Unix
Bash
C
GNU
Systems

Software Systems

Lectures Week 7

Introduction to C part 3

(Functions, Scope, Files, Structures)

Prof. Joseph Vybihal

Computer Science

McGill University

Week 7 Lecture 1

Functions and Scope

Functions

Syntax:

- RETURN_TYPE FN_NAME (PARAMETERS) { BODY; return VAR; }

Where:

- RETURN_TYPE any legal C type declaration
  - The type void can be used to mean nothing will be returned. In this case the "return VAR;" statement is not used.
- FN_NAME the functions name, must be unique
- PARAMETERS a comma separated list of TYPE VAR, TYPE2 V2
- BODY C code
- Return VAR statement specifying what is returned, must agree with RETURN_TYPE

Example

```
int max(int a, int b, int c) {                    // Function Declaration
    int theMax = a; // we assume 'a' is the max

    if (b > theMax) theMax = b;
    if (c > theMax) theMax = c;

    return theMax;
}
int main() {
    int x, y, z, result;
    scanf("%d %d %d", &x, &y, &z);
    result = max(x,y,z);                          // Function "call" or "invocation"
    printf("%d\n", result);
}
```

Compilation Order

2-pass vs 1-pass compilers

- 2-pass compiler: scans the source file twice
- 1-pass compiler: scans the source file once
- Conclusion: 1-pass is faster than 2-pass
- Restriction: 1-pass, by definition, has a declaration restriction

C is a 1-pass compiler

Compilation Order

The 1-pass declaration order restriction:

- All identifiers must be declared before they are used
- Example identifiers:
  - Variables, function names, pre-processor directives, user-defined types, libraries

```
int main() {
  x = 10;
  int x;
  x = negate(x);
}

int negate(int a) {
  return a * -1;
}
```

Fails at “int x” and function call.

```
int negate(int a) {
  return a * -1;
}

int main() {
  int x;
  x = 10;
  x = negate(x);
}
```

Fixed.

Scope

Scope defines how the compiler determines which identifier declaration is being used in a statement.

First come first server scope rule:

1. Declaration and usage is in the same Block
2. Declaration and usage is in the same Local space
3. Declaration and usage is in the same Global file space
4. Declaration and usage is in the same External space
5. Generate syntax error

Example

```
#include <stdio.h>
int x;                                  // x is in global file space
int max(int a, int b, int c) {          // a, b, c are in the “max” local space
    int theMax = a;                     // theMax is in the “max” local space
    if (b > theMax) theMax = b;
    if (c > theMax) theMax = c;
    return theMax;
}
int main() {
    int limit = 10;                     // limit is in the “main” local space
    for(int x=0; x<limit; x++) {        // x is in the “for” block space
     int result = max(x,x+1,x+2);       // result is in the “for” block space
    }
    printf(“%d”, result);
}
```

In the main program: what happens with ‘x’ and ‘result’ in terms of scope?

Function Prototypes

```
#include <stdio.h>
int x;
int max(int,int,int);                    // this is a function prototype
int main() {
    int limit = 10;
    for(int x=0; x<limit; x++) {
      int result = max(x,x+1,x+2);
    }
    printf("%d", result);
}
int max(int a, int b, int c) {
    int theMax = a;
    if (b > theMax) theMax = b;
    if (c > theMax) theMax = c;
    return theMax;
}
```

A function prototype is a promise to the compiler.

"I promise that somewhere in this source file there is a declaration for this function. And, that function will look exactly like this prototype."

Notice that a prototype does not have a body. It simply ends with a semi-colon.

Call-by-value

Definition:

- Passing a copy of a variable to a function.

Conclusion:

- Changing the value of the variable in the function does not effect the value of the variable in the calling environment.

Example:

```
void increment (int a) {
  a = a + 1;
}

int main() {
  int x = 5;
  increment(x);     // since void was used we don't return anything
  printf("%d", x);  // even though 'a' was incremented the value of 'x' is still 5
}
```

Call-by-reference

Definition:

- Passing a pointer to the original variable to a function.

