Introduction to C

Martin Read

Lecture aims

- To introduce C programming

Lecture Objectives

- To examine the funtamentals/syntax of the C programme

- Take a look at Development Tools

- Introduce Reverse engineering

- Practical in C & Development Tools next week

History

- Assembly language considered lowest level programming language
  - human readable version of machine code
- C created to provide a structural programming language
  - considered a low-level programming language with little to no loss in performance - generating efficient code
- Made C the natural choice for building OS & low-level software
  - allowed for easier development at near-assembly performance
- Easy to scale up from assembly programming
  - replaced assembly in 1980s

C programme

- Developed at AT & T’s Bell Laboratories in USA in 1972 & was written by Dennis Ritchie
- Popular because of its reliability, simplicity and user friendly facility

Prior to this:

- COBOL was the language which was used for commerce
- FORTRAN was used for engineering/scientific applications
- BASIC was used as beginner’s language

C

- A functional, structured programming language
  - uses top-down approach & is function-driven
  - easier to understand & implement than object-oriented programming
- Low-level programming language
  - usually compiles to assembly language
  - performs almost as efficiently as assembly code
  - provides base-level access to memory
- Can be ported & coded for any platform
  - low level of abstraction provides breadth of access to underlying machine functionality like direct memory access
- Does not provide error or exception handling
- General focus on applications that work directly with hardware or that need better performance than other languages can offer

C

- Complete binary data transparency
- Consecutive data is placed consecutive in memory

- Data allocated in a function gets allocated on the stack exactly as you declare it (usually in the same order)
- Memory layout of data is completely under your control
- Direct memory access through **pointers**
    - used to manipulate memory

C or C#?

- C is a minimal & fast compiled language, whilst C# is a simple and easy to use interpreted language
- C is a better solution for applications where performance is important
- If your application is a simple web or desktop application
  - use C# (or whatever is your language of choice)
- Many other languages are derived from C
  - or borrow heavily from its syntax
  - E.g. C++, Java, C#, PHP, Objective-C, Perl, Javascript

C or C#?

- If you want an application that works directly with computer hardware or deals with application development
  - C# is not efficient
  - Likely better with C
- C still used in operating systems, kernel-level software, hardware drivers & applications that need to work with older code
- ‘C’ languages are widely used in the electronic devices
  - cellular phones, laptops, microwaves, etc...
- It is majorly used in 3D applications like video games
  - powerful graphical interface needed and .. fast speed

Fundamentals of C

Commenting is similar to C#

# symbol -a “preprocessor directive”

- directs compiler to include header file
- **st**andar**d** **I**nput/**O**utput **h**eader file
    - contains info about function printf()
- printf() used to send output to stdout
- function main() can take arguments, but usually doesn’t, & returns an integer

```
# include <stdio .h>
int main ( void )
{
        printf ("Hello , World !\n");
        return (0);
}
```

C is Case Sensitive

Variables

<u>Must</u> be defined before they can be used

- Variable (or function) name can be up to 52 characters long
- Alphabetic characters: A-Z, a-z
- Numeric characters: 0-9
- Underscore symbol: _

However

- Must start with a letter or underscore
- <u>Cannot be a keyword</u> (e.g. for, if, return)

Declaring Variables as Constants

- Constant variable values once assigned cannot be changed
- Can be declared using the keyword 'const'.

```
const float pi = 3.14
```

Keywords

Auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	size of	volatile
do	if	static	while

"New" reserved words:

restrict	Bool	Complex	Imaginary

Data types

- five built-in data types & permits almost all data type conversion

Integer	int	- 32768 to +32767
Floating point	float	3.4e-38 to 3.4e+e38
Double floating point	double	1.7e+308
Character	char	
Void	void	used for functions

Type Qualifiers

declare the variables along with the data types

- Short
- Long
- Unsigned
- Unsigned long

- Also **arrays** & **vectors**

Strings in C

C does not have a **string** data type

- uses arrays of type char

```
char greeting[20]="Hello. How are you?";
```

System stores each character as an element of the array greeting[]

- Can modify the string one character at a time:

```
greeting[0]='H'; greeting[1]='e'; ...
```

