The Design of C

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."

– Dennis Ritchie

Goals of this Lecture

Help you learn about:

- The decisions that were made by the designers* of C
- **Why** they made those decisions

… and thereby…

- The fundamentals of C

Why?

- Learning the design rationale of the C language provides a richer understanding of C itself
- A power programmer knows both the programming language and its design rationale

* Dennis Ritchie & members of standardization committees

Goals of C

Designers wanted C to:	But also:
Support system programming	Support application programming
Be low-level	Be portable
Be easy for people to handle	Be easy for computers to handle

- Conflicting goals on multiple dimensions!
- Result: different design decisions than Java

Operators

Issue: What kinds of operators should C have?

Thought process

- Should handle typical operations
- Should handle bit-level programming ("bit twiddling")
- Should provide a mechanism for converting from one type to another

Operators

Decisions

- Provide typical arithmetic operators: `+ - * / %`
- Provide typical relational operators: `== != < <= > >=`
    - Each evaluates to 0 $\Rightarrow$ FALSE, 1 $\Rightarrow$ TRUE
- Provide typical logical operators: `! && ||`
    - Each interprets 0 $\Rightarrow$ FALSE, non-0 $\Rightarrow$ TRUE
    - Each evaluates to 0 $\Rightarrow$ FALSE, 1 $\Rightarrow$ TRUE
- Provide bitwise operators: `~ & | ^ >> <<`
- Provide a cast operator: `(type)`

Logical vs. Bitwise Ops

Logical AND (&&) vs. bitwise AND (&)

- **2 (TRUE) && 1 (TRUE) => 1 (TRUE)**

```
Decimal  Binary
      2  00000000 00000000 00000000 00000010
   && 1  00000000 00000000 00000000 00000001
   ----  -----------------------------------
      1  00000000 00000000 00000000 00000001
```

- **2 (TRUE) & 1 (TRUE) => 0 (FALSE)**

```
Decimal  Binary
      2  00000000 00000000 00000000 00000010
    & 1  00000000 00000000 00000000 00000001
   ----  -----------------------------------
      0  00000000 00000000 00000000 00000000
```

Implication:

- Use **logical** AND to control flow of logic
- Use **bitwise** AND only when doing bit-level manipulation
- Same for OR and NOT

Assignment Operator

Issue: What about assignment?

Thought process

- Must have a way to assign a value to a variable
- Many high-level languages provide an assignment **statement**
- Would be more expressive to define an assignment **operator**
  - Performs assignment, and then *evaluates to the assigned value*
  - Allows assignment to appear within larger expressions

Decisions

- Provide assignment operator: **=**
- Define assignment operator so it changes the value of a variable, and also evaluates to that value

Assignment Operator Examples

Examples

```
i = 0;
   /* Side effect: assign 0 to i.
      Evaluate to 0.

j = i = 0; /* Assignment op has R to L associativity */
   /* Side effect: assign 0 to i.
      Evaluate to 0.
      Side effect: assign 0 to j.
      Evaluate to 0. */

while ((i = getchar()) != EOF) …
   /* Read a character (maybe).
      Side effect: assign that character to i.
      Evaluate to that character.
      Compare that emitted value to EOF.
      Evaluate to 0 (FALSE) or 1 (TRUE). */
```

Special-Purpose Assignment

Issue: Should C provide tailored assignment operators?

Thought process

- The construct `a = b + c` is flexible
- The construct `i = i + c` is somewhat common
- The construct `i = i + 1` is very common
- Special-purpose operators make code more expressive
  - Might reduce some errors
  - May complicate the language and compiler

Decisions

- Introduce `+=` operator to do things like `i += c`
- Extend to `-= *= /= ~= &= |= ^= <<= >>=`
- Special-case increment and decrement: `i++ i--`
- Provide both pre- and post-inc/dec: `x = ++i; y = i++;`

iClicker Question

Q: What are **`i`** and **`j`** set to in the following code?

```
i = 5;
j = i++;
j += ++i;
```

A. 5, 7

B. 7, 5

C. 7, 11

D. 7, 12

E. 7, 13

sizeof Operator

Issue: How to determine the sizes of data?

