ICGE Module 4 Session 3   C programming

Topics for today:

- Why program in C?
- Microchess example
- Language speed comparisons
- Tour of C features and idiosyncracies
  - Compilation process
  - C delimiters
  - Variable declaration & initialization
  - C for loops
  - Memory management

Some possible reasons to learn and use C/C++

Speed: (but it's complicated)

http://basharalfallouji.com/2009/03/18/

	language	|-	|---	25%	median	75%	---|	-|
☑	C++ GNU g++	1.00	1.00	1.00	**1.02**	1.12	1.30	21.46
☐	C GNU gcc	1.00	1.00	1.00	**1.09**	3.58	7.45	20.41
☐	ATS	1.00	1.00	1.09	**1.22**	3.33	6.69	20.44
☑	Java 6 -server	1.15	1.15	1.37	**1.81**	2.86	5.08	38.33
☑	Haskell GHC	1.00	1.00	1.36	**2.19**	3.02	5.33	5.33
☑	C# Mono	1.22	1.22	1.79	**2.63**	4.64	8.91	42.80
☐	OCaml	2.17	2.17	3.13	**3.98**	7.42	13.85	19.14
☑	Lisp SBCL	1.70	1.70	1.86	**4.37**	6.49	13.45	39.87
☐	Pascal Free Pascal	1.12	1.12	1.71	**4.50**	6.39	13.42	16.79
☐	Scala	1.19	1.19	2.21	**4.97**	8.33	14.68	14.68
☐	Clean	1.40	1.40	2.30	**5.64**	6.63	9.44	9.44
☐	Ada 2005 GNAT	1.28	1.28	1.65	**6.49**	9.73	21.85	60.99
☐	Fortran Intel	1.17	1.17	1.35	**6.61**	9.64	10.33	10.33
☑	Erlang HiPE	3.89	3.89	4.24	**8.23**	19.88	43.33	64.61
☐	Lua LuaJIT	1.33	1.33	6.11	**11.74**	20.77	42.77	88.88
☑	Scheme PLT	1.05	1.05	8.52	**16.22**	41.46	90.87	139.46
☐	Java 6 -Xint	1.37	1.37	9.79	**17.60**	22.65	41.94	48.17
☑	Smalltalk VisualWorks	4.31	4.31	11.95	**22.41**	36.03	72.15	179.29
☐	F# Mono	2.43	2.43	4.24	**24.05**	30.51	43.04	43.04
☑	Lua	1.34	1.34	17.46	**27.51**	108.33	154.07	154.07
☑	Python	1.44	1.44	13.92	**37.83**	305.64	446.67	446.67

Ubiquity: (ditto)

Language Rank	Types	Spectrum Ranking
1. C		100.0
2. Java		98.1
3. Python		98.0
4. C++		95.9
5. R		87.9
6. C#		86.7
7. PHP		82.8
8. JavaScript		82.2
9. Ruby		74.5
10. Go		71.9

http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

Robustness and security:

Precision and control:

In the hands of an expert*, C can be amazingly concise, for example a 73-line chess program

Log in to your account on Merced

```
cp -r /tmp/icgep .
cd icgep
more microchess.c
gcc -o microchess microchess.c
microchess
```

```
t p l d k l p t  8
+ + + + + + + +  7
. . . . . . . .  6
. . . . . . . .  5
. . . . . . . .  4
. . . . . . . .  3
* * * * * * * *  2
T P L D K L P T  1
a b c d e f g h
```

e2e4

```
t p l d k l p t
+ + + + + + + +
. . . . . . . .
. . . . . . . .
. . . . * . . .
. . . . . . . .
* * * * . * * *
T P L D K L P T
```

Enter

```
t p l d k l . t
+ + + + + + + +
. . . . . p . .
. . . . . . . .
. . . . * . . .
. . . . . . . .
* * * * . * * *
T P L D K L P T
```

*Program by H.G. Muller (http://home.hccnet.nl/h.g.muller/chess.html)

