

“Prof” Joe's Tutorial on

***C Programming for Java & C++
Developers***

Joseph Phillips
2015 June 17

What is C?

- Programming language from which the syntax (and some of the mentality) of C++, Java, and C# are based
- The “portable assembly language”
 - No virtual machine here!
 - It does not hide the underlying implementation of datatypes
- A pre-object oriented language
 - It has structures, but think of these as classes:
 - without inheritance
 - no methods, only member vars
 - all member variables are `public`

Why use C?

- 1) Because you don't want *No stinkin' virtual machine* between you and the *soul* of the computer (CPU + OS)
- 2) Because you think you can do a *faster* job than:
 - the JVM's garbage collector at managing memory
 - C++'s virtual tables at figuring out which function to do with a given object at run-time
- Depending on the program (and your coding ability), maybe you can.

C is not so scary!

- Conditionals work pretty-much the same:
 - `if (..) {..}, if(..){..} else{..}`
 - `switch (..) { case ... default }`
- Loops work pretty-much the same:
 - `for(..;..;..)`
 - `do {..} while(..);`
 - `while (..) {..}`
- Functions are simpler: not in classes.
 - Just call them without any `obj.meth()` or `objPtr->meth()`
 - Just say `func()`

Programming in C (1)

(gnu/Linux)

(1) Type in the following program:

```
#include <stdlib.h>
#include <stdio.h>

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

Programming in C (2)

(2) Compile it:

```
$ gcc hello.c -o hello -g
```

(3) Run it:

```
$ ./hello
```

Hello world!

(Yeah, I know. No big surprise . . . but that is a good thing!)

Let's discuss this bad boy (1)

- Header file inclusion:

```
#include <stdlib.h>
```

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

- 1) Anything beginning with a “#” is a command for the pre-processor, not the compiler proper
 - The pre-processor's job is just to get and substitute text (in this case, two header files)

Let's discuss this bad boy (2)

2) C (and C++) distinguish between header (.h) files and source (.c or .cpp) files.

- Header files: tell the compiler “***Oh, this thing **might** exist. If it does then it has name such-and-such, parameters such-and-such, and return type such-and-such***”
- Source files: tell the compiler “***Here is the **actual variable or function** that does it. Allocate space, dammit!***”

Let's discuss this bad boy (3)

```
int main (int    argc,  
          char* argv[])  
    )
```

- Like in C++ the action commonly starts at **main()**.
 - It's just another function. It takes parameters and returns a value.
 - First parameter (**int argc**) tells how many command line arguments there are. (It is almost always at least one, **argv[0]** tells the process the name of the program being run.)
 - Second parameter (**char* argv[]**) is an array of char ptrs that points to each command line argument
 - By convention always called **argc** and **argv**.

Let's discuss this bad boy (4)

```
printf("Hello world!\n");
```

- Just call a function without any **this** reference/ptr.
 - No **object.function()**
 - No **objectPtr->function()**
- No classes
- There are “structures” (**struct**)
 - Structs' sole purpose is to group member vars
 - Only have member variables, *no* methods
 - All member vars are **public**.

Let's discuss this bad boy (5)

return(EXIT_SUCCESS);

- I **told** you **main ()** is just another function and returns a value!
 - Returned to OS.
- **EXIT_SUCCESS** is the integer **0**.
 - Means “*I did what I was supposed to do.*”
- **EXIT_FAILURE** is the integer **1**.
 - Means “*I encountered a problem, but didn't crash.*”

Let's discuss this bad boy (6)

```
$ gcc hello.c -o hello -g
```

- **gcc** is the GNU C compiler
 - **g++** is the GNU C++ compiler
- **-o hello** means “Name the output file **hello**”
 - If this is missing your program will have name **a.out**
- **-g** means “Add debugging information”
 - More about the GNU debugger **gdb** later.
- Many, many more options to **gcc** and **g++**

Let's discuss this bad boy (7)

\$ **./hello**

- Runs our program **hello**
- The “./” before the **hello** means “*Run the **hello** program that is in the current directory.*”
 - **../hello** means “*Run the **hello** program that is in the directory above the current one.*”

The Mentality of C

- **Be efficient!** (Safety is important, but speed is better)
 - Short-circuiting when computing `&&` and `||`
 - No index checking for arrays
- **Be flexible!**
 - C does not have constants `true` or `false`. The integer `0` means “false”, any other `int` means “true”.
- **Functions tells whether or not they succeeded in their return value.**
 - `printf()` returns `-1` on error (rarely checked)
 - Unfortunately there are two standards. Sometimes “0” means “success” and “1” means “failure”. Other times it is the other way around.

Output with printf()

- Output in C with `printf()` (“print-formatted”)
 - `printf("template" , expr1 , ... exprn)`
- Constant formatting:
 - `printf("\tI just print \"hello\".\n");`
 - What do these mean? `\t` `\"` `\n`
- Substitution formatting:
 - `int i=1; printf("%d %d %d Go!\n" , 3 , 1+1 , i);`
 - `%d` = decimal integer
 - `%x` `%X` = hexadecimal integer
 - `%c` = single char
 - `%s` = C-string (*i.e.* pointer to char: `char*`)
 - `%f` `%g` = double or floating point
 - `%p` = An address (*e.g.* a pointer's value)

Your turn!

Write a C program to print the multiplication table
from $1*1$ to $10*10$

Input with fgets():

- `fgets(charArrayToWriteInto,
 sizeofCharArray,
 fileStreamFromWhichToRead)`

- Example:

```
char array[64];  
printf("Some text? ");  
fgets(array, 64, stdin);
```

- **stdin** is C's **System.in** or **cin**.

Input with fgets(): Text

- **fgets()** places the '**n**' of the enter key in the string
- This removes it:

```
char* cPtr=strchr(text, '\n');  
if (cPtr != NULL)  
    *cPtr = '\0';
```

```
#include <stdlib.h>  
#include <stdio.h>  
#include <string.h>  
#define    LEN 64  
  
int main ()  
{ char* cPtr;  
  char name[LEN];  
  printf("Name? ");  
  fgets(line, LEN, stdin);  
  cPtr=strchr(name, '\n');  
  if (cPtr!=NULL)  
      *cPtr='\0';  
  printf("Hello ");  
  printf("%s\n", name);  
  return(EXIT_SUCCESS);  
}
```

Input with fgets(): Numbers

- **Almost always** should get any input as string
 - Convert string to integer or float
- `atoi()`:
 - C's `Integer.parseInt()`
 - Instead of throwing exception on error, returns 0
- `atof()`:
 - C's `Double.parseDouble()`
 - Instead of throwing exception on error, returns 0.0

```
#include <stdlib.h>
#include <stdio.h>
#define    LEN 64

int main ()
{
    char line[LEN];
    printf("Enter a #: ");
    fgets(line,LEN,stdin);
    int    i = atoi(line);
    float f = atof(line);
    printf("i = %d\n",i);
    printf("f = %g\n",f);
    return(EXIT_SUCCESS);
}
```

Your turn!

Write a program that allows the user to enter a positive integer n and writes the multiplication table from $1 * 1$ to $n * n$

Pointers (0)

- Declaring pointers: *Type* typePtr*

```
int* intPtr;
```

```
char* charPtr;
```

```
MyClass* myClassPtr;
```

- Getting addresses: *&var*

```
int i = 10;
```

```
printf("i=%d and lives at%p\n", i, &i);
```

- Putting it together:

```
int i = 10;
```

```
int* intPtr = &i;
```

```
const char* charPtr = "string const";
```

Pointers (00)

- All a pointer is a variable that holds the address of an object (like another var)

(Ain't so scary, is it? :)

- Consider the simple program to the right
 - It increments variable **i** using a pointer from 10 to 11, and prints it.

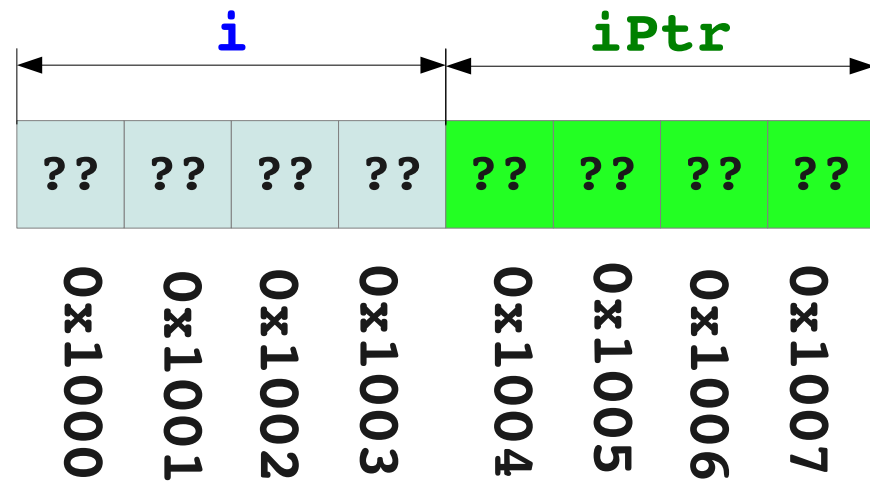
```
#include <stdlib.h>
#include <stdlib.h>

int main()
{
    int    i;
    int*   iPtr;

