計算機程式（一）
Fall 2004

Yukon Chang

Computer System

- A computer system is made up of hardware and software
- Hardware:
  - CPU, executes machine instructions one at a time
  - RAM, provides storage for program and data
  - Mother board, hard disk drive, peripherals
- Software:
  - System software, mainly operating system
  - Applications software

© Yukon Chang 2004

Software

- Also called program
- Written by programmers in some kind of programming language ➔ source program or source code
- Processed by a computer to create an equivalent program that can be executed by a computer ➔ executable program or executable code
- Only machine code is directly executable by a computer

© Yukon Chang 2004

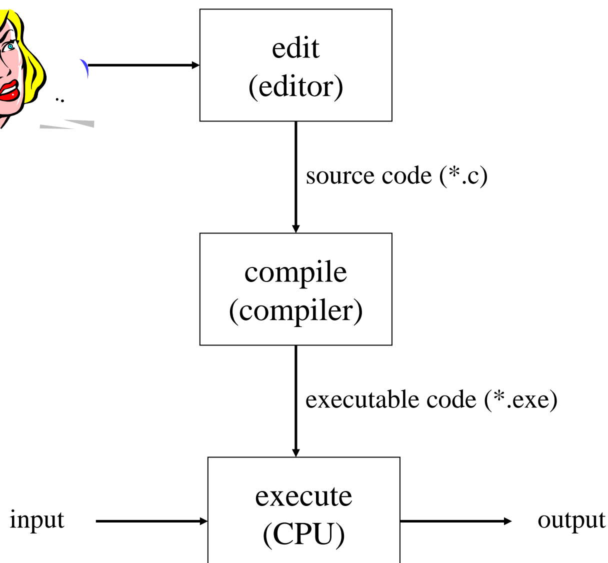
Programming Languages

- High-level language,
  one line of source code = many machine instructions
- Low-level language,
  one line of source code = several machine instructions
- Assembly language,
  one line of source code = one machine instruction

© Yukon Chang 2004

Programming

- Refers to the process of designing, writing, testing, and debugging a program.
- Tools are needed in every step of the programming process
- Roughly speaking, a programmer needs
  - Editor
  - Compiler
  - Debugger



Example – hello, world (p. 6)

```
#include <stdio.h>
main( )
{
    printf("hello, world\n");
}
```

- Read [K&R] pp. 6-8 for explanations of this program
- All example programs can be found at the course website: freefall.csie.isu.edu.tw

© Yukon Chang 2004

Steps in DOS environment

1. Use an editor to create the source program – hello.c
2. Use a compiler to compile hello.c, the result is the executable program – hello.exe
3. Run hello.exe by typing it on the command line
4. Hello.exe takes no input, but it prints out "hello, world" on the screen

© Yukon Chang 2004

Steps in UNIX environment

1. Use an editor to create the source program – hello.c
2. Use a compiler to compile hello.c, the result is the executable program – a.out
3. Run a.out by typing it on the command line
4. a.out takes no input, but it prints out "hello, world" on the screen

© Yukon Chang 2004

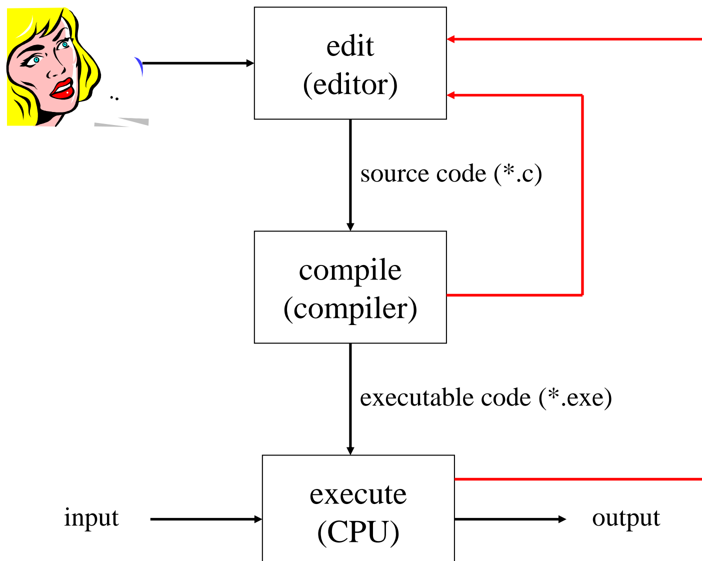
Microsoft Visual C++ 6.0

- Part of Microsoft Visual Studio (along with Visual Basic, Fox Pro, Java, …)
- An IDE (Integrated Development Environment) consists of all required tools: editor, compiler, linker, debugger, profiler, …
- Will be the primary tool for this course
- You must learn how to use it, especially the debugger, as quickly and thoroughly as possible,

© Yukon Chang 2004

It's Time to Experiment

- Play with hello.c (e.g., add/remove space or line, add/remove/change characters) to see what happens
- The more errors you make, the better
- Try to make sense of the error messages and note their sources
- Try to relate the error messages to the mistakes you think you make
- Write down your findings on paper and turn it in as assignment 1



Programming Errors

- "Compile-time" error: error that occurs before the executable code can be created
  - Preprocessor error
  - Compiler error
  - Linker error
- Run-time error: error that occurs during program execution
  - System crash
  - Abnormal program termination
  - Infinite loop
  - Wrong answer

Temperature Conversion (p. 9)

```
#include <stdio.h>
/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */
main()
{
   int fahr, celsius;
   int lower, upper, step;

   lower = 0;      /* lower limit of temperature scale */
   upper = 300;    /* upper limit */
   step = 20;      /* step size */

   fahr = lower;
   while (fahr <= upper) {
      celsius = 5 * (fahr-32) / 9;
      printf("%d\t%d\n", fahr, celsius);
      fahr = fahr + step;
   }
}
```

Variables

- Storage located somewhere in RAM (in binary form) to store final or partial result of computation
- Value may change during computation
- Must be declared before use
- Choose variable name to reflect its purpose
- Value can be observed in debugger

© Yukon Chang 2004

Program Tracing

- One of the most important features provided by a debugger
- Single step: executing one line of source code at a time
- Step over: single step without going into functions
- Other variations: step into, step out

© Yukon Chang 2004

Comments

- Anything between /* and *
- Explains what the program does so that the program is easier to understand
- Ignored completely by the compiler
- Unlike ( ) or { }, comments don't nest, the following is wrong

```
/*    /* comments */      */
```

- C++ allows // as comments
- You should use comments in your program

Data Types in C

- C has basically two types of numbers:
  - integers, also called fixed point numbers
  - floating-point numbers
- Basic data types:
  - int (an integer, including short and long)
  - char (a small integer, can also store a character)
  - float (single-precision floating-point numbers)
  - double (double-precision floating-point numbers)
- Can be signed or unsigned

Data Type

- int and char store integer values, such as 0, 100, -25
- int i; i = 2.5;
  i = ?
- float and double store numbers with decimal points, such as 333.33, 1.25, -5.0
- float f; f = 3;
  f = ?