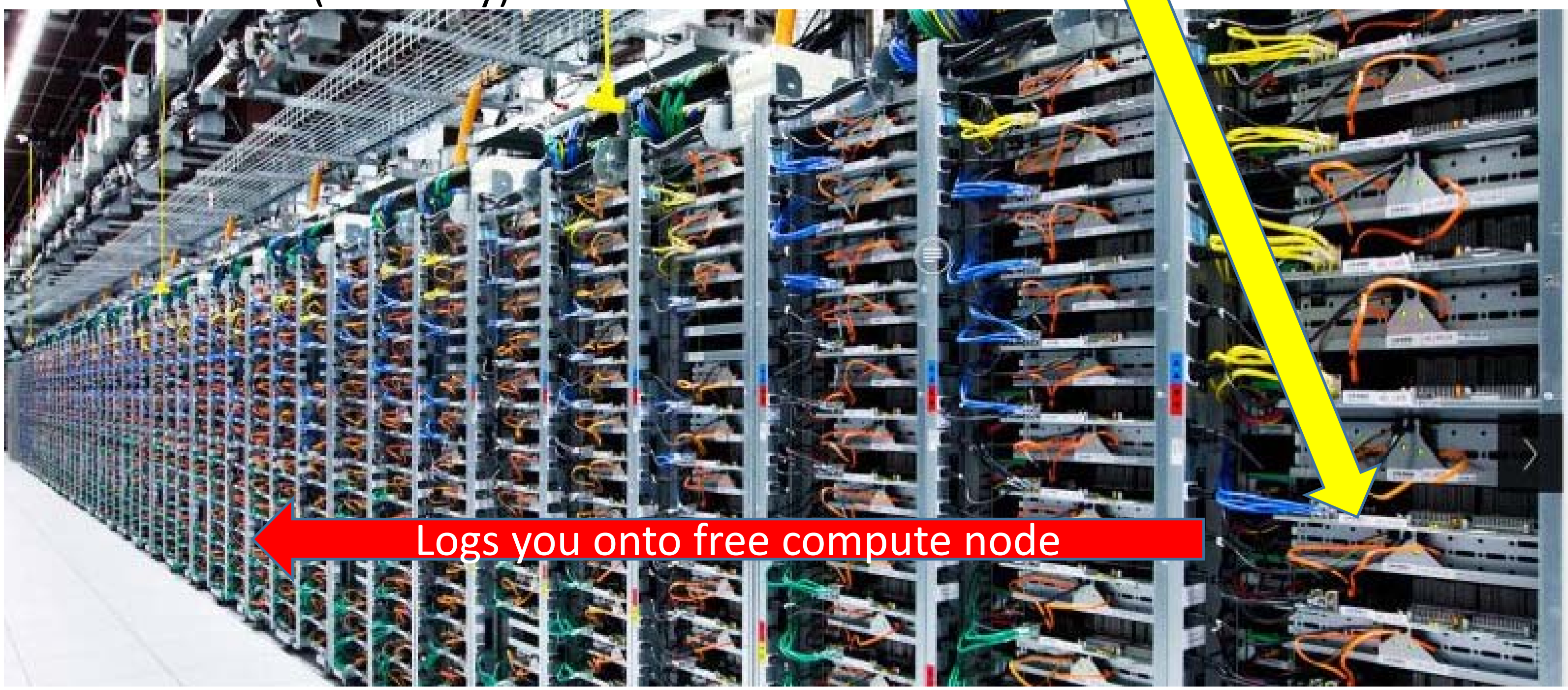
Using the Merced Cluster interactively (not via queue)

On your computer type:

`ssh username@merced.merced.edu`

Logs you onto
the head node

Merced cluster (someday)



On head node type: **`qlogin`**

Your home directory files are still
available on the compute node

Let's compare the speeds of the π Monte Carlo program in different languages

1. R: (see final slide on how to set up R environment on Merced)

```
Rscript slowpi.R 1000000
slowpi.R: Ntrials=1000000 Error=0.001017 Run time in seconds=7.557000
```

```
Rscript fastpi.R 1000000
```

← Uses R vector operations

```
fastpi.R: Ntrials=1000000 Error=-0.001995 Run time in seconds=0.205000
```

2. Python:

```
./slowpi.py 1000000
```

```
./fastpi.py 1000000
```

← Uses numpy array operations

3. C: (Need to compile first)

```
gcc -o pi pi.c
./pi 10000000
```

← 10x more steps than R or Python runs

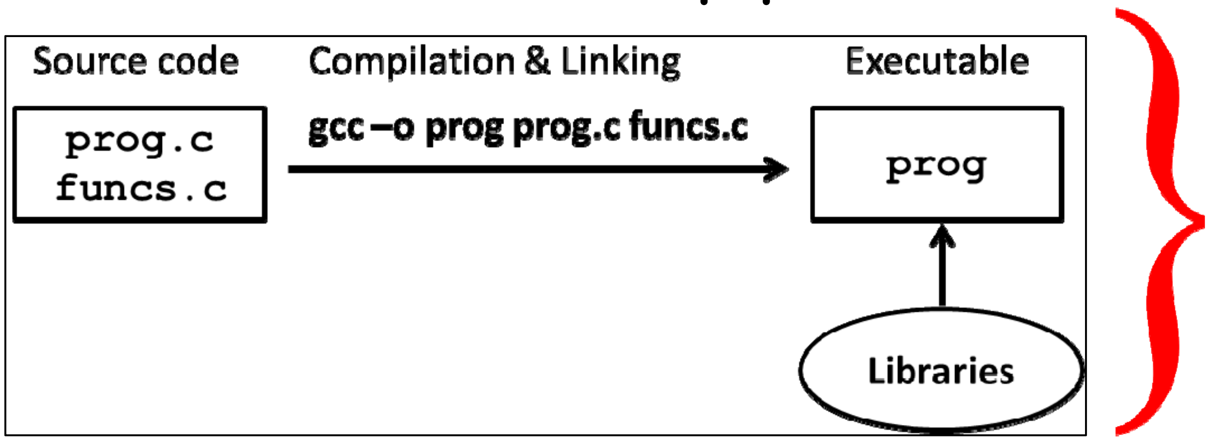
```
gcc -O3 -o pi pi.c    #Note capital "O" in -O3
./pi 10000000
```

