C Programming
(1) Overview and Basic Types

Acknowledgement: adapted from Karen
Reid's lecture slides for CSC209 winter 2007

History of PLs

- 1 GL
  - machine language (early 1950s)
- 2 GL
  - assembly (late 1950s)
- 3 GL
  - fortran (1954), lisp, cobol, algol
  - pascal, Basic, C (1971), prolog, simula, smalltalk, ada
  - C++ (1983), Java (1995), Delphi, C# (2000), etc.
- 4 GL
  - SQL, CodeFusion, PostScript, SPSS, etc.

http://www.levenez.com/lang/history.html

C Overview

- C is a high-level language — structured
- C is a low-level language — machine access
- C is a small language, extendable with libraries
- C is a permissive language
  - gives programmer more power
  - programmer responsilbe for memory release
  - programmer responsible for error-checking
- Characteristics
  - uninitialized variables have no default value
  - no run-time checking
  - no polymorphism
  - no objects

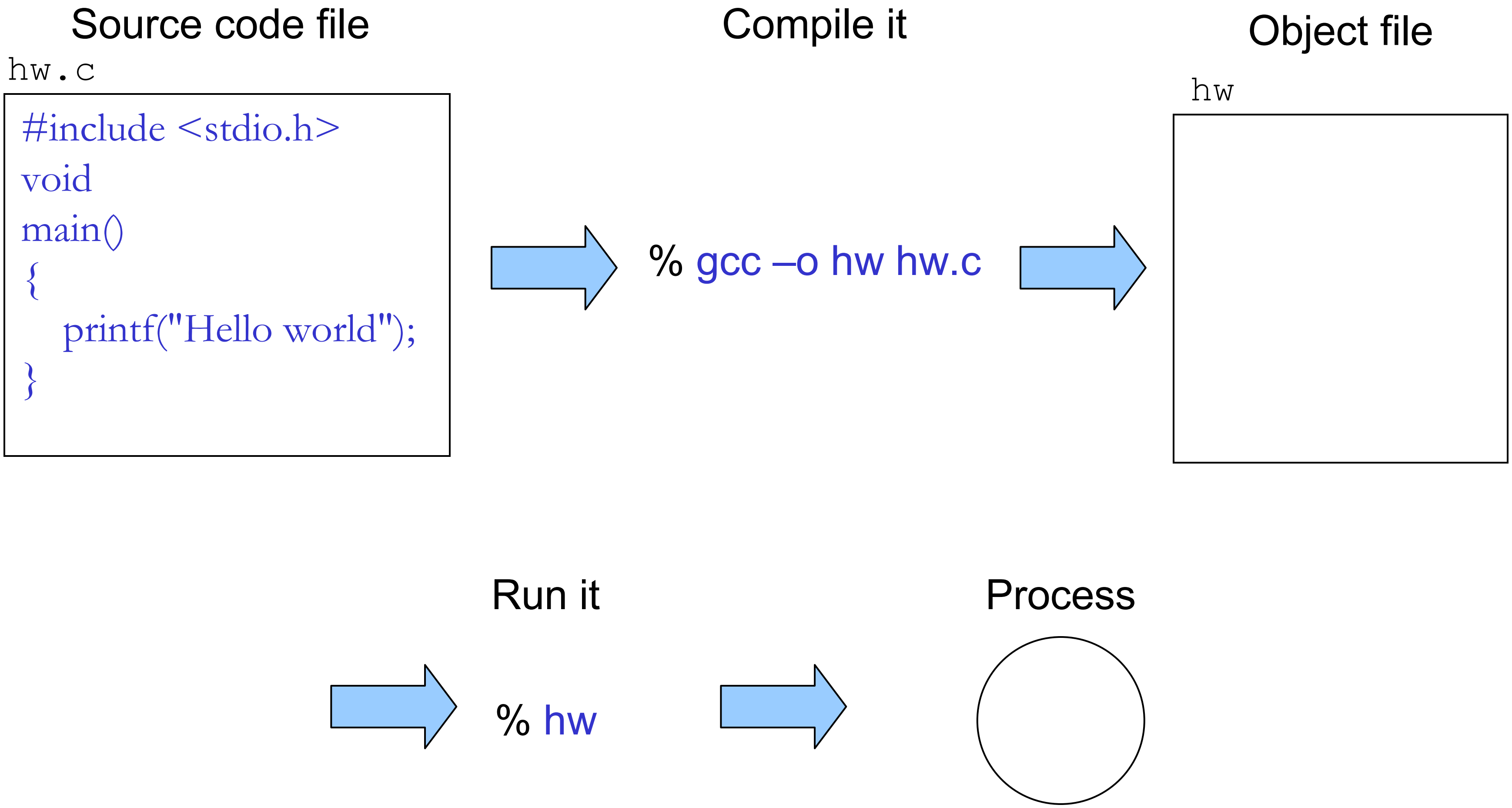
C Overview ...