    i      = 10;
    iPtr = &i;
    (*iPtr)++;
    printf("%d\n", i);
    return(EXIT_SUCCESS);
}
```

Pointers (1)

(1) `int i;`
`int* iPtr;`



- The act of declaring the variables makes the compiler write code that allocates space for them
 - Assume 32-bit architecture (4 bytes per var)
 - Assume little-endian byte ordering

Pointers (2)

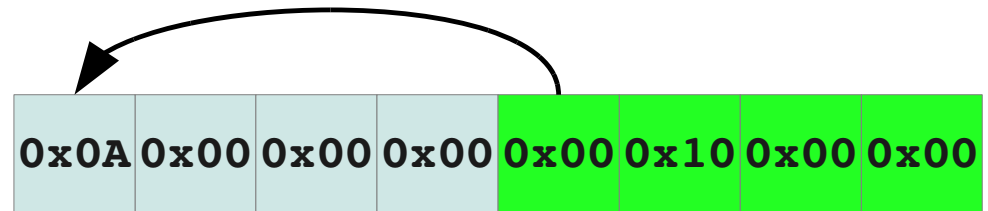
(2) **i = 10;**

0x0A	0x00	0x00	0x00	??	??	??	??
------	------	------	------	----	----	----	----

- Assigning 10 to **i** places **0x0A** (= 10 decimal) in the first byte of **i**.
 - **Remember**: little endian byte ordering

Pointers (3)

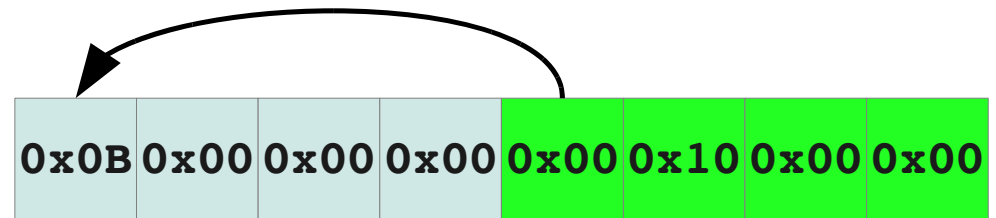
(3) **iPtr** = &**i**;



- Assigning the address of **i** (&**i** in C notation) to **iPtr** puts where **i** lives in **iPtr**.
 - **i** lives at **0x1000**, so **iPtr** now holds **0x00001000**.
 - We say “**iPtr** points to **i**” Drawing the arrow helps too.

Pointers (4)

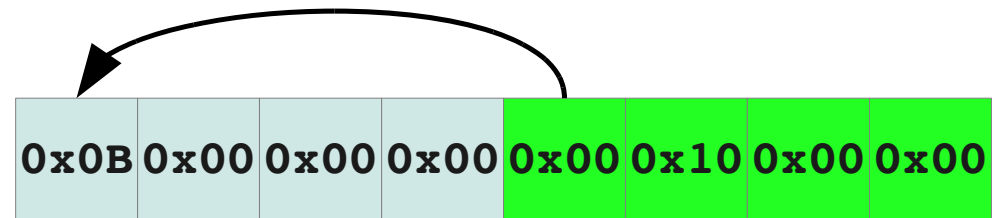
(4) `(*iPtr)++;`



- `*iPtr` “dereferences” `iPtr`. This just means “*Follow the pointer to the object*”.
- Thus `(*iPtr)++` means the same thing as `i++`
 - `i` goes from **10 (0x0A)** to **11 (0x0B)**.
- Must say `(*iPtr)++`.
 - `*iPtr++` means “Return the thing to which `iPtr` points, and make `iPtr` point to the next object”.
 - Thus, `iPtr` would change to **0x1004** instead of `i` changing to **11**.

Pointers (5)

(5) `printf("%d", i);`



- When we print `i` after doing `(*iPtr)++` we get `i`'s new value: **11 (0x0B)**.

Pointers (6)

- No legally-accessible object is allowed at address **NULL** (generally integer 0).
- **NULL** is a nice marker for “*This pointer has not been initialized, or has no legal value.*”

```
#include <stdlib.h>
#include <stdlib.h>
```

```
// What will I do?
```

```
int main()
{
    int* iPtr;

    iPtr = NULL;
    printf("%d\n", *iPtr);
    return(EXIT_SUCCESS);
}
```

Functions (1)

- C compilers want to know about functions before they are called:

```
#include <stdlib.h>
```

```
#include <stdlib.h>
```

```
//Compiler will complain:
```

```
int main()
```

```
{
```

```
    foo();
```

```
    return(EXIT_SUCCESS);
```

```
}
```

```
void foo ()
```

```
{
```

```
    printf("Hello world!\n");
```

```
}
```

Functions (2)

```
#include <stdlib.h>
#include <stdlib.h>
```

// Solution 1: reorder

```
void foo ()
{
    printf("Hello world!\n");
}

int main()
{
    foo();
    return(EXIT_SUCCESS);
}
```

```
#include <stdlib.h>
#include <stdlib.h>
```

// Soln 2: declare before use

```
extern void foo();

int main()
{
    foo();
    return(EXIT_SUCCESS);
}

void foo ()
{
    printf("Hello world!\n");
}
```