The C Programming Language

(with material from Dr. Bin Ren, William & Mary Computer Science)

Overview

- Motivation
- Hello, world!
- Basic Data Types
- Variables
- Arithmetic Operators
- Relational Operators
- Assignments
- Boolean Operators

Why C?

- **Age has its advantages**
  - C has been around for ~40 years
- **Easy to understand**
  - C is a great language for expressing common ideas in programming in a way that most people are comfortable with (procedural language)
- **Reasonably close to the machine**
  - Low-level access to memory
  - Language constructs that map efficiently to machine instructions
  - Minimal run-time support
- **Best combination of speed, low memory use, low-level access to the hardware, and popularity**

Transitioning to C from Python

- lower level – more for you to program
- sometimes unsafe
- standard library is smaller
- different syntax
- structured vs. script
- paradigm shift: not object-oriented

- like going from automatic transmission to stick shift

Programming in C

- C is procedural, not object-oriented
- C is fully compiled (to machine code)
- C allows direct manipulation of memory via pointers
- C does not have garbage collection
- C has many important, yet subtle, details

Source: xkcd.com/571

Hello, world!

```
#include <stdio.h>
void main(void)
{
    printf(“Hello, world!\n”);
}
```

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

```
#include <stdio.h>
void main(void)  {
    printf(“Hello,  “);
    printf(“world!”);
    printf(“\n”);  }
```

Which one is best?

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

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

- Reminder: many different ways to solve the same problem
- Experiment with leaving out parts of the program, to see what error messages you get

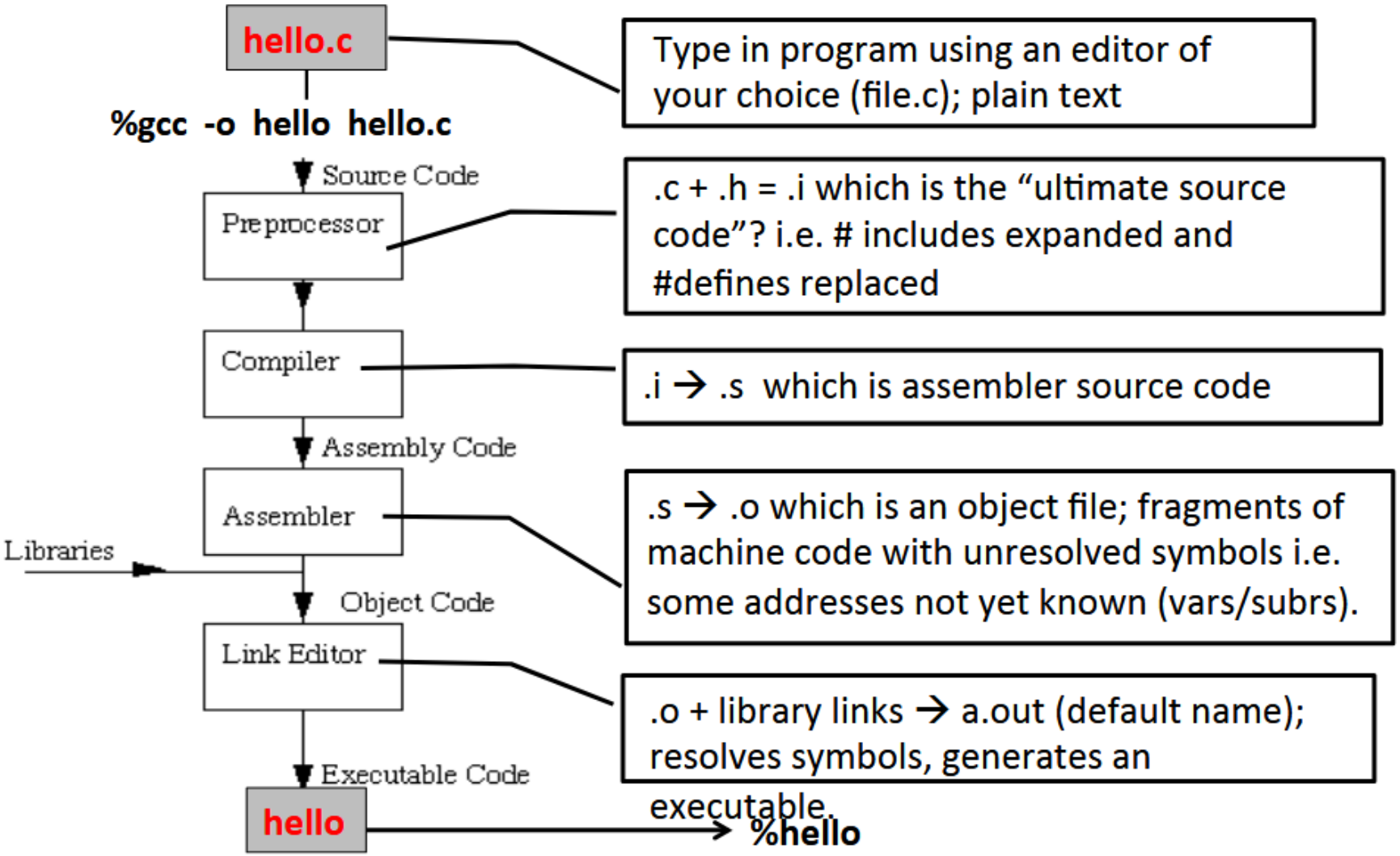
Hello world!