Thought process

- The sizes of most primitive types are un- or under-specified
- Provide a way to find size of a given variable programmatically

Decisions

- Provide a **`sizeof`** operator
  - Applied at compile-time
  - Operand can be a **data type**
  - Operand can be an **expression**,
    from which the compiler infers a data type

Examples, on armlab using gcc217

- **`sizeof(int)`** evaluates to 4
- **`sizeof(i)`** – where **`i`** is a variable of type **`int`** – evaluates to 4

iClicker Question

Q: What is the value of the following **`sizeof`** expression on the armlab machines?

```
int i = 1;

sizeof(i + 2L)
```

A. 3

B. 4

C. 8

D. 12

E. error

Other Operators

Issue: What other operators should C have?

Decisions

- Function call operator
  - Should mimic the familiar mathematical notation
  - **`function(param1, param2, …)`**
- Conditional operator: **`?:`**
  - The only ternary operator: "inline if statement"
  - Example: **`(i < j) ? i : j`** evaluates to min of **`i`** and **`j`**
  - See King book for details
- Sequence operator (rarely used):   **`,`**
  - See King book for details
- Pointer-related operators: **`& *`**
  - Described later in the course
- Structure-related operators: **`. ->`**
  - Described later in the course

Operators Summary: C vs. Java

Java only

• `>>>`	right shift with zero fill
• `new`	create an object
• `instanceof`	is left operand an object of class right operand?

C only

• `->`	structure member select
• `*`	dereference
• `&`	address of
• `,`	sequence
• `sizeof`	compile-time size of

Control Statements: History

What the computer does "under the hood":

```
/* add up numbers from 1 to
   whatever is stored in R2 */
1. R0 = 0
2. R1 = 1
3. compare R1, R2
4. if greater goto 8
5. R0 = R0 + R1
6. R1 = R1 + 1
7. goto 3
8. /* answer in R0 */
```

Early programming languages (1950s)

```
/* add up numbers from 1 to n */

sum = 0
i = 1
LOOP:
if (i > n) goto DONE
sum = sum + i
i = i + 1
goto LOOP
DONE: /* answer in sum */
```

Some high-level conveniences (variable names, labels)
but control flow based on `if` and `goto`

Control Statements

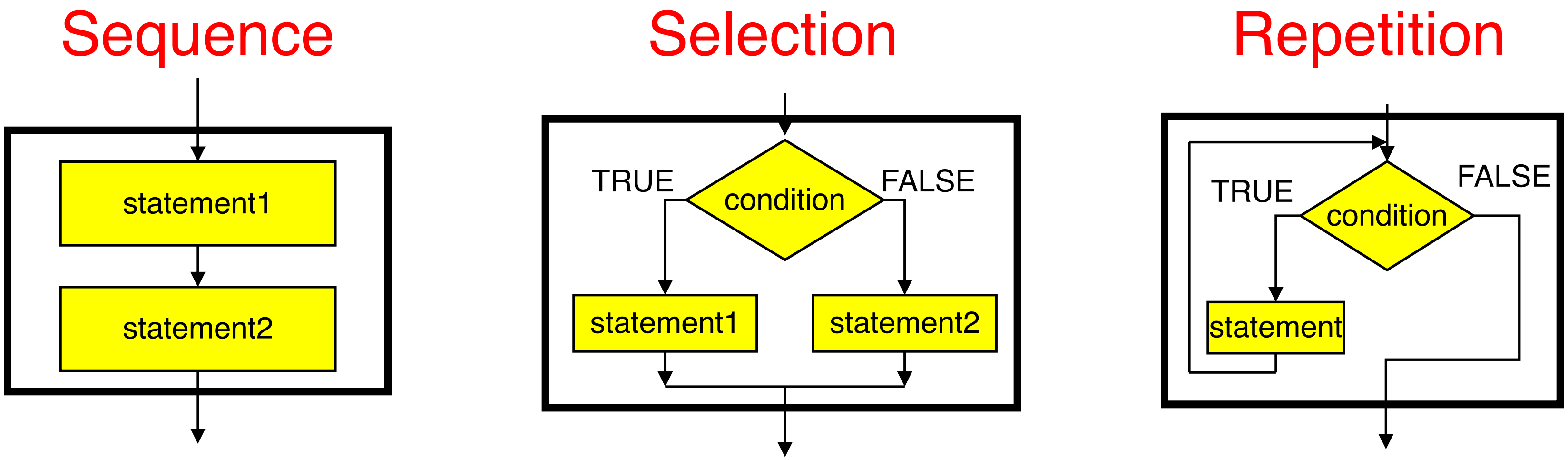
Algol-60 language (1960)

