System SW

Lecture 7 – Basics of C programming language – Part 6

Jarno Tuominen

Lecture 9 – Basics of C programming language – Part 6

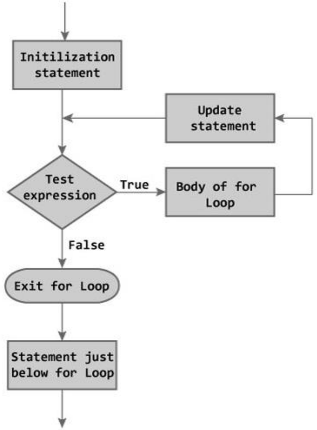
- Review
- Functions (cont)
- The C preprocessor

Review of last lecture

- Loop statements
- Functions

Review: Loops

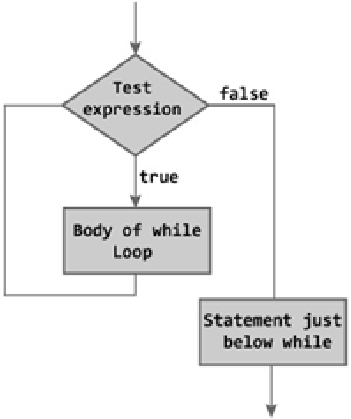
for



Body executed zero or more times

Repeats n times

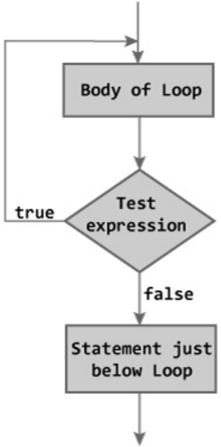
while



Executed zero or more times

Repeats until certain condition is met

do-while

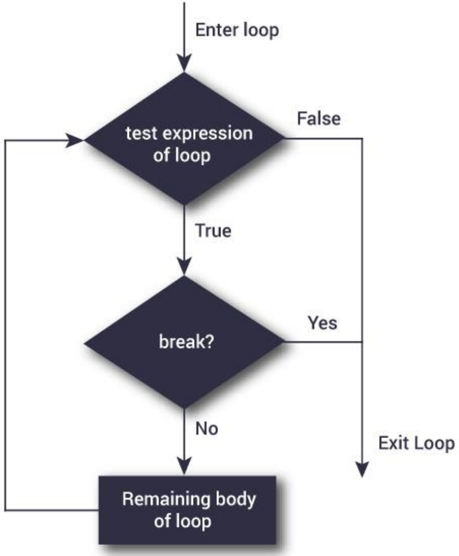


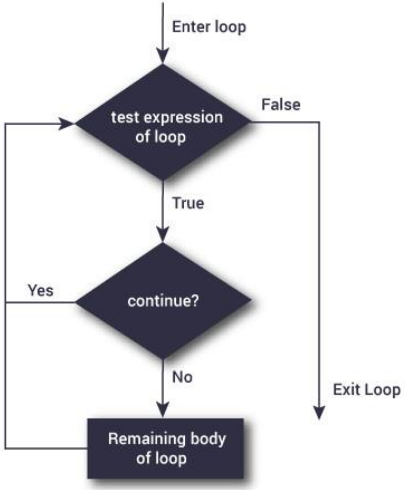
Executed one or more times

Repeats until certain condition is met

Review: break and continue in loops

- **break** causes exit from the loop
- In case of nested loops, exits only the inner loop





- **continue** stops the current iteration and moves to next iteration (and checks the condition)

Review: goto

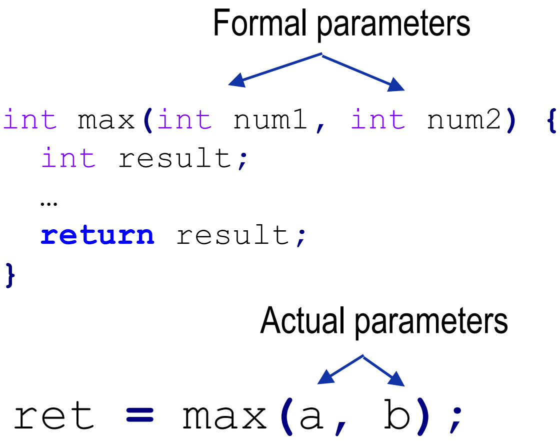
- goto allows you to jump unconditionally to arbitrary part of your code (within the same function)
- Generally avoid, except in error handlers where it might be handy

```
start:
{
    if (cond )
        goto outside;
    /* some code here */
    goto start;
}
outside:
```

