Goal

• Quick introduction to C
  – Enough to program assignments
  – Background for lectures
• Assume you know Java or C#
  – E.g. from Parallel Programming
• Non-goal:
  – Teach details and strict definition of C
  – Teach advanced features/idioms/techniques in C
Further reading

Old, but a great tutorial (how I learned C)

Definitive.

Very advanced: all the stuff you never wanted to know about C 😊

Online: http://www.iu.hio.no/~mark/CTutorial/CTutorial.html
Compared to Java or C#

- No objects, classes, features, methods, or interfaces
  - Only functions/procedures
  - Function pointers will be met later...
- No memory management
  - Lots of things on the stack
  - Heap structures must be explicitly created and freed
- No fancy built-in types
  - Mostly just what the hardware provides
  - Type constructors to build structured types
- No exceptions
  - Convention is to use integer return codes
Compared to Java or C#

• Powerful macro pre-processor (cpp)
• Very fast
  – Almost impossible to write assembly as fast as a good C compiler
  – Pretty much impossible to compile Java to run as fast as C
• Pointers: real machine addresses
• Close to the metal: you can know what the code is doing to the hardware
⇒ Language of choice for
  – Operating System developers
  – Embedded systems
  – People who really care about speed
  – Authors of security exploits
A feel of C programs

• A C program is characterized by:
  – Functions, grouped by header files and libraries
  – Data structures built using structs and pointers
  – Created dynamically using malloc and free
  – Symbolic constants defined with cpp macros

• More advanced features:
  – Polymorphism and object dispatch with function pointers
Syntax: the good news

• Similar to Java or C#
  – Java or C# syntax almost entirely lifted from C
  – Comments (/*...*/, //) the same
  – Identifiers the same as in Java (C# allows more characters in identifiers)
  – Block structure using { ... }
  – Many other constructs the same or similar

• Main differences
  – List of reserved words is different
  – C is run through a macro preprocessor
    • String and file substitution
    • Conditional compilation
    • Although C# has preprocessor directives, it does not have a separate preprocessor. Moreover there are no macros.
Hello World

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("hello, world\n");
    return 0;
}
```

“header file” – bit like an interface file in Java or C#

Every program has to have a “main” function, which takes a list of command line arguments.

Generic function for printing formatted strings. The “newline” is not included automatically!

Returning 0 indicates everything is OK – C has no exceptions.
Workflow

1. Macro substitution, Include header files

2. Compile each C file into assembly language

3. Assemble each file into object code

4. Link object files into program binary

C source ➔ cpp ➔ cc1 ➔ as ➔ ld ➔ Binary
The C Preprocessor

- Include the “header” file inline in the source code
- Basic mechanism for defining APIs
- Use of <> or “” determines where to look for the file
  - Use <> for system headers
  - Use “” for your own headers
- Included files can include other files
  - Beware of including files twice!

```
#include <file1.h>
#include "file2.h"
```
The C Preprocessor

```c
#define FOO BAZ
#define BAR(x) (x+3)
...
#undef FOO
```

- Token-based macro substitution
- Any subsequent occurrence of `FOO` is replaced with `BAZ`
  - Until a `#undef FOO`

- `BAR(4)` is replaced with `(4+3)`
  - Not 7!
- `BAR(hello)` is replaced with `(hello+3)`
The C Preprocessor

#ifdef FOO
... (text 1)
#else
... (text 2)
#endif

#endif __FILE_H
#define __FILE_H
... (contents of file.h)
#endif // __FILE_H

• Text 1 is used if a macro FOO is defined, otherwise Text 2
• Opposite for BAR
• #else is optional
• Idiom for header files:
  #ifndef __FILE_H
    #define __FILE_H
    ...
    ... (contents of file.h)
  #endif // __FILE_H
• Ensures file contents only appear once!
Types in C
Declarations

• Are like Java or C#:
  ```
  int my_int;
  double some_floating_point = 0.123;
  ```

• Inside a block:
  – Scope is just the block
  – `static` → value `persists` between calls

• Outside a block:
  – Scope is the `entire program`!
  – `static` → scope limited to the file (compilation unit)
Integers and floats

• Types and sizes:

<table>
<thead>
<tr>
<th>C data type</th>
<th>Typical 32-bit</th>
<th>ia32</th>
<th>Intel x86-64</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>float</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>double</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>long double</td>
<td>8</td>
<td>10/12</td>
<td>10/16</td>
</tr>
</tbody>
</table>

• Integers are signed by default
  – use **signed** or **unsigned** to clarify
Integers and floats

• Rules for arithmetic on integers and floats are complex
  – Implicit conversions between integer types
  – Implicit conversions between floating point types
  – Explicit conversions between anything (casts)

• Behavior is either:
  – Determined by the hardware
  – Was decided by hardware, a long time ago

• We’ll cover this more in lectures
Booleans

• Boolean values are just integers
  – False → zero
  – True → anything non-zero
  – Negation (“!”) turns zero into non-zero, and vice-versa

• Any statement in C is also an expression, hence idioms like:

```c
int rc;
if (!(rc = call_some_fn())) {
    printf("Failed with return code %d
", rc);
    exit(1);
}

// Carry on: call succeeded.
```
Casting

• Most C types can be *cast* to another:

```c
unsigned int ui = 0xFFEEDDCC;
signed int i    = (signed int)ui;
```

⇒ *i* has value -1122868.

• Bit-representation does not change!
• Frequently used with pointer types...
Arrays

- Finite set of variables, all the same type
- For an N-element array `a`:
  - First element is `a[0]`
  - Last element is `a[N-1]`
- C compiler **does not** check the array bounds!
  - Very typical bug!
  - Always check array bounds!

```c
#include <stdio.h>
float data[5]; /* data to average and total */
float total;  /* total of the data items */
float average; /* average of the items */

int main() {
    data[0] = 34.0;
    data[1] = 27.0;
    data[2] = 45.0;
    data[3] = 82.0;
    data[4] = 22.0;

    total = data[0] + data[1] + data[2] +
            data[3] + data[4];
    average = total / 5.0;
    printf("Total %f Average %f\n", total, average);
    return (0);
}
```
Multi-dimensional arrays

```c
int a[3][3];

// First approach
for (i=0; i < 3; i++)
    for (j=0; j < 3; j++)
        matrix[i][j] = a++;

// Second approach
int a = 1;
for (i=0; i < 3; i++)
    for (j=0; j < 3; j++)
        matrix[j][i] = a++;
```

```plaintext
\begin{array}{cccccccccc}
  & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  a[0][0] & \ldots & a[1][0] & \ldots & a[1][2] & \ldots & a[2][2] \\
\end{array}
```

```
\begin{array}{cccccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
```
More on arrays

- Arrays can be initialized when they are defined:
  ```
  /* a[0] = 3, 
     a[1] = 7, 
     a[2] = 9 */
  int a[3] = {3, 7, 9};
  /* list[0]=0.0, …, 
     list[99]=0.0 */
  float list[100] = {};
  ```

- Strings are arrays of characters terminated with the null character \0:
  ```
  char str[6] = {'h','e','l','l','o','\0'}
  ... is the same as:
  char str[6] = "hello";
  ```

- Secretly, arrays are (almost) the same as pointers
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[]) {
    char name1[12], name2[12], mixed[25];
    char title[20];

    strcpy(name1, "Rosalinda");
    strcpy(name2, "Zeke");
    strcpy(title, "This is the title.");

    printf("     %s

", title);
    printf("Name 1 is %s
", name1);
    printf("Name 2 is %s
", name2);

    if(strcmp(name1, name2) > 0) {
        /* returns 1 if name1 > name2 */
        strcpy(mixed, name1);
    } else {
        strcpy(mixed, name2);
    }

    printf("The biggest name alphabetically is %s\n", mixed);

    strcpy(mixed, name1);
    strcat(mixed, " ");
    strcat(mixed, name2);

    printf("Both names are %s\n", mixed);
    return 0;
}
Sizes

- How much memory does a value take up?
- Depends on machine and compiler!
- Use:
  
  `sizeof(type) or sizeof(value)`

- Evaluates at compile time to size in bytes
- e.g.

