15April, 2002

Important announcements

- Your Federal and State income taxes are due tonight
- No labs this week to due "April Advance"

C programming language

- Designed in 1972 as a high-level language for Unix operating system
  - Until then operating systems were usually written in assembler
- High-level language that supports low-level interfaces
  - Memory mapped I/O example
- Based on BCPL
- ANSI C standard in 1989
  - Addition of prototypes
- C++ and Java are its extensions

Compilation processor

- Preprocessor
  - Processes `#define`, `#include`, `#ifdef`
- Compiler
  - Generates machine code in an *object file*
- Linker
  - Joins compiled code and libraries into an *executable file*
- Loader (run-time linker)
  - Puts executable in memory with references to *shared* or *dynamic libraries*

C++ vs. C

- Almost every C program is a legal C++ program
- C does not have
  - classes
  - templates
  - reference parameters
  - `cin` and `cout` and <<= and +>> for I/O
- C comments are `/* your comments here */`

Java vs. C/C++

C and C++ have pointers

- Java hides these in object references
- C++ does not have large collections of standard "useful" classes
  - Although there is a standard template library
- C and C++ I/O is much simpler
- C does not have a Boolean type
- Microsoft likes C++

The main routine

Entry point for C programs

```
main( int argc, char *argv[], char *envp[])
```

- `argc` Number of command line arguments
- `argv` Command line arguments
- `envp` *Environment variables* can also be read

A simple program for both Linux and Windows

A brief introduction to C I/O

See `how.stuffworks.com` for a much better introduction

Standard I/O or <stdio.h>

`stdin`	≈ `cin`	≈ `system.in`
`stdout`	≈ `cout`	≈ `system.out`
`stderr`	≈ `cerr`	≈ `system.err`

Formatted I/O

`scanf`	for input
`fprintf`	for output

Some simple format specifiers

%d	for decimal integers
%x	for hexadecimal integers
%c	for characters
%f	for floating point

A short example

```
if (fscanf(stdin, "%d", &I) == 1)
      fprintf(stdout, "i is %d or %x or %c\n", i, i, i);
```

The simple types – just like in C++ and Java

```
int
char
float
double
```

C operators

Arithmetic operators

+, –, *, /, %, unary +, unary –

Bitwise operators

&, |, ~, ^, <<, >>

Logical operators

&&, ||, !

Relational operators

==, !=, >, <, >=, <=

Increment/decrement operators

prefix ++, prefix ––, postfix ++, postfix ––

Assignment operators

=, +=, –=, /=, …, <<=, >>=

Assignment operators

```
X += 5     ⇒    X = X + 5
X *= 5     ⇒    X = X * 5
X <<= 5    ⇒    X = X << 5
```

Precedence and associativity

Operators	Associativity
`()` `[]` `->` `.`	left to right
`!` `~` `++` `--` `+` `-` `*` `&` `(`*`type`*`)` `sizeof` *these are **all** unary operators*	right to left
`*` `/` `%`	left to right
`+` `-`	left to right
`<<` `>>`	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&`	left to right
`^`	left to right
`!`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=` `+=` `~=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=`	right to left
`,`	left to right

So what is …

```
p & q ^ r
x << 5 + z
x / y / z
a == b & c
a == b && c
```

Pointers in C and C++

Given declaration

```
CType *XP ;
```

`XP` may obtain the address of a value of type *`CType`*

Given declaration

```
CType P ;
```

the assignment

```
XP = &P ;
```

makes `XP` "point" to `P`

that is, contain the address of `P`

The expression

```
*XP
```

yields the value of the *`CType`* to which `XP` points

Everyone's favorite pointer routine

```
void swap(int *x, int *y) {
     int t ;
     t = *x ;
     *x = *y ;
     *y = t ;
}
```

What about the LC/2?

```
; X = &P ;
     LEA  R0,P
     ST   R0,X
; *Y = *X + 3
     LDI  R0,X
     ADD  R0,R0,#3
     STI  R0,Y
```

Scope of variables

- Local variables
  - Declared within a function
  - Scope is the smallest enclosing `{ }`
    - Nested scope can be handled by renaming inner variables
- Global variables
  - Declared outside a function

Implementation of scope in LC/2

- Local variables are on the *stack*
  - Accessed relative to R6
- Global variables are in the *data* area
  - Accessed relative to R5

Symbol table

- LC-2 (chapter 251)

Name	**Type**	**Offset**	**Scope**
`cost`	`int`	2	`PrintPayroll`
`X`	`int`	3	`PrintPayroll`
`MidInit`	`char`	17	`global`

Variable access in real computers

- Local variables are accessed relative to *stack pointer* or *base pointer* (IA-32)
- Global variable access isn't easy
  - Locations are known until *all* procedures are joined
    - Can relocate references in procedures
    - Could access through a table of *global pointers*

Trying out an example

```
extern int gx, gy ;
int f(int a, int b) {
      int p ;
      p = a*a + 7*b
      gx = gy + a + 3*p ;
      return gx ;
}
```