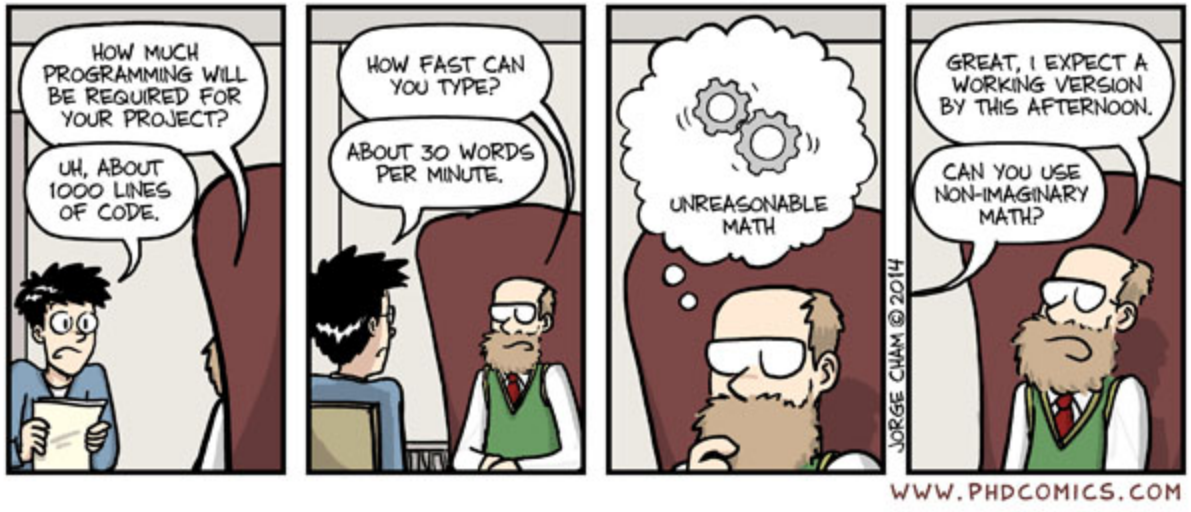
Introduction to programming in C for scientists and engineers

by Marcin Ziolkowski and Ashwin Srinath



Agenda for this week

1. Introduction
2. Data types, variables, pointers, compilation, loops, conditionals, functions, arrays
3. Functions, structures, pointers, I/O, conditionals
4. Preprocessor, compiler, linking, Makefiles
5. GSL, BLAS/LAPACK, debugger, profiler
6. OpenMP, performance programming, controlling the program, good practice, Doxygen

Development enviroment

1. Your own laptop/workstation (you need a C compiler)
2. Palmetto cluster (use your account if you have one)
3. If you do not have an account use `hpcguestXX` where `XX` is your number on the list

```
$ ssh hpcguestXX@user.palmetto.clemson.edu
```

```
$ qsub -I -l walltime=1:30:00
```

```
$ module load gcc/4.8.1
```

`$ gcc file.c -o file.x` or `make` to build all programs

The class repository

All materials for the class will be put in a Github repository. You can clone the whole repository using command

```
git clone https://github.com/zziolko/c-class.git
```

We will be updating the repository as we go with the class. You can always use command `git pull` to update your copy.

Why C

1. Simple language for elegant code
2. Readable code, yet offers great performance
3. Huge number of scientific codes are written in C
4. Large number of libraries available
5. Important for learning C++
6. Not FORTRAN so friends won't laugh at you *:-)*

Like in every language, also in C the things you need to know to start coding are

1. General structure of the program
2. Definition of variables
3. Conditional statements
4. Loops
5. Functions

Hello world program

```
01. #include <stdio.h>
02.
03. int main(){
04.
05.   printf("Hello world!\n");
06.   return 0;
07.
08. }
```

The most simple C program then looks like this

```
01. int main(){
02.   return 0;
03. }
```

which defines a `main` function which is the only executed part.

Before we begin you need to know this

1. C requires a compiler : `$ gcc hello.c -o hello.x`
2. Compiler's main job is to translate `int a = 1` (human readable code) to `%$^#$@%&@@4` (machine readable code)
3. C is case sensitive
4. Names can be long (use that fact)
5. It is all about functions
6. Comments `/* ... */` and sometimes `//`
7. Every line (with some exceptions) ends with `;`

and this ...

