Control flow in C

EECS 2031

Summer 2014

Przemyslaw Pawluk

June 3, 2014

What we will discuss today

- Review
- Control Flow
- Functions and Program Structure
- Compilation
  - Preprocessor
- Homework

Table of Contents

Array, Structures and Lists

Questions

- What is an Array?
- What is a Pointer?
- What is a Structure?
- Which data structure can be constructed with Structures?

Array, Structures and Lists

Questions

- What is an Array?
- What is a Pointer?
- What is a Structure?
- Which data structure can be constructed with Structures?

Array, Structures and Lists

Questions

- What is an Array?
- What is a Pointer?
- What is a Structure?
- Which data structure can be constructed with Structures?

Array, Structures and Lists

Questions

- What is an Array?
- What is a Pointer?
- What is a Structure?
- Which data structure can be constructed with Structures?

Array, Structures and Lists

Questions

- What is an Array?
- What is a Pointer?
- What is a Structure?
- Which data structure can be constructed with Structures?

Array, Structures and Lists

Questions

- What is an Array?
- What is a Pointer?
- What is a Structure?
- Which data structure can be constructed with Structures?

Linked List vs Binary Tree

Questions

- What is a Linked List?
- What is a Binary Tree?
- How to construct a Linked List using Structures?
- How to construct a Binary Tree using Structures?
- Which of the mentioned data structure can be used to improve searching?

Linked List vs Binary Tree

Questions

- What is a Linked List?
- What is a Binary Tree?
- How to construct a Linked List using Structures?
- How to construct a Binary Tree using Structures?
- Which of the mentioned data structure can be used to improve searching?

Linked List vs Binary Tree

Questions

- What is a Linked List?
- What is a Binary Tree?
- How to construct a Linked List using Structures?
- How to construct a Binary Tree using Structures?
- Which of the mentioned data structure can be used to improve searching?

Linked List vs Binary Tree

Questions

- What is a Linked List?
- What is a Binary Tree?
- How to construct a Linked List using Structures?
- How to construct a Binary Tree using Structures?
- Which of the mentioned data structure can be used to improve searching?

Linked List vs Binary Tree

Questions

- What is a Linked List?
- What is a Binary Tree?
- How to construct a Linked List using Structures?
- How to construct a Binary Tree using Structures?
- Which of the mentioned data structure can be used to improve searching?

Linked List vs Binary Tree

Questions

- What is a Linked List?
- What is a Binary Tree?
- How to construct a Linked List using Structures?
- How to construct a Binary Tree using Structures?
- Which of the mentioned data structure can be used to improve searching?

Linked List vs Binary Tree

Questions

- What is a Linked List?
- What is a Binary Tree?
- How to construct a Linked List using Structures?
- How to construct a Binary Tree using Structures?
- Which of the mentioned data structure can be used to improve searching?

Table of Contents

Statements and Blocks

Statement

Is a code followed by a semicolon ;

Block

- enclosed by { and }
- syntactically equivalent to a single statement
- no semicolons after the right brace (})

Variables and Blocks

Variable can be defined in any block and it's visibility is limited to this block only!

Control flow statements

- `if-else`
- `switch`
- `while`
- `for`
- `do-while`

- `break`
- `continue`
- `goto`
- labels (`labelName:`)

if–else

Usage

A basic branching instruction taking a logical expression.

Example

```
if (x < v[mid])
    high = mid + 1;
else if (x > v[mid])
    low = mid + 1;
else /* found match */
    return mid;
```

switch

Usage

Similar to the if-else statement but allows for multiple branches based on the value of an expression

Example

```
switch (c) {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
        ndigit[c-'0']++;
        break;
    case ' ':
    case '\n':
    case '\t':
        nwhite++;
        break;
    default:
        nother++;
        break;
}
```

switch

All cases must be

- unique (cannot duplicate cases)
- constant, e.g. case 2*x: is invalid

Guidelines

- avoid deliberate fall-through
- put a "break" at the end of the switch statement

Loops: while and for

`while`

Allows you to decide if to repeat an expression based on a logical expression

`for`

Allows you to decide if to repeat an expression based on a counter/calculated expression

Loops: while and for

Example: `while`

```
while ((c = getchar()) == ' ' || c == '\n'
  || c == '\t')
```

Example: `for`

```
for (i = 0; i < n; i++)
  ...
```

continue

Usage

Command `continue` allows you to skip an execution of the loop and go directly to the next iteration

Example

```
for (i = 0; i < n; i++) {
    if (a[i] < 0) /* skip negative */
        continue;
    a[i]++; /* increment non-negative */
}
```

break

Usage

Allows you to break loop (stop execution and continue to the next command after the loop)

Example

```
for (i = 0; i < n; i++)
    if (a[i] < 0) /* 1st negative element */
        break;
    if (i < n)
        return i;
    ...
```

goto and Labels

Usage

Allows you to make "jumps" in your program.

Example

```
if (a[i] == b[j])
    goto found;
 /* didn't find any common element */
 ...
found:
 /* got one: a[i] == b[j] */
```

Note

Using goto is considered bad programming style

goto and Labels

Usage

Allows you to make "jumps" in your program.

Example

```
if (a[i] == b[j])
    goto found;
 /* didn't find any common element */
 ...
found:
 /* got one: a[i] == b[j] */
```

Note

Using goto is considered bad programming style

goto and Labels

- Code that relies on goto statements is generally harder to understand and to maintain. So goto statements should be used rarely, if at all.