© Yukon Chang 2004

Integer Arithmetic

- C has many operators, such as +, -, *, /
- 3/2 should be 1.5, but …
- CPU has one machine instruction to add two integers; it has a different machine instruction to add two floating-point numbers
- If both operands are integers, the addition is an integer addition (-, *, / are similar)
- If one operand is int and the other float, then the int is converted to a float before doing a floating-point addition

© Yukon Chang 2004

printf – formatted print

- Can be used to print characters, strings, and numbers
- The first argument is the format string, other arguments provide values to be printed
- Special patterns in format string tell printf how to print those values:
  - %d, print as integer
  - %f, print as floating number
  - %c, print as character
  - %s, print as string

printf – formatted print

- printf("%d\t%d\n", fahr, celsius) means
  - Print the value of fahr as an integer, print a tab to jump over a few blank spaces, print the value of celsius as an integer, print a new line character
- Other variations:
  - %3d, print as an integer, at least 3 characters wide
- Experiment and then explain the following:
  - printf("%s\n", "hello, world");
  - printf("%s, %s\n", "hello", "world");

Temperature Conversion (p. 12)

```
#include <stdio.h>
/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300; floating-point version */
main()
{
   float fahr, celsius;
   float lower, upper, step;

   lower = 0;       /* lower limit of temperature scale */
   upper = 300;     /* upper limit */
   step = 20;       /* step size */
   fahr = lower;
   while (fahr <= upper) {
      celsius = (5.0/9.0) * (fahr-32.0);
      printf("%3.0f %6.1f\n", fahr, celsius);
      fahr = fahr + step;
   }
}
```

© Yukon Chang 2004

Temperature Conversion (p. 13)

```
#include <stdio.h>

/* print Fahrenheit-Celsius table */
main()
{
   int fahr;

   for (fahr = 0; fahr <= 300; fahr = fahr + 20)
      printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

- for (initial condition; test condition; increment)

© Yukon Chang 2004

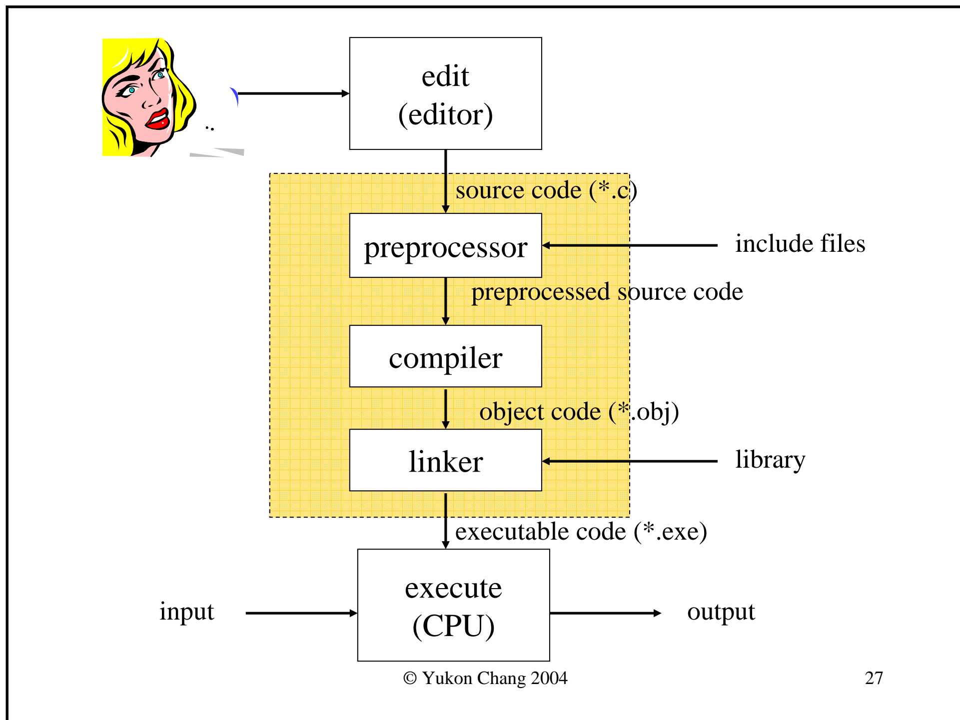
Temperature Conversion (p. 15)

```
#include <stdio.h>

#define LOWER  0     /* lower limit of table */
#define UPPER  300   /* upper limit */
#define STEP   20    /* step size */

/* print Fahrenheit-Celsius table */
main()
{
   int fahr;

   for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
      printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

- `LOWER`, `UPPER`, `STEP` are called symbolic constants

C Preprocessor

- Before the source code (*.c) is sent to the compiler proper, it first goes through the C preprocessor
- C preprocessor does text replacement only
- **`#include`** and **`#define`** are both C preprocessor directives
- The line containing **`#include`** is replaced by the content of the specified file
- **`#define`** defines a macro; subsequent occurrence of the macro is substituted by the replacement text
- There are other useful preprocessor directives



© Yukon Chang 2004

A Non-programming Example

- Create a file called "happy" containing a couple of lines of "happy birthday to you"
- Create a file called "test.c" containing:

```
#define YOU Mary
#include "happy"
happy birthday to YOU
/* this will disappear */
happy birthday to you
```

- In a DOS box, run "cl –E test.c"

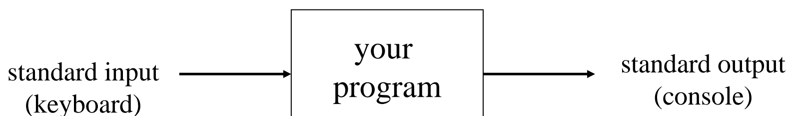
© Yukon Chang 2004

Repeat Experiment in VC++

- Create a new workspace in VC++ and create the files "happy" and "test.c" as before
- Add happy as a header file, add test.c as a source file
- Under "Project →Settings", find C/C++ tab and add /E at the beginning of the project options textbox
- Compile the project and observe the output window

Character Input and Output

- Every console mode program has a standard input (stdin) and a standard output (stdout)
- Normally, standard input is the keyboard, standard output is the console window; but either one can be redirected
- Input and output can be done one character at a time



What is a character?

- A character (or ASCII character) is a piece of information that can be stored in a byte.
- The letter 'A' looks like $0100\ 0001_2$, or 65_{10}, the letter 'a' looks like $0110\ 0001_2$, or 97_{10}, the new line character '\n' looks like $0000\ 1010_2$, or 10_{10}
- The ASCII table can be found here
- In C, a char can be used either as a (printable or unprintable) character or as a small integer
- A signed char stores a value between –128 and 127, an unsigned char, between 0 and 255

© Yukon Chang 2004

getchar( )

- **`c = getchar( );`**
  reads in one byte from stdin and stores the character in c
- If there is no more input, getchar( ) returns a special value EOF
- All 256 values in a byte may potentially be read in from stdin, EOF cannot be confused with any one of them; therefore, c cannot be a char
- c must be an int

© Yukon Chang 2004

putchar( )

- `putchar(c);`
  prints out the variable c to stdout
- c can be an int in putchar( )
- To print out the letter 'A', use
  `putchar('A');`
- To print out the new line character, use
  `putchar('\n');`
- To print out the byte AE_{16} , use
  `putchar('\xAE');`

Pseudo Code

- Before start typing in the code, it is a good idea to think about the problem you face and try to come up with good ideas to solve the problem
- Once you have a rough idea, express the idea in pseudo code, e.g., to copy stdin to stdout

```
read a character
while (character is not EOF) {
  output the character just read
  read a character
}
```

- Translate the pseudo code into real C program

File Copying (p. 16)

```
#include <stdio.h>
/* copy input to output; 1st version  */
main()
{
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

- != means “not equal” (== means “equal to”)
- Trace the program in the debugger now

© Yukon Chang 2004

File Copying v.2 (p. 17)

```
#include <stdio.h>
/* copy input to output; 2nd version  */
main()
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

- c = getchar() is first carried out
- The value of (c=getchar()), which is c, is then compared with EOF

© Yukon Chang 2004

Character Counting (p. 18)

```
#include <stdio.h>
/* count characters in input; 1st version */
main()
{
    long nc;
    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

- nc++; is equivalent to nc = nc + 1;

© Yukon Chang 2004

Character Counting v.2 (p. 18)

```
#include <stdio.h>
/* count characters in input; 2nd version */
main()
{
    double nc;
    for (nc = 0; getchar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

- C is notorious for terse programs like this

© Yukon Chang 2004

Line Counting (p. 19)

```
#include <stdio.h>
/* count lines in input */
main()
{
    int c, nl;
    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    printf("%d\n", nl);
}
```