- Easier to use strcpy() – the “string copy” function:

```
strcpy(greeting, "Not too bad");
```

found in the string.h header file

Variables and Size

- Defining a variable allocates space in memory
- We often need to know:
  - how much memory each variable consumes
  - The range of variable
- **These vary with platform**

sizeof function

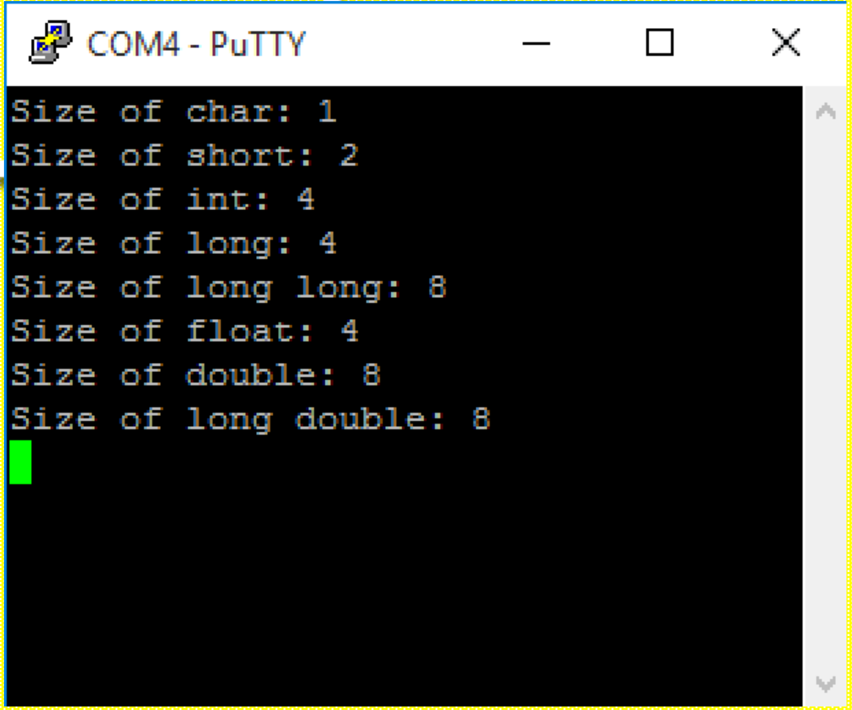
```
int main(void) {

    unsigned int x = 12345;
    unsigned int size = sizeof(x);
    unsigned int Nbits = size*8;
    unsigned int xMax = (1 << (Nbits-1));

    printf("The variable x consumes %u "
           "bytes and is located at address "
           "%X \n", size, (unsigned int)&x);

    printf("The number of bits is %u \n", Nbits);
    printf("The range of x is 0..%u", xMax);
    sleep();
    return 0;
}
```

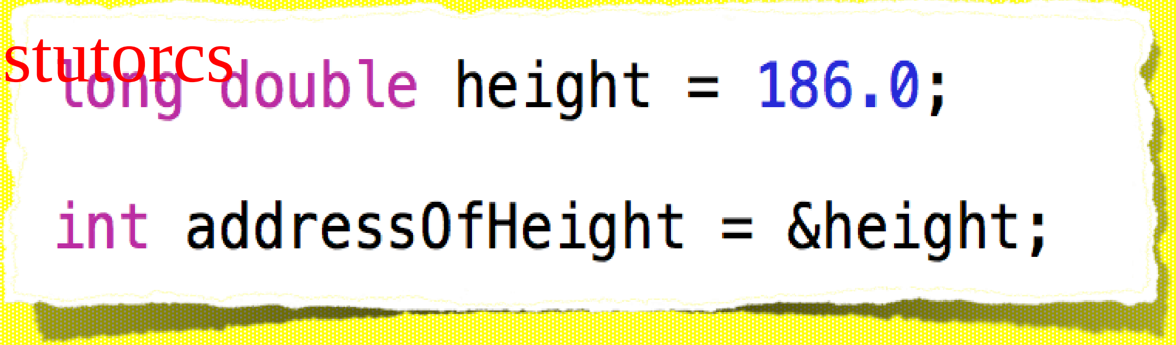
```c
int main()
{
   // insert code here...
   printf("Size of char: %lu\n", sizeof(char) );
   printf("Size of short: %lu\n", sizeof(short) );
   printf("Size of int: %lu\n", sizeof(int) );
   printf("Size of long: %lu\n", sizeof(long) );
   printf("Size of long long: %lu\n", sizeof(long long) );
   printf("Size of float: %lu\n", sizeof(float) );
   printf("Size of double: %lu\n", sizeof(double) );
   printf("Size of long double: %lu\n", sizeof(long double) );
   sleep();
   return 0;
}
```



```
Size of char: 1
Size of short: 2
Size of int: 4
Size of long: 4
Size of long long: 8
Size of float: 4
Size of double: 8
Size of long double: 8
```