1. Language is compact : `auto, break, case, char, const, continue, default, do, double, else, enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile, while`
2. There are several of standards : C90 (ANSI), C99, C11, e.g. `gcc -ansi hello.c -o hello.x` or `gcc -std=c99 hello.c -o hello.x`

Data types

Data types

1. Two basic data types 1) integer type and 2) floating point
2. Enumerated types
3. Derived types (pointers, arrays, structures, unions, functions)
4. `void`

Data types

Name	Storage	Range
`char`	1 byte	-128 to 127 or 0 to 255
`unsigned char`	1 byte	0 to 255
`signed char`	1 byte	-128 to 127
`int`	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647

Data types

Name	Storage	Range
`unsigned int`	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
`short`	2 bytes	-32,768 to 32,767
`unsigned short`	2 bytes	0 to 65,535
`long`	4 bytes	-2,147,483,648 to 2,147,483,647
`unsigned long`	4 bytes	0 to 4,294,967,295

Data types

Name	Storage	Range	Precision
`float`	4 byte	1.2E-38 to 3.4E+38	6 decimal places
`double`	8 byte	2.3E-308 to 1.7E+308	15 decimal places
`long double`	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

Variables

Variables

1. All variables need to be defined by `type name;` before they are used
2. Variables have limited scope (life time) to parent block `{ ... }`
3. Special `void` type. Variable which does not hold a value.

Variables are initialized like this `type name;`

```
01. int a;
02. float b = 1.0; double c, d, e;
```

Variables

1. Name of a variable can contain letters, numbers, underscore
2. Name has to start with a letter or an underscore character
3. Variables can be defined as constant using `const` keyword or `#define` macro, e.g. `const type variable = value;` or `#define identifier value`

Variables

Variables have storage class assigned to it

1. `auto` - defaults, limits the scope
2. `register` - not in RAM but on a register for quick access
3. `static` - variable has a lifetime of the program, no creation and destruction every time it is needed
4. `extern` - variable visible to all program files

Operators

1. Arithmetic operators `+`, `-`, `*`, `/`, `%` (modulus), `++` (increments by 1), `--` (decrements by 1)
2. Relational operators `==` (equivalence) , `!=` (negation), `>`, `<`, `>=`, `<=`
3. Logical operators `&&` (AND), `||` (OR), `!` (NOT)
4. Binary operators
5. Assignment operators `=`, `+=`, `-=`, `*=`, `/=`, `%=`
6. Misc `sizeof()`, `&` (address), `*` (pointer), `? :` (conditional)

Pointers

Pointers

Pointer is a variable with its value being an **address** to another variable (address of a memory location). Pointers are initialized as `type *name`

```
01. int *i;
02. double *f;
03. char *c;
```

The actual value is the same for all, i.e. a memory location. Value can be accessed by `&variable`

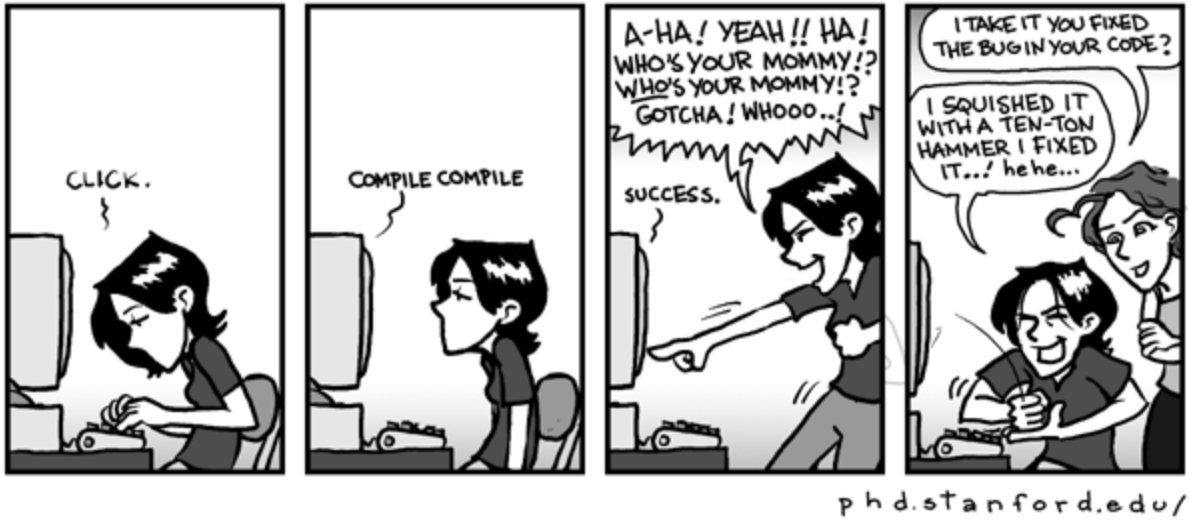
Pointers

```
01. int main(){
02.   int var = 1;
03.   int *i = NULL;
04.
05.   i = &var;
06. }
```

`NULL` is not necessary is this case but defines a pointer that “points” to nothing.

