Introduction to C

```
#include <stdio.h>

int main ()
{
        printf("Welcome to CS 1621!\n");
}
```

Outline

II. Program Basics
- A. Program skeleton
  - preprocessor directives
  - global declarations
  - functions
    - local declarations
    - statements
- B. Comments and Documentation
- C. Names (identifiers)
  - reserved words

Outline (cont)

II. Program Basics (cont)
- D. Variable declarations
  1. Memory allocation
  2. Atomic types
     - void, int, float, char
- E. Constants
  1. literal
  2. defined
  3. memory

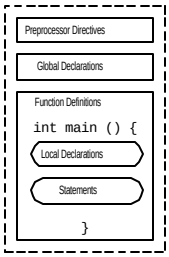
Outline (cont)

II. Program Basics (cont)
- F. Formatted input/output
  1. Files
  2. Printf (monitor output)
     - a. format strings
       - field specifications
     - b. data list
  3. Scanf (keyboard input)
     - a. format strings
     - b. address list
  4. Prompting for Input

History of C

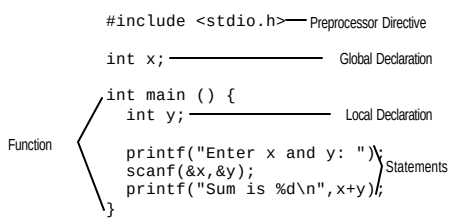
1960: ALGOL (ALGOrithmic Language)
1967: BCPL (Basic Combined Programming Language)
1970: B programming language (typeless)
1972: C: BCPL plus B with types
1978: Kernighan + Ritchie standard for C
1989: ANSI standard for C

C Program Structure



- Program defined by:
  - global declarations
  - function definitions
- May contain preprocessor directives
- Always has one function named *main*, may contain others

Parts of a Program



Preprocessor Directives

- Begin with #
- Instruct compiler to perform some transformation to file before compiling
- Example: #include <stdio.h>
  - add the *header* file stdio.h to this file
  - .h for header file
  - stdio.h defines useful input/output functions

Declarations

- Global
  - visible throughout program
  - describes data used throughout program
- Local
  - visible within function
  - describes data used only in function

Functions

- Consists of *header* and *body*
  - header: int main ()
  - body: contained between { and }
    - starts with location declarations
    - followed by series of statements
- More than one function may be defined
- Functions are *called* (invoked) - more later

Main Function

- Every program has one function **main**
- Header for main: int main ()
- Program is the sequence of statements between the { } following main
- Statements are executed one at a time from the one immediately following to main to the one before the }

Comments

- Text between /* and */
- Used to "document" the code for the human reader
- Ignored by compiler (not part of program)
- Have to be careful
  - comments may cover multiple lines
  - ends as soon as */ encountered (so no internal comments - /* An /* internal */ comment */)

Comment Example

```
#include <stdio.h>

/* This comment covers
 * multiple lines
 * in the program.
 */

int main () /* The main header */ {
  /* No local declarations */

  printf("Too many comments\n");
} /* end of main */
```

Documentation

- Global - start of program, outlines overall solution, may include structure chart
- Module - when using separate files, indication of what each file solves
- Function - inputs, return values, and logic used in defining function
- Add documentation for key (tough to understand) comments
- Names of variables - should be chosen to be meaningful, make program readable

Syntax of C

- Rules that define C language
  - Specify which tokens are valid
  - Also indicate the expected order of tokens
- Some types of tokens:
  - reserved words: include printf int…
  - identifiers: x y ...
  - literal constants: 5 'a' 5.0 ...
  - punctuation: { } ; < > # /* */

Identifier

- Names used for objects in C
- Rules for identifiers in C:
  - first char alphabetic [a-z,A-Z] or underscore (_)
  - has only alphabetic, digit, underscore chars
  - first 31 characters are significant
  - cannot duplicate a reserved word
  - case (upper/lower) matters

Reserved Words

- Identifiers that already have meaning in C
- Examples:
  - include, main, printf, scanf, if, else, …
  - more as we cover C language

Valid/Invalid Identifiers

Valid	Invalid
sum	7of9
c4_5	x-name
A_NUMBER	name with spaces
longnamewithmanychars	1234a
TRUE	int
_split_name	AXYZ&

Program Execution

- Global declarations set up
- Function *main* executed
  - local declarations set up
  - each statement in statement section executed
    - executed in order (first to last)
    - changes made by one statement affect later statements

Variables

- Named memory location
- Variables declared in global or local declaration sections
- Syntax: *Type Name*;
- Examples:

```
int sum;
float avg;
char dummy;
```

Variable Type

- Indicates how much memory to set aside for the variable
- Also determines how that space will be interpreted
- Basic types: char, int, float
  - specify amount of space (bytes) to set aside
  - what can be stored in that space
  - what operations can be performed on those vars