Variable address

- For built-in variable types (inc. char, short, int, long, float, double)
  - Referencing the name refers to the content (data)
  - Build tools manage the address in memory
  - You can also **acquire the address with the <u>&</u> prefix**



```
long double height = 186.0;

int addressOfHeight = &height;
```

Formatted output - printf

```
printf(const char*,arg1, arg2,...,argN);
```

Example 1:

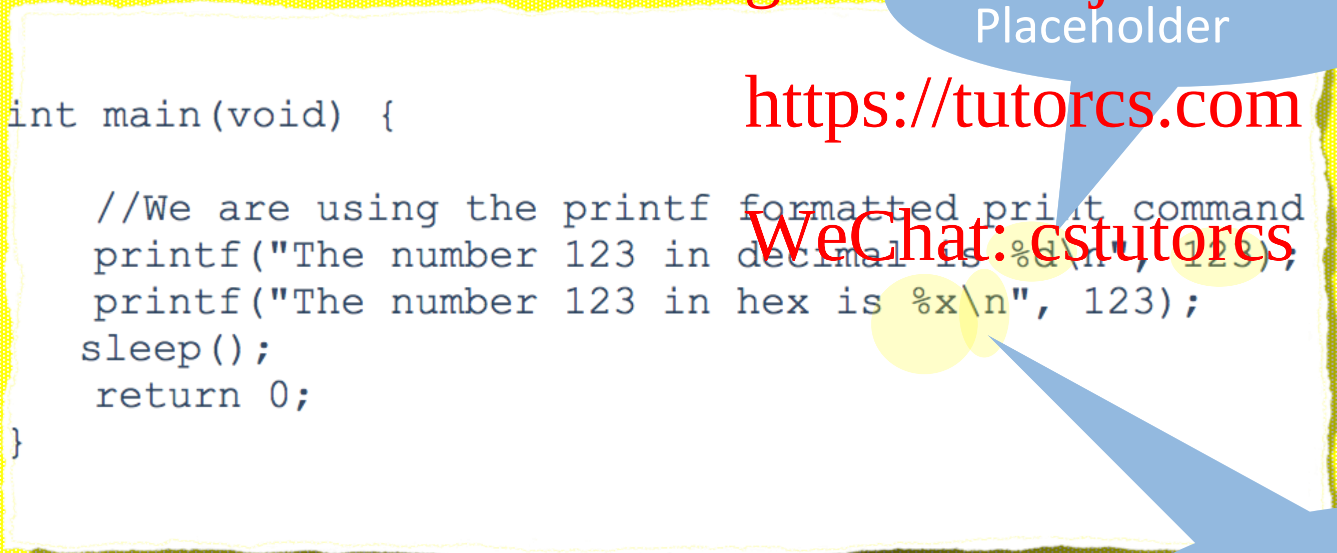
```
printf("The year %d was a long time ago\n", 1966);
```

Example 2:

```
printf("The %s was invented in %d","transistor",1948);
```

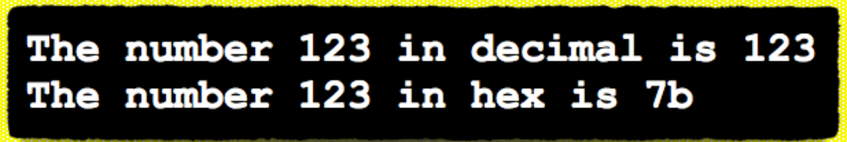
Formatted Output (constant integers)

We can use `printf` to display integers in different bases

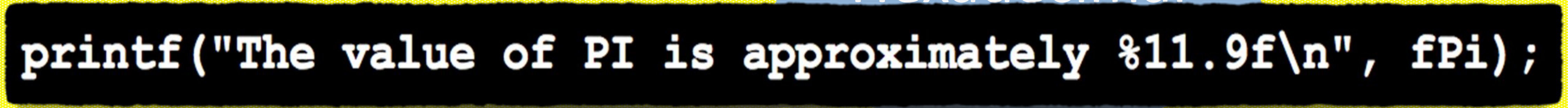


```
int main(void) {

    //We are using the printf formatted print command
    printf("The number 123 in decimal is %d\n", 123);
    printf("The number 123 in hex is %x\n", 123);
   sleep();
    return 0;
}
```



```
The number 123 in decimal is 123
The number 123 in hex is 7b
```



```
printf("The value of PI is approximately %11.9f\n", fPi);
```

