C, C++, and Assembly

Languages for Embedded Systems

Prof. Stephen A. Edwards
Summer 2004
NCTU, Taiwan

What are Embedded Systems?

Computers masquerading as non-computers.

Casio Camera Watch

Nokia 7110 Browser Phone

Sony Playstation 2

Philips DVD Player

Philips TiVo Recorder

Embedded System Challenges

Differs from general-purpose computing:

Real-time Constraints

Power Constraints

Exotic Hardware

Concurrency

Control-dominated systems

Signal-processing

User Interfaces

Laws of Physics

The Role of Languages

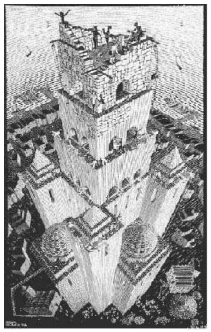
Language shapes how you solve a problem.

Java, C, C++ and their ilk designed for general-purpose systems programming.

Do not address timing, concurrency.

Domain-specific languages much more concise.

Problem must fit the language.



Syllabus

Software languages: Assembly, C, and C++

Concurrency in Java and Real-Time Operating Systems

Dataflow Languages (SDF)

Hardware Languages (Verilog)

SystemC

Syntax, Semantics, and Model

Marionette Model

You have control through the syntax of the language

The semantics of the language connect the syntax to the model

You ultimately affect a model

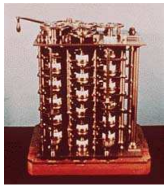
Syntax

Formally:

Language: infinite set of strings from an alphabet

Language	Alphabet
DNA	A T G C
Student Transcripts	w1007-02 w1009-01 w4995-02
English	aardvard abacus abalone . . .
Verilog	always module . . .

Computation Model



What the string ultimately affects

A language may have more than one

Language	Model
DNA	Proteins suspended in water
Student Transcripts	Your knowledge
	The admiration of others
English	Natural Language Understanding
Verilog	Discrete Event Simulator
	Netlist of gates and flip-flops

Semantics

How to interpret strings in the model

Also not necessarily unique

Language	Semantics
DNA	⟦AGA ⟧= Arginine
	⟦TAG ⟧= STOP
Student Transcripts	⟦w1007-02 ⟧= Java
English	⟦Look out! ⟧= Somebody's warning me
Verilog	⟦always @posedge clk ⟧= Flip-flop

Defining Syntax

Generally done with a grammar

Recursively-defined rules for constructing valid sentences

"Backus-Naur Form"

```
expr ::
      literal
   || expr + expr
   || expr * expr
```

Not a focus of this class: I'm assuming you've had a compilers class.

Operational Semantics

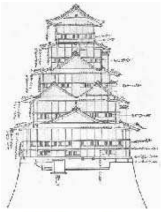
Describes the effect a program has on an abstract machine

Typical instruction observes and then advances machine state

Close to implementation, fairly easy to use to create the "obvious" implementation

Often includes too many details, can be hard to show that a particular implementation conforms

Specification and Modeling



How do you want to use the program?

Specification langauges say "build this please."

Modeling languages allow you to describe something that does or will exist

Distinction a function of the model and the language's semantics

Specification Versus Modeling

C is a specification language

- Semantics very operational
- Clear how the language is to be translated into assembly language

Verilog is a modeling language

- Semantics suggestive of a simulation procedure
- Good for building a model that captures digital hardware behavior (delays, unknown values)
- Not as good for specification: how do you build something with a specific delay?

Concurrency

Photo by Thomas Danoghue

Why bother?

Harder model to program

Real world is concurrent

Good architecture: one concurrently-running process controls each independent system component

E.g., process for the right brake, process for the left brake, process for a brake pedal

Approaches to Concurrency

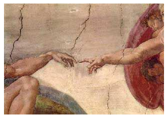
Shared memory / Every man for himself

- Adopted by Java, other software languages
- Everything's shared, nothing synchronized by default
- Synchronization through locks/monitors/semaphores
- Most flexible, easy to get wrong

Synchronous

- Global clock regulates passage of time
- Robust in the presence of timing uncertainty
- Good for hardware; but has synchronization overhead

Communication and Concurrency



Idea: Let processes run asynchronously
Only force them to synchronize when they communicate

C. A. R. Hoare's Communicating Sequential Processes

- Rendezvous-style communication
- Processes that wish to communicate both wait until the other is ready to send/receive

Kahn Process Networks (later in the course)

- Communicate through channels
- Reader waits for data; writer never waits

Nondeterminism

Does a program mean exactly one thing?

Example from C:

```
a = 0;
printf("%d %d %d", ++a, ++a, ++a);
```

Argument evaluation order is undefined

Program behavior subject to the whim of the compiler

Are you sure your program does what you think?

Nondeterministic is not Random

Deterministic: 1 + 1 = 2 *always*

Random: 1 + 1 = 2 *50% of the time*, 3 *otherwise*

Nondeterministic: 1 + 1 = 2 or 3, but I'm not telling

Nondeterministic behavior can look deterministic, random, or something worse.