- `break` and `continue` should be used only when necessary.

Table of Contents

Program Structure

- C programs are comprised of variables and functions.
- We have discussed variables, expressions and control flow.
- We now want to combine these into a program

Functions

Function is a set of statements that may have

- a number of arguments, that is values that can be passed to the function
- a return type that describes the value of this function in an expression

Functions

Definition

Defining a function describes its return value, its arguments and provides the code that implements the function

Returning values

If the function returns void

- finish execution (end of the block)
- `return;`

Else, you have to use

- `return value;`

Declaring Functions

Just declaration

- Sometimes we want to use a function without describing how it works
- Declaring a function tells us its return type and arguments but not its code.
  `int putchar(int c);`
- Like a function definition but with ; instead of a block

No param names

- We can omit argument names `int putchar(int);`
- The type of arguments is what matters
- Good practice recommends putting names

void

void means nothing

Void as an argument means that there is nothing expected, as a result that nothing is returned

Example

```
int getchar(void);

void exit(int status);
```

Returning value

int main()?

- The return value of `main()` is the programs exit status
- In `main()`, `return x;` is the same as `exit(x);`

Note

- Returning a value from a function that should return void is an error
- Returning nothing from a function that should return a value is valid but unpredictable

Local, global, external

Local

Variables only exist within their block

Global

Variable exist on a global level, and is visible,

External

Variable external – define in some other file in the program

Example

```
int x; //this is a global variable
extern int y; //declared in other file
void add_n_to_x(int n) {
    x += n;
}
void set_x_to_m(int m) {
    int x; //this is local variable
    x = m;
}
```

Table of Contents

Compilation

Three phases of compilation

- Preprocessor - handles `#define` and `#include`
- Compiler - converts C code into binary processor instructions ("object code")
- Linker - puts multiple files together and creates an executable program

Preprocessor

- Handles `#define` and `#include`
- Removes comments
- Preprocesses C file – processes it before compiling it
- Output is C code

Compilation

- When compiling multiple files, all .c files are converted to .o files
- Then all .o files are combined (linked) to make a program.
- `gcc -c` compiles .c files to .o files
- `gcc -o` links .o files into ecexutable

Hiding symbols

- By default, all global symbols (functions and global variables) in a source file are visible to the world.
- This is undesirable as it "pollutes" the global namespace and may expose sensitive data.

Hiding symbols – `static`

Keyword `static` has two meanings depending on where it appears

Global symbols

It is used to hide a global variable e.g.

```
static int variable;
```

Inside function

To declare persistent variables (the value is kept between executions of the function)

Question: How can we use such variable?

Table of Contents

The C Preprocessor

- Handles `#define` and `#include`
- Removes comments
- Preprocesses C file before compiling it
- Output is C code

#define

Usage

`#define` defines macros that substitute one value for another

Examples

```
#define IN 1
#define SQUARE(x) x*x
```

```
state = IN;
y = SQUARE(4);
```

```
state = 1;
y = 4*4;
```

#define

Usage

`#define` defines macros that substitute one value for another

Examples

```
#define IN 1
#define SQUARE(x) x*x
```

```
state = IN;
y = SQUARE(4);
```

```
state = 1;
y = 4*4;
```

#define

Be careful with parameters

Examples

```
#define SQUARE(x) x*x

y = 5+2*5+2 ;//=17
// not 7*7=49

y = SQUARE(5+2);
```

#define

Be careful with parameters

Examples

```
#define SQUARE(x) x*x

y = SQUARE(5+2);
```

```
y = 5+2*5+2 ;//=17
// not 7*7=49
```

#define

Makro has to be unique

Name of the makro has to be unique, parameters don't count

Macros in substituted values are also evaluated

Macros in substituted values are also evaluated but not recursively!

Example

```
#define Y Z y
#define Z z
```

from `Y` will produce `zy`

```
#define Y Z y
#define Z Y z
Y becomes Y z y
```

#define

Makro has to be unique

Name of the makro has to be unique, parameters don't count

Macros in substituted values are also evaluated

Macros in substituted values are also evaluated but not recursively!

Example

```
#define Y Z y
#define Z z
```

from `Y` will produce `zy`

```
#define Y Z y
#define Z Y z
```

`Y` becomes `Y z y`

Other operators used in makros

- `#` – can be used to make a string
- `##` – macro concatenation operator
- `# undef` – undefined what has been defined before
- `#if` and `#endif` – conditional makros (e.g. `#if isdefined(DEBUG)`)

Putting everything together

Headers

Are usually used for header files, and header files are really just C code

- Function declarations
- Macro definitions
- External variable declarations

Preventing multiple inclusions

A common use of `#ifndef` is to protect header files from being included more than once

Compilation of multiple sources

```
gcc file1.c file2.c ...  filen.c -o myprog
```

Table of Contents

Homework

Create a simple C program that:

- Reads integers from the stdin
- Puts them into a binary tree
  - if new element is smaller add it as a left leaf
  - if new element is bigger add it as a right leaf
- Travers the tree in-order (left-parent-right) and prints the values of all nodes
- **All functions, structures and definitions should be isolated in a separate files called `tree.c` and `tree.h`**