Conclusion:

- Changes to the value of the local variable will also effect the value in the original variable.

Example:

```
void increment (int *a) {
  *a = *a + 1;
}

int main() {
  int x = 5;
  increment(&x);     // sends the address of 'a' to the function (like in scanf!)
  printf("%d", x);     // this will print out 6
}
```

Call-by-reference and value

(arrays and strings)

```
int findA(char array[], int length, char key) {       // array is by reference, the others not
    int pos;
    for(pos=0; pos<length; pos++) if (array[pos] == key) return pos;
    return -1; // to indicate not in array since arrays have index numbers >=0
}
int findS(char *s, char key) {        // string is by reference (but strings are constants)
    char *p;
    for(p=s; *p!='\0'; p++) if (*p == key) return p-s; // like pos above, distance.
    return -1;
}
int main() {
    char name[30], address[100];
    printf("%d", findA(name,30,'w'));          // notice this works for different sized arrays
    printf("%d", findA(address,100,'5'));      //    due to the [] in the function declaration
    printf("%d",findS("My name is bob", 'b'));
```

Question

How can we write a function called swap:

void swap(value1, value2)

So that after this function is called the values in the two original variables have been exchanged?

```
int x=5, y=10;
swap(x, y);
printf("%d %d",x,y); // 10 5
```

Week 7 Lecture 2

Sequential Text Files

Sequential Files

Letter.txt

Dear Mom,
Please send money.
Love Bob.

LOGICAL VIEW

Files on disk are actually linear structures like 1D arrays but without cell index numbers.

Start address

D	e	a	r	/r	/n	/t	P	….	EOF

End of file character

ACTUAL PHYSICAL VIEW

STDIO.H

The stdio.h library has file commands:

- fopen - to access the file from the start address
- fclose - to terminate file access
- fgetc - to read a single character from the file
- fgets - to read one entire line from a file
- fputc - to write a single character to the file
- fputs - to write one entire line to the file
- fscanf - to read a formatted line from the file
- fprintf - to write a formatted line to the file
- feof - end of the file test

The get, put and printf commands work much like their console and stream versions

fopen

To access a file.

Syntax:

- FILE *fopen(FILENAME, MODE);

Where:

- FILE - a built-in pointer type to reference a file
  - On success returns a pointer to the file
  - On failure returns a NULL pointer
- FILENAME - a Unix path/filename descriptor as a string
- MODE:
  - rt - read from text file (file must exist)
  - wt - write to text file (if file exists, overwrites)
  - at - append to text file (if file exists it appends, or creates file)

Example

```
#include <stdio.h>
#include <stdlib.h>
void displayFile (FILE *p) {
    char c;
    while(!feof(p)) {
        c = fgetc(p);
        putc(c);
    }
}
void main() {
    FILE *q = fopen("letter.txt","rt");
    if (q == NULL) exit(1); // terminate with an error code
    displayFile(q);
    fclose(q);
}
```

Example

```
#include <stdio.h>
#include <stdlib.h>
void copyFile (FILE *source, FILE *destination) {
    char c;
    while(!feof(source)) {
        c = fgetc(source);
        fputc(c, destination);
    }
}
void main() {
    FILE *s = fopen("letter.txt","rt"), *d = fopen("copy.txt","wt");
    if (s == NULL || d == NULL) exit(1); // terminate with an error code
    copyFile(s, d);
    fclose(s); fclose(d);
}
```

Example

```
#include <stdio.h>
#include <stdlib.h>  #include<string.h>  // cannot define beside in real life…
void copySkipWord (FILE *source, FILE *destination, char *word) {
    char array[1000]; // must assume a max size…
    while(!feof(source)) {
        fgets(array,999,source); // 999 since fgets inserts a \0 at the end
        if (strstr(array, word) == 0) // the word is not in the array
            fputs(array, destination);
    }
}
void main() {
    FILE *s = fopen("letter.txt","rt"), *d = fopen("copy.txt","wt");
    if (s == NULL || d == NULL) exit(1); // terminate with an error code
    copySkipWord(s, d, "bob");
    fclose(s); fclose(d);
}
```

Important

End of file issue: in the previous example the last line of the file would be repeated twice. This is the correct way to do it.