- nl is incremented if and only if c is a newline
- Don’t confuse == with =
- Can be modified to count other characters

© Yukon Chang 2004

Word Counting (p. 20)

```
#include <stdio.h>
#define IN   1  /* inside a word */
#define OUT  0  /* outside a word */
/* count lines, words, and characters in input */
main()
{
    int c, nl, nw, nc, state;
    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c == '\t')    /* || means OR */
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    printf("%d %d %d\n", nl, nw, nc);
}
```

© Yukon Chang 2004

When Not to Trace

- Tracing a program is a very important technique that gives you a lot of information about what a program does
- With a complicated program, tracing alone will not help you understand the program
- You need to learn how to step back and study the program to discover the data structures and/or algorithms it uses
- Mathematical models help, too

© Yukon Chang 2004

Analyzing wc

- Two new variables: **nw** and **state**
- New code:

```
if (c == ' ' || c == '\n' || c == '\t' )    /* || means OR */
     state = OUT;
else if (state == OUT) {
     state = IN;
     ++nw;
}
```

- **nw** keeps track of the number of words read so far; it is incremented when certain condition is met
- **state** alternates between IN and OUT, representing currently in a word or not in a word

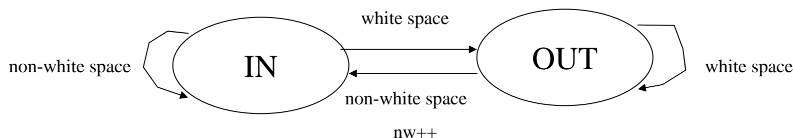
© Yukon Chang 2004

Try Out Idea with Example

- Suppose the input is
  happy birthday to you
  happy birthday dear Mary
- As characters are read in one by one, how should **`nw`** change to reflect the correct number of words we have seen so far?
- How can we tell when a word has ended?
- How can we tell when a new word begins?

State Transition Diagram

- The idea is represented in the following state transition diagram:



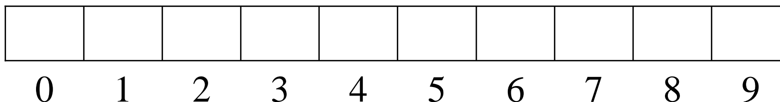
- Can be modified to count sequence of characters

Counting digits, etc. (p. 22)

```
#include <stdio.h>
/* count digits, white space, others */
main()
{
    int c, i, nwhite, nother;
    int ndigit[10];
    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;
    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;
    printf("digits =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n", nwhite, nother);
}
```


Arrays

- **`int ndigit[10];`** allocates a section of consecutive memory space large enough to store 10 integers
- All 10 integers share the same variable name
- Each integer is identified as **`ndigit[0]`**, **`ndigit[1]`**, **`ndigit[2]`**, …, **`ndigit[9]`**
- Array index starts with 0



Program Analysis

- Has the same structure as the previous ones
- The array **`ndigits`** must be initialized just like any other variable, but this time with a loop

```
for (i = 0; i < 10; ++i)
    ndigit[i] = 0;
```

- To check if character c is a digit (0-9), use

```
if (c >= '0' && c <= '9')
```

- Since a character is a (small) integer, **`c-'0'`** is a legal operation, the result is the numerical value of **`c`**
- **`++ndigit[c-'0'];`** then increments the corresponding counter in the array

© Yukon Chang 2004

Binary Number System

- Computer stores numbers in binary form, e.g.
  00010110_2
  $= 0*2^7+0*2^6+0*2^5+1*2^4+0*2^3+1*2^2+1*2^1+0*2^0$
  $= 22_{10}$
- $35_{10} = ?_2$

```
2 )_35_--- 1
2 )_17_--- 1
2 )__8_--- 0
2 )__4_--- 0
2 )__2_--- 0
     1
```

Answer: 100011_2 or 00100011_2

© Yukon Chang 2004

Octal Representation

- 1 octal digit = 3 binary digits (bits)

binary	octal
001	1
010	2
011	3
100	4
101	5
110	6
111	7

- $10110010_2 = 262_8$

© Yukon Chang 2004

Hexadecimal Representation

- 1 hexadecimal digit = 4 bits

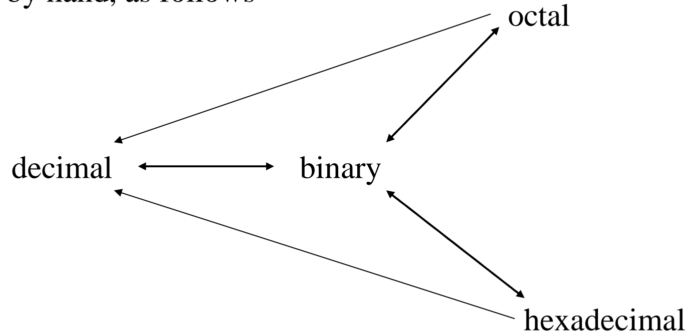
binary	hex	binary	hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

- $10011110_2 = 9E_{16}$
- Extremely important, must memorize them

© Yukon Chang 2004

Conversions

• by hand, as follows



• or use a calculator

© Yukon Chang 2004

Negative Numbers

- To represent negative numbers, the leading bit is interpreted as a sign bit
- If the sign bit is 0, the number is non-negative
- If the sign bit is 1, the number is negative
- Several ways to interpret the remaining bits
  - sign-and-magnitude
  - one's complement
  - two's complement
  - excess N

© Yukon Chang 2004

Sign-and-Magnitude

- Interpret the remaining bits as an unsigned integer as before
- For a 8-bit char, $-35_{10} = 1010\ 0011_2$
- Why? because $35_{10} = 10\ 0011_2$
- For a 16-bit short integer,
  $-35_{10} = 1000\ 0000\ 0010\ 0011_2$
- For a 32-bit integer,
  $-35_{10} = 1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 0011_2$
- There are two different representations for 0: +0 and –0

© Yukon Chang 2004

One's Complement

- Complement every bit in the number
- For a 8-bit char,
  $35_{10} = 0010\ 0011_2$
  $-35_{10} = 1101\ 1100_2$
- For a 16-bit short integer,
  $35_{10} = 0000\ 0000\ 0010\ 0011_2$
  $-35_{10} = 1111\ 1111\ 1101\ 1100_2$
- For a 32-bit integer,
  $35_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 0011_2$
  $-35_{10} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101\ 1100_2$
- Still two different representations for 0: +0 and –0

© Yukon Chang 2004

Two's Complement

- Complement every bit in the number, then add 1
- For a 8-bit char, $35_{10} = 0010\ 0011_2$ and $-35_{10} = 1101\ 1101_2$
- For a 16-bit short integer,
  $35_{10} = 0000\ 0000\ 0010\ 0011_2$ and
  $-35_{10} = 1111\ 1111\ 1101\ 1101_2$
- For a 32-bit integer,
  $35_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010\ 0011_2$
  $-35_{10} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101\ 1101_2$
- For a n-bit integer, Range = $-2^{n-1} \sim 2^{n-1}-1$
- Sum of +x and –x = 0 (with overflow bit = 1)
- Most commonly used

Special Bit Patterns

- All 0's ➔ 0
- All 0's except one bit ➔ powers of two (2^0, 2^1, 2^2, 2^3, 2^4, …), or
  1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048,…
- All 1's ➔ -1
- Some 0's followed by some 1's ➔ 2^k-1
- Left shift by one bit ➔ multiply by two
- Right shift by one bit ➔ divide by two

What Is Data Type

- Data type defines the way a data storage is interpreted
- For a byte,

  $1010\ 0011_2 = 163_{10}$, if it is an unsigned char

  $1010\ 0011_2 = -93_{10}$, if it is a signed char
- A 32-bit information can be interpreted as
  - a signed integer
  - an unsigned integer
  - an address of another data storage (this is called a pointer)

