The C Language

COMS W4995-02

Prof. Stephen A. Edwards
Fall 2002
Columbia University
Department of Computer Science

The C Language

Currently, the most commonly-used language for embedded systems

"High-level assembly"

Very portable: compilers exist for virtually every processor

Easy-to-understand compilation

Produces efficient code

Fairly concise

C History

Developed between 1969 and 1973 along with Unix

Due mostly to Dennis Ritchie

Designed for systems programming

- Operating systems
- Utility programs
- Compilers
- Filters

Evolved from B, which evolved from BCPL

BCPL

Martin Richards, Cambridge, 1967

Typeless

- Everything a machine word (n-bit integer)
- Pointers (addresses) and integers identical

Memory: undifferentiated array of words

Natural model for word-addressed machines

Local variables depend on frame-pointer-relative addressing: no dynamically-sized automatic objects

Strings awkward: Routines expand and pack bytes to/from word arrays

C History

Original machine (DEC PDP-11) was very small:

24K bytes of memory, 12K used for operating system

Written when computers were big, capital equipment

Group would get one, develop new language, OS

C History

Many language features designed to reduce memory

- Forward declarations required for everything
- Designed to work in one pass: must know everything
- No function nesting

PDP-11 was byte-addressed

- Now standard
- Meant BCPL's word-based model was insufficient

Euclid's Algorithm in C

```
int gcd(int m, int n )
{
  int r;
  while ((r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```

← "New syle" function declaration lists number and type of arguments. Originally only listed return type. Generated code did not care how many arguments were actually passed, and everything was a word. Arguments are call-by-value

Euclid's Algorithm in C

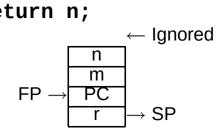
```
int gcd(int m, int n )
{
  int r;
  while ((r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```

Automatic variable (→ int r;)

Allocated on stack when function entered, released on return

Parameters & automatic variables accessed via frame pointer

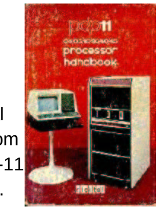
Other temporaries also stacked



Euclid on the PDP-11

```
     .globl _gcd              GPRs: r0–r7
     .text                    r7=PC, r6=SP, r5=FP
_gcd:
     jsr r5, rsave            Save SP in FP
L2:  mov 4(r5), r1            r1 = n
     sxt r0                   sign extend
     div 6(r5), r0            r0, r1 = m ÷ n
     mov r1, -10(r5)          r = r1 (m % n)
     jeq L3                   if r == 0 goto L3
     mov 6(r5), 4(r5)         m = n
     mov -10(r5), 6(r5)       n = r
     jbr L2
L3:  mov 6(r5), r0            r0 = n
     jbr L1                   non-optimizing compiler
L1:  jmp rretrn               return r0 (n)
```

Euclid on the PDP-11




```
    .globl _gcd
    .text
_gcd:
    jsr r5, rsave
L2: mov 4(r5), r1
    sxt r0
    div 6(r5), r0
    mov r1, -10(r5)
    jeq L3
    mov 6(r5), 4(r5)
    mov -10(r5), 6(r5)
    jbr L2
L3: mov 6(r5), r0
    jbr L1
L1: jmp rretrn
```

Very natural mapping from C into PDP-11 instructions.

Complex addressing modes make frame-pointer-relative accesses easy.

Another idiosyncrasy: registers were memory-mapped, so taking address of a variable in a register is straightforward.

Pieces of C

Types and Variables

- Definitions of data in memory

Expressions

- Arithmetic, logical, and assignment operators in an infix notation

Statements

- Sequences of conditional, iteration, and branching instructions

Functions

- Groups of statements invoked recursively

C Types

Basic types: char, int, float, and double

Meant to match the processor's native types

- Natural translation into assembly
- Fundamentally nonportable: a function of processor architecture

Declarators

Declaration: string of specifiers followed by a declarator

basic type

`static unsigned int (*f[10])(int, char*)[10];`

specifiers — declarator

Declarator's notation matches that of an expression: use it to return the basic type.

Largely regarded as the worst syntactic aspect of C: both pre- (pointers) and postfix operators (arrays, functions).

Struct bit-fields

Aggressively packs data into memory

```
struct {
  unsigned int baud : 5;
  unsigned int div2 : 1;
  unsigned int use_external_clock : 1;
} flags;
```

Compiler will pack these fields into words.

Implementation-dependent packing, ordering, etc.

Usually not very efficient: requires masking, shifting, and read-modify-write operations.

Code generated by bit fields

```
struct {
  unsigned int a : 5;
  unsigned int b : 2;
  unsigned int c : 3;
} flags;

void foo(int c) {
  unsigned int b1 =
              flags.b;
  flags.c = c;
}
```

```
# unsigned int b1 = flags.b
   movb    flags, %al
   shrb    5, %al
   movzbl  %al, %eax
   andl    3, %eax
   movl    %eax, -4(%ebp)

# flags.c = c;
   movl    flags, %eax
   movl    8(%ebp), %edx
   andl    7, %edx
   sall    7, %edx
   andl    -897, %eax
   orl     %edx, %eax
   movl    %eax, flags
```

C Unions

Like **structs**, but only stores the most-recently-written field.

```
union {
  int ival;
  float fval;
  char *sval;
} u;
```

Useful for arrays of dissimilar objects

Potentially very dangerous: not type-safe

Good example of C's philosophy: Provide powerful mechanisms that can be abused

Layout of Records and Unions

Modern processors have byte-addressable memory.