Murphy's law of nondeterminism: Something nondeterministic will choose the worst possible outcome at the worst possible time.

Nondeterminism is Awful

Much harder to be sure your specification or model is correct

True nondeterminstic language difficult to simulate

Should produce "any of these results"

Must maintain all possible outcomes, which grows exponentially

Idiosyncrasies of a particular implementation of a nondeterministic language often become the de facto standard

Example from Verilog

Concurrent procedure execution order undefined

```
always @(posedge clk) $write( a )
always @(posedge clk) $write( b )
```

First simulator moved procedures between two push-down stacks, producing

a b b a a b b a a b b a a b a

Later simulators had to match this now-expected behavior.

Nondeterminism is Great

True nondeterministic specification often exponentially smaller than deterministic counterpart

Implicit "all possible states" representation

E.g., nondeterministic finite automata for matching regular expressions

If system itself is truly nondeterministic, shouldn't its model also be?

Can be used to expose design errors

More flexible: only there if you want to use it

Correctness remains more elusive

Communication

Memory

- Value written to location
- Value stays until written again
- Value can be read many times
- No synchronization

FIFO Buffer

- Value written to buffer
- Value held until read
- Values read in written order

Communication

Wires

- May or may not have explicit write operation
- Value immediately seen by all readers
- More like a system of equations than a sequence of operations

Hierarchy

Most languages can create pieces and assemble them

Advantage: Information hiding

- User does not know details of a piece
- Easier to change implementation of piece without breaking whole system
- Easier to get small piece right
- Facilitates abstraction: easier to understand the whole

Advantage: Reuse

- Pieces less specific; can be used again

E.g., Functions in C, Classes in Java, Modules in Verilog

Assembly Language

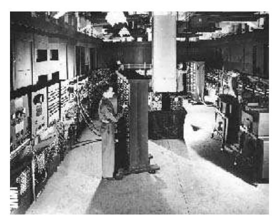
Assembly Languages

One step up from machine language

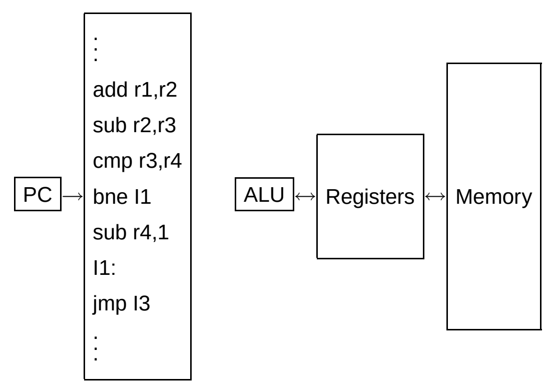
Originally a more user-friendly way to program

Now mostly a compiler target

Model of computation: stored program computer



Assembly Language Model



Assembly Language Instructions

Built from two pieces:

```
add R1, R3, 3
```

Opcode — What to do with the data

Operands — Where to get the data

Types of Opcodes

Arithmetic, logical

- add, sub, mult
- and, or
- Cmp

Memory load/store

- ld, st

Control transfer

- jmp
- bne

Complex

- movs

Operands

Each operand taken from a particular addressing mode:

Examples:

Register	add r1, r2, r3
Immediate	add r1, r2, 10
Indirect	mov r1, (r2)
Offset	mov r1, 10(r3)
PC Relative	beq 100

Reflect processor data pathways

Types of Assembly Languages

Assembly language closely tied to processor architecture

At least four main types:

CISC: Complex Instruction-Set Computer

RISC: Reduced Instruction-Set Computer

DSP: Digital Signal Processor

VLIW: Very Long Instruction Word

CISC Assembly Language

Developed when people wrote assembly language

Complicated, often specialized instructions with many effects

Examples from x86 architecture

- String move
- Procedure enter, leave

Many, complicated addressing modes

So complicated, often executed by a little program (microcode)

Examples: Intel x86, 68000, PDP-11

RISC Assembly Language

Response to growing use of compilers

Easier-to-target, uniform instruction sets

“Make the most common operations as fast as possible”

Load-store architecture:

- Arithmetic only performed on registers
- Memory load/store instructions for memory-register transfers

Designed to be pipelined

Examples: SPARC, MIPS, HP-PA, PowerPC

DSP Assembly Language

Digital signal processors designed specifically for signal processing algorithms

Lots of regular arithmetic on vectors

Often written by hand

Irregular architectures to save power, area

Substantial instruction-level parallelism

Examples: TI 320, Motorola 56000, Analog Devices

VLIW Assembly Language

Response to growing desire for instruction-level parallelism

Using more transistors cheaper than running them faster

Many parallel ALUs

Objective: keep them all busy all the time

Heavily pipelined

More regular instruction set

Very difficult to program by hand

Looks like parallel RISC instructions

Examples: Itanium, TI 320C6000

Example: Euclid’s Algorithm

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

i386 Programmer's Model

31 0	
eax	Mostly
ebx	General-
ecx	Purpose-
edx	Registers
esi	Source index
edi	Destination index
ebp	Base pointer
esp	Stack pointer
eflags	Status word
eip	Instruction Pointer