Formatted Output Placeholders

WITH PLYMOUTH UNIVERSITY

d,i	int as a signed decimal number.
u	unsigned decimal
f, F	double in normal (fixed-point) notation.
e, E	double value in standard form ([-]d.ddd e[+/-]ddd).
g, G	double in either normal or exponential notation, whichever is more appropriate for its magnitude.
x, X	unsigned int as a hexadecimal number. 'x' uses lower-case letters and 'X' uses upper-case.
o	unsigned int in octal.
s	null-terminated string.
c	char (character).
p	void * (pointer to void) in an implementation-defined format.
n	Print nothing, but write number of characters successfully written so far into an integer pointer parameter.
%	a literal '%' character (this type doesn't accept any flags, width, precision or length).

Arithmetic, Rational & Logical Operators

- **Arithmetic Operators**

  + - * / %

  - No built-in function for powers or truncating division
- **Rational Operators**

  > < ≤ ≥ == !=
- **Assignment Operators**

  ++ -- += -= *= /= %=
- **Logical Operators**

  AND, OR & NOT

  && - Logical AND – True if all conditions are true

  || - Logical OR – True if any one or all conditions are true

  ! - Logical NOT – Negation

Precedence?

Selection and loop

- **Selection Statements:**
  - main ones are if/else, else if & switch
- **Iteration statements:**
  - for, while & do
- **jump statements:**
  - break, continue & goto

Your turn….

Write a programme that :

takes 2 integers,

checks that they are both 5 or more (or displays an error),

multiplies them & then displays the result

(don’t worry about input).

```
#include <stdio.h>


int main() {
        int x, y;
        int sum;
        sum = 0;
      x = 2;
      y = 5;

        if(x >= 5 && y >=5)
                sum = x*y;
        printf("sum: %d\n", sum);
}
```

```
if( ( (i%3) == 0) && ( (i%5)==0) )
        printf("%d divisible by 15\n", i);

if( ( (i%3) == 0) || ( (i%5)==0) )
        printf("%d divisible by 3 or by 5\n", i);
```

Functions

- All executable code resides within a **function**
  - named block of code that performs a task
  - then returns control to caller

- Other languages may distinguish between:

  "function", "subroutine“

  "subprogram", "procedure", or "method“
- In C, these are all functions

```
# include <stdio .h>
# include <stdlib .h>

float average(float c, float d)
{
        return((a+b)/2);
}

int main ()
{
        float a, b;
        printf (" Enter a and b: ");
        scanf ("%f", &a);
        scanf ("%f", &b);
        printf (“Average(a,b) = %f\n", average(a,b));
        return ( );
}
```

Functions from the C Standard Library

- C language doesn't itself contain functions
- Usually linked with the C Standard Library
  - need to add an #include directive at the top of the C file
  - may be one of the following from C89/C90:

<assert.h>, <ctype.h>, <errno.h>, <float.h>

<limits.h>, <locale.h>, <math.h>, <setjmp.h>

<signal.h>, <stdarg.h>, <stddef.h>, <stdio.h>

<stdlib.h>, <string.h>

Embedded code

<mbed.h>

```
# include <stdio .h>
# include <stdlib .h>

int main (){
        int k, m, a [8];
        printf ("\ nThe list is: ");
        for (k=0; k <8; k++) {
                a[k] = rand ()%21;
                printf ("\t%d", a[k ]);
        }
        m = a [0];
        for (k=1; k <8; k++)
                if (m < a[k])
                        m = a[k];

        printf ("\ nWhat is appearing her? %d\n",
m);
        return (0);
}
```

- creates a list of 8 randomly chosen integers between 0 & 20
- finds the largest one & prints to the screen
- Use modulus operator to get one between 0 & 20
- rand produces a number between 0 and 2147483647

Formatted Input - scanf

Example 1:

```
int x;
scanf("%d", &x);
```

Example 2:

```
char myString[32];  //32 character string
scanf("%s",myString);
```

Formatted Input

- Reads input from standard input
- Formats it - using a **conversion character**
- stores it in a specified address

```
int i;
char s;
printf("Enter an integer and a char: ");
scanf("%d %c", &i, &s);

printf("The int is %d, char is %c\n", i, s);
```

getchar()