- Advantages
  - efficient, powerful, flexible, portable
  - standard library
  - integration with Unix
- Disadvantages
  - easy to make errors
  - obfuscation (not easy to understand)
  - little support for modularization (difficult to change)
  - programmer responsilbe for memory release
  - platform dependent

C Overview ...

- Similar to Java
  - Java adopted many syntax, operators and conventions of C/C++
- Functions must be
  - declared: tells compiler how to use function
  - defined: creates the item
- Declarations must appear before code

The Big Picture



Source code file

hw.c

```
#include <stdio.h>
void
main()
{
   printf("Hello world");
}
```

Compile it

```
% gcc –o hw hw.c
```

Object file

hw

Run it

```
% hw
```

Process

Compilation Process

- Preprocessing
  - modify the C source program following directives (commands beginning with #)
- Compiling
  - translates preprocessed C source program into assembly code
- Assembling
  - translates assembly code into machine instructions (object code)
- Linking
  - combines multiple object code to generate executable program
- Execution
  - runs the generated executable program

An Example

hello.c

```
#include <stdio.h>

main() {
    printf(“Hello World!\n”);
}
```

use nm, objdump, readelf to view object code

- gcc -E hello.c > hello1.c
- gcc -S hello1.c
  – generate hello1.s
- gcc -c hello1.s
  – generate hello1.o
- gcc hello1.o
  – generate a.out

- gcc -o hello hello.c
  – generates hello
- Other useful options
  – -ansi
  – -l (lower case of L)
  – -L -I (upper case of i)
  – -g
  – -Wall

Another Example

gcdtest.c

```
#include <stdio.h>

int main() {
        int i, j;
        extern int gcd(int x, int y);
        scanf("%d", &i);
        scanf("%d", &j);
        printf("gcd of %d and %d is %d\n", i, j, gcd(i,j));
        return 0;
}
```

gcd.c

```
int gcd(int x, int y) {
      int t;
      while (y) {
            t = x;
            x = y;
            y = t % y;
      }
      return (x);
}
```

% gcc -c gcd.c

   => generate gcd.o

% gcc -c gcdtest.c

   => generate gcdtest.o

% gcc -o mygcd gcd.o gcdtest.o

   => generate executable mygcd

where are C
header files
stored?

where is C
standard library
located?

General Form of C Programs

```
directives

main()
{
     statements
}

function definitions
```

- Directives
  - file inclusion: #include
  - macro definition: #define, #undef
  - conditional compilation: #if, #ifdef, #ifndef, #elif, #else, #endif
  - others: #error, #line, #pragma
- Functions
  - return-type function-name (parameters)
  - if return-type is omitted, default is int
  - if return-type is void, then no return value
  - the main() function returns int
  - if no return statement, then last statement is returned

Five Forms of *main()*

```
// mymain1.c
// no return type
// no return statement
main()
{
        /* statements */
}
```

```
// mymain2.c
// has return type
// no return statement
int main()
{
        /* statements */
}
```

```
// mymain3.c
// no return type
// has return statement
main()
{
        /* statements */
        return 3;
}
```

```
// mymain4.c
// has return type
// has return statement
int main()
{
        /* statements */
        return 4;
}
```

```
// mymain5.c
// no return type
// no return statement
// the last statement returns
#include <ctype.h>
main()
{
        /* statements */
        toupper('a');
}
```

try compiling
with -ansi

try echo $?
after running
each program

Formatted Output

- printf(format_string, expr1, expr2, ...)
  - printf(“i=%d, j=%.3f, k=%c\n”, i, j, k);
  - %d, %.3f, %c are format conversion specifiers
- Conversion specifier: %[-][m][.p]X
  - the – character is optional, meaning left alignment
  - m is an integer specifying the minimum (total) number of character spaces to print
  - if X=d, then p is an integer specifying the minimum number of digits to explicitly print out
  - if X=f, then p means how many digits to appear after the decimal point