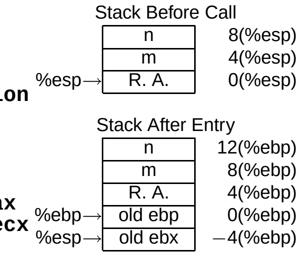
15 0	
cs	Code segment
ds	Data segment
ss	Stack segment
es	Extra segment
fs	Data segment
gs	Data segment

Euclid on the i386

```
 .file "euclid.c"          # Boilerplate
 .version "01.01"
gcc2_compiled.:
 .text                     # Executable
 .align 4                  # Start on 16-byte boundary
 .globl gcd                # Make "gcd" linker-visible
 .type gcd,@function
gcd:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx
   movl 8(%ebp),%eax
   movl 12(%ebp),%ecx
   jmp .L6
.p2align 4,,7
```

Euclid on the i386

```
 .file "euclid.c"
 .version "01.01"
gcc2_compiled.:
 .text
 .align 4
 .globl gcd
 .type gcd,@function
gcd:
   pushl %ebp
   movl %esp,%ebp
   pushl %ebx
   movl 8(%ebp),%eax
   movl 12(%ebp),%ecx
   jmp .L6
.p2align 4,,7
```



Euclid in the i386

```
  jmp .L6             # Jump to local label .L6
.p2align 4,,7         # Skip ≤ 7 bytes to a multiple of 16
.L4:
  movl %ecx,%eax
  movl %ebx,%ecx
.L6:
  cltd                # Sign-extend eax to edx:eax
  idivl %ecx          # Compute edx:eax / ecx
  movl %edx,%ebx
  testl %edx,%edx
  jne .L4
  movl %ecx,%eax
  movl -4(%ebp),%ebx
  leave
  ret
```

Euclid on the i386

```
  jmp .L6
.p2align 4,,7
.L4:
  movl %ecx,%eax      # m = n
  movl %ebx,%ecx      # n = r
.L6:
  cltd
  idivl %ecx
  movl %edx,%ebx
  testl %edx,%edx     # AND of edx and edx
  jne .L4             # branch if edx was ≠ 0
  movl %ecx,%eax      # Return n
  movl -4(%ebp),%ebx
  leave               # Move ebp to esp, pop ebp
  ret                 # Pop return address and branch
```

SPARC Programmer's Model

31 0	
r0	Always 0
r1	Global Registers
⋮	
r7	
r8/o0	Output Registers
⋮	
r14/o6	Stack Pointer
r15/o7	
r16/l0	Local Registers
⋮	
r23/l7	

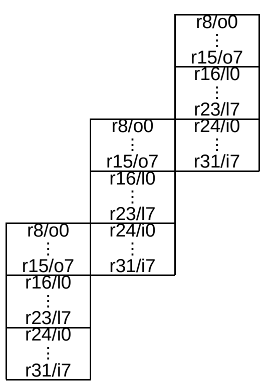
31 0	
r24/i0	Input Registers
⋮	
r30/i6	Frame Pointer
r31/i7	Return Address
PSW	Status Word
PC	Program Counter
nPC	Next PC

SPARC Register Windows

The output registers of the calling procedure become the inputs to the called procedure

The global registers remain unchanged

The local registers are not visible across procedures



Euclid on the SPARC

```
 .file    "euclid.c"        # Boilerplate
gcc2_compiled.:
 .global .rem               # make .rem linker-visible
 .section ".text"           # Executable code
 .align 4
 .global gcd                # make gcd linker-visible
 .type gcd, #function
 .proc    04
gcd:
  save %sp, -112, %sp       # Next window, move SP

  mov  %i0, %o1             # Move m into o1
  b    .LL3                 # Unconditional branch
  mov  %i1, %i0             # Move n into i0
```

Euclid on the SPARC

```
   mov   %i0, %o1
   b     .LL3
   mov   %i1, %i0
.LL5:
   mov   %o0, %i0    # n = r
.LL3:
   mov   %o1, %o0    # Compute the remainder of
   call  .rem, 0     # m / n, result in o0
   mov   %i0, %o1

   cmp   %o0, 0
   bne   .LL5
   mov   %i0, %o1    # m = n (always executed)
   ret               # Return (actually jmp i7 + 8)
   restore           # Restore previous window
```

Digital Signal Processor Apps.