Review: functions

```
int max(int num1, int num2);
int max(int, int);
```

- A function **declaration** (prototype) tells the compiler about a function's name, return type, and parameters
- A function **definition** provides the actual body of the function
- Definition ”header” must match the declaration
- By default, formal parameters are **local copies** of the actual parameters given at the fucntion call
- Modifying the formal parameters inside a function, does not change the value of actual parameters! (Call by Value)



Formal parameters

```
int max(int num1, int num2) {
  int result;
  …
  return result;
}
```

Actual parameters

```
ret = max(a, b);
```

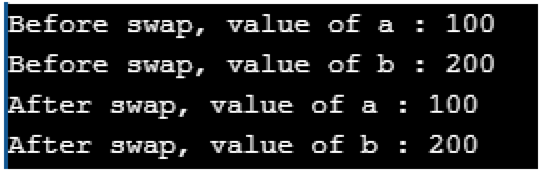
Review: Passing function arguments

- The parameters (or arguments) of a function, that are given in function call are **actual parameters**
- The parameters listed in function declaration are **formal parameters**
- Formal parameters behave like other **local** variables inside the function and are created upon entry into the function and destroyed upon exit
- While calling a function, there are two ways in which arguments can be passed to a function
  1. Call by value (the default, as in earlier example)
  2. Call by reference

Function call by value vs call by reference

Does not work!

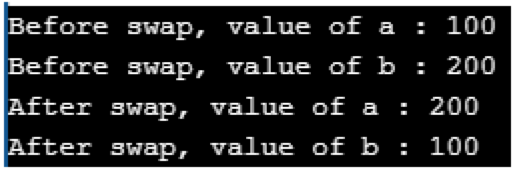
```
void swap(int x, int y) {
   int temp;
   temp = x; /* save the value of x */
   x = y;    /* put y into x */
   y = temp; /* put temp into y */
   return;
}

int main () {
  ...
  swap(a, b);
  ...
}
```



```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 100
After swap, value of b : 200
```

Works

```
void swap(int *x, int *y) {
   int temp;
   temp = *x;    /* save the value at address x */
   *x = *y;      /* put y into x */
   *y = temp;    /* put temp into y */
   return;
}

int main () {
  ...
  swap(&a, &b);
  ...
}
```



```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

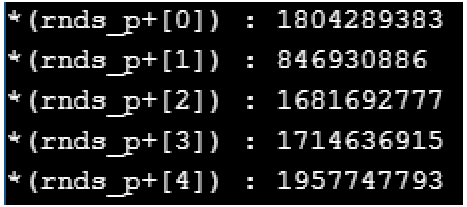
Lecture 9 – Basics of C programming language – Part 6

- Review
- Functions (cont)
- The C preprocessor

Functions returning pointers

- C allows function to **return a pointer** to a
  - local variable (bad idea!) Why?
  - static variable
  - dynamically allocated memory
  - Function
- Syntax: add ”*” in front of the function name to indicate that return value is a pointer (of type int, in the example)
- In this example the function returns an array – which is a pointer, remember?

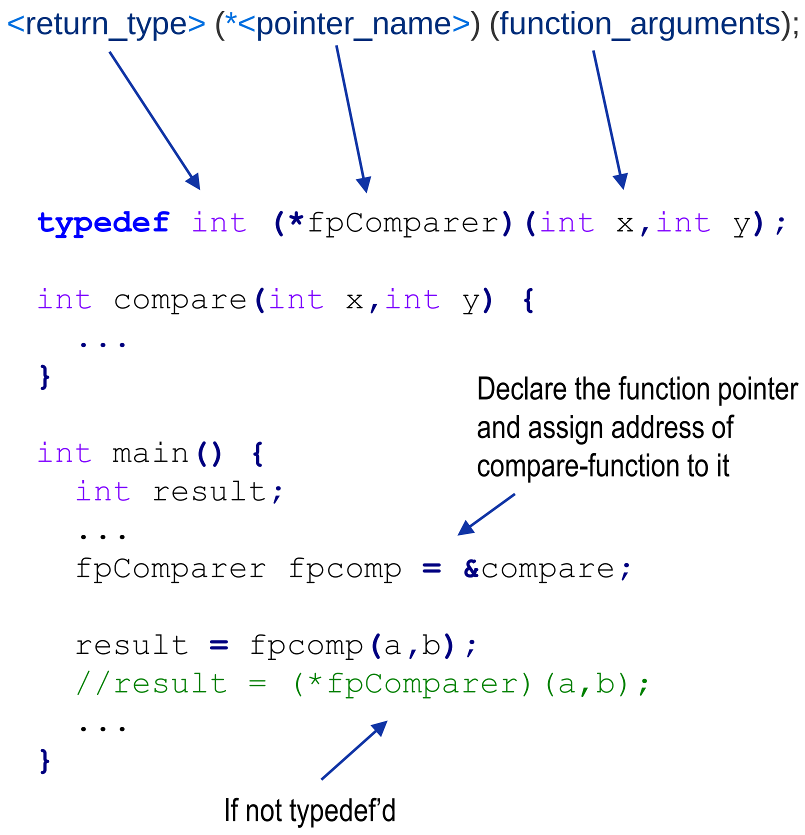
```
int *getrandom(void) {
    static int r[5]; //must be static!
    for (int i = 0; i<5; i++) {
        r[i] = rand();
    }
    //Note: r is same as &r[0]
    return r;
}