Formatted Output - Examples

```
#include<stdio.h>

main {
    int i=40;
    float x=839.21;
    char c=‘C’;
    printf(“|%d|%5d|%-5d|%5.3d|\n”, i, i, i, i);
    printf(“|%10.1f|%10.4e|%-10f|\n”, x, x, x);
    printf(“|%c|%4c|%-4c|\n”, c, c, c);
    return 0;
}
```

check the output, what does %-10f generate? Try %-10g instead.

Formatted Input

- scanf(format_string, &var1, &var2, ...)
  - scanf(“%d%f%c”, &i, &j, &k);
  - %d, %f, %c are format conversion specifiers
  - the & sign is important
  - putting delimeters between specifiers may not be a good idea
  - putting \n at the end of format string may not be a good idea
- How to detect errors
  - scanf returns the number of items successfully read

Formatted Input - Examples

```
scanf(“%d%d%f%f%c”,&i,&j,&x,&y,&c);
Input:
   1
-12     .3
      4.2e-2
    X
Result:
i=1, j=-12, x=0.3, y=0.042, c=‘X’
```

```
n=scanf(“%d%d”, &i, &j);
Input:12 , 34
Result:n=1,i=12,j=?

n=scanf(“%d,%d”, &i, &j);
Input:12 , 34
Result:n=1,i=12,j=?

n=scanf(“%d ,%d”, &i, &j);
Input:12 , 34
Result:n=2,i=12,j=34

n=scanf(“%d, %d”, &i, &j);
Input:12 , 34
Result:n=1,i=12,j=?

n=scanf(“%d , %d”, &i, &j);
Input:12 , 34
Result:n=2,i=12,j=34
```

Basic Types

- Integer types
  - short, int, long
  - signed vs. unsigned
- Floating types
  - float, double, long double
  - signed
- Character type
  - char
  - signed vs. unsigned

check
limits.h

Integer Types

how are these
values obtained?

Type	Short Form	Size	Min	Max
short int	short	16 bits	-32768 (-2^{15})	32767 ($2^{15}-1$)
unsigned short int	unsigned short	16 bits	0	65535 ($2^{16}-1$)
int	int	32 bits	-2147483648 (-2^{31})	2147483647 ($2^{31}-1$)
unsigned int	unsigned int	32 bits	0	4294967295 ($2^{32}-1$)
long int	long	32 bits	-2147483648 (-2^{31})	2147483647 ($2^{31}-1$)
unsigned long int	unsigned long	32 bits	0	4294967295 ($2^{32}-1$)

Two’s Complement

Number (int)	left-most byte (1st bit is sign bit)		2nd byte	3rd byte	right-most byte
0	**0**	0000000	00000000	00000000	00000000
1	**0**	0000000	00000000	00000000	00000001
2^{31}-1	**0**	1111111	1111111	1111111	1111111
-2^{31}	**1**	0000000	0000000	0000000	0000000
-2^{31}+1	**1**	0000000	0000000	0000000	0000001
-1	**1**	1111111	1111111	1111111	1111111

Integer Constants (Literals)

- An integer constant by default is type decimal
- An octal constant begins with a zero
  - 09, 027, 063
- A hexadecimal constant begins with 0x
  - 0x32, 0x6f, 0Xac
- An integer constant is typed int if it fits in the range of int; otherwise typed long
  - force a long integer constant by adding L or l
    - 15L, 063l, 0x6fL
- An integer constant is signed by default
  - force an unsigned integer constant by adding U or u
    - 15U, 063Lu, 0x6flU

Floating Types

Type	Size	Size of exponent	Size of fraction	Rough range
float	32 bits	8 bits	23 bits	10^{-44} to 10^{38}
double	64 bits	11 bits	52 bits	-10^{-323} to 10^{308}
long double	96 bits	NA	NA	NA

Floating Constants (Literals)