© Yukon Chang 2004

Variable Names

- Letters (a-z, A-Z), digits (0-9), and '_'
- The first character must be a letter
- Cannot be keywords, such as if, else, int, float, etc
- Case sensitive, xyz and XYZ are different
- Usually in lower case
- Choose variable name to reflect its purpose

© Yukon Chang 2004

Basic Data Types in C

- Two basic types: integer and floating point numbers
- **`char`**, **`int`**, **`float`**, **`double`**
- Can be qualified with **`short`** or **`long`**
- Can be qualified as **`signed`** or **`unsigned`**
- The operator **`sizeof`** returns the number of bytes used to represent a data type

© Yukon Chang 2004

Constants

- Suffix can be used after integer or floating point constants:
  U ➔ unsigned, L ➔ long, F ➔ float
- Can be decimal, octal (0…), or hexadecimal (0x…)
- Character constant can be written in several ways:
  - as a (small) integer, 97
  - as a letter **`'a'`**
  - in octal **`'\141'`**
  - in hexadecimal **`'\x61'`**
- Special characters: **`'\n'`**, **`'\t'`**, **`'\\'`**, etc
- **`'\0'`** represents the null character

© Yukon Chang 2004

Constants

- Suffix can be used after integer or floating point constants:
  U ➔ unsigned, L ➔ long, F ➔ float
- Can be decimal, octal (0…), or hexadecimal (0x…)
- Character constant can be written in several ways:
  - as a (small) integer, 97
  - as a letter `'a'`
  - in octal `'\141'`
  - in hexadecimal `'\x61'`
- Special characters: `'\n'`, `'\t'`, `'\\'`, etc
- `'\0'` represents the null character

String Constants

- `"hello, world"` is a C string constant, also called a string *literal*
- The length of this string is 12, but it is stored in an array of size 13
- The first 12 cells store the characters in the string, the last cell stores `'\0'` to indicate the end of the string

h	e	l	l	o	,		w	o	r	l	d	\0

Strings in C

- Strings are stored in character arrays in C
- Every string must have a terminating null character
- A string literal is created with the terminating null character automatically
- When writing code to create your own string, you must add `'\0'` yourself.
- Write a short program to create "hello, world" string manually in an array.

© Yukon Chang 2004

More on Strings

- What happens if we change "hello, world\n" to "hello\0 world\n"? Why?
- Compare the following:
  - `"0"`
  - `'0'`
  - `"\0"`
  - `'\0'`
  - `""`
  - `''`

© Yukon Chang 2004

Enumeration Constant

- An alternative to **`#define`**
- **`enum boolean { NO, YES };`**
- **`enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC };`**

  `/* FEB = 2, MAR = 3, etc. */`
- Easier to understand
- Try experiment with it

© Yukon Chang 2004

Declarations

- All variables must be declared before use
- **`int i`** tells the compiler that a variable named **`i`** exists and its data type is **`int`**
- Declarations may appear in or outside of a block (block ➔ **`{ . . . }`**)
- Declarations in a block must be put in the beginning of the block
- Undeclared variable causes an compilation error
- Declared but unused variable causes a warning (check for a mistyped variable)

© Yukon Chang 2004

Declaration (cont.)

- Most variables have unknown content in the when they are first created, i.e., uninitialized
- `int i, j;`
  is the same as
  `int i;`
  `int j;`
- `int i = 0;`
  is the same as
  `int i;`
  `i = 0;`
- `const double e = 2.71828182845905;`
  makes **e** a constant, it cannot be changed anymore

Operators

- C has a large number of operators, see p. 53 of [K&R]
- Unary operator takes one operand
- Binary operator takes two operands
- Ternary operator takes three operands
- An expression consists of properly organized variables, constants, operators, parentheses, etc.

Arithmetic Operators

- 5 arithmetic operators:
  `+ - * / %`
- May operate on integers and floating-point numbers, except `%`
- Integer division truncates any fractional part
- The modulus operator `%` takes two integers,
  `x % y`
  produces the remainder when `x` is divided by `y`
- Experiment with these operands

© Yukon Chang 2004

Limitation of Experimentation

- Sometimes the same experiment produces different results on different machines
- What is the value of `-5 / 3` ? Or `-5 % 3` ?
- The direction of truncation for `/` and the sign of the result for `%` are machine-dependent for negative operands
- Machine-dependent features are defined in the language specification and must be avoided

© Yukon Chang 2004

For Loops (midterm review)

- For loop has 3 parts: (1) initialization, (2) test for continuation, and (3) increment

```
for (i = 0; i < 10; i++)
    printf("%d ", i);
```

  - `i` is set to 0
  - `i` is compared to 10, keep looping
  - `i` is printed
  - `i` is incremented, now `i` is 1
  - `i` is compared to 10 , keep looping
  - `i` is printed
  - `i` is incremented, now `i` is 2
  - …
  - `i` is incremented, now `i` is 10
  - `i` is compared to 10, exit loop

© Yukon Chang 2004

Example of Modulus (%)

- A year is a leap year if it is divisible by 4, but not divisible by 100, except that years divisible by 400 are leap years
- See KR041.c
- **`scanf("%d", &year)`** reads an decimal integer from the standard input into **`year`**
- **`scanf( )`** returns **`EOF`** if there is no more input or if there is error in integer conversion
- Try to add a loop to this program

© Yukon Chang 2004

Operator Precedence (2.12)

- Operator with a higher precedence is carried out first
- 3 + 5 / 2 = ?

  4, if + is carried out first

  5, if / is carried out first
- The answer is 5 because * and / have higher precedence than + and /
- Table 2.1 on p. 53 of [K&R] lists operators in decreasing precedence
- If 4 is what you want, write the expression as (3 + 5) / 2

© Yukon Chang 2004

Operator Associativity (2.12)

- What if two operands have the same precedence?
- 9 – 3 + 2 = ?

  8, if the subtraction on the left is done first

  4, if the addition on the right is done first
- The answer is 8 because + and – associates left to right, i.e., 9 – 3 + 2 means (9 – 3) + 2, not 9 – (3 + 2)
- Most operators associates left to right
- When in doubt, use parentheses to clarify for your reader (and yourself, too!)

© Yukon Chang 2004

Relational Operators (2.6)

- Relational operators:

  `>   >=   <   <=   ==   !=`

- Each operator returns 1 (true) or 0 (false)
- Do not confuse **`==`** with **`=`**

  `x == 5` tests if the value stored in **`x`** is 5

  `x = 5` stores 5 in **`x`**

- But **`if ( x = 5 ) y = 7;`** is still valid
- You can verify that **`y`** will be set to 7 no matter what the value of **`x`** is
- Why?

Two Aspects of C Expression

- In C, every expression has a value and a side effect
- For example, after **`a = 2; b = 3; c = 4;`**
  - **`a+b*c`** has a value of 14
  - **`a`**, **`b`**, **`c`** remains unchanged, i.e., null or no side effect
- **`x == 5`**
  - has a value of 1 if x is 5, 0 otherwise
  - has null side effect
- **`x = 5`**
  - has a value of 5
  - has a side effect of assigning 5 to x
- **`if (x = 5) y = 7;`** always assigns 5 to **`x`** and 7 to **`y`**

Increment and Decrement (2.8)

- `int i = 5; i++;` `i` becomes 6
- `int i = 5; ++i;` `i` also becomes 6
- No difference at all?
- `int i = 5, j; j = i++;` what are the values of `i` and `j`?
- `int i = 5, j; j = ++i;` what are the values of `i` and `j`?
- The value of `i++` is the OLD value of `i`, the value of `++i` is the NEW value of `i`
- `(i+j)++;` is illegal
- What does `s[i++] = c;` mean?

