CSCI-UA.0201

Computer Systems Organization

C Programming – Basics (Part 1)

Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies
In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of *The C Programming Language* by Kernighan & Ritchie caused a revolution in the computing world.
Why C?

• Mainly because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:
  – Operating Systems
  – Language Compilers
  – Assemblers
  – Text Editors
  – Print Spoolers
  – Network Drivers
  – Language Interpreters
  – Utilities
Interesting Opinion About C

You might never use it professionally, but it contains a lifetime of lessons. And the hardest problems, the ones that the top engineers are asked to solve, will sooner or later hit some foundational C code.

Here are some things that are written in C:

- The Java virtual machine is written in ANSI C
- Linux is written in C (and some assembly, but mostly C)
- Python is written in C
- Mac OS X kernel is written in C
- Windows is written in C and C++
- The Oracle database is written in C and C++
- Cisco routers, those things which connect the Internet, also C

Name anything that is foundational, complex, and performance critical. It was written in C, with a sprinkling of assembly thrown in.

C will make you a better Java programmer. You'll know when the JVM is using the stack and when it's using the heap, and what that means. You'll have a more intuitive sense of what garbage collection does. You'll have a better sense of the relative performance cost of objects versus primitives.
Your first goal: Learn C!

• Resources
  – These lectures
  – Additional online resources (some links on the course website)

• Learning a Programming Language
  – The best way to learn is to write programs
# Writing and Running Programs

```c
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

1. Write text of program (source code) using a text editor, save as text file e.g. my_program.c

2. Run the compiler to convert program from source to an “executable” or “binary”:
   ```
   $ gcc -Wall -g -o my_program my_program.c
   ```

3. Compiler gives errors and warnings; edit source file, fix it, and re-compile

Run it and see if it works 😊
```bash
$ ./my_program
Hello World
$ |
```
$ gcc -Wall -g -o my_program my_program.c

- generate all warnings
- keep debugging information
- name the generated executable (default: a.out)
- one or more C files
About C

• **Procedural language**
  – Functions calling each other, starting with main().

• **Case-sensitive**
C Syntax and Hello World

#include <stdio.h>

/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}

This is a comment. The compiler ignores this.

#include inserts another file. "h" files are called "header" files. They contain stuff needed to interface to libraries and code in other "c" files.

Blocks of code are marked by { ... }

The main() function is always where your program starts running.

Print out a message. '\n' means "new line".

Return '0' from this function
The simplest C Program

```c
#include <stdio.h>

/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

Preprocessing

```
__extension__ typedef unsigned long long int __dev_t;
__extension__ typedef unsigned int __uid_t;
__extension__ typedef unsigned int __gid_t;
__extension__ typedef unsigned long int __ino_t;
__extension__ typedef unsigned long long int __ino64_t;
__extension__ typedef unsigned int __nlink_t;
__extension__ typedef long int __off_t;
__extension__ typedef long int __off64_t;
extern void flockfile (FILE *__stream) ;
extern int ftrylockfile (FILE *__stream) ;
extern void funlockfile (FILE *__stream) ;
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}
```

my_program

Compile
Include <stdio.h>
/* The simplest C Program */int main(int argc, char **argv) {
    printf("Hello World"");
    return 0;
}

__extension__ typedef  unsigned long long int   __dev_t;
__extension__ typedef  unsigned int   __uid_t;
__extension__ typedef  unsigned int   __gid_t;
__extension__ typedef  unsigned long int   __ino_t;
__extension__ typedef  unsigned long long int   __ino64_t;
__extension__ typedef  unsigned int   __nlink_t;
__extension__ typedef  long int   __off_t;
__extension__ typedef  long long int   __off64_t;
extern void flockfile (FILE *__stream)  ;
extern int ftrylockfile (FILE *__stream)  ;
extern void funlockfile (FILE *__stream)  ;
int main(int argc, char **argv) {
    printf("Hello World"");
    return 0;
}

Preprocessing

In Preprocessing, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#’ is a line that is interpreted by the Preprocessor.

- Include files are “pasted in” (#include)
- Macros are “expanded” (#define)
- Comments are stripped out (/* */ , //)
- Continued lines (i.e. very long lines) are joined (\ )
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello World\n");
    return 0;
}

my_program

The compiler then converts the resulting text into binary code the CPU can run directly.
The compilation process involves really several steps:
- **Compiler**: high level language → assembly
- **Assembler**: assembly → machine code
- **Linker**: links all machine code files and needed libraries into one executable file.

When you type `gcc` you really invoke the compiler, assembler, and linker.
What is “Memory”?

- Is like a big table of numbered slots.
- Each slot stores a byte.

- The number of a slot is its Address.
- One byte Value can be stored in each slot.

Some “logical” data values span more than one slot, like the character string “Hello\n”

A Type names a logical meaning to a span of memory. Some simple types are:

- **char**: a single character (1 slot)
- **char [10]**: an array of 10 characters
- **int**: signed 4 byte integer
- **float**: 4 byte floating point

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>7</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>8</td>
<td>‘o’ (111)</td>
</tr>
<tr>
<td>9</td>
<td>‘\n’ (10)</td>
</tr>
<tr>
<td>10</td>
<td>‘\0’ (0)</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
What is a Variable?

A Variable names a place in memory where you store a Value of a certain Type.

You first Define a variable by giving it a name and specifying the type, and optionally an initial value.

```
char x;  // Initial value of x is undefined
char y='e';  // Initial value of y is 'e' (101)
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>x</td>
<td>5</td>
<td>'e' (101)</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Type is single character (char)

The compiler puts them somewhere in memory.
Multi-byte Variables

Different types consume different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char).

```
char x;
char y='e';
int z = 0x01020304;
```

- `char` consumes 1 byte
- `int` consumes 4 bytes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
<td>?</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

- `0x` means the constant is written in hex
- An int consumes 4 bytes
- Padding may occur to align on word boundaries
Scope

Every Variable is Declared within some scope. A Variable cannot be referenced from outside of that scope.

Scopes are defined with curly braces { }.

The scope of Function Arguments is the complete body of the function.

The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block.

The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called Global Vars.

```c
void p(char x) {
    char y;
    char z;
}
char z;

void q(char a) {
    char b;

    {
        char c;
    }
}
char d;
```
Now that we know about variables, let’s combine them to form expressions!

Expression

\[ X = 2 \times Y + Z; \]

Statement
How Expressions Are Evaluated?

Expressions combine **Values** using **Operators**, according to **precedence**.

\[
\begin{align*}
1 + 2 * 2 & \rightarrow 1 + 4 \rightarrow 5 \\
(1 + 2) * 2 & \rightarrow 3 * 2 \rightarrow 6
\end{align*}
\]

Comparison operators are used to compare values.

In C: 0 means “false”, and **any other value** means “true”.

\[
\begin{align*}
\text{int } x=4; \\
(x < 5) & \rightarrow (4 < 5) \rightarrow \text{<true>} \\
(x < 4) & \rightarrow (4 < 4) \rightarrow 0 \\
((x < 5) || (x < 4)) & \rightarrow (\text{<true>} || (x < 4)) \rightarrow \text{<true>}
\end{align*}
\]

Not evaluated because first clause was true
Precedence

• Highest to lowest
  • ()
  • *, /, %
  • +, -

When in doubt, use parenthesis.