Compilation

Compilation

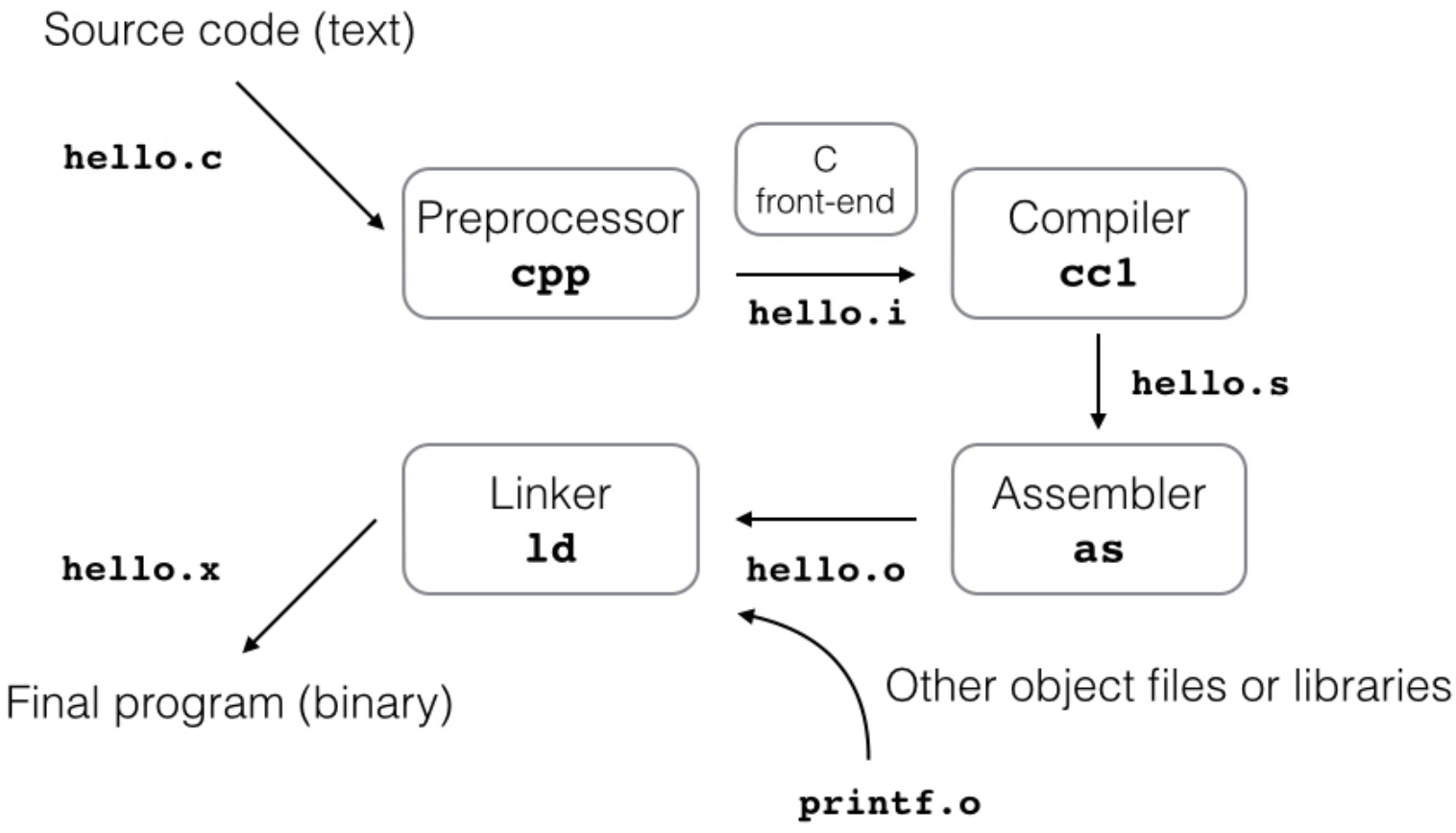


Compiler in general is your friend, read the messages and you will fix all problems with your code ... unless your code is buggy

Consider the simple Hello World code

```
01. #include <stdio.h>
02.
03. int main(){
04.
05.    printf("Hello world!\n");
06.    return 0;
07.
08. }
```

