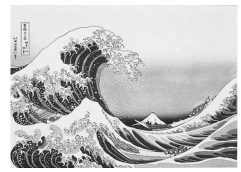
## The C Language

Prof. Stephen A. Edwards



## The C Language

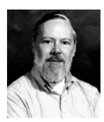
- Currently, the most commonly-used language for embedded systems
- "High-level assembly"
- Very portable: compilers exist for virtually every processor
- Easy-to-understand compilation
- Produces efficient code
- Fairly concise





## 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

- Designed by Martin Richards (Cambridge) in 1967
- Typeless
  - Everything an n-bit integer (a machine word)
  - Pointers (addresses) and integers identical
- Memory is an undifferentiated array of words
- Natural model for word-addressed machines
- Local variables depend on frame-pointer-relative addressing: dynamically-sized automatic objects not permitted
- 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

## Hello World in C

```
#include <stdio.h>

void main()
{
  printf("Hello, world!\n");
}
```

Preprocessor used to share information among source files

- Clumsy

+ Cheaply implemented

+ Very flexible



## Hello World in C

```
#include <stdio.h>

void main()
{
  printf("Hello, world!\n");
}
```

Program mostly a collection of functions

"main" function special: the entry point

"void" qualifier indicates function does not return anything

I/O performed by a library function: not included in the language



## 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 Style" function declaration lists number and type of arguments

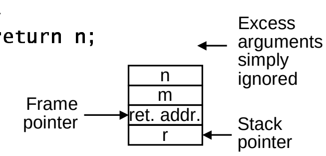
Originally only listed return type. Generated code did not care how many arguments were actually passed.

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

Storage allocated on stack when function entered, released when it returns.

All parameters, automatic variables accessed w.r.t. frame pointer.

Extra storage needed while evaluating large expressions also placed on the stack



## Euclid's algorithm in C

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

Expression: C's basic type of statement.

Arithmetic and logical

Assignment (=) returns a value, so can be used in expressions

% is remainder

!= is not equal



## Euclid's algorithm in C

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

Each function returns a single value, usually an integer. Returned through a specific register by convention.

High-level control-flow statement. Ultimately becomes a conditional branch.

Supports "structured programming"

## Euclid Compiled on PDP-11

```
.globl  _gcd               r0-r7
.text                      PC is r7, SP is r6, FP is r5
_gcd:
   jsr  r5,rsave           save sp in frame pointer r5
L2:mov  4(r5),r1           r1 = n
   sxt  r0                 sign extend
   div  6(r5),r0           m / n = r0,r1
   mov  r1,-10(r5)         r = m % n
   jeq  L3
   mov  6(r5),4(r5)        m = n
   mov  -10(r5),6(r5)      n = r
   jbr  L2
L3:mov  6(r5),r0           return n in r0
   jbr  L1
L1:jmp  rretrn             restore sp ptr, return
```

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



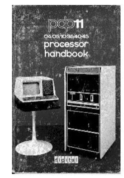
## Euclid Compiled on PDP-11

```
.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 and variables 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**
- **Declaration syntax: string of specifiers followed by a declarator**
- **Declarator's notation matches that in an expression**
- **Access a symbol using its declarator and get the basic type back**



## C Type Examples

| | |
|---|---|
| `int i;` | **Integer** |
| `int *j, k;` | **j: pointer to integer, int k** |
| `unsigned char *ch;` | **ch: pointer to unsigned char** |
| `float f[10];` | **Array of 10 floats** |
| `char nextChar(int, char*);` | **2-arg function** |
| `int a[3][5][10];` | **Array of three arrays of five …** |
| `int *func1(float);` | **function returning int *** |
| `int (*func2)(void);` | **pointer to function returning int** |



## C Typedef

- **Type declarations recursive, complicated.**
- **Name new types with typedef**

- **Instead of**

```
int (*func2)(void)
```

**use**

```
typedef int func2t(void);
func2t *func2;
```

## C Structures

- **A struct is an object with named fields:**

```
struct {
  char *name;
  int x, y;
  int h, w;
} box;
```

- **Accessed using "dot" notation:**