int main() {
    int * rnds_p;
    rnds_p = getrandom();

    for (int i = 0; i<5; i++) {
      printf("*(rnds_p+[%d]) : %d\n",i,*(rnds_p+i));
    }
  return 0;
}
```



```
*(rnds_p+[0]) : 1804289383
*(rnds_p+[1]) : 846930886
*(rnds_p+[2]) : 1681692777
*(rnds_p+[3]) : 1714636915
*(rnds_p+[4]) : 1957747793
```

Function pointers

- A **function pointer** is a pointer variable that contains an **address of a function**, instead of a data object
- The syntax of declaration is similar to the syntax of declaring a function – but instead of using a function name, you use a pointer name inside the parenthesis
- Like with normal pointer variables, before using function pointer you need to assign it a value, i.e. the address of a function
- It is a good practice to type define declaration of function pointers as it will make your code much nicer
- Note: Function pointers are potentially very dangerous, as a loose pointer does not code access to wrong data, but if will cause your program to branch to a random address!



<return_type> (*<pointer_name>) (function_arguments);

```
typedef int (*fpComparer)(int x,int y);

int compare(int x,int y) {
  ...
}

int main() {
  int result;
  ...
  fpComparer fpcomp = &compare;

  result = fpcomp(a,b);
  //result = (*fpComparer)(a,b);
  ...
}
```

Declare the function pointer and assign address of compare-function to it

If not typedef'd

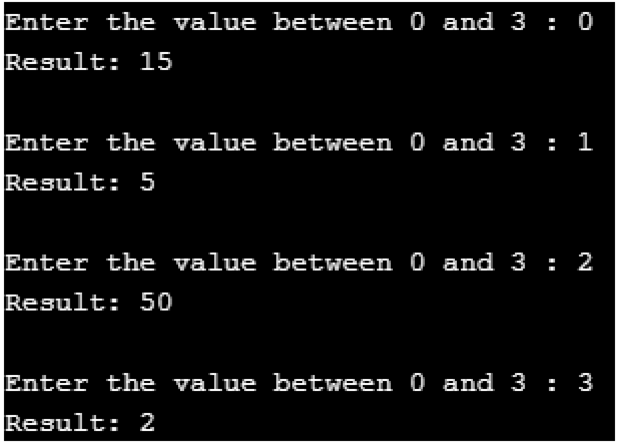
Practical example of using function pointers

```c
#include <stdio.h>

int add(int i, int j)  { return (i+j); }
int sub(int i, int j)  { return (i-j); }
int mul(int i, int j)  { return (i*j); }
int divi(int i, int j) { return (i/j); }

int (*oper[4])(int a, int b) = {add, sub, mul, divi};

int main() {
    int ch,result;
    int a=10, b=5;
    while(1) {
        printf("Enter value between 0 and 3 : ");
        scanf("%d",&ch);
        result  = oper[ch](a,b);
        printf("\nResult: %d\n\n",result);
    }
}
```

Function definitions (no declaration needed as they are before main())

Create an array of function pointers and initialize them to contain addresses of add,sub,mul,divi

Based on input, call the corresponding function



```
Enter the value between 0 and 3 : 0
Result: 15

Enter the value between 0 and 3 : 1
Result: 5

Enter the value between 0 and 3 : 2
Result: 50

Enter the value between 0 and 3 : 3
Result: 2
```

What happens if you enter value 4?

Function pointers packed in a struct

```c
typedef uint8_t (*sensor_fp)(uint8_t SensorID, uint8_t param);
```

Type define: sensor_fp

```c
//Struct for generic sensor instance
typedef struct sensor_t {
          char* name;
          bool enabled;
          sensor_fp init;
          sensor_fp power_ctrl;
} sensor_t;
```

Sensor-related data and its functions packed in a single ”instance” (close to object-oriented thinking but still plain C!)