- Simple function, reads a single character from “stdin”
- “stdin” is by default the terminal keyboard input

```
//Define two integer variables
int c1, c2;

//Read two characters from stdin
//(terminal keyboard)
c1 = getchar();
c2 = getchar();

//Print out the ASCII codes
printf("The ASCII code for the input "
       "data are %d and %d \n", c1, c2);
```

scanf - example

```
int main(void) {
   //Define two integer variables
   int height, weight;
   float h, w, bmi;

   printf("Enter your height in cm: "); //Print out the terminal prompt
   scanf("%d", &height);                //Read input (including newline)

   printf("Enter your weight in kg: "); //Print out the terminal prompt
   scanf("%d", &weight);                //Read input (including newline)

   //Calculate result
   h = (float)height * 0.01f;
   w = (float)weight;
   bmi = w/(h*h);
   printf("Your body mass index is %4.1f\n", bmi);

   return 0;
}
```

Call-by-value & call-by-reference

Call-by-value

- parameters are passed in a function
- can be changed within the function
- remain unchanged in the calling function

- **Call-By-Reference**
- parameter passes the value, stored in a parameter
- modify the contents of the memory space referred to by parameter
- e.g. if i is a variable, then &i is its location in memory

Pointers

- A variable that stores the address of another variable
  - commonly used in C
  - sometimes only way to express a computation
  - pointers are one of the powerful tools of ‘C’ programming
  - very efficient
- Accesses variable indirectly, via the pointer
- Advantages include
  - save memory space
  - process data very fast
  - usually lead to more compact code

Pointers

- **Declaration**

```
        type *<pointer name>
```

Example

```
        int     i , y;
        int     *pointer_to_i;

        i = 3;
        pointer_to_i = &i;
```

- Assigns the address of i to the pointer

```
        y = *pointer_to_i;
```

- Equivalent to y = 3

```
y = *px + 1;
```

Sets y to x+1

```
printf("The contents of x are: %d\n", *px);
```

```
*px = 0;
```

Sets contents of x to 0

Pointers & Functions

```
swap(a, b);          - passes values of a & b

swap(&a, &b);        - passes pointers to a & b
```

Example

```
printf("i=%2d and j=%2d\n", i, j);
printf("Swapping...\n");
Swap_by_Reference(&i, &j);
printf("i=%2d and j=%2d\n", i, j);

void Swap_by_Reference(int *i, int *j)
{
int tmp;
tmp=*i; *i=*j; *j=tmp;
}
```

Pointers & Functions

- Some functions return/compute a single value
- Many important functions return more than one value, or modify one of its own arguments

```
int n, v, array[SIZE];

for (n=0; n < SIZE && getint(&v) ! = EOF; n++)
        array(n) = v;
```

Each call sets v to the next value found in the input file

Pointer example

```
getint (pn)
int     *pn;
{
        int c;
        for (*pn = 0; c >= ‘0’ && c <= ‘9’; c = getch())
                *pn = 10 * *pn + c – ‘0’;
        if (c != EOF)
                ungetch(c);
        return(c);
```

C programme compilation

- C is a **compiled** language, not an interpretive one
- **Compiler** coverts human-readable source code into machine code
- C is a very **small** language
  - relies heavily on external libraries that contain **function**s to achieve many important tasks, including input & output
- But compiler has to be told in advance how these functions should be used
  - So before the compilation process, the **preprocessor** is run to include the function descriptions that the programmer thinks are necessary
- Code is then compiled into **object code**
- Object code is **linked** with library functions to produce executable code

DEVELOPMENT TOOLS

- Assembler – converts assembly code into CPU instruction code
- Linker – provides a link to libraries etc. to make coding easier
- Debugger – for running code safely and to help test it as it runs
- Compiler
- Object code disassembler

ASSEMBLER

- Converts the Assembly language source code to instruction code for the processor
- Instruction code varies for different assemblers (even for the same architecture)
- Assembler Directives instruct the assembler how to construct the instruction code program
    - They are unique to the individual assembler

LINKER

- Can make things easier e.g. printing output to terminal
- Using libraries can save you time
  - if well developed, will be faster
- For GDB, linkers are added when compiling using something like this:

```
$ ld -dynamic-linker /lib/ld-linux.so.2 -o cpuid2_gcc -lc cpuid2_gcc.o
```

LINKER

- Links objects: resolves all defined functions and memory address labels declared in the program code
- For external functions (eg printf) usually a second step is required to link the assembly object code with other external dynamic libraries and allow the executable to run on the host system
- *ld* command

  ld –o test test.o
- Creates executable file *test* from the object file *test.o*

COMPILER

- Converts high level code into assembly language and then into instruction code for the processor to execute.
- GNU Common Compiler (gcc) can use the GNU assembler to assemble and link

