Com S // Cpr E // Math 5250

Numerical Analysis of High-Performance Computing

Instructor: Songting Luo

Lecture 7: Summary up until now & Intro to C

Outline

1. Summary up until now
2. Introduction to C

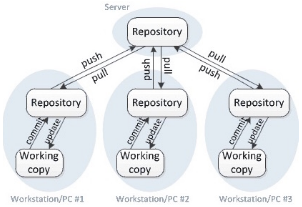
Summary up until now

Unix shell

Basic commands:

- `pwd`, `ls`, `cd`
- `cp`, `mv`, `rm`
- `rm -r`, `mkdir`

Git: distributed version control



https://homes.cs.washington.edu/~mernst/advice/version-control.html

- When you git clone a repository you get all the history (stored in .git)
- Git commands:
  - git clone – clone an existing repository
  - git status – check what files have been modified
  - git stage – add current changes into your next commit
  - git commit – commits to your clone's .git directory
  - git push – sends your recent changesets to another clone
  - git pull – pulls changesets from another clone
- The `.gitignore` file: ignore certain files in commit list

Python: multi-purpose scripting language

- Freely available, open source scripting language
- Basic Python (supports ints, floats, lists, and tuples)
- **Scripts:** can write a `.py` file that contains lists of python commands
- **Functions:** functions that can be called directly in the Python command window or by other scripts/functions
- **Modules:** a collection of related functions
- Python modules often end with a section that looks like:

```
if __name__ == "__main__":
        # some code
```

This code is not executed if the file is imported as a module, only if it is run as a script, e.g., from Unix command line:

```
$ python3 filename.py
```

Python: multi-purpose scripting language

- File output:

```
fid = open('input.data', 'w')
for k in range(0,kmax):
    value = ...
    fid.write("%12.6e" % value)
    fid.write("\n");
fid.close()
```

- File input:

```
A = np.zeros(kmax,dtype=float)
fid = open('input.data', 'r')
for k in range(0,kmax):
    linestring = fid.readline()
    linelist   = linestring.split()
    A[k]       = np.float(linelist[0])
fid.close()
```

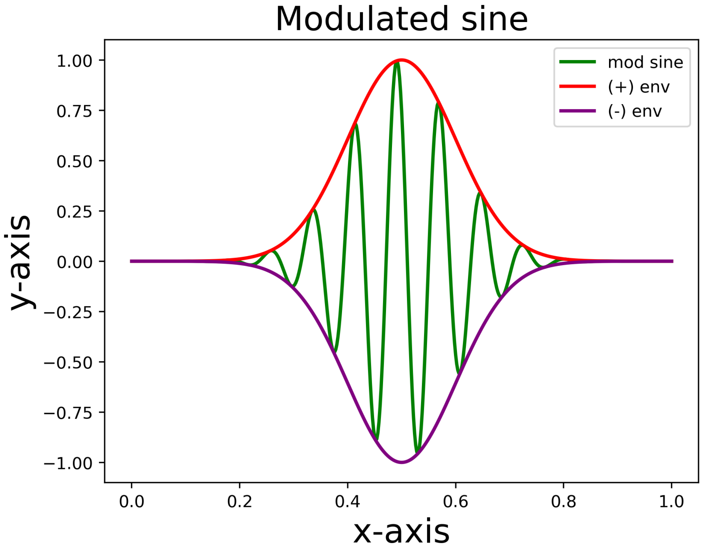
- **NumPy:** module for numerical linear algebra (e.g., `numpy.array`)
- **SciPy:** module for scientific computing (e.g., numerical integration)

Python: multi-purpose scripting language

Matplotlib: module for visualizing data in Python

```
import numpy as np
import matplotlib.pyplot as plt

x  = np.linspace(0,1,501)
y1 = np.exp(-50.0*(x-0.5)**2)*np.sin(80.0*x)
y2 = np.exp(-50.0*(x-0.5)**2)
y3 = -y2

plt.plot(x,y1,linewidth=2,color="green",label="mod sine")
plt.plot(x,y2,linewidth=2,color="red",label="(+) env")
plt.plot(x,y3,linewidth=2,color="purple",label="(-) env")
plt.legend()

plt.xlabel("x-axis",size=20); plt.ylabel("y-axis",size=20);
plt.title("Modulated sine",size=20)

plt.savefig('example2.png',dpi=400,bbox_inches='tight')
plt.show()
```

Python: multi-purpose scripting language



Introduction to C

Compiled vs. interpreted language

Not so much a feature of language syntax as of how language is converted into machine instructions.