```
  box.x = 5;
  box.y = 2;
```



## Struct bit-fields

- **Way to aggressively pack data in memory**

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

- **Compiler will pack these fields into words**
- **Very implementation dependent: no guarantees of ordering, packing, etc.**
- **Usually less efficient**
  - **Reading a field requires masking and shifting**



## C Unions

- **Can store objects of different types at different times**

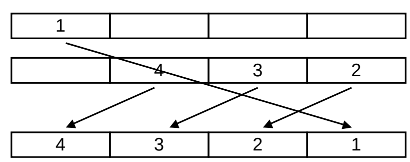
```
union {
 int ival;
 float fval;
 char *sval;
};
```

- **Useful for arrays of dissimilar objects**
- **Potentially very dangerous**
- **Good example of C's philosophy**
  - **Provide powerful mechanisms that can be abused**



## Alignment of data in structs

- **Most processors require n-byte objects to be in memory at address n*k**
- **Side effect of wide memory busses**
- **E.g., a 32-bit memory bus**
- **Read from address 3 requires two accesses, shifting**

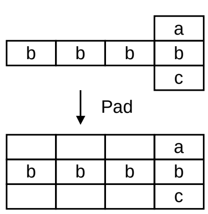





## Alignment of data in structs

- **Compilers add "padding" to structs to ensure proper alignment, especially for arrays**
- **Pad to ensure alignment of largest object (with biggest requirement)**

```
struct {
  char a;
  int b;
  char c;
}
```




- **Moral: rearrange to save memory**



## C Storage Classes

```
#include <stdlib.h>

int global_static;
static int file_static;

void foo(int auto_param)
{
  static int func_static;
  int auto_i, auto_a[10];
  double *auto_d = malloc(sizeof(double)*5);
}
```

Linker-visible. Allocated at fixed location

Visible within file. Allocated at fixed location.

Visible within func. Allocated at fixed location.

## C Storage Classes

```
#include <stdlib.h>

int global_static;
static int file_static;

void foo(int auto_param)
{
  static int func_static;
  int auto_i, auto_a[10];
  double *auto_d = malloc(sizeof(double)*5);
}
```

Space allocated on stack by caller.

Space allocated on stack by function.

Space allocated on heap by library routine.



## 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 automatic variables (stacked)**
- **More costly in time and space**
  - **malloc() and free() use complicated non-constant-time algorithms**
  - **Each block generally consumes two additional words of memory**
    - **Pointer to next empty block**
    - **Size of this block**
- **Common source of errors**
  - **Using uninitialized memory**
  - **Using freed memory**
  - **Not allocating enough**
  - **Neglecting to free disused blocks (memory leaks)**



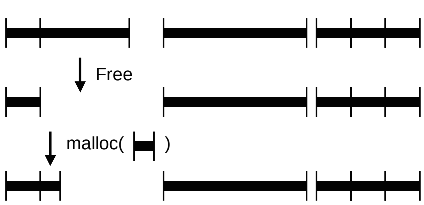
## malloc() and free()

- **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**



## Dynamic Storage Allocation

- **What are malloc() and free() actually doing?**
- **Pool of memory segments:**






## Dynamic Storage Allocation

- **Rules:**
  - **Each segment contiguous in memory (no holes)**
  - **Segments do not move once allocated**
- **malloc()**
  - **Find memory area large enough for segment**
  - **Mark that memory is allocated**
- **free()**
  - **Mark the segment as unallocated**

## Dynamic Storage Allocation

- **Three issues:**
- **How to maintain information about free memory**
- **The algorithm for locating a suitable block**
- **The algorithm for freeing an allocated block**

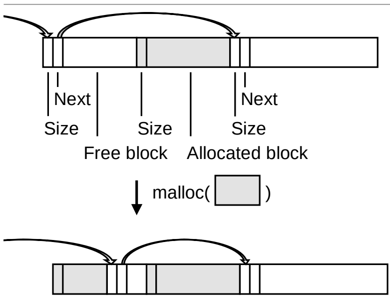


## Simple Dynamic Storage Allocation

- **Three issues:**
- **How to maintain information about free memory**
  - **Linked list**
- **The algorithm for locating a suitable block**
  - **First-fit**
- **The algorithm for freeing an allocated block**
  - **Coalesce adjacent free blocks**



## Simple Dynamic Storage Allocation




First large-enough free block selected

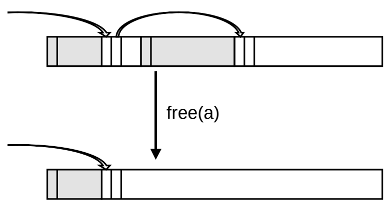
Free block divided into two

Previous next pointer updated

Newly-allocated region begins with a size value



## Simple Dynamic Storage Allocation




Appropriate position in free list identified

Newly-freed region added to adjacent free regions



## Dynamic Storage Allocation

- **Many, many variants**
- **Other “fit” algorithms**
- **Segregation of objects by sizes**
  - **8-byte objects in one region, 16 in another, etc.**
- **More intelligent list structures**



## Memory Pools

- **An alternative: Memory pools**
- **Separate management policy for each pool**
- **Stack-based pool: can only free whole pool at once**
  - **Very cheap operation**
  - **Good for build-once data structures (e.g., compilers)**
- **Pool for objects of a single size**
  - **Useful in object-oriented programs**
- **Not part of the C standard library**

## Arrays

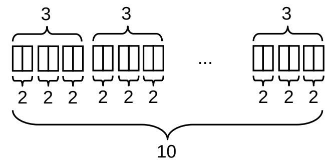
Filippo Brunelleschi, Ospdale degli Innocenti, Firenze, Italy, 1421

- **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**



## Multidimensional Arrays

- **Array declarations read right-to-left**
- **int a[10][3][2];**
- **“an array of ten arrays of three arrays of two ints”**
- **In memory**




Seagram Building, Ludwig Mies van der Rohe,1957



## Multidimensional Arrays

- **Passing a multidimensional array as an argument requires all but the first dimension**