Low-cost embedded systems

- Modems, cellular telephones, disk drives, printers

High-throughput applications

- Halftoning, base stations, 3-D sonar, tomography

PC based multimedia

- Compression/decompression of audio, graphics, video

Embedded Processor Requirements

Inexpensive with small area and volume

Deterministic interrupt service routine latency

Low power: ≈50 mW (TMS320C54x uses 0.36 μA/MIPS)

Conventional DSP Architecture

Harvard architecture

- Separate data memory/bus and program memory/bus
- Three reads and one or two writes per instruction cycle

Deterministic interrupt service routine latency

Multiply-accumulate in single instruction cycle

Special addressing modes supported in hardware

- Modulo addressing for circular buffers for FIR fi lters
- Bit-reversed addressing for fast Fourier transforms

Instructions to keep the pipeline (3-4 stages) full

- Zero-overhead looping (one pipeline flush to set up)
- Delayed branches

Conventional DSPs

	Fixed-Point	Floating-Point
Cost/Unit	\$5–\$79	\$5–\$381
Architecture	Accumulator	load-store
Registers	2–4 data, 8 address	8–16 data, 8–16 address
Data Words	16 or 24 bit	32 bit
Chip Memory	2–64K data+program	8–64K data+program
Address Space	16–128K data	16M–4G data
	16–64K program	16M–4G program
Compilers	Bad C	Better C, C++
Examples	TI TMS320C5x	TI TMS320C3x
	Motorola 56000	Analog Devices SHARC

Conventional DSPs

Market share: 95% fixed-point, 5% floating-point

Each processor comes in dozens of configurations

- Data and program memory size
- Peripherals: A/D, D/A, serial, parallel ports, timers

Drawbacks

- No byte addressing (needed for image and video)
- Limited on-chip memory
- Limited addressable memory on most fixed-point DSPs
- Non-standard C extensions to support fixed-point data

Example

Finite Impulse Response filter (FIR)

Can be used for lowpass, highpass, bandpass, etc.

Basic DSP operation

For each sample, computes

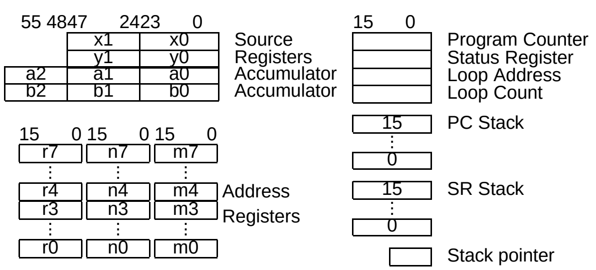
$$y_n = \sum_{i=0}^{k} a_i x_{n+i}$$

where

$a_0, \ldots, a_k$ are filter coeffecients,

x_n is the nth input sample, y_n is the nth output sample.

56000 Programmer's Model



56001 Memory Spaces

Three memory regions, each 64K:

- 24-bit Program memory
- 24-bit X data memory
- 24-bit Y data memory

Idea: enable simultaneous access of program, sample, and coefficient memory

Three on-chip memory spaces can be used this way

One off-chip memory pathway connected to all three memory spaces

Only one off-chip access per cycle maximum

56001 Address Generation

Addresses come from pointer register r0 . . . r7

Offset registers n0 . . . n7 can be added to pointer

Modifier registers cause the address to wrap around

Zero modifier causes reverse-carry arithmetic

Address	Notation	Next value of r0
r0	(r0)	r0
r0 + n0	(r0+n0)	r0
r0	(r0)+	(r0 + 1) mod m0
r0 - 1	-(r0)	r0 - 1 mod m0
r0	(r0)-	(r0 - 1) mod m0
r0	(r0)+n0	(r0 + n0) mod m0
r0	(r0)-n0	(r0 - n0) mod m0

FIR Filter in 56001

```
n       equ 20     # Define symbolic constants
start   equ $40
samples equ $0
coeffs  equ $0
input   equ $ffe0  # Memory-mapped I/O
output  equ $ffe1

        org p:start  # Locate in prog. memory
        move #samples, r0  # Pointers to samples
        move #coeffs, r4   # and coefficients
        move #n-1, m0    # Prepare circular buffer
        move m0, m4
```

FIR Filter in 56001

```
movep y:input, x:(r0)  # Load sample into memory
      # Clear accumulator A
                # Load a sample into x0
                           # Load a coefficient
clr   a         x:(r0)+, x0  y:(r4)+, y0

rep   #n-1      # Repeat next instruction n-1 times
      # a = x0 × y0
                # Next sample
                           # Next coefficient
mac   x0,y0,a   x:(r0)+, x0  y:(r4)+, y0

macr  x0,y0,a   (r0)-
movep a, y:output  # Write output sample
```

TI TMS320C6000 VLIW DSP

Eight instruction units dispatched by one very long instruction word

Designed for DSP applications

Orthogonal instruction set

Big, uniform register file (16 32-bit registers)

Better compiler target than 56001

Deeply pipelined (up to 15 levels)

Complicated, but more regular, datapath

Pipelining on the C6

One instruction issued per clock cycle

Very deep pipeline

- 4 fetch cycles
- 2 decode cycles
- 1-10 execute cycles

Branch in pipeline disables interrupts

Conditional instructions avoid branch-induced stalls

No hardware to protect against hazards

- Assembler or compiler's responsibility

FIR in One 'C6 Assembly Instruction

Load a halfword (16 bits)