Many languages use elements of both.

Interpeter:

- Takes commands one at a time, converts into machine code, and executes.
- Allows interactive programming at shell prompt (e.g., Python or Matlab).
- Cannot take advantage of optimizing over an entire program – does not know what instructions are coming next.
- Must translate each command while running the code, possibly many times over in a loop.
- **Bottom line:** a language that uses only an interpreter, sacrifices some **computational speed** for **coding efficiency** & **interactivity**

Compiled language

The program must be written in 1 or more files $\Longrightarrow$ source code.

Compiled language

The program must be written in 1 or more files $\Longrightarrow$ source code.

These files are input the compiler, which is a computer program that analyzes the source code and converts it into object code.

Compiled language

The program must be written in 1 or more files $\Longrightarrow$ source code.

These files are input the compiler, which is a computer program that analyzes the source code and converts it into object code.

The object code is then passed to a linker or loader that turns one or more objects into an executable.

Compiled language

The program must be written in 1 or more files $\Longrightarrow$ source code.

These files are input the compiler, which is a computer program that analyzes the source code and converts it into object code.

The object code is then passed to a linker or loader that turns one or more objects into an executable.

Why two steps?

Object code contains symbols such as variables that may be defined in other objects. The linker resolves the symbols and converts them into addresses in memory.

Compiled language

The program must be written in 1 or more files $\Longrightarrow$ source code.

These files are input the compiler, which is a computer program that analyzes the source code and converts it into object code.

The object code is then passed to a linker or loader that turns one or more objects into an executable.

Why two steps?

Object code contains symbols such as variables that may be defined in other objects. The linker resolves the symbols and converts them into addresses in memory.

Often large programs consist of many separate files and/or library routines – we do not want to re-compile them all when only one is changed.

Later, we will use Makefiles to handle this.

The C language: Some background

1940s: first electronic digital computers (Atanasoff-Berry, Zuse Z3, ENIAC, Colossus, Harvard Mark 1, . . .)

The C language: Some background

1940s: first electronic digital computers (Atanasoff-Berry, Zuse Z3, ENIAC, Colossus, Harvard Mark 1, . . .)

1940s: programming had to be done in assembly language (low-level programming language; very strong correspondence between language and architecture's machine code instructions; not portable from on platform to another)

The C language: Some background

1940s: first electronic digital computers (Atanasoff-Berry, Zuse Z3, ENIAC, Colossus, Harvard Mark 1, . . .)

1940s: programming had to be done in assembly language (low-level programming language; very strong correspondence between language and architecture's machine code instructions; not portable from on platform to another)

1957: Fortran (Formula Translating System) developed at IBM – a general-purpose, imperative programming language, that is especially suited to numeric computation and scientific computing.

The C language: Some background

1940s: first electronic digital computers (Atanasoff-Berry, Zuse Z3, ENIAC, Colossus, Harvard Mark 1, . . .)

1940s: programming had to be done in assembly language (low-level programming language; very strong correspondence between language and architecture's machine code instructions; not portable from on platform to another)

1957: Fortran (Formula Translating System) developed at IBM – a general-purpose, imperative programming language, that is especially suited to numeric computation and scientific computing.

1972: C (name is based on its predecessor B) developed by Brian Kernighan and Dennis Ritchie at AT&T Bell Labs.

The C language: Some background

1940s: first electronic digital computers (Atanasoff-Berry, Zuse Z3, ENIAC, Colossus, Harvard Mark 1, . . .)

1940s: programming had to be done in assembly language (low-level programming language; very strong correspondence between language and architecture's machine code instructions; not portable from on platform to another)

1957: Fortran (Formula Translating System) developed at IBM – a general-purpose, imperative programming language, that is especially suited to numeric computation and scientific computing.

1972: C (name is based on its predecessor B) developed by Brian Kernighan and Dennis Ritchie at AT&T Bell Labs.

Purpose and design of C:

- Designed to provide low-level access to memory and to provide language constructs that map efficiently to machine instructions (similar to assembly)
- Despite its low-level capabilities, the language was designed to encourage cross-platform programming and much simpler to read and write compared to assembly.