Variable Name

- Legal identifier
- Not a reserved word
- Must be unique:
  - not used before
  - variable names in functions (local declarations) considered to be qualified by function name
  - variable x in function main is different from x in function f1

Multiple Variable Declarations

- Can create multiple variables of the same type in one statement:

```
int x, y, z;
```

is a shorthand for

```
int x;
int y;
int z;
```

- stylistically, the latter is often preferable

Variable Initialization

- Giving a variable an initial value
- Variables not necessarily initialized when declared (value is unpredictable - *garbage*)
- Can initialize in declaration:
- Syntax: *Type Name = Value*;
- Example:

```
int x = 0;
```

Initialization Values

- Literal constant (token representing a value, like 5 representing the integer 5)
- An expression (operation that calculates a value)
- Function call

- The value, however specified, must be of the correct type

Multiple Declaration Initialization

- Can provide one value for variables initialized in one statement:
  int x, y, z = 0;
- Each variable declared and then initialized with the value

Type

- Set of possible values
  - defines size, how values stored, interpreted
- Operations that can be performed on those possible values
- Data types are associated with objects in C (variables, functions, etc.)

Standard Types

- Atomic types (cannot be broken down)
  - void
  - char
  - int
  - float, double
- Derived types
  - composed of other types

Literal Constants

- Sequences of characters (tokens) that correspond to values from that type
  `-35` is the integer -35
  `3.14159` is the floating pointer number 3.14159
  `'A'` is the character A
- Can be used to initialize variables

Void Type

- Type name: void
- Possible values: none
- Operations: none
- Useful as a placeholder

Integer Type

- Type name:
  - int
  - short int
  - long int
- Possible values: whole numbers (within given ranges) as in 5, -35, 401
- Operations: arithmetic (addition, subtraction, multiplication, …), and others

Integer Types/Values

Type	Bytes	Bits	Min Val	Max Val
short int	2	16	-32768	32767
int	4	32	-2147483648	2147483647
long int	4	32	-2147483648	2147483647

Why Limited?

- With a fixed number of bits, only a certain number of possible patterns
- 16 bits, 65,536 possible patterns
  - 32768 negative numbers
  - 1 zero
  - 32767 positive numbers
- Overflow: attempt to store a value to large in a variable (40000 in short int)

Two's Complement

Integers:

positive number: 0, number in binary

97 in binary 1*64 + 1*32 + 1*1 (1100001)

pad with leading zeroes (0 00000001100001) - 16 bits

zero: 0, all zeroes

negative number: 1, (inverse of number + 1)

-97 (1, 111111110011110 + 1)

1 111111110011111

Unsigned Integers

- Type: unsigned int
- No negative values
- unsigned int:
  - possible values: 0 to 65536
- Representation: binary number

Integer Literal Constants

Syntax:

1 or more digits

Optional leading sign (+ or -)

Optional l or L at the end for long

Optional u or U for unsigned

Examples:

5, -35, 401, 4010L, -350L, 2000UL

Floating-Point Type

- Type names:
  - float
  - double
  - long double
- Possible values: floating point numbers, 5.0 -3.5, 4.01
- Operations: arithmetic (addition, subtraction, multiplication, …), and others

Floating-Point Representation

- float: 4 bytes, 32 bits
- double: 8 bytes, 64 bits
- long double: 10 bytes, 80 bits
- Representation:
  - magnitude (some number of bits) plus exponent (remainder of bits)
  - 3.26 * 10^4 for 32600.0

Floating-Point Limitations

- Maximum, minimum exponents
  - maximum possible value (largest positive magnitude, largest positive exponent)
  - minimum value (largest negative magnitude, largest positive exponent)
  - can have overflow, and underflow
- Magnitude limited
  - cannot differentiate between values such as 1.00000000 and 1.00000001

Floating-Point Literals

- Syntax:
  - Zero or more digits, decimal point, then zero or more digits (at least one digit)
  - Whole numbers also treated as float
  - Optional sign at start
  - Can be followed by e and whole number (to represent exponent)
  - f or F at end for float
  - l or L at end for long double
- Examples: 5, .5, 0.5, -1.0, 2.1e+3, 5.1f

Character Type

- Type name: char
- Possible values: keys that can be typed at the keyboard
- Representation: each character assigned a value (ASCII values), 8 bits
  - A - binary number 65
  - a - binary number 97
  - b - binary number 98
  - 2 - binary number 50

Character Literals

- Single key stroke between quote char ‘
- Examples: ‘A’, ‘a’, ‘b’, ‘1’, ‘@’
- Some special chars:
  - ‘\0’ - null char
  - ‘\t’ - tab char
  - ‘\n’ - newline char
  - ‘\’’ - single quote char
  - ‘\\’ - backslash char