Do this on unit D1

```
FIRLOOP:
        LDH .D1   *A1++, A2   ; Fetch next sample
||      LDH .D2   *B1++, B2   ; Fetch next coeff.
|| [B0] SUB .L2   B0, 1, B0   ; Decrement count
|| [B0] B   .S2   FIRLOOP     ; Branch if non-zero
||      MPY .M1X  A2, B2, A3  ; Sample × Coeff.
||      ADD .L1   A4, A3, A4  ; Accumulate result
```

Use the cross path

Predicated instruction (only if B0 non-zero)

Run these instruction in parallel

Peripherals

Peripherals

Often the whole point of the system

Memory-mapped I/O

- Magical memory locations that make something happen or change on their own

Typical meanings:

- Configuration (write)
- Status (read)
- Address/Data (access more peripheral state)

Example: 56001 Port C

Nine pins each usable as either simple parallel I/O or as part of two serial interfaces.

Pins:

Parallel	Serial	
PC0	RxD	Serial Communication Interface (SCI)
PC1	TxD	
PC2	SCLK	
PC3	SC0	Synchronous Serial Interface (SSI)
PC4	SC1	
PC5	SC2	
PC6	SCK	
PC7	SRD	
PC8	STD	

Port C Registers for Parallel Port

Port C Control Register

Selects mode (parallel or serial) of each pin

X: $FFE1 Lower 9 bits: 0 = parallel, 1 = serial

Port C Data Direction Register

I/O direction of parallel pins

X: $FFE3 Lower 9 bits: 0 = input, 1 = output

Port C Data Register

Read = parallel input data, Write = parallel data out

X: $FFE5 Lower 9 bits

Port C SCI

Three-pin interface

422 Kbit/s NRZ asynchronous interface (RS-232-like)

3.375 Mbit/s synchronous serial mode

Multidrop mode for multiprocessor systems

Two Wakeup modes

- Idle line
- Address bit

Wired-OR mode

On-chip or external baud rate generator

Four interrupt priority levels

Port C SCI Registers

SCI Control Register

X: $FFF0	**Bits**	**Function**
	0–2	Word select bits
	3	Shift direction
	4	Send break
	5	Wakeup mode select
	6	Receiver wakeup enable
	7	Wired-OR mode select
	8	Receiver enable
	9	Transmitter enable
	10	Idle line interrupt enable
	11	Receive interrupt enable
	12	Transmit interrupt enable
	13	Timer interrupt enable
	15	Clock polarity

Port C SCI Registers

SCI Status Register (Read only)

X: $FFF1	**Bits**	**Function**
	0	Transmitter Empty
	1	Transmitter Reg Empty
	2	Receive Data Full
	3	Idle Line
	4	Overrun Error
	5	Parity Error
	6	Framing Error
	7	Received bit 8

Port C SCI Registers

SCI Clock Control Register

X: $FFF2	**Bits**	**Function**
	11–0	Clock Divider
	12	Clock Output Divider
	13	Clock Prescaler
	14	Receive Clock Source
	15	Transmit Clock Source

Port C SSI

Intended for synchronous, constant-rate protocols

Easy interface to serial ADCs and DACs

Many more operating modes than SCI

Six Pins (Rx, Tx, Clk, Rx Clk, Frame Sync, Tx Clk)

8, 12, 16, or 24-bit words

Port C SSI Registers

SSI Control Register A $FFEC

Prescaler, frame rate, word length

SSI Control Register B $FFED

Interrupt enables, various mode settings

SSI Status/Time Slot Register $FFEE

Sync, empty, oerrun

SSI Receive/Transmit Data Register $FFEF

8, 16, or 24 bits of read/write data.

The C Language

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

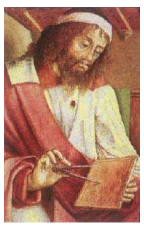
PDP-11 was byte-addressed

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

Euclid's Algorithm in C

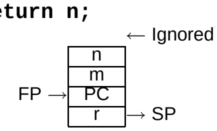
```
int gcd(int m, int n )
{
  int r;
  while ((r = m % n) != 0) {
    m = n;
    n = r;
  }
  return n;
}
```

← "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

Allocated on stack when function entered, released on return

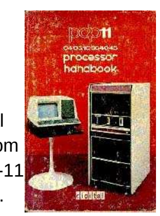
Parameters & automatic variables accessed via frame pointer

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:

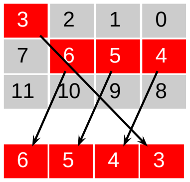
3	2	1	0
7	6	5	4
11	10	9	8

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 */
};
```

x	x	x	x
y	y		z
			w

(grey) = Added padding

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;
}
```

Dynamic Memory Allocation

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

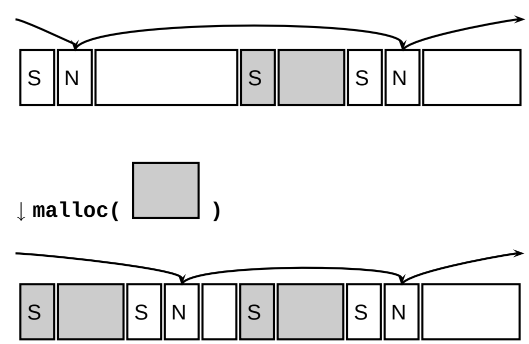
The algorithm for locating a suitable block