```c
sensor_t sensors[MAX_SENSOR_COUNT];
```

Create an array of sensor instances

```c
sensors[0].name = "BMI_1";
sensors[0].enabled = true;
sensors[0].init = bmi160_initialize_sensor;
sensors[0].power_ctrl = bmi160_power_ctrl;
```

set-up the sensor instance

```c
sensors[1].name = “ECG";
sensors[1].enabled = true;
sensors[1].init = ads1293_init;
sensors[1].power_ctrl = ads1293_power_ctrl;
```

and another one

Init all the sensors! Beatiful code <3

```c
void init_sensors(uint8_t count) {
  for (uint32_t i=0; i < count; i++)
  {
    e = sensors[i].init(i, NULL);
  }
}
```

Functions with variable arguments list

- C library ”**stdarg.h**” defines data types and macros which can be used to get arguments in a function when the number of arguments is not known
- The function declaration has ellipses (=three dots) to indicate the variable amount of arguments
- The first arguments indicates the number of arguments

```
int func(int, ... ) {
  <code here>
}

int main() {
   func(2, 1, 2);
   func(3, 1, 2, 3);
}
```

https://www.tutorialspoint.com/c_standard_library/stdarg_h.htm

Variable argument list example

```c
#include <stdio.h>
#include <stdarg.h>

double avg(int num,...) {
   va_list valist;
   double sum = 0.0;
   int i;
   /* initialize valist for num number of arguments */
   va_start(valist, num);
   /* access all the arguments assigned to valist */
   for (i = 0; i < num; i++) {
      sum += va_arg(valist, int);
   }
   va_end(valist); /* clean memory reserved for valist */
   return sum/num;
}

int main() {
   printf("Avg = %f\n", avg(4, 2,3,4,5));
   printf("Avg = %f\n", avg(3, 5,10,15));
}
```

num is number of arguments

va_list is a data type that can hold list of arguments.
Used by macros **va_start**, **va_arg** and **va_end**

populate *valist* using macro **va_start**

Get arguments of type int
using macro **va_arg**

va_end releases memory

Lecture 9 – Basics of C programming language – Part 6

- Review
- Functions (cont)
- The C preprocessor

The C Preprocessor

- **C preprocessor** is the macro preprocessor, which provides the ability for the
  - inclusion of header files
  - macro expansions
  - conditional compilation
  - line control
  - Handling of pragma operators (in C99)
- invoked by the compiler as the first part of code translation
- Preprocessor macros begin with **#**

https://en.wikipedia.org/wiki/C_preprocessor

Preprocessor macros and directives

- **#include**
  - Inclusion of header files
  - #include <stdio.h> , #include ”path/my_file.h”
    - Compiler replaces that line with the entire contents of named source file
    - Standard headers use < > - notation (the .h file is found in the standard compiler include paths), user files use ” ” –notation.

- .h files can include other .h –files, and sometimes two or more files include same (third) header file.
  - Problem, as variables, macros and function prototypes get re-defined multiple times
  - Compiler error or warning will occur
  - A good solution is to use **include guards** – the inclusion takes place on once and you don’t have to worry about #include hierarchy
  - Still, try to avoid including unnecessary .h-files, because they slow down the compilation

```
File ”child.c”:

#include "grandparent.h‟
#include "parent.h"
```

```
File ”parent.h”:

#include "grandparent.h‟
```

```
File ”grandparent.h”:


#ifndef GRANDPARENT_H
#define GRANDPARENT_H

   struct foo { int member; };

#endif /* GRANDPARENT_H */
```

Defining expression macros

- **#define, #undef**
  - Defining/undefining macros (and macro constants)

A good coding practice is to use UPPERCASE letters for all macro names

```
// object-like macro
#define <identifier> <replacement token list>
#define PI 3.14159

// function-like macro, note parameters
#define <identifier>(<parameter list>) <replacement token list>
#define RADTODEG(x) ((x) * 57.29578)
#define add3(x,y,z) ((x)+(y)+(z))
```

parentheses ensure order of operations

```
// delete the macro
#undef <identifier>
#undef PI
```

the gcc option **-Dname=value** sets a preprocessor define that can be used

Conditional preprocessor macros

- Enable conditional compilation of the code
- Evaluated before code itself is compiled, so conditions must be preprocessor defines or literals
- #if, #elif, #endif
- #ifdef, #ifndef

```
#if VERBOSE >= 2
printf (“lots of trace messages\n");
#elif VERBOSE >1
printf (“some trace messages\n");
#endif
```

```
#ifdef DEBUG
some_debug_function();
#endif
```

True with any value of DEBUG (as long as it is defined)

Custom errors/warnings, pragmas

- #error, #warning

```
#ifdef CHIP_VERSION_3
#error  Sorry, chip version not supported
#endif


#if VERBOSE >5
#warning  Lots of msgs going to UART!
#endif
```

- #pragma

The `#pragma' directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself

http://gcc.gnu.org/onlinedocs/gcc-4.0.3/cpp/Pragmas.html