© Yukon Chang 2004

Logical Operators (2.6)

- Logical operators:

  `!    ||    &&`

- C has no boolean type
  - Any nonzero value is interpreted as 1, or true
  - Zero is interpreted as false
- C experts write

  ```
  if (!valid)
  ```

  instead of

  ```
  if (valid == 0)
  ```

© Yukon Chang 2004

Lazy Evaluation

- For binary operations such as x + y, C normally does not specify which operand is evaluated first
- Two exceptions: `||`, `&&`
  - If x is true, then x||y must be true, y is not evaluated
  - If x is false, the x`&&`y must be false, y is not evaluated
- Important when y has side effect

© Yukon Chang 2004

Bitwise Operators (2.9)

- Bitwise AND: `&` (Not logical AND, `&&`)
  - used to turn bits off
- Bitwise inclusive OR: `|` (Not logical OR, `||`)
  - used to turn bits on
- Bitwise exclusive OR: `^`
  - used to do XOR on two bit patterns
-

```
      n 1010 0101 1100 0011
   0xff 0000 0000 1111 1111
 n&0xff 0000 0000 1100 0011
 n|0xff 1010 0101 1111 1111
 n^0xff 1010 0101 0011 1100
```

© Yukon Chang 2004

Bitwise Operators (2.9)

- Left shift: **`<<`**
  - **`x<<3`** is equivalent to multiplying **`x`** by 8, or 2^3
- Right shift: **`>>`**
  - **`X>>5`** is equivalent to applying integer division to **`x`** by 32, or 2^5
- One's complement: **`~`**
  - converts each 1-bit to a 0-bit and visa versa
  - adjust to data size automatically
  - **`n & 0xff00`** sets the last 8 bits to 0 in 16-bit short
  - **`n & ~0xff`** does the same regardless of the integer size
- Bitwise operations are very important in your future study

© Yukon Chang 2004

Assignment Operators (2.10)

- Most binary operators have a corresponding assignment operator

  `a = a + 3;` can be written as **`a += 3;`**
  `a = a - 3;` can be written as **`a -= 3;`**
  `a = a * 3;` can be written as **`a *= 3;`**
  `a = a / 3;` can be written as **`a /= 3;`**
  `a = a % 3;` can be written as **`a %= 3;`**
  `a = a << 3;` can be written as **`a <<= 3;`**
  `a = a >> 3;` can be written as **`a >>= 3;`**
  `a = a & 3;` can be written as **`a &= 3;`**
  `a = a ^ 3;` can be written as **`a ^= 3;`**
  `a = a | 3;` can be written as **`a |= 3;`**

© Yukon Chang 2004

Conditional Expressions (2.11)

- The statement

```
if (a > b)
    z = a;
else
    z = b;
```

  can be written as

```
z = (a > b) ? a : b;
```

- expr1 ? expr2 : expr3
  - evaluate expr1 first
  - if expr1 is true, expr2 is the value of this expression
  - if expr1 is false, expr3 is the value of this expression

Type conversion (2.7)

- C does very little covert operations, that is, most machine code can be traced back to explicit source code
- Exceptions: type conversion and function calls
- Operand may need to be converted to a different type, e.g., when adding an int and a float
- Usually, conversions go from "narrower" operand to "wider" one without losing information
- Conversion the other way around may draw a warning, e.g., assigning an int to a char
- Better use explicit type conversion, or type casting
  - `c = (char) i;`

Practice Problem (p. 53)

- How to interpret

```
a + b || c ++  && ! d  >= e && f
```

- See Table 2-1

```
a + b || (c++) && (!d) >= e && f

(a+b) || (c++) && (!d) >= e && f

(a+b) || (c++) && ((!d)>=e) && f

(a+b) || ((c++)&&((!d)>=e)) && f

(a+b) || (((c++)&&((!d)>=e))&&f)
```

© Yukon Chang 2004

Practice Problem (p. 48)

- If **`i = 0x1234`**, what is the value of
  `( (i & 0x0fff) | 0x000f ) ^ 0x0ff0`
- 0001 0010 0011 0100 (before)
- 0000 1101 1100 1111 (after)
- **`i & 0x0fff`** turns the first 4 bits off
- **`i | 0x000f`** turns the last 4 bits on
- **`i ^ 0x0ff0`** inverts the middle 8 bits

© Yukon Chang 2004

Practice Problem (p. 52)

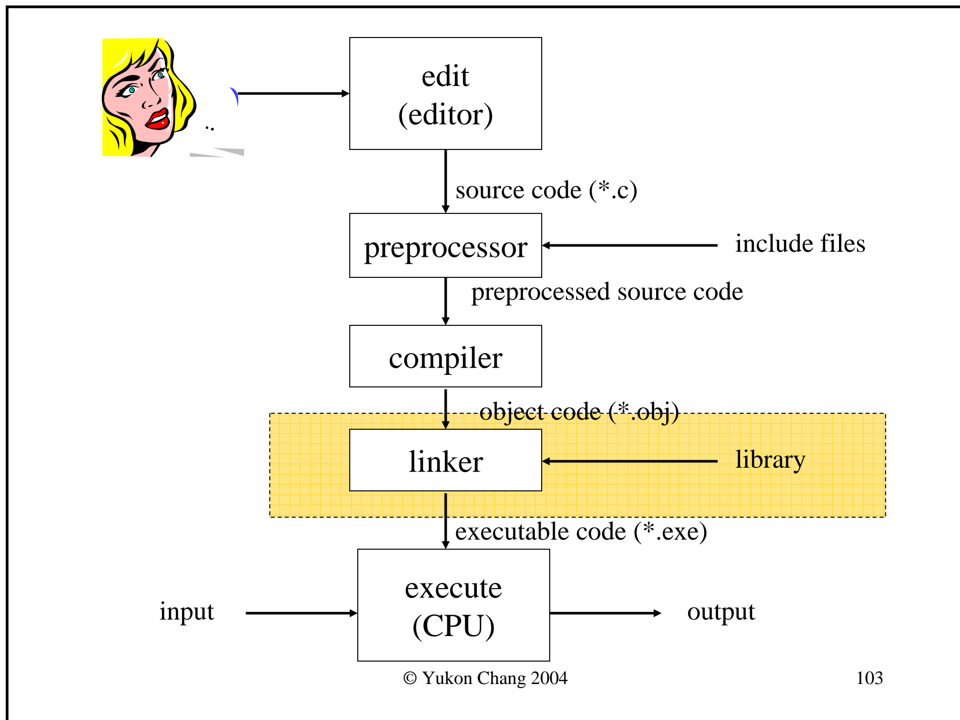
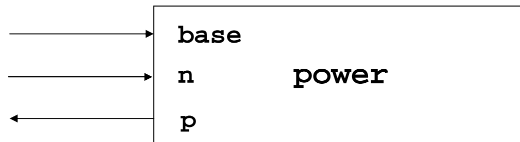
- ```
  printf("I saw %d boy%s.\n",
      n, n==1 ? "" : "s");
  ```
- Possible outputs:
  ```
  I saw 5 boys.
  I saw 1 boy.
  ```
- Can we do
  ```
  printf("I saw %d boy%c.\n",
      n, n==1 ? ' ' : 's');
  ```

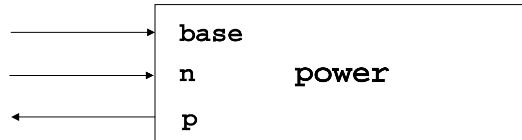
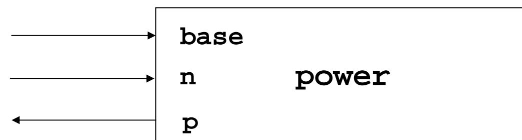