- An floating constant must contain a decimal point or an exponent
  - 62., 62.0, 62.0e0, 62E0
  - 6.2e1, 6.2e+1, .062e3, 620.e-1, 6200E-2
- An floating constant is typed double by default
  - force a float constant by adding F or f
    - 57.3F, .648f
  - force a long double constant by adding L or l
    - 15.30L, .64E-3l
- A floating constant is always signed

Character Type

- char ch = ‘B’; char flag = ‘0’; char dollar = ‘$’;
- ASCII character set
  - 7-bit code representing 128 characters
  - 8-bit code representing 256 characters
- Unicode character set
  - 16-bit code representing 65536 characters
- Characters are integers in C
  - ‘a’ to ‘z’ map to intergers 97 to 122
  - ‘A’ to ‘Z’ map to integers 65 to 90
  - ‘0’ to ‘9’ map to integers 48 to 57
- Don’t assume char is signed or unsigned by default
  - use signed char or unsigned char to explictly specify

Character Type

- Escape sequences

– backspace	\b
– new line	\n
– carriage return	\r
– horizontal tab	\t
– backslash	\\
– single quote	\’
– double quote	\”

- Utility functions
  - toupper(‘a’); tolower(‘A’); (include<ctype.h>)
  - getchar(); putchar(‘a’);

```
printf(“Enter an integer:”);
scanf(“%d”, &i);
printf(“Enter a command:”);
cmd=getchar();
```

what are stored in i and cmd?

The sizeof Operator

- sizeof(type-name)
  - returns number of bytes to store a type
  - often used in dynamic memory allocation
  - output is machine-dependent

```
printf("size of char: %lu\n", (unsigned long)sizeof(char));
printf("size of short: %lu\n", (unsigned long)sizeof(short));
printf("size of int: %lu\n", (unsigned long)sizeof(int));
printf("size of long: %lu\n", (unsigned long)sizeof(long));
printf("size of long long: %lu\n", (unsigned long)sizeof(long long));
printf("size of float: %lu\n", (unsigned long)sizeof(float));
printf("size of double: %lu\n", (unsigned long)sizeof(double));
printf("size of long double: %lu\n", (unsigned long)sizeof(long double));
```

Type Conversions

- Implicit conversions
  - arithmetic conversions
  - assignment conversions
  - function calls
  - function returns
- Explicit conversions – cast

```
int i; float f=13.54;
i = (int)f;
```

Arithmetic Conversions

widest

- long double
- double
- float
- long int
- unsigned int
- int
- short, char

narrowest

```
char c;
short int s;
int i;
unsigned int u;
long int l;
unsigned long int ul;
float f;
double d;
long double ld;
s = s + c; /* c and s converted to int */
i = i + c; /* c converted to int */
i = i + s; /* s converted to int */
u = u + i; /* i converted to unsigned int */
l = l + u; /* u converted to long int */
ul = ul + l; /* l converted to unsigned long */
f = f + ul; /* ul converted to float */
d = d + f; /* f converted to double */
ld = ld + d; /* d converted to long double */
```

Assignment Conversions

- In an assignment, the expression on the right side is converted to the type of the variable on the left.

```
char c;
int i = c;    /* c is converted to int */
double d = i; /* i is converted to double */
```

- This is no problem as long as the variable’s type is at least as “wide” as the expression; otherwise, precision may be lost, and compiler may warn.

```
int i = 1121;
float f = 313.252;
char c = i;
i = f;
printf("c = %c, i = %d\n”, c, i);
```

Data Type Capacity

- What happens when this code is executed?

```
char c = 127;
short s = 32767;

printf(“c = %d\n”, c);
printf(“s = %hd\n”, s);
c++;
s++;

printf(“c = %d\n”, c);
printf(“s = %hd\n”, s);
```

Mixed Mode Arithmetic

```
double m = 5/6; /* int / int = int */
printf("Result of 5/6 is %f\n", m);
```

```
Result of 5/6 is 0.000000
```

```
double n = (double)5/6; /* double / int = double */
printf("Result of (double)5/6 is %f\n", n);
```

```
Result of (double)5/6 is 0.833333
```

```
double o = 5.0/6; /* double / int = double */
printf("Result of 5.0/6 is %f\n", o);
```

```
Result of 5.0/6 is 0.833333
```

```
int p = 5.0/6; /* double / int = double but then
                  converted to int */
printf("Result of 5.0/6 is %d\n", p);
```

```
Result of 5.0/6 is 0
```