- BEGIN-END, IF-THEN-ELSE, WHILE-DO, FOR, (and also GOTO)

Scientific background

- **Böhm** and **Jacopini** proved (1966) that any algorithm *can* be expressed as the nesting of only 3 control structures:

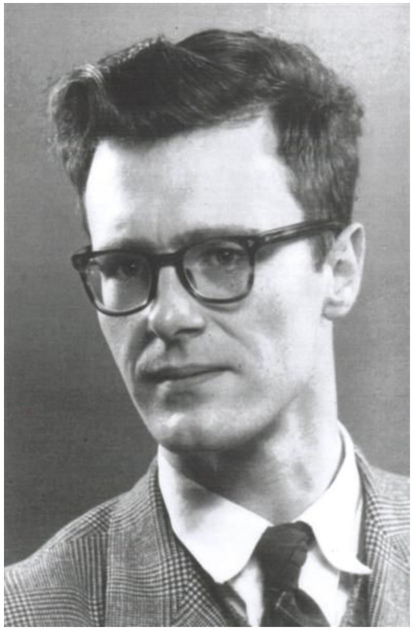
Corrado Böhm



Control Statements (cont.)

Thought Process

- **Dijkstra** argued that any algorithm ***should*** be expressed using only those control structures (*GOTO Statement Considered Harmful*, 1968)

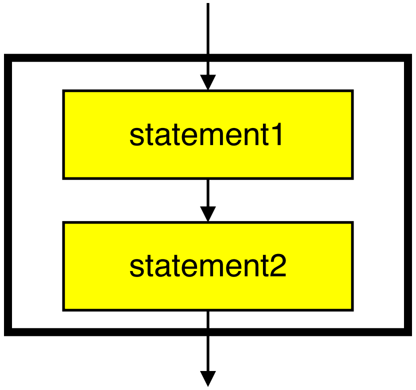


Edsger Dijkstra

C language design (1972)

- Basically follow ALGOL-60, but use { braces } instead of the more heavyweight BEGIN – END

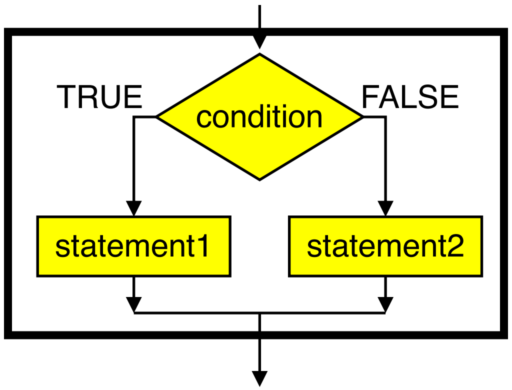
Sequence Statement



Compound statement, alias **block**

```
{
    statement1;
    statement2;
     ...
}
```

Selection Statements



if and **if…else** statements

```
if (expr)
    statement1;
```

```
if (expr)
    statement1;
else
    statement2;
```