Why?

```
fgets(array,999,source);
while(!feof(source)) {
        if (strstr(array, word) == 0) // the word is not in the array
            fputs(array, destination);
        fgets(array,999,source);
    }
}
```

Week 7 Lecture 3

Struct and Union

Complex Data Structures

User-defined types composing primitive elements into a single complex structure.

Two structures exist in C:

- The struct data structure
- The union data structure

struct

Syntax:

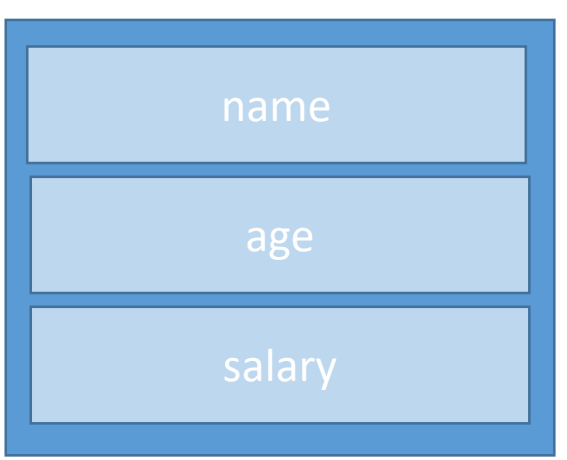
```
struct OPTIONAL_NAME {
    FIELDS;
} OPTIONAL_VAR_NAME ;
```

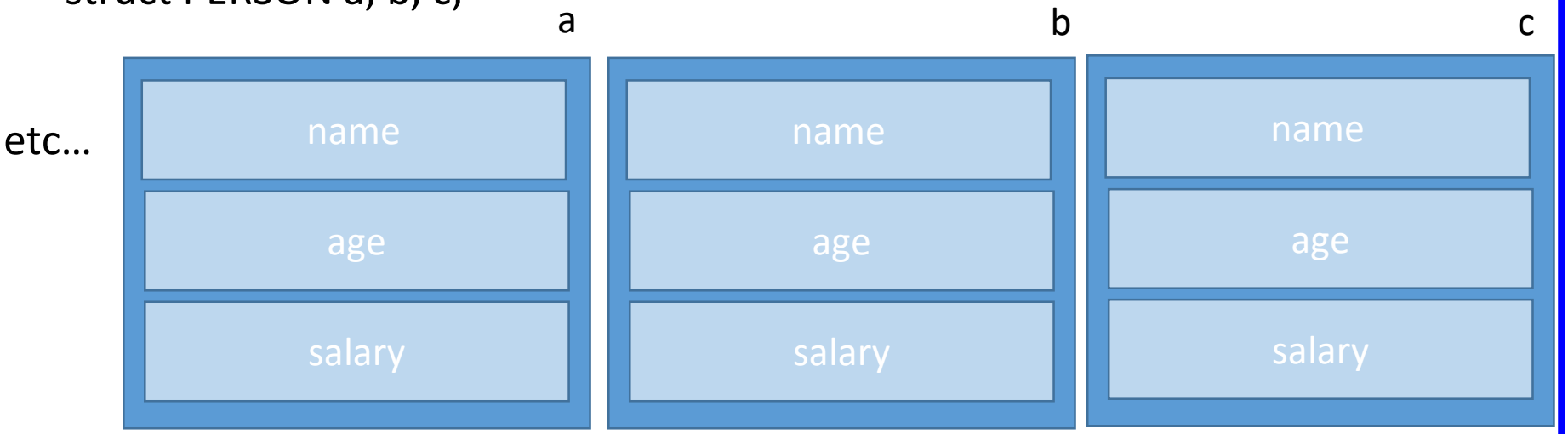
Where:

- OPTIONAL_NAME - is its user-defined type name
- OPTIONAL_VAR_NAME- is a variable containing this structure
- FIELDS
  - Is a list of semi-colon separated variable declarations
  - TYPE1 VAR1; TYPE2 VAR2; etc.