The price of this inherent speed is that you do more of the work than in Python or R

- Variables must be “declared” to be a particular type (int, float, etc.) and not change
- There are few standard built-in complex data types or other short cuts (but many libraries exist)
- Language provides few run-time safety checks, like testing array bounds
- You usually need to be aware of how variables are stored in memory and accessed
- You manually allocate and deallocate memory for large data structures

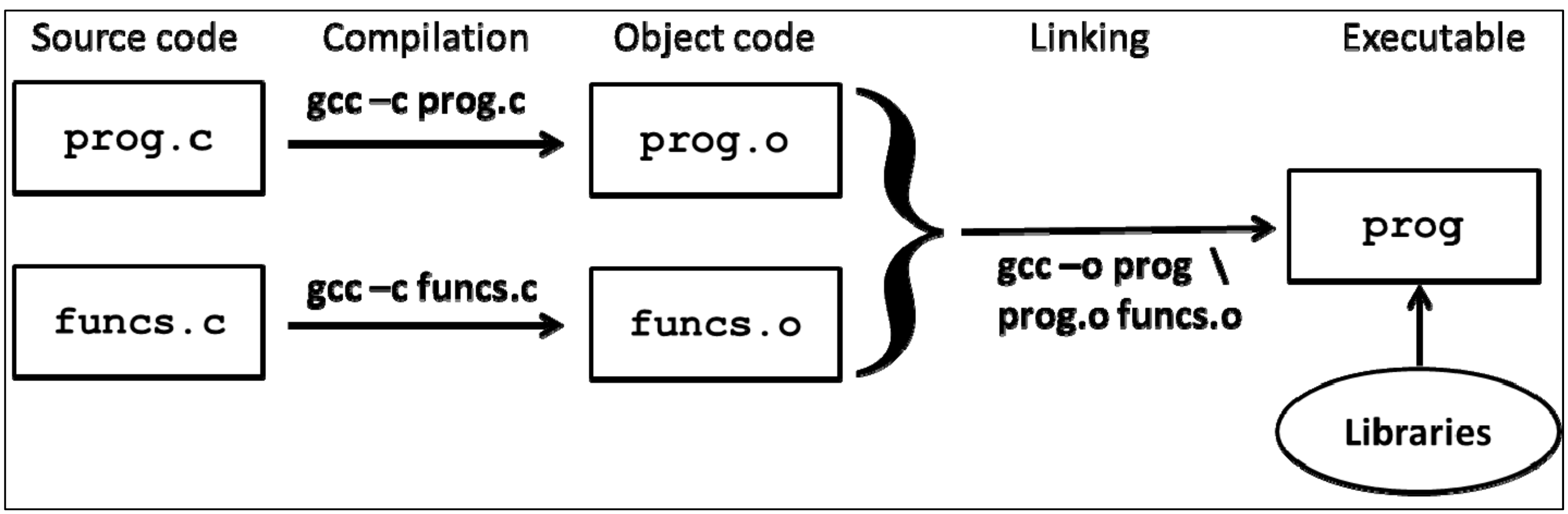
C is a compiled language—it is converted into machine code all at once before it can be run

This can be a one-step process:



Many program errors are found at compile time

Or a two-step process:



C looks different from Python because it uses delimiters (";", "{}", etc.) & not white space

space.c

```
#include <stdio.h>
int main() {
  for (int i=0; i<10; i++){
    printf("%d\n",i);
  }
}
```

```
gcc -o space space.c -std=c99
```

To compile on Merced

Initialization in for-loop not allowed in older C standards

```
./space
```

nospace.c

```
#include <stdio.h>
int main(){for(int i=0;i<10;i++){printf("%d\n",i);}}
```

Many organizations have recommended C/C++ style guides, e.g. Google, GNU, NASA, etc

C has several numerical and string data types, but no high-level built in types (e.g. list or dict)

Description	Name	Max value (IEEE Std)	Accuracy (IEEE Std)	Bytes*
Character	`char`	N/A	N/A	1
Integer	`int`	2147483647	N/A	4
Long integer	`long`	9223372036854775807	N/A	8
Single precision floating point	`float`	3.402823e+38	1.192092e-7	4
Double precision floating point	`double`	1.797693e+308	2.220446e-16	8

