

CSE 374 Lecture 9

Declarations, control, printf



Hello World in C

```
#include <stdio.h>

/**
 * Compile this file with:
 *     gcc -o hello hello.c
 */
int main(int argc, char **argv)
{
    printf("Hello, World!\n");
    return 0;
}
```

- Compile: `gcc hello.c`
 - ◆ creates executable `a.out`
- Or: `gcc -Wall -std=c11 -o hello hello.c`
 - ◆ `Wall` - turns all warnings on
 - ◆ `C11` - specifies using C11 standard libraries
 - ◆ Creates executable `hello`
- Run: `./a.out` or `./hello`
 - ◆ Exits with '0' (`return 0;`)

```
// includes for functions & types
defined elsewhere
#include <stdio.h>
#include "localstuff.h"
// symbolic constants
#define MAGIC 42
// global variables (if any)
static int days_per_month[ ] = { 31,
28, 31, 30, ...};
// function prototypes
// (to handle "declare before use")
void some_later_function(char, int);
// function definitions
void do_this( ) { ... }
char *return_that(char s[ ], int n)
{ ... }
int main(int argc, char ** argv) { ... }
```

Source File Structures

Preprocessor

Pre-processes your C code before the compiler gets to it.

- Follows commands prefaced by ‘#’
- Includes content of header files
- Defines constants and macros
- Conditional compilation (not covered right now)

File inclusion

- `#include <foo.h>`
 - ◆ Searches for `foo.h` in “system include” directories (`/usr/include`, etc)
- `#include "foo.h"`
 - ◆ Starts by searching in current directory (allows coder to break project into smaller files)
- Include include file’s preprocessed contents
- Recursively include all the includes from original file
- Use `gcc -I dir1` to tell gcc to look for include in `dir1`

Preprocessor Cont.

Define constants

```
#define PI 3.14
#define NULL 0 // in stdlib

#define TRUE 1
#define FALSE 0
```

And macros

```
#define min(X, Y) ((X) < (Y) ? (X) : (Y))
```

Constants are ALL_CAPS to differentiate them from other variables.

Defined constants will override variables of the same name used in the code.

Shadow with another #define, or #undef

Declarations

Introduces a name, and defines properties (such as the datatype)

Does not actually create the named thing

Things can be declared more than once (but once per scope)

Often shared, declared in #include files

MUST be declared before they are used.

```
Int count; // an integer called count
```

```
int* countptr;
```

```
int count[10]; // an integer array
```

```
Int adding(int a, int b);  
// function the returns an int
```

```
void printing (char *s);
```

Declarations Cont.

You can put multiple declarations on one line, e.g., `int x, y;` or `int x=0, y;` or `int x, y=0;`, or ...

But `int *x, y;` means `int *x; int y;` – you usually mean (want) `int *x, *y;`

Common style rule: one declaration per line (clarity, safety, easier to place comments)

Array types in function arguments are pointers(!)

Definitions

Defines properties of item; this happens only ONCE, even if the item is declared more than once.

Linker-error will occur if an item is used but not defined.

To use something before it is defined, you must declare it before you use it (forward declaration).

```
int count=4

countptr = &count;

int count[3] = {1,2,3};

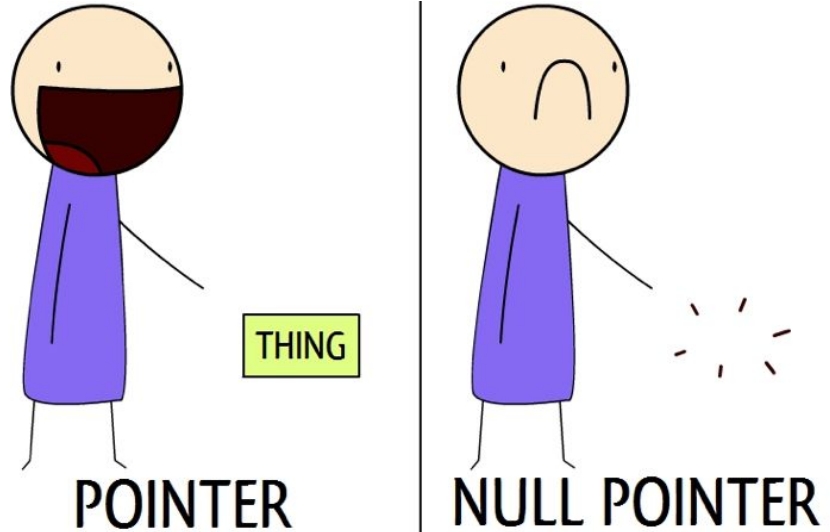
int adding(int a, int b) {
    return (a+b);
}

void printing (char *str){
    printf("%s\n", str);
}
```

Definitions

- `Int *arrspace = myArr;`
- Arrays that rely on run-time info to determine size are dynamically allocated to the heap (and declared `*array` syntax)
- Define as `NULL` until otherwise defined.

