,7},{4,6,7}}; // Bad
int arr[3];      // uninitialized values
arr = {2,5,7};  // Bad (compilation): array assignment only
                // in initialization
```

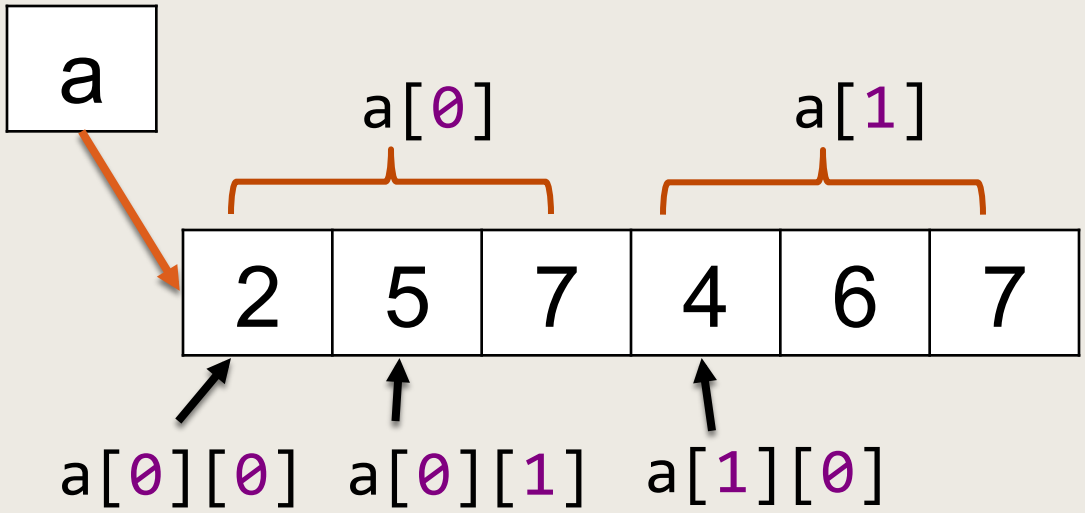
2D Array Memory Map

```
int a[2][3] = {{2,5,7},{4,6,7}};
```

Generally we would look at arrays as

```
int a[ROWS][COLS];
```

2	5	7
4	6	7



Think about a[n][m][k] etc...

Passing arguments to a program with argc and argv

- it is a good practice to print program arguments at the beginning of the program
- when the number of arguments is not what you expect, it is a good practice to print program usage

```
int main(int argc, char *argv[])
{
   for(int i=0; i<argc; i++)
   {
     printf("%s ", argv[i]);
   }

   if(argc < 2) // no arguments given
   {
     printf("Usage: myprog <num1> <num2>\n");
   }
}
```

Boolean types

Boolean types

Boolean type **doesn't exist in C**!

Use *char/int instead (It's possible to manipulate bits)*

zero => false

non-zero => true

```
#define TRUE 1
while (TRUE)
{
}
```

Examples: (Take a second)

```
while (1)
{
}
```

```
if (-1974)
{
}
```

```
i = (3==4);
```

Boolean types

Boolean type **doesn't exist in C**! (unlike C++ or Java)

Use *char/int instead (It's possible to manipulate bits)*

zero => false

non-zero => true

```
#define TRUE 1
while (TRUE)
{
}

(infinite loop)
```

Examples:

```
while (1)
{
}

(infinite loop)
```

```
if (-1974)
{
}

(true statement)
```

```
i = (3==4);

(i equals zero)
```

Boolean variables – example

```
int main()
{
    int a = 5;
    while(1)
    {
        if(!(a-3))
        {
            printf(”3");
            break;
        }
        printf(”%d", a--);
    }
    return 0;
}
```

Why does it evaluate to TRUE iff (a==3)?