Example

```
#include <stdio.h>
struct PERSON {      // customary to write in all caps
    char name[30];
    int age;
    float salary;
};     // notice the semi-colon
int main() {
    struct PERSON a, b, c;
```





The dot operator

To access the fields within the structure we use the dot operator:

```
struct PERSON a;

scanf("%s", a.name);
scanf("%d", &(a.age));
a.salary = 50.25;

printf("%s %d %f", a.name, a.age, a.salary);
```

Example

```
#include <stdio.h>
struct PERSON { char name[30]; int age; float salary; } a, b, c; // 3 variables
int main() {
    printf("Enter the name for person a: ");
    scanf("%s", a.name);
    printf("Enter the age: ");
    scanf("%d", &(a.age));
    printf("Enter the salary: ");
    scanf("%f", &(a.salary));
    // repeat the above for b and c
    printf("Enter the name for person b: ");
    scanf("%s", b.name);
}
```

Array of struct

```
#include <stdio.h>
struct PERSON { char name[30]; int age; float salary; } ;
int main() {
    struct PERSON people[100];
    for(int x=0; x<100; x++) {
            scanf("%s", people[x].name);
            scanf("%d", &(people[x].age));
            scanf("%f", &(people[x].salary));

            if (people[x].age == 20) printf("Hey you are 20!!\n");
    }
}
```

union

Syntax:

```
union OPTIONAL_NAME {
    FIELDS;
} OPTIONAL_VAR_NAME ;
```

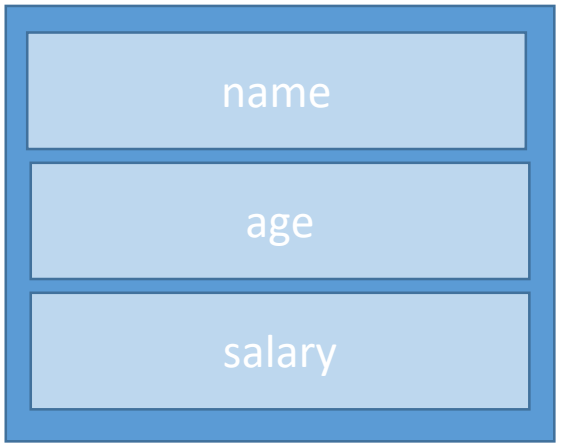
It is identical to struct, except what is built is different.

Example

```
struct PERSON {
    char name[30];
    int age;
    float salary;
};

union PERSON2 {
    char name[30];
    int age;
    float salary;
};
```

PERSON



PERSON2


```
union PERSON2 x;
x.age = 20;
x.salary = 30.7; // destroys the 20
```

Example

```
struct PERSON {
    char type;   // ‘p’=prof, ‘s’=student, ‘f’=staff
    char name[30];
    int age;
    char ID[10];
    union SPECIFIC_DATA {
            struct {
                    float evaluation;
                    int position;
            } prof;
            struct { float GPA; float fees; } stud;
            struct { char level; float salary; } staff;
    } specific;
}
```

Example

```
struct PERSON mcgill[1000];

int findPerson(char ID[]) {
    int pos;
    for (pos=0; pos<1000; pos++) {
            if (strcmp(mcgill[pos].ID, ID)==0) return pos;
    }
    return -1;
}

int main() {
    int location = findPerson("3219678");
    if (location != -1) printf("%s", mcgill[location].name);
```

Example

```
struct PERSON mcgill[1000];
int main() {
    int location = findPerson("3219678");
    if (location != -1) {
            printf("%s", mcgill[location].name);
            switch (mcgill [location].type ) {
                case 'p':
                    printf("Evaluation= %f\n", mcgill[location].specific.prof.evaluation);
                    break;
                case 's':
                    printf("GPA = %f\n", mcgill[location].specific.stud.GPA);
            }
    }
}
```

Question

If a bank has a checking account and a savings account. How might we build a struct and union data structure to represent this information?

CHECKING	SAVINGS
Account number	Account number
Balance	Balance
	Withdraw fee