<https://www.codewithc.com/understanding-c-pointers-beginners-guide/>



L-values v. R-values

Left Side

Evaluated to locations (addresses) 

Right Side

Evaluated to values (the contents
at the address)

Values may be numbers (or characters) OR addresses

```
9 = x;          // Nonsense, because 9 isn't a LOCATION
int x = 1;      // Stores the VALUE 1 at a LOCATION which has the LABEL x.
x = 2;         // Stores the VALUE 2 at the LOCATION x.
int* xPtr = &x; // Stores VALUE of address of x at a LOCATION labelled xPtr.
*xPtr = 3;     // Stores VALUE 3 at a LOCATION defined by address stored in xPtr.
int** xx = &(&x); // Nonsense, the r-value needs to resolve to a value.
                // &x does indeed represent a value (the address x), but
                // &(&x) refers to the address of the address of x -
                // which is just a number and not stored anywhere
```

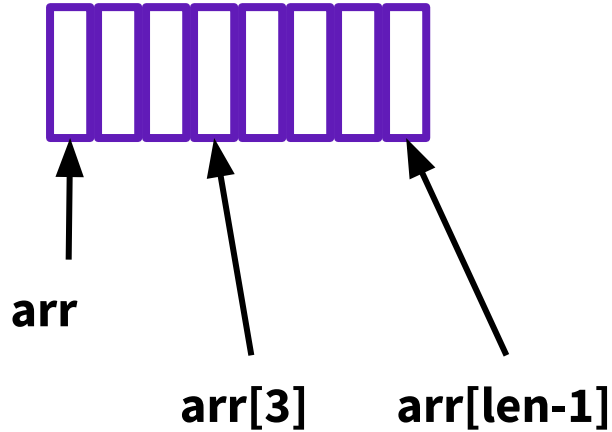
Initialization

Memory allocation and initialization are not the same thing

Unlike Java, you MUST provide a value to initialize a bit of memory

It is possible to access un-initialized bits
unlike Java with sets defaults and checks for initialization
best case scenario: you crash

Arrays



- `int myArr[10];`
 - User must store length (10).
- `Int *arrspace = myArr;`
 - Implicit conversion
- `myArr[3]` is ??
 - (Not automatically initialized to any value.)
- Arrays MUST be declared with a known length (the compiler needs to allocate space)
- Arrays that rely on run-time info to determine size are dynamically allocated to the heap (and declared `*array` syntax)

Arrays

Contiguous blocks in memory

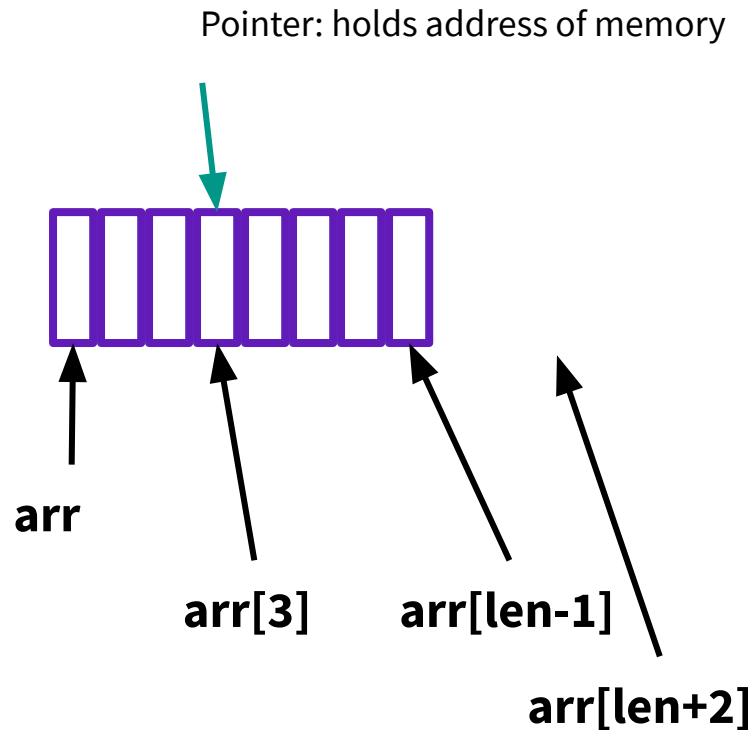
Declare as

```
Datatype arr[len]
```

Has type

```
Datatype*
```

Stores the location in memory of the first value; when arrays are passed passes this memory location



Danger, Will Robinson!!

Control constructs

Similar to Java: if, while, switch

Break, continue, etc.

<https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html#Statements>

No Boolean type!

Use integers, can declare constants.

Generally, 0/NULL => False

Anything else => True

I/O : Printf, scanf

Printf and scanf are two I/O functions, prototyped in `stdio.h`

- `Printf` (print-format)
- `int printf(const char *format, ...)`
- 'Format' is a string that can contain format tags
- + additional arguments to match tags
- Number of arguments better match number of %
- Corresponding arguments better have the right types (`%d`, int; `%f`, float; `%e`, float (prints scientific); `%s`, \0- terminated `char*`; ... Compiler might check, but not guaranteed
 - ◆ best case scenario: you crash
- `printf("%s: %d %g\n", p, y+9, 3.0)`
- `scanf` (gets input, formatted)
- `int scanf(const char *format, ...)`
- 'Format' is a string that can contain format tags
- + additional arguments to match tags - should be pointers to the right data type so input can be stored in them
- `scanf("%d %s", &n, str);`
- `scanf("%*s %d", &a);`
 - ◆ `%*s` ignores string until space, then reads in an integer