- **#include <stdio.h>**
  - tells the compiler to include this header file for compilation
  - to access the standard functions that come with your compiler, you need to include a header with the #include directive.
- **main()**
  - main function, where execution begins
- **{ }**
  - curly braces are equivalent to stating "block begin" and "block end“
  - the code in between is called a “block”
- **printf()**
  - the actual print statement
- **return 0**
  - returns a value

Header Files

- **Functions, types and macros of the standard library are declared in standard headers**
- **A header can be accessed by**
  - #include <header>
  - Note that this does not end with a semicolon
- **Headers can be included in any order and any number of times**
- **Must be included outside of any external declaration or definition; and before any use of anything it declares**
- **should not include C source files**

C Compilation



Coding Style

- **always explicitly declare the return type on the function**
  - defaults to a type integer
- **replace return 0 with return EXIT_SUCCESS (in <stdlib.h>)**
- **comments**
  - /* comment */
  - comments cannot be nested
  - // is a single line comment from // to the end of the line
- **blanks, tabs, and newlines (or "white space"), as well as comments, are ignored except to separate tokens**
  - free-form spacing

Hello, world! (v2)

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

/* Main Function
 *   Purpose: Controls program, prints Hello, World!
 *   Input: None
 *   Output: Returns Exit Status
 */
int main(int argc, char **argv) {
    printf("Hello, world!\n");
    return EXIT_SUCCESS;
}
```

- works exactly the same as previous versions
- easier to understand

C Basic Data Types

- **Integer Types**
  - char – smallest addressable unit; each byte has its own address
  - short – short int; not used as much
  - int – default type for an integer constant value
  - long – do you really need it?
- **Floating point Types**
  - inexact
  - float – single precision (about 6 digits of precision)
  - double – double precision (about 15 digits of precision)
    - constant default unless suffixed with ‘f’

C Basic Data Types