Quiz Problems Wanted

- See if you can come up with good problems for me to use in quizzes
- State why your problem is tricky and what are the probable incorrect answers when someone takes the bite
- Extra points will be awarded if your problem is adopted
  ```

Control Flow

- Program executes line by line
- Semicolon is a statement terminator and must be present
- Braces { and } are used to group declarations and statements into block
- Variables can be declared inside any block
- Variables declared outside block are called global variables

If-Else

- **`if (`***expression***`)`**
  statement1
  `else`
  statement2
- If *expression* is true, *statement1* is executed
- Otherwise, *statement2* is executed
- C experts use
  `if (`*expression***`)`**
  instead of
  `if (`*expression* **`!= 0)`**

Dangling Else Problem

```
int a = 1, b = 2, c = 3, d = 4;
if (a == 2)
    if (b < c)
        d = 7;
else
    d = 9;
```

- **`d`** = ?
- C compiler interprets the above statement this way:

```
if (a == 2) {
    if (b < c)
        d = 7;
    else
        d = 9;
}
```

© Yukon Chang 2004

Multiple If-then-else

- **`if`** **(***expression***)**

  statement

  `else if` **(***expression***)**

  statement

  `else if` **(***expression***)**

  statement

  `else if` **(***expression***)**

  statement

  `else`

  statement

© Yukon Chang 2004

Switch

```
if (a == 3)
   x = 6;
else if (a == 7)
   y++;
else if (a == -1)
   z += 3;
else
   printf("hi");
```

```
switch (a) {
case 3:
   x = 6;
   break;
case 7:
   y++;
   break;
case -1:
   z += 3;
   break;
default:
   printf("hi");
}
```


Switch

- **`case`** specifies condition
- **`break`** provides immediate exit from the switch
- Omitting **`break`** causes "falling through cases"
- **`default`** equivalent to the final **`else`**
- KR059.c is easier to understand than KR022.c

While Loops

- **while (***expression***)**
  statement
- Repeat *statement* until *expression* is false
- Do not execute *statement* if *expression* is false initially
- Explicit infinite loop
  while (1)
  statement
- May get out of loop with **break** or **goto**

© Yukon Chang 2004

For Loops

- For loop has 3 parts: (1) initialization, (2) test for continuation, and (3) increment
  for (*expr1***;** *expr2***;** *expr3***)**
  statement
- Equivalent to
  expr1
  while (*expr2***) {**
  statement
  expr3
  }

© Yukon Chang 2004

For Loops

- Fundamental type of for loop
  `for (i = 0; i < 10; i++)`
  statement
- Either of *expr1*, *expr2*, or *expr3* may be empty
- Empty *expr1* ➔ no initialization
- Empty *expr2* ➔ no test, always loop
- Empty *expr3* ➔ no increment
- Explicit infinite loop
  `for ( ; ; )`
  statement

Loop Exercises

```
int a[10] = { 3, 20, 7, -5, ... };
int i, sum = 0;

for (i=0; i<10; i++)
   printf("%d%c", a[i], i==9 ? '\n' : ' ');

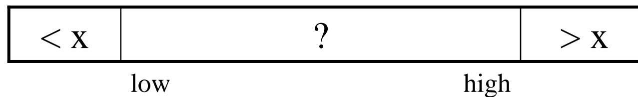
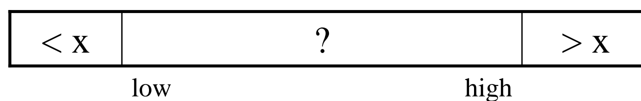
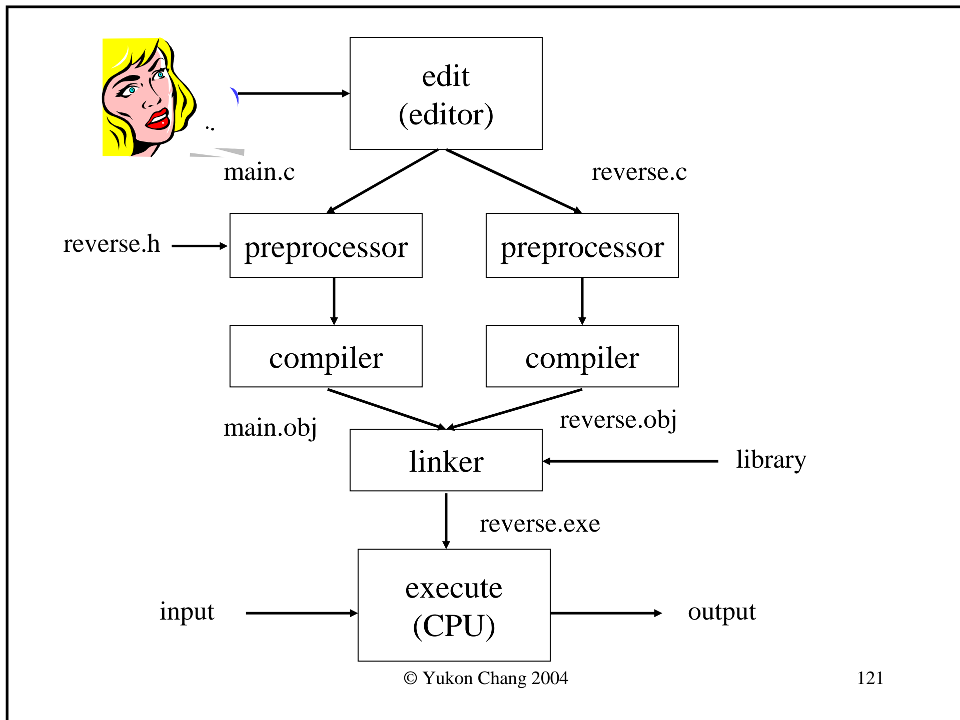
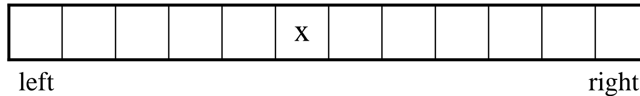
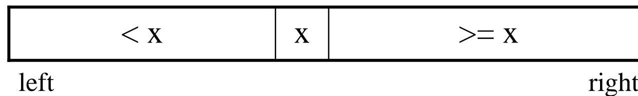
for (i=0; i<10; i++)
   sum = sum + a[i];

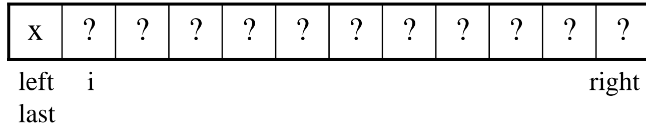
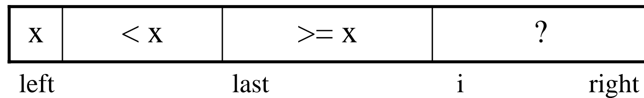
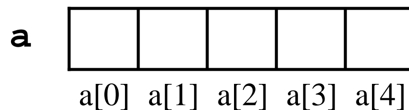
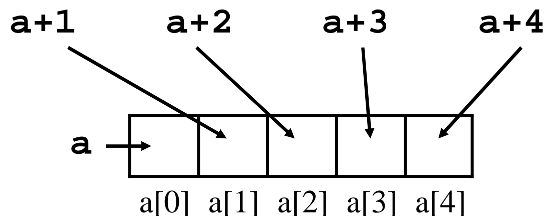
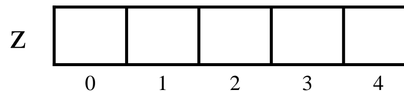
printf("%d\n", sum);
```

- Try other variations of the loop problem in assignment 5

Do –while Loops

- **`do`**

  statement

  `while (`*expression***`)`**
- Similar to while loop but less used
- Execute at least once

© Yukon Chang 2004

Break and Continue

- **`break`** can be used in both switches and loops
- **`break`** causes program control to move out of the (innermost) loop

```
while (...) {
   ...
   break;
   ...
}
```

- **`continue`** can be used only in loops
- **`continue`** causes program control to move past the last statement of the loop

```
while (...) {
   ...
   continue;
   ...
}
```

© Yukon Chang 2004

Goto and Labels

- Goto is legal in C, but …
- Don’t use them except in special cases