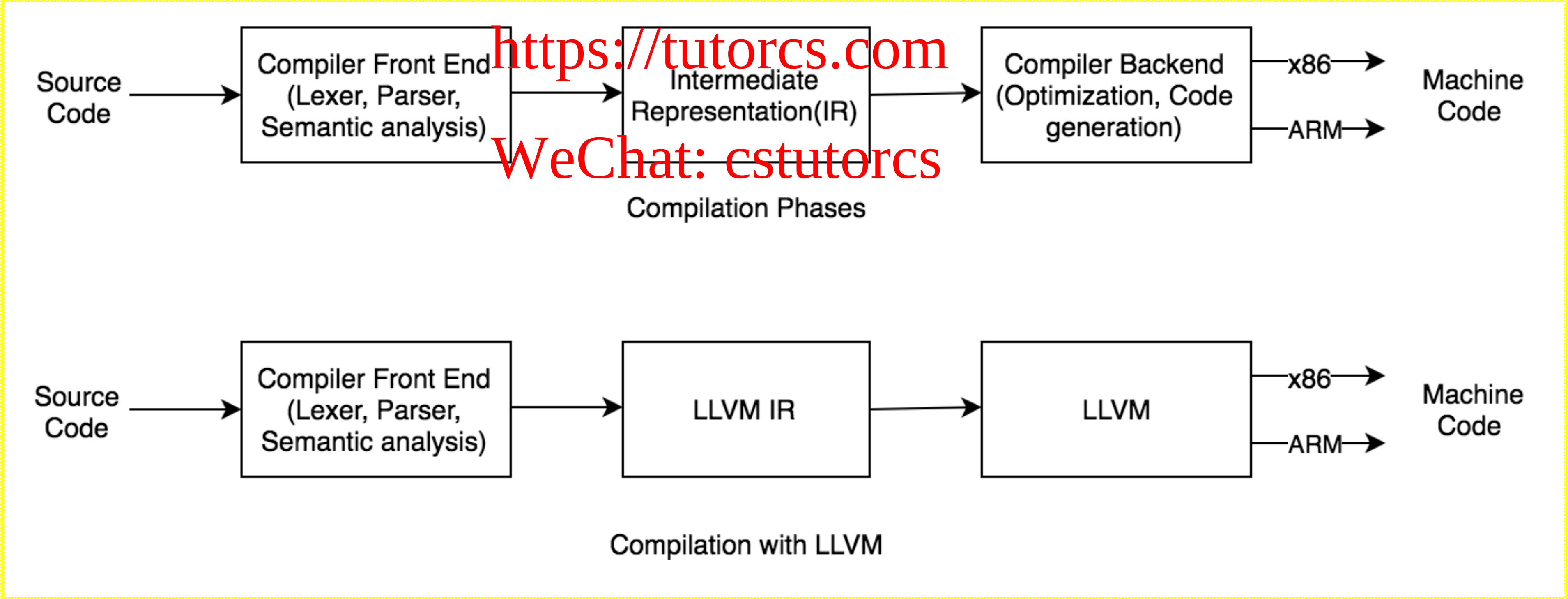
```
gcc –o test test.c
```

- Creates executable file *test* from C language program *test.c*

- Compilers can usually catch typos, etc.
- Complex assembly can get a bit messy when assigning registers & memory locations