  ```c
  int nr = 1919;
  int size = sizeof(nr);
  ```
void

- There is a type called **void**.
- It has **no** value.
- Used for:
  - Untyped pointers (to raw memory): “**void** *”
  - Declaring functions with no return value (procedures)

- `sizeof(void)` shouldn’t work
  - Why?
  - (Non-standard) GCC allows `sizeof(void) == 1`
  - Why?
# Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>() [] -&gt; .</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>! ~ ++ -- + - * &amp; (type) sizeof</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>* / %</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>+ -</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&lt;= &gt;=</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>== !=</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>^</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>l</td>
<td>Left-to-right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left-to-right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Right-to-left</td>
</tr>
<tr>
<td>= += -= *= /= %= &amp;= ^=</td>
<td>= &lt;&lt;= &gt;&gt;=</td>
</tr>
</tbody>
</table>

- ( ) is a function call
- -> means struct pointer indirection
- Unary +, -, *
- * here is pointer indirection
- Ternary if-else operator
Control flow
Control flow statements (like Java or C#)

```java
if (Expression) Statement_when_true
    else  Statement_when_false

switch (Expression) {
    case Constant_1:  Statement;  break;
    case Constant_2:  Statement;  break;
    ...
    case Constant_n:  Statement;  break;
    default:  Statement;  break;
}

return (Expression)
```
Control flow statements
(just like Java)

for (initial; Test; Increment) Statement

while (Expression) Statement

do Statement while (Expression)
Control flow statements
(not like Java, same as C#)

controversial, but occasionally very useful indeed!
Functions

- **Main unit of composition for programs**
  - **Return type**: type of the value returned by the function when it terminates
  - **Name**: identifies the function
  - **Arguments** of defined types: parameters to pass to the function

- **Arguments passed by value**
  - function gets copy of the value of the parameters but cannot modify the actual parameters
  - Values can be passed by reference using *pointers to the values* instead

General syntax:

```
returntype function_name(def of parameters) {
    localvariables
    functioncode
}
```

Example:

```
float findavg(float a, float b) {
    float average;
    average=(a+b)/2;
    return(average);
}
```

Must be declared as prototypes *before* they are defined:

```
float findavg(float a, float b);
```
/* Compute factorial function */
/* fact(n) = n * (n-1) * ... * 2 * 1 */

#include <stdio.h>

int fact(int n)
{
    if (n == 0) {
        return(1);
    } else {
        return(n * fact(n-1));
    }
}

int main(int argc, char *argv[])
{
    int n, m;

    printf("Enter a number: ");
    scanf("%d", &n);
    m = fact(n);
    printf("Factorial of %d is %d.\n", n, m);
    return 0;
}
main() is also a function

/* program to print arguments from command line */
#include <stdio.h>

int main(int argc, char **argv) {
    int i;

    printf("argc = %d\n\n",argc);
    for (i=0;i<argc;++i)
        printf("argv[%d]: %s\n",i, argv[i]);
    return 0;
}

- argc: argument count. Number arguments passed in the command line
- argv: argument vector (array). All the arguments as strings
- argc is always at least 1 since argv[0] is the name of the program

/* append one file to the another */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv) {
    int c;
    FILE *from, *to;
    if (argc != 3) {   /* Check the arguments. */
        fprintf(stderr, "Usage: %s from-file to-file\n", *argv);
        exit(1);
    }
    if ((from = fopen(argv[1], "r")) == NULL) {
        perror(argv[1]);          /* Open the from-file */
        exit(1);
    }
    if ((to = fopen(argv[2], "a")) == NULL) {
        perror(argv[2]);     /* Open the to-file */
        exit(1);
    }
    /* Read one file and append to the other until EOF */
    while ((c = getc(from)) != EOF)
        putc(c, to);
    /*close the files */
    fclose(from);
    fclose(to);
    exit(0);
}
©Gustavo Alonso, ETH Zürich.
printf

• Just another function, but very useful!

```c
#include <stdio.h>
int i = 79;
const char *s="Mothy";
printf("My name is %s and I work in CAB F %d\n", s, i);
```

• First argument is format string
  – see “man 3 printf” for all the (many) options
• Remaining arguments are arbitrary
  – but must match the format
• You will see other “printf-like” functions