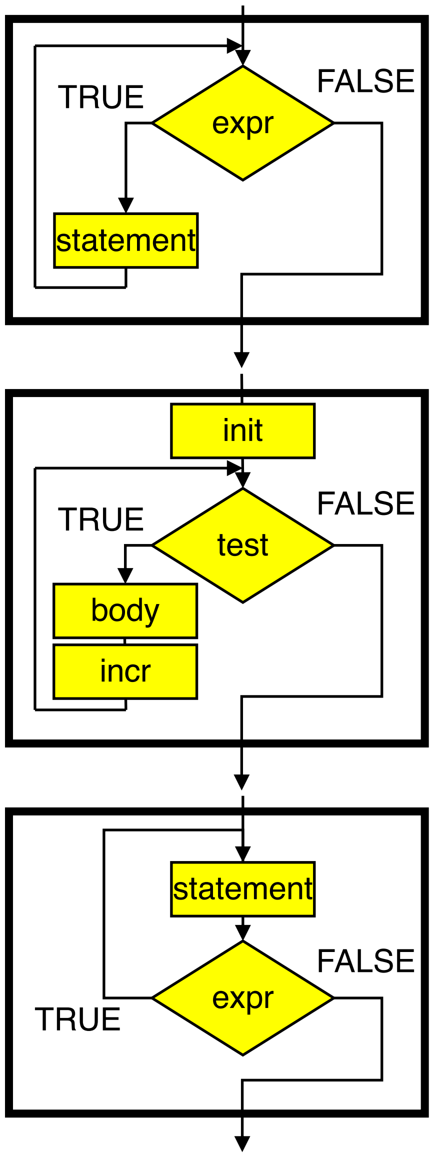
Selection Statements

`switch` and **`break`** statements, for multi-path decisions on a single ***`integerExpr`***

```
switch (integerExpr)
{   case integerLiteral1:
          …
          break;
    case integerLiteral2:
          …
          break;
    …
    default:
          …
}
```

What happens if you forget to **`break`**?

Repetition Statements



while statement: test at leading edge

```
while (expr)
    statement;
```

for statement: test at leading edge, increment at trailing edge

```
for (initExpr; testExpr; incrExpr)
    bodyStatement;
```

do…while statement: test at trailing edge

```
do
    statement;
while (expr);
```

Other Control Statements

Issue: What other control statements should C provide?

Decisions

- `break` statement
  - Breaks out of closest enclosing `switch` or repetition statement
- `continue` statement
  - Skips remainder of current loop iteration
  - Continues with next loop iteration
  - When used within `for`, still executes ***`incrementExpr`***
- `goto` statement grudgingly provided
  - Jump to specified **label**

Declaring Variables

Issue: Should C require variable declarations?

Thought process:

- Declaring variables allows compiler to check “spelling”
- Declaring variables allows compiler to allocate memory more efficiently
- Declaring variables produces fewer surprises about types of variables
- (But, requires more typing; invites “do what I mean, not what I say” complaints)

Declaring Variables

Decisions:

- Require variable declarations
- Provide **declaration statement**
- Programmer specifies type of variable (and other attributes too)

Examples

- `int i;`
- `int i, j;`
- `int i = 5;`
- `const int i = 5;   /* value of i cannot change */`
- `static int i;     /* covered later in course */`
- `extern int i;     /* covered later in course */`

Declaring Variables

Decisions (cont.):

- Unlike Java, declaration statements in C90 ***must*** appear before any other kind of statement in compound statement

```
{
   int i;
   /* Non-declaration
      stmts that use i. */
   …
   int j;
   /* Non-declaration
      stmts that use j. */
   …
}
```

Illegal in C

```
{
   int i;
   int j;
   /* Non-declaration
      stmts that use i. */
   …
   /* Non-declaration
      stmts that use j. */
   …
}
```

Legal in C

Repetition Statements

Decisions (cont.)

- Similarly, cannot declare loop control variable in **`for`** statement

```
{
   …
   for (int i = 0; i < 10; i++)
      /* Do something */
   …
}
```

Illegal in C

```
{
   int i;
   …
   for (i = 0; i < 10; i++)
      /* Do something */
   …
}
```

Legal in C

Statements Summary: C vs. Java

Java only

- Declarations anywhere within block
- Declare immutable variables with **`final`**
- Conditionals of type **`boolean`**
- "Labeled" **`break`** and **`continue`**
- No **`goto`**

C only

- Declarations only at beginning block
- Declare immutable variables with **`const`**
- Conditionals of any type (checked for zero / nonzero)
- No "labeled" **`break`** and **`continue`**
- **`goto`** provided (but using it in COS217 is a hanging offense)

iClicker Question

Q: What does the following code print?

```
int i = 1;
switch (i++) {
   case 1: printf("%d", ++i);
   case 2: printf("%d", i++);
}
```

A. 1

B. 2

C. 3

D. 22

E. 33

iClicker Question

Q: What does the following code print?

```
int i = 1;
switch (i=i++) {
   case 1: printf("%d", ++i);
   case 2: printf("%d", i++);
}
```

A. 1

B. 2

C. 3

D. 22

E. 33

I/O Facilities

Issue: Should C provide I/O facilities?