Simplest: First-fit

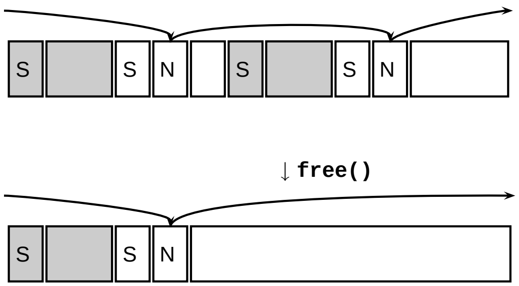
The algorithm for freeing an allocated block

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:

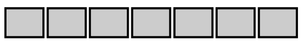
Fit, allocation, etc. much faster

Good for object-oriented programs

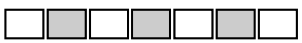
On unix, implemented on top of **`sbrk()`** system call (requests additional memory from OS).

Fragmentation

`malloc(` ▭ **`)`** seven times give



`free()` four times gives



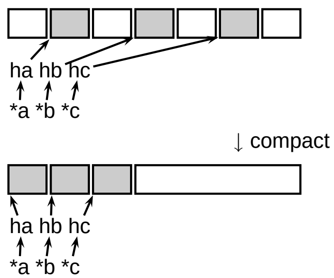
`malloc(` ▭ **`)`** ?

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.

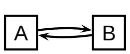
Automatic Garbage Collection

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) {



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 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

The C++ Language

The C++ Language

Bjarne Stroupstrup, the language's creator, explains

C++ was designed to provide Simula's facilities for program organization together with C's efficiency and flexibility for systems programming.

C++ Features

Classes
  User-defined types

Operator overloading
  Attach different meaning to expressions such as a + b

References
  Pass-by-reference function arguments

Virtual Functions
  Dispatched depending on type at run time

Templates
  Macro-like polymorphism for containers (e.g., arrays)

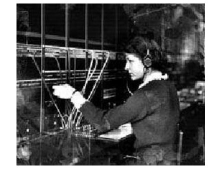
Exceptions
  More elegant error handling

Implementing Classes

Simple without virtual functions.

```
C++                    Equivalent C
class Stack {          struct Stack {
  char s[SIZE];          char s[SIZE];
  int sp;                int sp;
public:                };
  Stack();
  void push(char);     void St_Stack(Stack*);
  char pop();          void St_push(Stack*,char);
};                     char St_pop(Stack*);
```

Operator Overloading



For manipulating user-defined "numeric" types

```
complex c1(1, 5.3), c2(5); // Create objects
complex c3 = c1 + c2;  // + means complex plus
c3 = c3 + 2.3;  // 2.3 promoted to a complex number
```

Complex Number Type

```
class Complex {
  double re, im;
public:
  complex(double);  // used, e.g., in c1 + 2.3
  complex(double, double);

  // Here, & means pass-by-reference: reduces copying
  complex& operator+=(const complex&);
};
```

References

Designed to avoid copying in overloaded operators

Especially efficient when code is inlined.

A mechanism for calling functions pass-by-reference

C only has pass-by-value: fakable with explicit pointer use

```
void bad_swap(int x, int y) {
  int tmp = x; x = y; y = tmp;
}

void swap(int &x, int &y) {
  int tmp = x; x = y; y = tmp;
}
```

Function Overloading

Overloaded operators a particular case of function/method overloading

General: select specific method/operator based on name, number, and type of arguments.

Return type not part of overloading

```
void foo(int);
void foo(int, int);  // OK
void foo(char *);    // OK
int  foo(char *);    // BAD
```

Const

Access control over variables, arguments, and objects.

```
const double pi = 3.14159265; // Compile-time constant

int foo(const char* a) { // Constant argument
  *a = 'a';              // Illegal: a is const
}

class bar {
  // "object not modifi ed"
  int get_field() const { return field; }
};
```

Templates

Macro-preprocessor-like way of providing polymorphism.

Polymorphism: Using the same code for different types

Mostly intended for containiner classes (vectors of integers, doubles, etc.)

Standard Template Library has templates for strings, lists, vectors, hash tables, trees, etc.