A very simple example

$ISUHPC/lectures/lecture7/codes/HelloWorld.c:

```c
#include <stdio.h>

int main()
{
   // Print a message to the screen
   printf("\n");
   printf("Hello World! My name is Cy.\n");
   printf("\n");

   // Return 0 to signify successful completion
   return 0;
}
```

- Compile by typing (in Unix):

```
$ gcc HelloWorld.c
```

- This will produce the executable: `a.out`
- Run by typing (in Unix):

```
$ ./a.out
```

A very simple example

$ISUHPC/lectures/lecture7/codes/HelloWorld.c:

```c
#include <stdio.h>

int main()
{
   // Print a message to the screen
   printf("\n");
   printf("Hello World! My name is Cy.\n");
   printf("\n");

   // Return 0 to signify successful completion
   return 0;
}
```

- Compile by typing (in Unix):

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

- This will produce the executable: `hello`
- Run by typing (in Unix):

```
$ ./hello
```

Some comments: A very simple example

Every C program must have a main function that is the entry point for the program.

When you run your program the operating system loads it into memory and then runs the code inside the main function.

Some comments: A very simple example

Every C program must have a main function that is the entry point for the program.

When you run your program the operating system loads it into memory and then runs the code inside the main function.

The convention is to make main of type int and then return 0 to indicate that the program was successfully executed.

Some comments: A very simple example

Every C program must have a main function that is the entry point for the program.

When you run your program the operating system loads it into memory and then runs the code inside the main function.

The convention is to make main of type int and then return 0 to indicate that the program was successfully executed.

`#include <stdio.h>` is the C version of `import somemodule`.

Some comments: A very simple example

Every C program must have a main function that is the entry point for the program.

When you run your program the operating system loads it into memory and then runs the code inside the main function.

The convention is to make main of type int and then return 0 to indicate that the program was successfully executed.

`#include <stdio.h>` is the C version of `import somemodule`.

Comments: `// something here` or `/* something here */`

Some comments: A very simple example

Every C program must have a main function that is the entry point for the program.

When you run your program the operating system loads it into memory and then runs the code inside the main function.

The convention is to make main of type int and then return 0 to indicate that the program was successfully executed.

`#include <stdio.h>` is the C version of `import somemodule`.

Comments: `// something here` or `/* something here */`

`"Hello World!  My name is Cy.\n"`, is a string. `"\n"` – newline character.

Some comments: A very simple example

Every C program must have a main function that is the entry point for the program.

When you run your program the operating system loads it into memory and then runs the code inside the main function.

The convention is to make main of type int and then return 0 to indicate that the program was successfully executed.

`#include <stdio.h>` is the C version of `import somemodule`.

Comments: `// something here` or `/* something here */`

`"Hello World!  My name is Cy.\n"`, is a string. `"\n"` – newline character.

`printf` is a function that takes a string as input.

Some comments: A very simple example

Every C program must have a main function that is the entry point for the program.

When you run your program the operating system loads it into memory and then runs the code inside the main function.

The convention is to make main of type int and then return 0 to indicate that the program was successfully executed.

`#include <stdio.h>` is the C version of `import somemodule`.

Comments: `// something here` or `/* something here */`

`"Hello World!  My name is Cy.\n"`, is a string. `"\n"` – newline character.

`printf` is a function that takes a string as input.

Note the semicolon at the end of each statement in the code. Every statement in C ends with a semicolon. In Python, the semicolon is optional.

Some comments: A very simple example

Every C program must have a main function that is the entry point for the program.

When you run your program the operating system loads it into memory and then runs the code inside the main function.

The convention is to make main of type int and then return 0 to indicate that the program was successfully executed.

`#include <stdio.h>` is the C version of `import somemodule`.

Comments: `// something here` or `/* something here */`

`"Hello World!  My name is Cy.\n"`, is a string. `"\n"` – newline character.

`printf` is a function that takes a string as input.

Note the semicolon at the end of each statement in the code. Every statement in C ends with a semicolon. In Python, the semicolon is optional.

Indentation is optional (but **HIGHLY** recommended). In Python: **NOT OPTIONAL**.