*The actual memory usage depends on the computer:

Let's check the type sizes on Merced:

```
gcc -o sizes sizes.c
./sizes
```

The types with a "*" suffix give the number of bytes used to "point" to the value in memory

C distinguishes variable "declaration" from "initialization", unlike Python, R, or Matlab

declare.c

```
#include <stdio.h>
int main(void) {
   int i;
   float a=2.4;
   printf("a=%f\n",a);
   float c;
}
```

Declare i

Declare & initialize a

Declaration after executable statement not allowed in old C standard

Compile:

```
gcc -o declare declare.c
```

Compile asking for picky warnings:

```
gcc -o declare declare.c -pedantic
```

Compile asking for picky warnings with C99 standard:

```
gcc -o declare declare.c -pedantic -std=c99
```

In general you can't assign one data type to another but there are lots of automatic conversions

conversion.c

```
#include <stdio.h>
int main(void) {
  int i=1, j=100;
  float a=2.99;
  char c='w';
  i=a;  /* float to int */
  a=j;  /* int to float */
  c=a;  // float to int (aka char)
  printf("i=%d, a=%f, c=%c\n",i,a,c);
}
```

Compile and run:

```
gcc -o conversion conversion.c
./conversion
```

Things to note:

Is "i" still an int after being set to a float value?
Is "a" still a float and being set to an int value?

The separation of declaring & initializing variables can lead to bugs that don't happen in Python or R

initial.c

```c
#include <stdio.h>
void func(void) {
  long j;
  printf("j=%ld\n",j);
}
int main(void) {
  long i;
  printf("i=%ld\n",i);
  func();
}
```

Compile and run:

```
gcc -o initial initial.c
./initial
```

Ask compiler to check for uninitialized variables

```
gcc -o initial initial.c -Wuninitialized
```

C for-loops are more complicated than in Python or R, but more concise and powerful

Initialize Test Increment

```
for (int i=0; i<10; i++){
  /* Code inside loop */
}
/* Loops vars out of scope here */
```

Example of complex for-loop:

$$a_{n+1} = \begin{cases} \frac{a_n}{2} & (a_n\ even) \\ 3a_n + 1 & (a_n\ odd) \end{cases}$$

Hailstone seq for 5: 5→16 →8 →4 →2 →1

Collatz conjecture: All Hailstone sequences reach 1

hailstone.c

```
#include <stdio.h>
int main(void) {
  for (int i=50; i!=1; i=i%2?3*i+1:i/2) printf("%d\n",i);
}
```

```
gcc -o hailstone hailstone.c -std=c99
```

For arrays of variables, you need to allocate the memory required at compile time or manually

Arrays allocated at compile time use memory from the "stack"

```
int iarray[100];   //Fixed array of size 100
```

The "stack" is usually small compared to total memory:

`ulimit -s` (for some reason that's not the case on Merced)

Arrays allocated at run time use memory from the "heap"

```
//Manually allocate & free memory for int array
int *array, n=100;
array=(int *)malloc(N*sizeof(int));
//Use array for stuff: array[0]=3; etc
free(array); //Skipping this step leads to memory leaks
```

The "heap" is usually very large and can be more easily recycled

```
gcc -o memalloc memalloc.c
./memalloc
```

← Allocates, then frees, bigger and bigger chunks of memory

Managing memory and passing data requires you to learn about how to use memory pointers—see article by Ted Jenson on Catcourses

How to learn more about C-programming…

UCM workshops/classes (covering Bash/R/Python/C):

1. CREST programming workshop June 5-16
2. Chem 260 Fall 2017 (Wednesdays 11:30-3:20)

Postings to CatCourses:

1. The C Book (thecbook.pdf)
2. C reference card
3. A Tutorial On Pointers And Arrays In C

Suggested C-programming websites (among many)

1. publications.gbdirect.co.uk/c_book/
2. gribblelab.org/CBootcamp/
3. www.linuxtopia.org/online_books/programming_books/learning_gnu_c/index.html
4. www.linuxtopia.org/online_books/programming_books/gnu_c_programming_tutorial
5. Many nice example C programs: www.cis.temple.edu/~giorgio/cis71/code/
6. www.cprogramming.com/tutorial/c-tutorial.html
7. www.tutorialspoint.com/cprogramming/index.htm

Loading R into a virtual environment on Merced

Load acaconda software modules:

```
module load anaconda
```

Install R in a vitual environment called "my-R"v

```
conda create -n my-R -c r r-essentials
```

Start virtual environment

```
source activate my-R
```

"`(my-R)`" will prepend prompt

Run R scripts:

```
Rscript slowpi.R 1000000
```

End virtual environment

```
source deactivate
```

To restart after logout or on new node:

```
module load anaconda
source activate my-R
```