String Literals

- No string type (more later)
- Contained between double quote chars (“)
- Examples:
  “” - null string
  “A string”
  “String with newline \n char in it”
  “String with a double quote \” in it”

Constants

- Literal constants - tokens representing values from type
- Defined constants
  – syntax: #define *Name Value*
  – preprocessor command, *Name* replaced by *Value* in program
  – example: #define MAX_NUMBER 100

Constants (cont)

- Memory constants
  – declared similar to variables, type and name
  – *const* added before declaration
  – Example: const float PI = 3.14159;
  – Can be used as a variable, but one that cannot be changed
  – Since the value cannot be changed, it *must* be initialized

Formatted Input/Output

- Input comes from files
- Output sent to files
- Other objects treated like files:
  – keyboard - standard input file (stdin)
  – monitor - standard output file (stdout)
- Generally send/retrieve characters to/from files

Formatted Output

- Command: ***printf*** - print formatted
- Syntax: printf(*Format String*, *Data List*);
  – Format string any legal string
  – Characters sent (in order) to screen
- Ex.: printf(“Welcome to\nCS 1621!\n”);
  causes
  `Welcome to`
  `CS 1621!`
  to appear on monitor

Formatted Output (cont)

- Successive printf commands cause output to be added to previous output
- Ex.
  printf(“Hi, how “);
  printf(“is it going\nin 1621?”);
  prints
  `Hi, how is it going`
  `in 1621?`
  To the monitor

Field Specifications

- Format string may contain one or more field specifications
  - Syntax: %*[Flag][Width][Prec][Size]Code*
  - Codes:
    - c - data printed as character
    - d - data printed as integer
    - f - data printed as floating-point value
  - For each field specification, have one data value after format string, separated by commas

Field Specification Example

printf("%c %d %f\n",'A',35,4.5);

produces

```
A 35 4.50000
```

(varies on different computers)

Can have variables in place of literal constants (value of variable printed)

Width and Precision

- When printing numbers, generally use width/precision to determine format
  - Width: how many character spaces to use in printing the field (minimum, if more needed, more used)
  - Precision: for floating point numbers, how many characters appear after the decimal point, width counts decimal point, number of digits after decimal, remainder before decimal

Width/Precision Example

printf("%5d%8.3f\n",753,4.1678);

produces

```
  753   4.168
```

values are right justified

If not enough characters in width, minimum number used

use 1 width to indicate minimum number of chars should be used

Left Justification (Flags)

Put - after % to indicate value is left justified

printf("%-5d%-8.3fX\n",753,4.1678);

produces

```
753  4.168   X
```

For integers, put 0 after % to indicate should pad with 0's

printf("%05d",753);

produces

```
00753
```

Size Indicator

- Use hd for small integers
- Use ld for long integers
- Use Lf for long double
- Determines how value is treated

Printf Notes

- Important to have one value for each field specification
  - some C versions allow you to give too few values (garbage values are formatted and printed)
- Values converted to proper type
  - printf("%c",97); produces the character a on the screen

Formatted Input

- Command: ***scanf*** - scan formatted
- Syntax: scanf(*Format String*, *Address List*);
  - Format string a string with one or more field specifications
  - Characters read from keyboard, stored in variables
- scanf("%c %d %f",&cVar,&dVar,&fVar);
  attempts to read first a single character, then a whole number, then a floating point number from the keyboard

Formatted Input (cont)

- Generally only have field specifications and spaces in string
  - any other character must be matched exactly (user must type that char or chars)
  - space characters indicate white-space is ignored
  - "white-space" - spaces, tabs, newlines
  - %d and %f generally ignore leading white space anyway (looking for numbers)
  - %d and %f read until next non-number char reached

Formatted Input (cont)

- More notes
  - can use width in field specifications to indicate max number of characters to read for number
  - computer will not read input until return typed
  - if not enough input on this line, next line read, (and line after, etc.)
  - inappropriate chars result in run-time errors (x when number expected)
  - if end-of-file occurs while variable being read, an error occurs

Address Operator

- & - address operator
- Put before a variable (as in &x)
- Tells the computer to store the value read at the location of the variable
- More on address operators later

Scanf Rules

- Conversion process continues until
  - end of file reached
  - maximum number of characters processed
  - non-number char found number processed
  - an error is detected (inappropriate char)
- Field specification for each variable
- Variable address for each field spec.
- Any character other than whitespace must be matched exactly

Scanf Example

```
scanf("%d%c %f",&x,&c,&y);
```

and following typed:

```
-543A

4.056 56
```

-543 stored in x, A stored in c, 4.056 stored in y, space and 56 still waiting (for next scanf)

Prompting for Input

- Using output statements to inform the user what information is needed:

  printf("Enter an integer: ");

  scanf("%d",&intToRead);
- Output statement provides a cue to the user:

  Enter an integer: *user types here*