```
int a[10][3][2];
void examine( a[][3][2] ) { … }
```

- **Address for an access such as** `a[i][j][k]` **is**

```
a + k + 2*(j + 3*i)
```

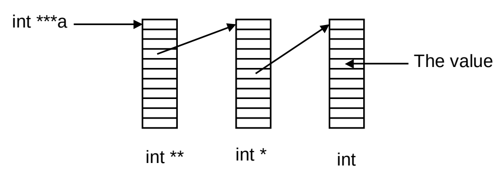


## Multidimensional Arrays

- **Use arrays of pointers for variable-sized multidimensional arrays**
- **You need to allocate space for and initialize the arrays of pointers**

```
int ***a;
```

- `a[3][5][4]` **expands to** `*(*(*(a+3)+5)+4)`






## C Expressions

- **Traditional mathematical expressions**

**y = a*x*x + b*x + c;**

- **Very rich set of expressions**
- **Able to deal with arithmetic and bit manipulation**



## C Expression Classes

- **arithmetic:** `+ - * / %`
- **comparison:** `== != < <= > >=`
- **bitwise logical:** `& | ^ ~`
- **shifting:** `<< >>`
- **lazy logical:** `&& || !`
- **conditional:** `? :`
- **assignment:** `= += -=`
- **increment/decrement:** `++ --`
- **sequencing:** `,`
- **pointer:** `* -> & []`

## Bitwise operators

- and: & or: | xor: ^ not: ~ left shift: << right shift: >>
- Useful for bit-field manipulations

```
#define MASK 0x040
if (a & MASK) { … }          /* Check bits */
c |= MASK;                   /* Set bits */
c &= ~MASK;                  /* Clear bits */
d = (a & MASK) >> 4;         /* Select field */
```



## 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) { … } } }
```

- Evaluation order (left before right) provides safety

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



## Conditional Operator

- c = a < b ? a + 1 : b – 1;
- Evaluate first expression. If true, evaluate second, otherwise evaluate third.
- Puts almost statement-like behavior in expressions.
- BCPL allowed code in an expression:

```
a := 5 + valof{ int i, s = 0; for (i = 0 ; i < 10 ; i++) s += a[I];
  return s; }
```



## Side-effects in expressions

- Evaluating an expression often has side-effects

| | |
|---|---|
| a++ | increment a afterwards |
| a = 5 | changes the value of a |
| a = foo() | function foo may have side-effects |



## Pointer Arithmetic

- From BCPL's view of the world
- Pointer arithmetic is natural: everything's an integer

`int *p, *q;`

`*(p+5)` equivalent to `p[5]`

- If p and q point into same array, `p – q` is number of elements between p and q.
- Accessing fields of a pointed-to structure has a shorthand:

`p->field` means `(*p).field`



## C Statements

- Expression
- Conditional
  - if (expr) { … } else {…}
  - switch (expr) { case c1: case c2: … }
- Iteration
  - while (expr) { … } — zero or more iterations
  - do … while (expr) — at least one iteration
  - for ( init ; valid ; next ) { … }
- Jump
  - goto label
  - continue; — go to start of loop
  - break; — exit loop or switch
  - return expr; — return from function

## The Switch Statement

- **Performs multi-way branches**

```
switch (expr) {
case 1: …
  break;
case 5:
case 6: …
  break;
default: …
  break;
}
```

```
tmp = expr;
if (tmp == 1) goto L1
else if (tmp == 5) goto L5
else if (tmp == 6) goto L6
else goto Default;
L1: …
  goto Break;
L5:;
L6: …
  goto Break;
Default: …
  goto Break;
Break:
```



