Introduction to C in Linux/Unix

COMP 1002/1402

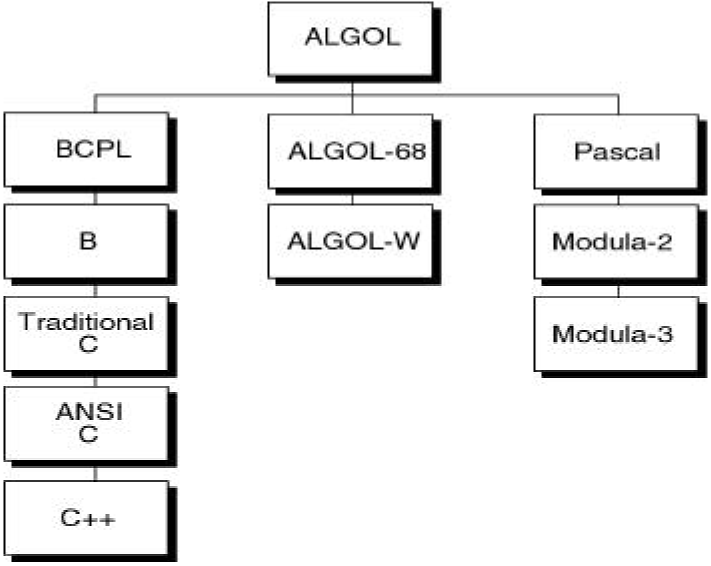
Machine Languages

- Basic instructions sets of chip
- Unique to manufacturer
- Each instruction is a circuit
- Examples
  - Add, subtract, shift bits, load bits into cpu…

High Level Languages

- Pascal, C, C++, Java, Cobol
- Must be translated into Machine Language
- Need translation programs (e.g. compilers)
- Machine Code is executed

History of C



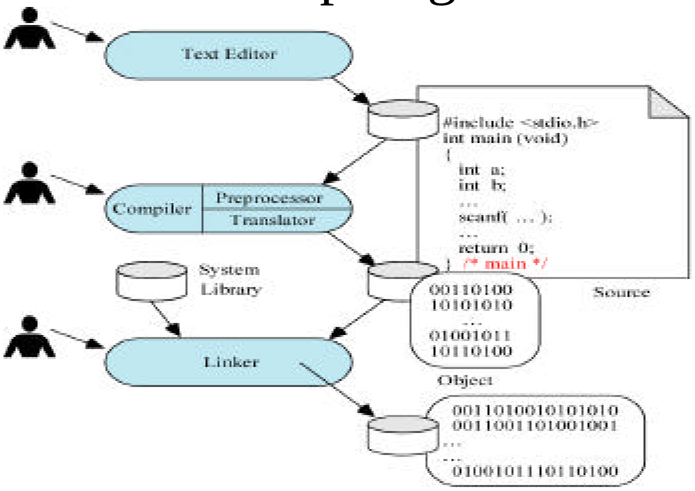
C Language

- Ken Thompson invents B (1967)
- Dennis Ritchie develops C (1970)
- C is high level language but
- Contains low level abilities

Creating a Program

- Create a text file with .c extension
- “compile” the program
- An executable program is the result
- Compiling is a multi-stage process
  (many stages are sometimes automatic)

Compiling



```
#include <stdio.h>
int main (void)
{
  int  a;
  int  b;
  ...
  scanf( ... );
  ...
  return  0;
}  /* main */
```

Preprocessing

- Takes human source code
- Outputs machine readable code
- May rely on specific files (-I includes)

  – Inserts “`#include`”
  – Uses “`#defines`”
  – Processes “`#if`”
  – Eliminates comments

Compiling

- Compiles the preprocessed code into assembler
- This is machine dependent
  - Certain options occur here (-c -o -Wall)
- Output is fed into the Assembler

Assembling

- The assembler converts assembler into object files (.o files)
- .o files are
  - Machine dependent
  - Not executables (unresolved calls)
  - Can act as libraries of code

Linking

- Linking produces executable from object files
- Object files are:
  - The assembler output
  - Necessary libraries
- Normal executable output is: a.out

Stages

- Compiling is either a:
  - Single stage process
  - Two stage process
- Many programs compile in one stage
- Usefulness of .o files:
  - Implies two stage process
  - (Preprocess, compile and assemble) then link later

Compiling in Linux

- `cc` is the default compiler for Sun
- `gcc` is the compiler from Gnu

```
gcc hello.c
```

Results in a.out

Compiling hello.c

```
gcc –o hello hello.c
```

`-o` is output
creates hello executable file

May not compile if there are errors
(spelling, brackets, semi-colons…)

Running hello

After compiling you can run the program:

```
./hello
```

If “.” is in your PATH then you can type:

```
  hello
```

Compiling hello.c

```
gcc –g –o hello hello.c
```

-g creates symbol table in hello executable

Allows debugging programs to use hello!

Remember compiler only does basic checks

Debugging with gdb

- When programs don’t work:
  e.g. produces **`Segmentation fault`**
- Runtime error doesn’t say anything
- Program may not output anything

gdb commands

`gdb <executable>`

`help` – get help

`list` – lists program

`run` – runs program

`break <line>` **--** sets a breakpoint

`backtrace` – shows where program ended

`quit` – end the program

Compiling Multiple Programs

```
example2a.c example2b.c
 example2c.c
```

All three at once:

```
gcc –g –o example2 example2a.c
 example2b.c example2c.c
```

Compiling Multiple Programs

Two stages:

1) to objects first

```
 gcc –g –c example2a.c
 gcc –g –c example2b.c
 gcc –g –c example2c.c
```

2) Link objects

```
 gcc –g –o example2 example2a.o
 example2b.o example2c.o
```

Using **`make`**

Step 1: Create a file called Makefile.

Step 2: Add the example as given

Step 3: When finished save the file and exit.

Step 4: At the command line, type the following command to build the hello executable file:

```
make
```

Rules for make

- Lines with : are dependency lines
  - Left of : is the target
  - Right of : are the dependencies
- After each line use a carriage return
- Tab on second line for command
- Lines are continued with \
- Comments begin #