Compilation



Loops

Loops

Purpose : When a block of code needs to be executed more than one time

General structure : condition -> execution

Loops in C:

1. `for` - execute based on internal counter
2. `do while` - execute and check condition
3. `while` - check condition and execute

for loop

```
01. #include <stdio.h>
02. int main(){
03.   int iter;
04.   int max_iter = 10;
05.   for(iter = 0; iter < max_iter; iter++) {
06.     printf("Value of iter is %d \n", iter);
07.   }
08.   return 0;
09. }
```

do while loop

```
01. do {
02.   int i;
03.   printf("Iter %d : ", iter);
04.   /* nested loop */
05.   for(i=0; i<5; i++) printf("%d", i);
06.   printf("\n");
07.   iter++;
08. } while( iter < max_iter );
```

while loop

```
01. #include <stdio.h>
02. int main(){
03.   int i = 0, max_i = 100;
04.   while( i < max_i ) {
05.     printf("Value of i = %d \n", i);
06.     i = i + 10;
07.   }
08.   return 0;
09. }
```

Making decisions

Conditionals

```
% phd.m
%
% author: Cecilia
% date: 09/08/05

load THESIS_TOPIC

while (funding==true)
    data = run_experiment(THESIS_TOPIC);
    GOOD_ENOUGH = query(advisor);
    if (data > GOOD_ENOUGH)
        graduate();
        break
    else
        THESIS_TOPIC = new();
        years_in_gradschool += 1;
    end
end
```

At some point in the program you may want to make a decision about further path ...

Conditional `if else`

```
01. /* generate normalized random numbers */
02. for(i; i<max_i; i++){
03.   float val = (float)rand()/(float)(RAND_MAX);
04.   if( val > 0.5 ) {
05.     printf("%f *** \n", val);
06.   } else {
07.     printf("%f \n", val);
08.   }
09. }
```

Conditional operator `?:`

```
01. #include <stdio.h>
02. #include <stdlib.h>
03. int main(){
04.   int status = 2;
05.   return ( status == 0 ) ? EXIT_SUCCESS : EXIT_FAILURE;
06. }
```

In general `condition ? do if true : do if false`

Functions

Functions

Function is a block of code that performs a specific task and can be reused.

Function needs to be declared (to tell the compiler about its existence) and then defined (the actual body of the function)

```
01. type function_name(arguments){
02.   body
03. }
```

Functions

```
01. float random_number() {
02.   float val;
03.   srand((unsigned int)time(NULL));
04.   val = (float)rand()/(float)(RAND_MAX);
05.   return val;
06. }
```

The function can be invoked in the program like this

```
float random_value = random_number();
```

Functions - declaration and definition

```
01. void generate_set(int);
02. float random_number();
03. int main(){
04.   generate_set(10);
05.   return 0;
06. }
07.
08. void generate_set(int n){
09. }
```

Functions - definitions in a header file

Definitions may be places in a separate file - header file which is then included by the preprocessor.

```
01. #include <stdio.h>
02. #include "my-functions.h"
03. int main(){
04. }
```

Note `#include "file.h"` not `#include <file.h>`

Arrays

Static arrays (size is known at the compile time)

```
01. int var0[10]; /* Simple array */
02. /* Simple array with assigned values */
03. int var1[10] = {0, 1 , 2, 3, 4, 5, 6, 7, 8, 9};
04. /* Size is not needed if we initialize all elements */
05. int var2[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
06. /* Size has to be a constant */
07. const int size3 = 10;
08. float var3[size3];
```

Accessing elements

Indexing starts with 0

```
01. int i;
02. for(i=0; i<size3; i++){
03.   var3[i] = (float)i;
04. }
05. for(i=0; i<size3; i++){
06.   printf("var3[%d] = %f \n", i, var3[i]);
07. }
```

Dynamic arrays (size does not need to be known at the compile time)

```
01. double *vector; /* pointer */
02. /* memory allocation */
03. vector = (double *)malloc(size*sizeof(double));
04. for(i=0; i<size; i++) vector[i] = (double)i;
05. for(i=0; i<size; i++)
06.     printf("vector[%d] = %f \n", i, vector[i]);
07. free(vector); /* memory release */
```

Exercise

Write a short program do generate Fibonacci numbers

1. Start with two numbers `0` and `1`
2. Generate next one by summing up last two i.e. `0`, `1`, `(0+1)=1`, `(1+1)=2`, `(1+2)=3`,
3. You should get: `0`, `1`, `1`, `2`, `3`, `5`, `8`, `13`, `21`,
4. Generte a golden ratio `n/(n-1)` where `n` is a Fibonacci number