Thought process

- Unix provides the **file** abstraction
  - A file is a sequence of characters with an indication of the current position
- Unix provides 3 standard files
  - Standard input, standard output, standard error
- C should be able to use those files, and others
- I/O facilities are complex
- C should be small/simple

I/O Facilities

Decisions

- Do not provide I/O facilities in the **language**
- Instead provide I/O facilities in **standard library**
  - **Constant**: `EOF`
  - **Data type**: `FILE` (described later in course)
  - **Variables**: `stdin`, `stdout`, and `stderr`
  - **Functions**: …

Reading Characters

Issue: What functions should C provide for reading characters from standard input?

Thought process

- Need function to read a single character from **`stdin`**
- Function must have a way to indicate failure, that is, to indicate that no characters remain

Decisions

- Provide **`getchar()`** function
- Make return type of **`getchar()`** wider than **`char`**
    - Make it **`int`**; that's the natural word size
- Define **`getchar()`** to return **`EOF`** (a special non-character **`int`**) to indicate failure

Reminder: there is no such thing as "the **`EOF`** character"

Writing Characters

Issue: What functions should C provide for writing a character to standard output?

Thought process

- Need function to write a single character to **`stdout`**

Decisions

- Provide a **`putchar()`** function
- Define **`putchar()`** to accept one parameter
  - For symmetry with **`getchar()`**, parameter is an **`int`**

Reading Other Data Types

Issue: What functions should C provide for reading data of other primitive types?

Thought process

- Must convert external form (sequence of character codes) to internal form
- Could provide `getshort()`, `getint()`, `getfloat()`, etc.
- Could provide one parameterized function to read any primitive type of data

Decisions

- Provide `scanf()` function
- Can read any primitive type of data
- First parameter is a **format string** containing **conversion specs**

See King book for details

Writing Other Data Types

Issue: What functions should C provide for writing data of other primitive types?

Thought process

- Must convert internal form to external form (sequence of character codes)
- Could provide `putshort()`, `putint()`, `putfloat()`, etc.
- Could provide one parameterized function to write any primitive type of data

Decisions

- Provide `printf()` function
- Can write any primitive type of data
- First parameter is a **format string** containing **conversion specs**

See King book for details

Other I/O Facilities

Issue: What other I/O functions should C provide?

Decisions

- `fopen()`: Open a stream
- `fclose()`: Close a stream
- `fgetc()`: Read a character from specified stream
- `fputc()`: Write a character to specified stream
- `fgets()`: Read a line/string from specified stream
- `fputs()`: Write a line/string to specified stream
- `fscanf()`: Read data from specified stream
- `fprintf()`: Write data to specified stream

Described in King book, and later in the course after covering files, arrays, and strings

Summary

C design decisions and the goals that affected them

- Data types (last time)
- Operators
- Statements
- I/O facilities

Knowing the design goals and how they affected the design decisions can yield a rich understanding of C

Appendix: The Cast Operator

Cast operator has multiple meanings:

(1) Cast between integer type and floating point type:

- Compiler generates code
- At run-time, code performs conversion

`f`	`11000001110110110000000000000000`	**-27.375**

```
i = (int)f
```

`i`	`11111111111111111111111111100101`	**-27**

Appendix: The Cast Operator

(2) Cast between floating point types of different sizes:

- Compiler generates code
- At run-time, code performs conversion

`f`	`11000001110110110000000000000000`	-27.375

```
d = (double)f
```

`d`	`11000000001110110110000000000000` `00000000000000000000000000000000`	-27.375

Appendix: The Cast Operator

(3) Cast between integer types of different sizes:

- Compiler generates code
- At run-time, code performs conversion

```
i  00000000000000000000000000000010   2

c = (char)i

c  00000010   2
```

Appendix: The Cast Operator

(4) Cast between integer types of same size:

- Compiler generates no code
- Compiler views given bit-pattern in a different way

```
i   11111111111111111111111111111110   -2

u = (unsigned int)i

u   11111111111111111111111111111110   4294967294
```