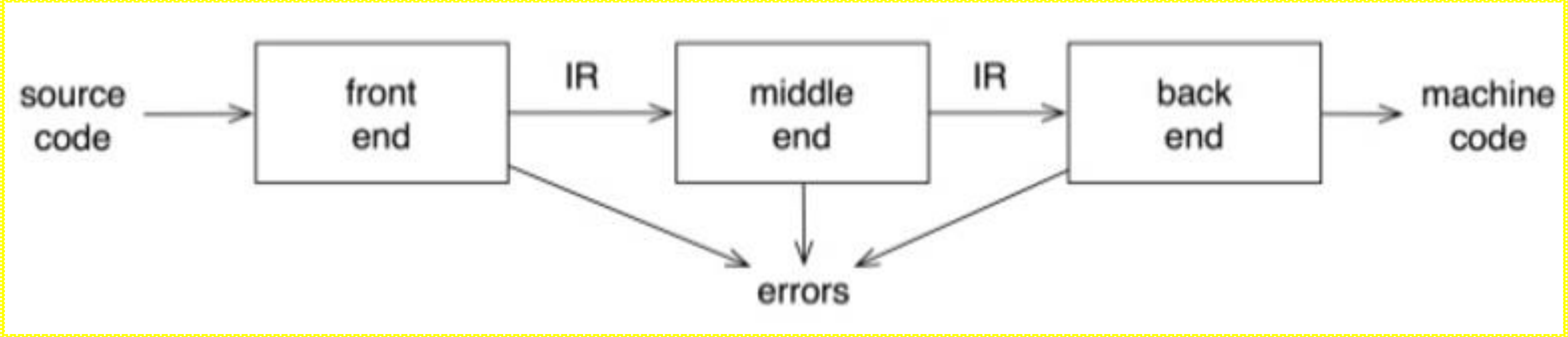
GCC and LLVM

- Two major compilers used today (Linux uses GCC, Apple uses LLVM)
- Both are multi-pass compilers


Source Code
Compiler Front End (Lexer, Parser, Semantic analysis)
Intermediate Representation(IR)
Compiler Backend (Optimization, Code generation)
x86
ARM
Machine Code
Compilation Phases
Source Code
Compiler Front End (Lexer, Parser, Semantic analysis)
LLVM IR
LLVM
x86
ARM
Machine Code
Compilation with LLVM

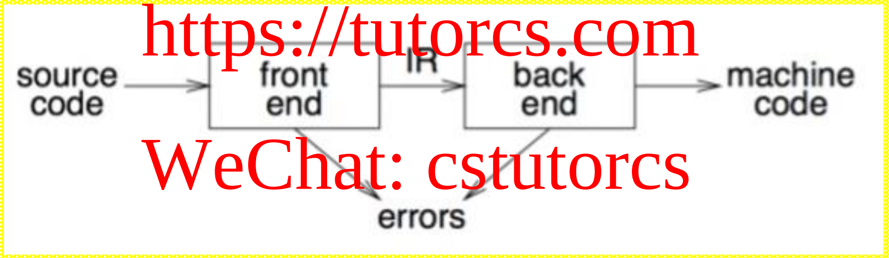
Multipass compiler

1. Essentially an extension of two-pass compilers, multipass compilers have even more middle stages dealing directly with IR.
2. The goal is to reduce execution runtime for the compiled program by spending a little more time compiling/optimizing it.



Two-pass compiler

There is a "front end" and "back end" to the compiler, with the code translated to an intermediate representation (IR) in the middle.