0
1
2
3
4

Many data types (integers, addresses, floating-point numbers) are wider than a byte.

16-bit integer:			1	0
32-bit integer:	3	2	1	0

Layout of Records and Unions

Modern memory systems read data in 32-, 64-, or 128-bit chunks:

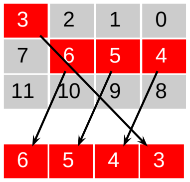
3	2	1	0
7	6	5	4
11	10	9	8

Reading an aligned 32-bit value is fast: a single operation.

3	2	1	0
7	6	5	4
11	10	9	8

Layout of Records and Unions

Slower to read an unaligned value: two reads plus shift.



SPARC prohibits unaligned accesses.

MIPS has special unaligned load/store instructions.

x86, 68k run more slowly with unaligned accesses.

Layout of Records and Unions

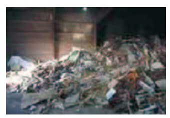
Most languages "pad" the layout of records to ensure alignment restrictions.

```
struct padded {
  int x;   /* 4 bytes */
  char z;  /* 1 byte */
  short y; /* 2 bytes */
  char w;  /* 1 byte */
};
```

C Storage Classes

```
/* fixed address: visible to other files */
int global_static;

/* fixed address: only visible within file */
static int file_static;

/* parameters always stacked */
int foo(int auto_param)
{
  /* fixed address: only visible to function */
  static int func_static;

  /* stacked: only visible to function */
  int auto_i, auto_a[10];

  /* array explicitly allocated on heap (pointer stacked) */
  double *auto_d =
     malloc(sizeof(double)*5);

  /* return value passed in register or stack */
  return auto_i;
}
```

malloc() and free()



Library routines for managing the heap

```
int *a;
a = (int *) malloc(sizeof(int) * k);
a[5] = 3;
free(a);
```

Allocate and free arbitrary-sized chunks of memory in any order

malloc() and free()

More flexible than (stacked) automatic variables

More costly in time and space

malloc() and free() use non-constant-time algorithms

Two-word overhead for each allocated block:

- Pointer to next empty block
- Size of this block

Common source of errors:

Using uninitialized memory	Using freed memory
Not allocating enough	Indexing past block
Neglecting to free disused blocks (memory leaks)	

malloc() and free()

Memory usage errors so pervasive, entire successful company (Pure Software) founded to sell tool to track them down

Purify tool inserts code that verifies each memory access

Reports accesses of uninitialized memory, unallocated memory, etc.

Publicly-available Electric Fence tool does something similar

malloc() and free()

```
#include <stdlib.h>
struct point {int x, y; };
int play_with_points(int n)
{
  struct point *points;
  points = malloc(n*sizeof(struct point));
  int i;
  for ( i = 0 ; i < n ; i++ ) {
    points[i].x = random();
    points[i].y = random();
  }
  /* do something with the array */

  free(points);
}
```

Dynamic Storage Allocation

Dynamic Storage Allocation

Rules:

Each allocated block contiguous (no holes)

Blocks stay fixed once allocated

`malloc()`

Find an area large enough for requested block

Mark memory as allocated

`free()`

Mark the block as unallocated

Simple Dynamic Storage Allocation

Maintaining information about free memory

- Simplest: Linked list

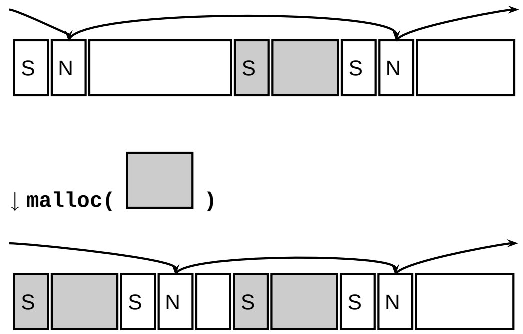
The algorithm for locating a suitable block

- Simplest: First-fit

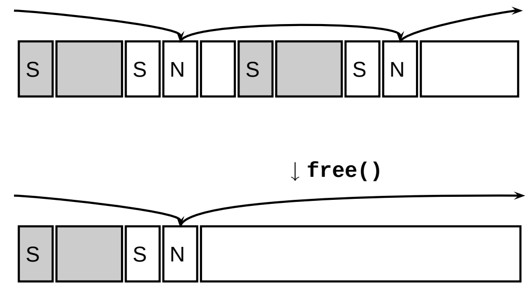
The algorithm for freeing an allocated block

- Simplest: Coalesce adjacent free blocks

Dynamic Storage Allocation



Simple Dynamic Storage Allocation



Dynamic Storage Allocation

Many, many other approaches.