```
char           1 bytes -128 to 127
unsigned char  1 bytes 0 to 255
short          2 bytes -32768 to 32767
unsigned short 2 bytes 0 to 65535
int            4 bytes -2147483648 to 2147483647
unsigned int   4 bytes 0 to 4294967295
long           4 bytes -2147483648 to 2147483647
unsigned long  4 bytes 0 to 4294967295
float          4 bytes 1.175494e-38 to 3.402823e+38
double         8 bytes 2.225074e-308 to 1.797693e+308
```

- **char guaranteed to be one byte**
- **no maximum size for a type, but the following relationships must hold:**
  - sizeof (short) <= sizeof (int) <= sizeof (long)
  - sizeof (float) <= sizeof (double) <= sizeof (long double)

C Basic Data Types

C Language Variable Types

Whether you're working with regular or unsigned variables in your C program, you need to know a bit about those various variables. The following table show C variable types, their value ranges, and a few helpful comments:

Type	*Value Range*	*Comments*
char	–128 to 127	
unsigned char	0 to 255	
int	–32,768 to 32,767	16-bit
	–2,147,483,648 to 2,147,483,647	32-bit
unsigned int	0 to 65,535	16-bit
	0 to 4,294,967,295	32-bit
short int	–32,768 to 32,767	
unsigned short int	0 to 65,535	
long int	–2,147,483,648 to 2,147,483,647	
unsigned long int	0 to 4,294,967,295	
float	1.17×10^{-38} to 3.40×10^{38}	6-digit precision
double	2.22×10^{-308} to 1.79×10^{308}	15-digit precision

Variable Declarations

- **purpose: define a variable before it is used**
- **format: type identifier [, identifier] ;**
- **initial value: can be assigned**
  - int i, j, k;
  - char a, b, c = 'D';
  - int i = 123;
  - float f = 3.1415926535;
  - double f = 3.1415926535;
- **type conversion: aka type casting (coercion: use with caution)**
  - (type) identifier;
    - int i = 65; /* what if 258 */
    - char a; /* range -128 to 127 */
    - a = (char) i; /* what is the value of a? */

Identifier Naming Convention

- **similar to Python**
- **rules for identifiers**
  - a-z, A-Z, 0-9, and _
  - case sensitive
  - first character must be a letter or _
  - keywords are reserved words, and may not be used as identifiers
- **identifier naming style**
  - separate words with '_' or capitalize the first character
  - use all UPPERCASE for symbolic constant, macro definitions, etc.
  - be consistent
  - use mnemonic names
- **sample identifiers**
  - i0, j1, abc, stu_score, __st__, data_t, MAXOF, MINOF ...

C 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	sizeof	volatile
do	if	static	while

- **reserves a word or identifier to have a particular meaning**
  - meaning of keywords — and, indeed, the meaning of the notion of keyword differs widely from language to language.
  - do not use them for any other purpose in a C program
  - allowed, of course, within double quotation marks

C Operators

C Language Operators

In programming with C, you occasionally want to use common mathematical operators for common mathematical functions and not-so-common operators for logic and sequence functions. Here's a look at C language operators to use:

Operator, Category, Duty	Operator, Category, Duty	Operator, Category, Duty
=, Assignment, Equals	!=, Comparison, Is not equal to	>, Bitwise, Shift bits right
+, Mathematical, Addition	&&, Logical, AND	~, Bitwise, One's complement
–, Mathematical, Subtraction	||, Logical, OR	+, Unary, Positive
*, Mathematical, Multiplication	!, Logical, NOT	–, Unary, Negative
/, Mathematical, Division	++, Mathematical, Increment by 1	*, Unary, Pointer
%, Mathematical, Modulo	--, Mathematical, Decrement by 1	&, Unary, Address
>, Comparison, Greater than	&, Bitwise, AND	sizeof, Unary, Returns the size of an object
>=, Comparison, Greater than or equal to	|, Bitwise, Inclusive OR	., Structure, Element access
<, Comparison, Less than	^, Bitwise, Exclusive OR (XOR or EOR)	->, Structure, Pointer element access
<=, Comparison, Less than or equal to	<<, Bitwise, Shift bits left	?:, Conditional , Funky if operator expression
==, Comparison, Is equal to		

Arithmetic Type Issues

- **type combination and promotion**
  - (‘a’ – 32) = 97 – 32 = 65 = ‘A’
  - smaller type (char) is “promoted” to be the same size as the larger type (int)
  - determined at compile time – based purely on the types of the values in the expressions
  - does not lose information – convert from type to compatible large type

Arithmetic Operators

- **mathematical symbols**
  - + - * / %
  - addition, subtraction, multiplication, division, modulus
- **works for both int and float**
  - + - * /
    - / operator performs integer division if both operands are integer, i.e., truncates; otherwise, float
- **% operator divides two integer operands with an integer result of the remainder**
- **precedence – left to right**
  - () always first
  - * / %
  - + -

Example