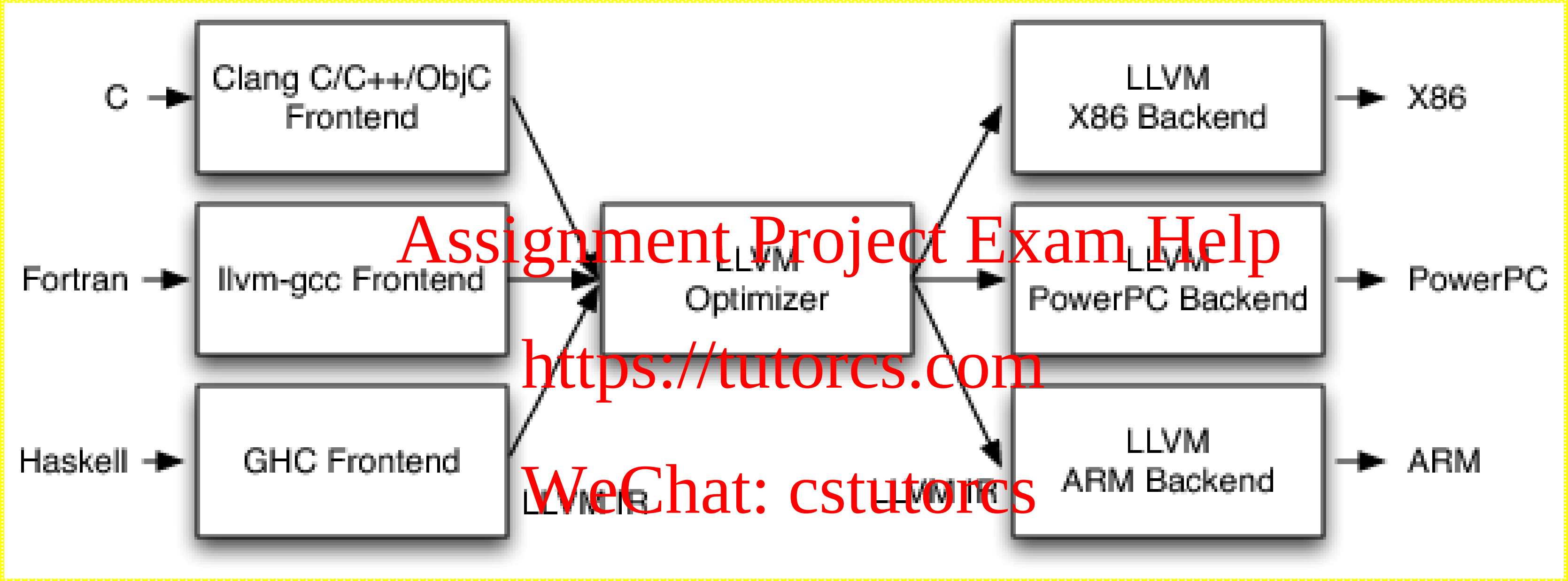


"front end"

1. Code can be broken into pieces (token) to understand context (lexing/scanning)
2. Context-free syntax guides context-sensitive analysis (forward referencing)

- alters the original code (whatever the compiler accepts) into IR

Multi pass example



“back end”

- Can optimize code to create faster/smaller (but semantically the same) code
- Depending on the hardware the executable will be execute on, the backend generates the right file for the native machine language of the system

DEBUGGER

- Runs the program within its own controlled “sandbox”
  - controlled environment, specifying runtime parameters
- Stops the program at any point within the program
- Examines data elements, such as memory locations or registers
- Changes elements in the program while it is running to help bug removal
- GDB debugger

  gcc –gstabs –o test test.c

  gdb test
- Compiles *test.c* using debugging information to create executable file *test*
  - Then opens it in debugging mode with gdb
- Debuggers are handy to step through the program & find the errors

OBJECT CODE DISASSEMBLER

- Takes a full executable program (or an object code file)
  - displays the instruction codes that will be ran by the processor
- Some disassemblers convert the instructions into Assembly language syntax (mnemonics)

gcc –c test.c

objdump –d test.o

- Creates an object file by compiling *test.c*
- Display the disassembled object code file with objdump

Disassemblers – assembly decompilers

Disassemblers are the opposite of assemblers

1. Disassemblers fetch the bits of one instruction (at a time)
2. Then decode the instructions (opcodes, operands)
3. Instead of executing the instruction, it is written to a file
4. The file contains the assembly instructions from the binary

Reverse engineering

Code becomes a binary via a compiler/assembler

How do we reverse the process?

- decompilers/disassemblers, and debuggers

"Reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made … is usually conducted to obtain missing knowledge, ideas, and design philosophy when such information is unavailable. In some cases, the information is owned by someone who isn't willing to share them. In other cases, the information has been lost or destroyed." – Secrets of Reverse Engineering, Eldad Eilam

Reversing – software developing

1. Most code is not stand-alone, it relies on other code. Sometimes reverse engineering gives you answers you need for integration faster than documentation
2. Probably the most popular use of reverse engineering is developing competing software
3. Evaluating the quality and robustness of the software in general

Three levels – hardware, system, code

- You can physically reverse engineer hardware
- Reverse engineering on system level runs several tools to inspect operating system level information
- Code-level reverse engineering is meant for extracting design concepts & algorithms
  - From the analyst point of view, it is probably more complex than system level reversing
  - You need a more detailed understanding of the hardware (CPU) etc.

Sega versus accolade – Interoperability

- **Interoperability:** reverse engineering a system to add more programs that can run on top is reverse engineering (owed) code

Example:

- In 1990 Japanese gaming company Sega Enterprises released their Genesis gaming console but not its programming interfaces
- Accolade, a California-based game developer, reverse engineered the system so they could write and sell games for the Genesis platform
- The courts eventually ruled in Accolade's favour

Reversing engineering – security

- Reverse engineering can your own software can show what others may learn
- If the software is supposed to “hide” information (e.g., encryption), make sure it works the way you think it should
- Preventing hackers from reverse engineering your code to find vulnerabilities
- Reverse engineering malicious software (malware) to prevent attacks
- Preventing stealing copyrighted information (digital rights)

Lecture aims

- To introduce C programming

Lecture Objectives

- To examine the fundamentals/syntax of the C programme

- Take a look at Development Tools

- Introduce Reverse engineering

- Practical in C & Development Tools next week