```
for ( ... )
     for ( ... ) {
          ...
          if (disaster)
               goto error;
     }
     ...
 error:
     /* clean up the mess */
     ...
```


Functions

- A blackbox encapsulating some computation
- To use a function, one only needs to know what is done, not how it is done
- Library functions and user-written functions
- Library functions are written, compiled, and archived by others
- **`printf`**, **`getchar`**, and **`scanf`** are library functions
- To use a library function, link it to your code



Functions

- See KR024.c
- **`int power(int m, int n);`** tells the compiler that **`power( )`** is a function with two **`int`** parameters and returns an **`int`**
- **`power( )`** is called twice in the printf line
- **`power(2,5)`** computes and returns 2^5, or 32
- Single step through KR024.c to see how control is transferred to **`power( )`** by "step into (F11)"

© Yukon Chang 2004

power( )




```
int power(int base, int n)
{
    int i,  p;

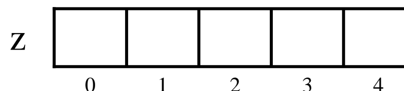
    p = 1;
    for (i = 1; i <= n; ++i)
        p = p * base;
    return p;
}
```

© Yukon Chang 2004

Function Definition

- return-type function-name (parameter declarations, if any)

  {
    declarations
    statements
  }
- **`void f(void)`** means f takes no input and returns no value
- Missing return-type means returning **`int`**
- **`main( ){ }`** returns **`int`**

© Yukon Chang 2004

Call By Value

- ALL C function calls pass parameter by value
- Arguments are copied into local variables in the function
- Original variables are not affected, see KR027.c



- What if we DO want to change the original variables? Answer: pointers in Chapter 5

© Yukon Chang 2004

power( ) p. 27




```
/* power:  raise base to n-th power; n >= 0; version 2 */
int power(int base, int n)
{
    int p;
    for (p = 1; n > 0; --n)
        p = p * base;
    return p;
}
```

© Yukon Chang 2004

Passing Array Argument

- Array may be passed as argument, too

```
/* strlen:  return length of s */   // KR039.c
int strlen(char s[])
{
    int i = 0;

    while (s[i] != '\0')
        ++i;
    return i;
}
```

- When passing array as argument, the address of the first element of the array is passed

© Yukon Chang 2004

Example Code from K&R

- Most sample programs contain loops
- Try to find regularity during loop execution
- Help you understand the sample code
- Help you learn to write your own program
- Add your own comments to the sample code to make sure you understand what the program does
- These comments may eventually lead to pseudo code

© Yukon Chang 2004

Example – atoi( ) p.43

```
/* atoi:  convert s to integer */
int atoi(char s[])
{
    int i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');
    return n;
}
```

- Always need a simple **`main( )`** as the test driver

© Yukon Chang 2004

Example – strcat( ) p.48

```
/* strcat:  concatenate t to end of s; s must be
  big enough */
void strcat(char s[], char t[])
{
    int i, j;

    i = j = 0;
    while (s[i] != '\0') /* find end of s */
        i++;
    while ((s[i++] = t[j++]) != '\0') /* copy t */
        ;
}
```

- Self-study: getbits( ) and bitcount( ) on p. 49 and p. 50

© Yukon Chang 2004

Binary Search

- v[0..n-1] is a sorted array
- If x is in v, return i where v[i] = x, else return –1
- Regularity is illustrated below:



- At the beginning?
- When to terminate?

© Yukon Chang 2004

Example – binsearch( ) p.58

```
/* binsearch:  find x in v[0] <= v[1] <= ... <= v[n-1] */
int binsearch(int x, int v[], int n)
{
    int low, high, mid;
    low = 0; high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else    /* found match */
            return mid;
    }
    return -1;   /* no match */
}
```

© Yukon Chang 2004

Binary Search

1	3	7	11	18	23	32	46	78	91	92	101	104

1	3	7	11	18	23	32	46	78	91	92	101	104

1	3	7	11	18	23	32	46	78	91	92	101	104

1	3	7	11	18	23	32	46	78	91	92	101	104

1	3	7	11	18	23	32	46	78	91	92	101	104

1	3	7	11	18	23	32	46	78	91	92	101	104

© Yukon Chang 2004

Binary Search

- Easy to make mistakes in writing binsearch
- What if we say
  ```
  mid = (low+high)/2-1;
  ```
- What if we say
  ```
  high = mid;
  ```
- What if we say
  ```
  low = mid;
  ```



© Yukon Chang 2004

Example – reverse( ) p.62

```
/* reverse:  reverse string s in place */
void reverse(char s[])
{
    int c, i, j;
    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
main()
{
    char a[] = "good morning, world";
    reverse(a);
    printf("%s\n", a);
    return 0;
}
```

© Yukon Chang 2004

Samples Skipped

- Study
  - getbit on p. 49
  - bitcount on p. 50
  - atoi( ) p.61
  - itoa on p. 64
  - trim on p. 65
- Skip
  - shellsort on p. 62
  - pattern matching in §4.1
  - calculator examples on §4.2 & §4.3

© Yukon Chang 2004

Separate Compilation

- A large program may contain several million lines of code
- A large student project may contain several thousand lines of code
- Break up the program into separate files
- Each file is compiled separately into object code
- All object code are linked together (along with library) to form the final executable program

Reverse( ) Revisited

```
/* reverse:  reverse string s in place, 2nd version */
void reverse(char s[])   // put in reverse.c
{
    int c, i, j;
    for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
        c = s[i];
        s[i] = s[j];
        s[j] = c;
    }
}
main()     // put in main.c
{
    char a[] = "good morning, world";
    reverse(a);
    printf("%s\n", a);
    return 0;
}
```



Linking in Separate Compilation

- Variables and functions must be declared before use
- In main.c, no one tells the compiler what type reverse( ) is, so compiler assumes it is a function returning an int and warns about it
- Adding one line

```
void reverse(char s[]);
```

  at the top of main.c solves the problem
- What if there are many functions?

Header File

- A text file that contains declarations of functions (and external variables)
- Create a header file reverse.h with one line

```
void reverse(char s[]);
```

- Add one line to main.c

```
#include "reverse.h"   // not <reverse.h>
```

- C preprocessor adds the declaration for reverse to main.c

Header File

- To include a system header file

```
#include <stdio.h>
```

- You have to know what header file to include in order to use a library function
- To include a user-defined header file

```
#include "reverse.h"
```

- Always remember to include proper header files to avoid warnings and errors
- Compiler may be lenient on functions but strict on external variables

Automatic Variables

- Also called local variables
- Defined at the top of a block

```
{
   int i;
}
```

- Tells compiler that i is an int
- Compiler prepares memory space to store i
- When control goes into the block
  – i comes alive but contains junk
- When control moves out of the block
  – system reclaims i's memory
  – i's content no longer exists

© Yukon Chang 2004

Automatic Variables

- Automatic variable has block scope, it cannot be used outside the block

```
{
   int i;
}
```

- i can be defined in multiple blocks

```
{
   int i;
   i = 5; ...
}
{
  ...      // can't use i in this block
}
{
   int i;  // this is a different i from the one above
   i = 100; ...
}
```

© Yukon Chang 2004

Automatic Variables

- i can be used in nested blocks

```
{
   int i = 5;
   ...    // i is 5
   {
       ...
       printf("%d", i); // print 5
       ...
   }
   printf("%d", i); // print 5
}
```

© Yukon Chang 2004

Automatic Variables

- i can be defined in nested blocks

```
{
   int i = 5;
   ...    // i is 5
   {
       int i = 100;
       printf("%d", i); // print 100
       ...
   }
   printf("%d", i); // print 5 again
}
```

- The inner i temperarily hides the outer I
- Warning: this is legal, but don’t do it

© Yukon Chang 2004

External Variables

- Also called global variables
- Defined outside of any function
- Seem easy to use, but you should not use them without a convincing reason
- A global variable can be defined only once