Template Stack Class

```
template <class T> class Stack {
  T s[SIZE]; // T is a type argument
  int sp;
public:
  Stack() { sp = 0; }
  void push(T v) {
    if (sp == SIZE) error("overflow");
    s[sp++] = v;
  }
  T pop() {
    if (sp == () error("underflow");
    return s[--sp];
  }
};
```

Using a Template

```
Stack<char> cs;  // Creates code specialized for char
cs.push('a');
char c = cs.pop();

Stack<double*> dps;  // Creates version for double*
double d;
dps.push(&d);
```

Implementing C++

Implementing Inheritance

Simple: Add new fields to end of the object

Fields in base class always at same offset in derived class

Consequence: Derived classes can never remove fields

C++

```
class Shape {
  double x, y;
};

class Box : Shape {
  double h, w;
};
```

Equivalent C

```
struct Shape {
  double x, y;
};

struct Box {
  double x, y;
  double h, w;
};
```

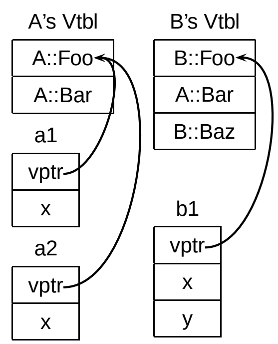
Virtual Functions

```
class Shape {
  virtual void draw(); // Invoked by object's class
};                     // not its compile-time type.
class Line : public Shape {
  void draw();
};
class Arc : public Shape {
  void draw();
};

Shape *s[10];
s[0] = new Line;
s[1] = new Arc;
s[0]->draw();     // Invoke Line::draw()
s[1]->draw();     // Invoke Arc::draw()
```

Virtual Functions

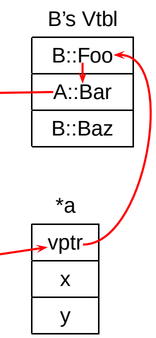
The Trick: Add a "virtual table" pointer to each object.

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};

A a1, a2; B b1;
```



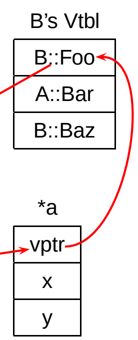
Virtual Functions

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar()
    { do_something(); }
};
struct B : A {
  int y;
  virtual void Foo();
  virtual void Baz();
};
A *a = new B;
a->Bar();
```



Virtual Functions

```
struct A {
  int x;
  virtual void Foo();
  virtual void Bar();
};
struct B : A {
  int y;
  virtual void Foo()
    { something_else(); }
  virtual void Baz();
};
A *a = new B;
a->Foo();
```



Multiple Inheritance

Rocket Science,
and nearly as dangerous

Inherit from two or more classes

```
class Window { ... };

class Border { ... };

class BWindow : public Window,
                public Border {
    :
};
```

Multiple Inheritance Ambiguities

```
class Window {
  void draw();
};

class Border {
  void draw();     // OK
};

class BWindow : public Window,
                public Border { };

BWindow bw;
bw.draw();     // Compile-time error: ambiguous
```

Resolving Ambiguities Explicitly

```
class Window { void draw(); };

class Border { void draw(); };

class BWindow : public Window,
                public Border {
  void draw() { Window::draw(); }
};

BWindow bw;
bw.draw();    // OK
```

Duplicate Base Classes

A class may be inherited more than once

```
class Drawable { ... };
class Window : public Drawable { ... };
class Border : public Drawable { ... };
class BWindow : public Window, public
Border { ... };
```

BWindow gets two copies of the Drawable base class.

Virtual Base Classes

Virtual base classes are inherited at most once

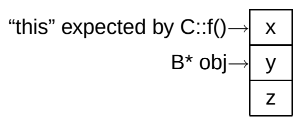
```
class Drawable { ... };
class Window : public virtual Drawable {
... };
class Border : public virtual Drawable {
... };
class BWindow : public Window, public
Border { ... };
```

BWindow gets two copies of the Drawable base class

Implementing Multiple Inheritance

A virtual function expects a pointer to its object

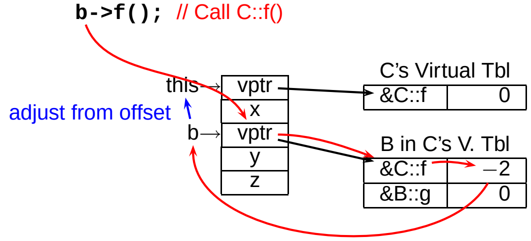
```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f(); }
struct C : A, B { int z; void f(); }

B *obj = new C;
b->f(); // Calls C::f()
```



“obj” is, by definition, a pointer to a B, not a C. Pointer must be adjusted depending on the actual type of the object. At least two ways to do this.

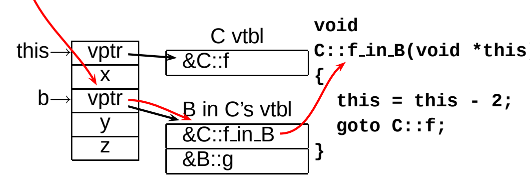
Implementation using Offsets

```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f();
                  virtual void g(); }
struct C : A, B { int z; void f(); }
B *b = new C;
b->f(); // Call C::f()
```



Implementation using Thunks

```
struct A { int x; virtual void f(); }
struct B { int y; virtual void f();
                  virtual void g(); }
struct C : A, B { int z; void f(); }
B *b = new C;
b->f(); // Call C::f()
```



Offsets vs. Thunks

Offsets	Thunks
Offsets to virtual tables	Helper functions
Can be implemented in C	Needs “extra” semantics
All virtual functions cost more	Only multiply-inherited functions cost
Tricky	Very Tricky