Other "fit" algorithms

Segregation of objects by size

More clever data structures

malloc() and free() variants

ANSI does not define implementation of malloc()/free().

Memory-intensive programs may use alternatives:

Memory pools: Differently-managed heap areas

Stack-based pool: only free whole pool at once

- Nice for build-once data structures

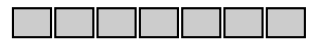
Single-size-object pool:

- Fit, allocation, etc. much faster
- Good for object-oriented programs

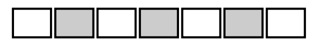
On unix, implemented on top of **`sbrk()`** system call (requests additional memory from OS).

Fragmentation

`malloc(`  **`)`** seven times give



`free()` four times gives



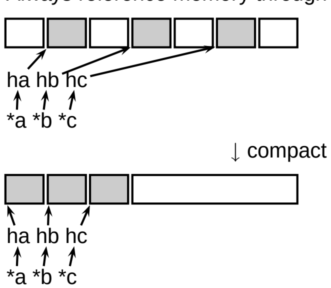
`malloc(`  **`)`** ?

Need more memory; can't use fragmented memory.

Fragmentation and Handles

Standard CS solution: Add another layer of indirection.

Always reference memory through "handles."



The original Macintosh did this to save memory.

Automatic Garbage Collection

Remove the need for explicit deallocation.

System periodically identifies reachable memory and frees unreachable memory.

Reference counting one approach.

Mark-and-sweep another: cures fragmentation.

Used in Java, functional languages, etc.

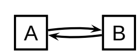
Automatic Garbage Collection

Challenges:

How do you identify all reachable memory?

(Start from program variables, walk all data structures.)

Circular structures defy reference counting:



Neither is reachable, yet both have non-zero reference counts.

Garbage collectors often conservative: don't try to collect everything, just that which is definitely garbage.

Arrays

Array: sequence of identical objects in memory

`int a[10];` means space for ten integers

By itself, **a** is the address of the first integer

`*a` and `a[0]` mean the same thing

The address of **a** is not stored in memory: the compiler inserts code to compute it when it appears

Ritchie calls this interpretation the biggest conceptual jump from BCPL to C. *Makes it unnecessary to initialize arrays in structures*

Lazy Logical Operators

"Short circuit" tests save time

```
if ( a == 3 && b == 4 && c == 5 ) { ... }
```

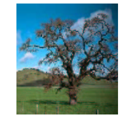
equivalent to

```
if (a == 3) { if (b ==4) { if (c == 5) { ... } }
```

Strict left-to-right evaluation order provides safety

```
if ( i <= SIZE && a[i] == 0 ) { ... }
```

The Switch Statment



```
switch (expr) {        tmp = expr;
                       if (tmp == 1) goto L1;
                       else if (tmp == 5) goto L5;
                       else if (tmp == 6) goto L6;
                       else goto Default;
case 1: /* ... */      L1: /* ... */
  break;                 goto Break;
case 5:                L5: ;
case 6: /* ... */      L6: /* ... */
  break;                 goto Break;
default: /* ... */     Default: /* ... */
  break;                 goto Break;
}                      Break:
```

Switch Generates Interesting Code

Sparse labels tested sequentially

```
if (e == 1) goto L1;
else if (e == 10) goto L10;
else if (e == 100) goto L100;
```

Dense cases uses a jump table:

```
/* uses gcc extensions */
static void *table[] =
   { &&L1, &&L2, &&Default, &&L4, &&L5 };
if (e >= 1 && e <= 5) goto *table[e];
```

setjmp/longjmp: Sloppy exceptions

```
#include <setjmp.h>
jmp_buf closure;  /* address, stack */
void top(void) {
  switch (setjmp(closure)) {
    case 0: child(); break;
    case 1: /* longjmp called */ break;
}}
void child() {child2(); }
void child2() {longjmp(closure, 1); }
```

Nondeterminism in C

Library routines

- malloc() returns a nondeterministically-chosen address
- Address used as a hash key produces nondeterministic results

Argument evaluation order

- myfunc(func1(), func2(), func3())
- func1, func2, and func3 may be called in any order

Nondeterminism in C

Word sizes

```
  int a;
  a = 1 << 16;  /* Might be zero */
  a = 1 << 32;  /* Might be zero */
```

Uninitialized variables

- Automatic variables may take values from stack
- Global variables left to the whims of the OS?

Nondeterminism in C

Reading the wrong value from a union

- union int a; float b; u; u.a = 10; printf("%g", u.b);

Pointer dereference

- *a undefined unless it points within an allocated array and has been initialized
- Very easy to violate these rules
- Legal: int a[10]; a[-1] = 3; a[10] = 2; a[11] = 5;
- int *a, *b; a - b only defined if a and b point into the same array

Nondeterminism in C

How to deal with nondeterminism? *Caveat programmer*

Studiously avoid nondeterministic constructs

Compilers, lint, etc. don't really help

Philosophy of C: get out of the programmer's way

C treats you like a consenting adult

Created by a systems programmer (Ritchie)

Pascal treats you like a misbehaving child

Created by an educator (Wirth)

Ada treats you like a criminal

Created by the Department of Defense