```
#include <stdio.h>

int main()
{
  int first, second, add;
  float divide;

  printf("Enter two integers\n");
  scanf("%d %d", &first, &second);

  add = first + second;
  divide = first / (float)second;

  printf("Sum = %d\n",add);
  printf("Division = %.2f\n",divide);

  return 0;
}
```

Relational Operators

- used to compare two values
  - < <= > >=
  - == !=
- precedence order given above; then left to right
- arithmetic operators have higher precedence than relational operators
- a true statement is one that evaluates to a nonzero number
- a false statement evaluates to zero
- when you perform a comparison with the relational operators, the operator will return 1 if the comparison is true, or 0 if the comparison is false
  - 0 == 2 evaluates to 0
  - 2 == 2 evaluates to a 1

Example

```
#include <stdio.h>
 /* print Fahrenheit-Celsius table for fahr = 0, 20, ..., 300
    where the conversion factor is C  =  (5/9) x (F-32)  */
main()
{
    int fahr, celsius;
    int lower, upper, step;
    lower = 0;                  /* lower limit of temperature scale */
    upper = 300;                /* upper limit */
    step = 20;                  /* step size */
    fahr = lower;
    while (fahr <= upper)     {
            celsius = 5 * (fahr-32) / 9;     // problem?  9.0? Typecast?
            printf("%d\t%d\n", fahr, celsius);
            fahr = fahr + step;  }
   return 0;
}
```

Example

```
#include <stdio.h>
#define MAGIC 10
int main(void)
{
    int i, fact, quotient;
    while (i++ < 3)          // value of i? need to initialize
    {
        printf("Guess a factor of MAGIC larger than 1: ");
        scanf("%d", &fact);
        quotient = MAGIC % fact;
        if (0 == quotient)
            printf("You got it!\n");
        else
            printf("Sorry, You missed it!\n");
    }
    return 0;
}
```

i++ is the same as:

i = i + 1

How evaluate?

i = i + 1 < 3

3 1 2

Problem, but...

(i = i + 1) < 3

Assignments

- **in C, assignments are expressions, not statements**
- **embedded assignments -- legal anywhere an expression is legal**
  - allows multiple assignment a = b = c = 1;
- **assignment operators**
  - same precedence: right to left
  - = assignment
  - perform the indicated operation between the left and right operands, then assign the result to the left operand
    - += add to
    - -= subtract from
    - *= multiply by
    - /= divide by
    - %= modulo by

Assignments

- example: a = x = y+3
- example: r = s + (t = u - v) / 3; give “same as” code

- NOTE: using an assignment operator (=) is legal anywhere it is legal to compare for equality (==), so it is not a syntax error (though, depending on the compiler, it may give a warning) because there is not a distinct boolean type in C

Boolean Operators

- **C does not have a distinct boolean type**
  - int is used instead
- **treats integer 0 as FALSE and all non-zero values as TRUE**

```
i = 0;
while (i - 10) { ... }
```

  - will execute until the variable i takes on the value 10 at which time the expression (i - 10) will become false (i.e., 0)
- **a sampling of Logical/Boolean Operators:**
  - &&, ||, and ! → AND, OR, and NOT
- **&& is used to compare two objects**
  - x != 0 && y != 0
- **short-circuit evaluation: above example, if x != 0 evaluates to false, the whole statement is false regardless of the outcome of y != 0 (same for or if first condition is true)**

Boolean Examples

Operator	Operator's Name	Example	Result
&&	AND	3>2 && 3>1	1(true)
&&	AND	3>2 && 3<1	0(false)
&&	AND	3<2 && 3<1	0(false)
||	OR	3>2 || 3>1	1(true)
||	OR	3>2 || 3<1	1(true)
||	OR	3<2 || 3<1	0(false)
!	NOT	!(3==2)	1(true)
!	NOT	!(3==3)	0(false)

A. ! (1 || 0)	ANSWER: 0
B. ! (1 || 1 && 0)	ANSWER: 0 (AND is evaluated before OR)
C. ! ((1 || 0) && 0)	ANSWER: 1 (Parenthesis are useful)