Exceptions

A high-level replacement for C’s setjmp/longjmp.

```
struct Except { };

void baz() { throw Except; }
void bar() { baz(); }

void foo() {
  try {
    bar();
  } catch (Except e) {
    printf("oops");
  }
}
```

One Way to Implement Exceptions

```
try {                push(Ex, Handler);

  throw Ex;          throw(Ex);
                     pop();
                     goto Exit;
} catch (Ex e) {   Handler:
  foo();             foo();
}                  Exit:
```

`push()` adds a handler to a stack

`pop()` removes a handler

`throw()` finds first matching handler

Problem: imposes overhead even with no exceptions

Implementing Exceptions Cleverly

Real question is the nearest handler for a given PC.

```
 1 void foo() {
 2
 3   try {
 4     bar();
 5   } catch (Ex1 e) { H1: a(); }
 6
 7 }
 8 void bar() {
 9
10   try {
11     throw Ex1();
12   } catch (Ex2 e) { H2: b(); }
13
14 }
```

Lines	Action
1–2	Reraise
3–5	H1
6–9	Reraise
10–12	H2
13–14	Reraise

1. look in table
2. H2 doesn't handle Ex1, reraise
3. look in table

The C++ Standard Template Library

Standard Template Library

I/O Facilities
- iostream, fstream

Garbage-collected String class

Containers
- vector, list, queue, stack, map, set

Numerical
- complex, valarray

General algorithms
- search, sort

C++ I/O

C's printing facility is clever but not type safe.

```
char *s; int d; double g;
printf("%s %d %g", s, d, g);
```

Hard for compiler to typecheck argument types against format string.

C++ overloads the **<<** and **>>** operators. This is type safe.

```
cout << 's' << ' ' << d << ' ' << g;
```

C++ I/O

Easily extended to print user-defined types

```
ostream &
operator <<(ostream &o, MyType &m) {
  o << "An Object of MyType";
  return o;
}
```

Input overloads the **>>** operator

```
int read_integer;
cin >> read_integer;
```

C++ String Class

Provides automatic garbage collection, usually by reference counting.

```
string s1, s2;
s1 = "Hello";
s2 = "There";
s1 += " goodbye";
s1 = ""; // Frees memory holding "Hello goodbye"
```

C++ STL Containers

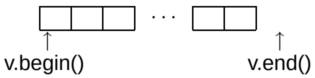
Vector: dynamically growing and shrinking array of elements.

```
vector<int> v;
v.push_back(3);  // vector can behave as a stack
v.push_back(2);
int j = v[0];    // operator[] defined for vector
```

Iterators

Mechanism for stepping through containers

```
vector<int> v;
for ( vector<int>::iterator i = v.begin();
      i != v.end() ; i++ ) {
  int entry = *i;
}
```



Associative Containers

Keys must be totally ordered

Implemented with trees—O(log n)

Set of objects

```
set<int, less<int> > s;
s.insert(5);
set<int, less<int> >::iterator i =
s.find(3);
```

Map: Associative array

```
map<int, char*> m;
m[3] = "example";
```

C++ In Embedded Systems

C++ In Embedded Systems

Dangers of using C++:

No or bad compiler for your particular processor

Increased code size

Slower program execution

Much harder language to compile

Unoptimized C++ code can be larger & slower than equivalent C

C++ Features With No Impact

Classes

- Fancy way to describe functions and structs
- Equivalent to writing object-oriented C code

Single inheritance

- More compact way to write larger structures

Function name overloading

- Completely resolved at compile time

Namespaces

- Completely resolved at compile time

Inexpensive C++ Features

Default arguments

- Compiler adds code at call site to set default arguments
- Long argument lists costly in C and C++ anyway

Constructors and destructors

- Function call overhead when an object comes into scope (normal case)
- Extra code inserted when object comes into scope (inlined case)

Medium-cost Features

Virtual functions

- Extra level of indirection for each virtual function call
- Each object contains an extra pointer

References

- Often implemented with pointers
- Extra level of indirection in accessing data
- Can disappear with inline functions

Inline functions

- Can greatly increase code size for large functions
- Usually speeds execution

High-cost Features

Multiple inheritance

- Makes objects much larger (multiple virtual pointers)
- Virtual tables larger, more complicated
- Calling virtual functions even slower

Templates

- Compiler generates separate code for each copy
- Can greatly increase code sizes
- No performance penalty

High-cost Features

Exceptions

- Typical implementation:
- When exception is thrown, look up stack until handler is found and destroy automatic objects on the way
- Mere presence of exceptions does not slow program
- Often requires extra tables or code to direct clean-up
- Throwing and exception often very slow

High-cost Features

Much of the standard template library

- Uses templates: often generates lots of code
- Very dynamic data structures have high memory-management overhead
- Easy to inadvertently copy large data structures

The bottom line

C still generates better code

Easy to generate larger C++ executables

Harder to generate slower C++ executables

Exceptions most worrisome feature

- Consumes space without you asking
- GCC compiler has a flag to enable/disable exception support **`-fexceptions`** and **`-fno-exceptions`**