## Switch Generates Interesting Code

- **Sparse case labels tested sequentially**

```
if (e == 1) goto L1;
else if (e == 10) goto L2;
else if (e == 100) goto L3;
```

- **Dense cases use a jump table**

```
table = { L1, L2, Default, L4, L5 };
if (e >= 1 and e <= 5) goto table[e];
```

- **Clever compilers may combine these**



## setjmp/longjmp

- **A way to exit from deeply nested functions**
- **A hack now a formal part of the standard library**

```
#include <setjmp.h>
jmp_buf jmpbuf;

void top(void) {
  switch (setjmp(jmpbuf)) {
  case 0: child(); break;
  case 1: /* longjmp called */ break;
} }

void deeplynested() { longjmp(jmpbuf, 1); }
```

Space for a return address and registers (including stack pointer, frame pointer)

Stores context, returns 0

Returns to context, making it appear setjmp() returned 1



## The Macro Preprocessor

- **Relatively late and awkward addition to the language**
- **Symbolic constants**
  `#define PI 3.1415926535`
- **Macros with arguments for emulating inlining**
  `#define min(x,y) ((x) < (y) ? (x) : (y))`
- **Conditional compilation**
  `#ifdef __STDC__`
- **File inclusion for sharing of declarations**
  `#include “myheaders.h”`



## Macro Preprocessor Pitfalls

- **Header file dependencies usually form a directed acyclic graph (DAG)**
- **How do you avoid defining things twice?**
- **Convention: surround each header (.h) file with a conditional:**

```
#ifndef __MYHEADER_H__
#define __MYHEADER_H__
/* Declarations */
#endif
```



## Macro Preprocessor Pitfalls

- **Macros with arguments do not have function call semantics**
- **Function Call:**
  - **Each argument evaluated once, in undefined order, before function is called**
- **Macro:**
  - **Each argument evaluated once every time it appears in expansion text**

## Macro Preprocessor pitfalls

- **Example: the "min" function**

```
int min(int a, int b)
 { if (a < b) return a; else return b; }
#define min(a,b) ((a) < (b) ? (a) : (b))
```

- **Identical for min(5,x)**
- **Different when evaluating expression has side-effect:**
  **min(a++,b)**
  - **min function increments a once**
  - **min macro may increment a twice if a < b**



## Macro Preprocessor Pitfalls

- **Text substitution can expose unexpected groupings**

```
#define mult(a,b) a*b
mult(5+3,2+4)
```

- **Expands to** `5 + 3 * 2 + 4`
- **Operator precedence evaluates this as**

**5 + (3*2) + 4 = 15 not (5+3) * (2+4) = 48 as intended**

- **Moral: By convention, enclose each macro argument in parenthesis:**

```
#define mult(a,b) (a)*(b)
```



## 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**
- **Word sizes**

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



## Nondeterminism in C

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



## Summary

- **C evolved from the typeless languages BCPL and B**
- **Array-of-bytes model of memory permeates the language**
- **Original weak type system strengthened over time**
- **C programs built from**
  - **Variable and type declarations**
  - **Functions**
  - **Statements**
  - **Expressions**

## Summary of C types

- Built from primitive types that match processor types
- char, int, float, double, pointers
- Struct and union aggregate heterogeneous objects
- Arrays build sequences of identical objects
- Alignment restrictions ensured by compiler
- Multidimensional arrays
- Three storage classes
  - global, static (address fixed at compile time)
  - automatic (on stack)
  - heap (provided by malloc() and free() library calls)



## Summary of C expressions

- Wide variety of operators
  - Arithmetic + - * /
  - Logical && || (lazy)
  - Bitwise & |
  - Comparison < <=
  - Assignment = += *=
  - Increment/decrement ++ --
  - Conditional ? :
- Expressions may have side-effects



## Summary of C statements

- Expression
- Conditional
  - if-else switch
- Iteration
  - while do-while for(;;)
- Branching
  - goto break continue return
- Awkward setjmp, longjmp library routines for non-local goto



## Summary of C

- Preprocessor
  - symbolic constants
  - inline-like functions
  - conditional compilation
  - file inclusion
- Sources of nondeterminsm
  - library functions, evaluation order, variable sizes



## The Main Points

- Like a high-level assembly language
- Array-of-cells model of memory
- Very efficient code generation follows from close semantic match
- Language lets you do just about everything
- Very easy to make mistakes