Declaring variables

$ISUHPC/lectures/lecture7/codes/example1.c:

```c
#include <stdio.h>

int main()
{
    // Declare some variables
    double x = 3.0;
    double y = 0.1;
    double z;

    // Add x and y and store in z
    z = x+y;

    // Print z to screen
    printf(" z = %010.4f\n",z);

    // Return 0 to signify successful completion
    return 0;
}
```

Declaring variables

Compiling & running this code:

```
$ gcc example1.c -o ex1

$ ./ex1
 z = 3.100000
```

Declaring variables

Compiling & running this code:

```
$ gcc example1.c -o ex1

$ ./ex1
 z = 3.100000
```

Formating:

```
printf(" z = %f\n",z);
 z = 3.100000

printf(" z = %6.4f\n",z);
 z = 3.1000

printf(" z = %10.4e\n",z);
 z = 3.1000e+00

printf(" z = %22.16e\n",z);
 z = 3.1000000000000001e+00
```

Declaring variables

$ISUHPC/lectures/lecture7/codes/types.c:

```c
#include <stdio.h>

int main()
{
   // Standard data types
   char cyclops; // character, format: %c
   int iceman;   // 16-bit int, format: %i
   float storm;  // 32-bit float, format: %f
   double rogue; // 64-bit float, format: %f or %lf

   // Extended data types
   long int beast;       // 32-bit int, format: %li
   long long int havok; // 64-bit int, format: %lli
   long double mimic;    // 128-bit float, format: %Lf

   // Return 0 to signify completion
   return 0;
}
```

Compile-time errors

$ISUHPC/lectures/lecture7/codes/example1a.c:

```c
    // Declare some variables
    double x = 3.0;
    double y = 0.1;
    double z;

    // Add x and y and store in z
    zz = x+y;

    // Print z to screen
    printf(" z = %f\n",z);
```

Generates compile-time error:

```
$ gcc example1a.c
example1a.c:11:4: error: use of undeclared identifier 'zz'
   zz = x+y;
   ^
1 error generated.
```

A more subtle mistake

$ISUHPC/lectures/lecture7/codes/example1b.c:

```c
    // Declare some variables
    double x = 3.0;
    double y = 0.1;
    double z,zz;

    // Add x and y and store in z
    zz = x+y;

    // Print z to screen
    printf(" z = %f\n",z);
```

Generates wrong answer:

```
$ gcc example1b.c
$ ./a.out
 z = 0.000000
```

For loops in C

$ISUHPC/lectures/lecture7/codes/loop1.c:

```c
#include <stdio.h>

int main()
{
   // Declare some variables
   const int n = 8;
   int nfactorial = 1;
   int i;
   // A loop
   for (i=1; i<=n; i++)
   {
      nfactorial = nfactorial*i;
   }

   // Print to screen
   printf(" nfactorial = %i\n",nfactorial);

   return 0;
}
```

For loops in C

Answer:

```
$ gcc loop1.c
$ ./a.out
 nfactorial = 40320
```

For loops in C

Answer:

```
$ gcc loop1.c
$ ./a.out
 nfactorial = 40320
```

Format of for loops:

```
for (int i=start_index; i<=end_index; i++)
{
  // 'i' will be incremented by 1 each time
}

for (int i=start_index; i>=end_index; i--)
{
  // 'i' will be decremented by 1 each time
}

for (int i=start_index; i<=end_index; i+=2)
{
  // 'i' will be incremented by 2 each time
}
```

If-else statement in C

$ISUHPC/lectures/lecture7/codes/ifelse1.c:

```
    // Declare some variables
    double x = -1.43;
    double y = 4.5014;
    double min;

    // Check which one is smaller
    if (x<y)
    {
       min = x;
    }
    else
    {
       min = y;
    }

    // Print to screen
    printf(" min = %f\n",min);
```

If-else statement in C

Answer:

```
$ gcc ifelse1.c
$ ./a.out
 min = -1.430000
```

If-else statement in C

Answer:

```
$ gcc ifelse1.c
$ ./a.out
 min = -1.430000
```

Logical operators:

```
x == a                      // == is the logical equality test

if ( (x==a) || (x==b) )     // || is the logical 'or'

if ( (x>a) && (x<b) )       // && is the logical 'and'

if ( x != 3.8 )             // ! is the logical 'not'

if ( ( (x>a) && (x<b) ) || x==0.0 ) // use parenthesis to clarify
```

Lab assignment

Develop a script ("main program") with name **lab.c**: computing factorial of an integer *n*, exponential of a real number x, and logarithm of a real number y.

submit code and screenshot