```
int x;
void f()
{
    ....
}
```

```
int x; // wrong, two
main() // copies of x
{
    ....
}
```

© Yukon Chang 2004

External Variables

- Have file scope, meaning it can be used anywhere in your program
- Defined in one file and declared before use in other files, then only one copy of x exists

```
int x;
void f()
{
    ....
}
```

```
extern int x; // OK
main()
{
    x = 5; ....
}
```

© Yukon Chang 2004

Declaration vs. Definition

- Two types of objects: variable and function
- An object takes up memory space
  - A variable needs memory space to store its value
  - A function needs memory space to store its code
- An object may be used elsewhere in the program
- A declaration is a notice to the compiler announcing the existence of certain object
- A definition is a declaration that also tells compiler to allocate storage for the object

© Yukon Chang 2004

How to Identify

- For functions,
  - declaration finishes with semicolon (;)
  - definition finishes with the function body in a block { … }
- For variables,
  - declaration starts with the keyword **`extern`**
  - definition does not start with **`extern`**
- Function declaration may start with **`extern`**, but this **`extern`** is optional

© Yukon Chang 2004

Reverse Polish Calculator

- Each operator follows its operands
- Normal arithmetic expression is called infix notation, for example, (1 – 2) * (4 + 5)
- 1 2 – 4 5 + * is the equivalent postfix or Reverse Polish Notation (RPN)
- No need for parenthesis in RPN
- Use a stack to evaluate an expression in RPN

© Yukon Chang 2004

Stack

- A data structure
- First in, last out
- Think of a stack of dishes in a kitchen or a train of shopping carts
- Two main operations: push and pop
- push(x) pushes x onto the top of the stack
- pop( ) removes the top of the stack and returns it

© Yukon Chang 2004

Pseudo Code

```
while (next operator or operand is not end-of-file indicator)
  if (number)
    push it
  else if (operator)
    pop operands
    do operation
    push result
  else if (newline)
    pop and print top of stack
  else
    error
```

© Yukon Chang 2004

First Attempt

- All program in one file, all.c
- See course website under program/RPCalculater/Attempt1
- About 130 lines of C code
- Try to mark
  - declarations of various functions
  - global variables
- Next, try separate compilation

© Yukon Chang 2004

Second Attempt

- Break up all.c into 4 separate files, main.c, stack.c, getop.c, getch.c
- Some C files compiles OK, but some don't
- Breaking up all.c makes some declarations missing
- Fix the problem by adding necessary declarations
- Now declarations are scattered over many files

Third and Final Attempt

- Now, move recurring declarations into a header file, calc.h
- Add #include "calc.h" in C files as needed
- See [K&R] p.82 for a clean illustration

Stack in Function Calls

- Stack is also used in function calls
- Function calls and their stack operations
  - main starts — push (main's info)
  - main calls f — push (f's info)
  - f calls g — push (g's info)
  - g returns — pop ( ) // g
  - f calls h — push (h's info)
  - h returns — pop ( ) // h
  - f returns — pop ( ) // f
  - main returns — pop ( ) // main

Static Variables

- Two unrelated uses
- Static automatic variables lives throughout the program

```
{
    static int bufp = 0;
    …
}
```

- External static variables and functions are visible only in the current file
- Two external variables CAN have the same name if they are declared static!

Program Design

- Try to break up a program's work into small, well-defined units
- Assign each unit of work to a function
- Write and test functions separately
- Higher level steps call these functions to get the work done
- Function f( ) thinks to itself: "If I can call a function g( ) to do this for me, how can I finish the work at hand"

© Yukon Chang 2004

Recursion

- Function f( ) starts by thinking to itself: "If I can call a function g( ) to do this for me, how can I finish the work at hand"
- What if it just so happens that g( ) solves the same problem f( ) does?
- Function f( ) calls f( ) to do part of the work, f( ) then finishes its own work
- Function f( ) solves easy cases directly
- A very powerful technique

© Yukon Chang 2004

Recursion Example: Summation

- Problem: Adding n numbers from x[i..i+n-1]
- Easy case: n = 1, return x[i]
- Otherwise, ask another function to compute s = sum of x[i..i+n-2], then return s+x[i+n-1]

```
int sum(int x[], int i, int n)
{
   if (n==1)
      return x[i];  //must have a base case
   return sum(x, i, n-1) + x[i+n-1];
}
```

© Yukon Chang 2004

Fundamentals of Recursion

- A very powerful technique
- Solve a problem by calling upon itself, i.e., a function f calls itself directly or indirectly
- Must have a base case, otherwise the function loops forever
- The “problem size” gets smaller after each round of recursion
- Think: “if someone gives the answer to a smaller problem to me, how can I make use of it”

© Yukon Chang 2004

Simple Recursion Examples

- Problems that are associated with recursive definitions:
  - Factorial,
    n! = 1, if n = 0, 1
    = n * (n-1)!, if n > 1
  - Fibonacci number,
    fib(n) = 1, if n = 0, 1
    = fib(n-1) + fib(n-2), if n > 1
    (Note: there is a big problem with this solution!)
  - Many recursively defined data structures

© Yukon Chang 2004

Observe Recursion in Action

- fact(6) calls fact(5), pushes "activation record" onto the stack
- fact(5) calls fact(4), another push
- fact(4) calls fact(3), push, etc.
- fact (2) calls fact(1), push
- fact(1) returns 1, pop back to fact(2)
- fact(2) returns 2, pop back to fact(3)
- fact(3) returns 6, pop back to fact(4), etc.
- fact(6) returns 720

© Yukon Chang 2004

Recursion Example 2: Quick Sort



- Use the middle element as a pivot and swap array elements one at a time to get this:



- Recursively call qsort( ) to sort the left half and the right half
- Done!

© Yukon Chang 2004

Recursion Example 2: Quick Sort

```
void qsort(int v[], int left, int right)
{
    int i, last;
    void swap(int v[], int i, int j);

    if (left >= right) return;
    swap(v, left, (left + right)/2);
    last = left;
    for (i = left + 1; i <= right; i++)
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);
    qsort(v, left, last-1);
    qsort(v, last+1, right);
}
```

© Yukon Chang 2004

Recursion Example 2: Quick Sort

- How to do partitioning?
- In the beginning



- After a while



© Yukon Chang 2004

Macro Substitution

- May be used to define function-like macros
  - `#define square(x) x*x`
- Avoids function call overhead
- More efficient, but has many problems
  - `square(5)`, OK
  - `square(2+3)`?
  - `5/square(5)`?
- Replaced in C++ by inline functions

© Yukon Chang 2004

Memory

- Recall that data type refers to the way a number of bits are interpreted
- An 8-bit storage (a byte) stores 8 bits of information, e.g., **`0xa3`**
- A 32-bit storage (an int) stores 32 bits of information, e.g., **`0x0328f94c`**
- Each byte or word is located by its address, also a 32-bit value, e.g., 0x084129cc
- Can we store the address of some storage in some variable and use it as an address?

Chapter 5 Pointers and Arrays

- Two concepts are closely related in C
- A storage has
  - Content
  - Address
- Debugger allows us to peek at the content of a storage
- It also allows us to find out the address of a storage

Pointers

```
int x, y;
int *p;
x = 5;
p = &x; // p points to x
*p = 6; // x is set to 6
p = &y; *p = 100;
```


- & ➔ address of operator
- * ➔ dereference, follow the pointer to get the content
- It really doesn't matter where x is located (or what the address of x is)

© Yukon Chang 2004

More Examples

```
int x = 1, y = 2, z[5];
int *ip, *iq;;
ip = &x;
y = *ip;
ip = &z[0];
*ip = 20;
*ip = *ip + 10;
y = *ip + 1;
*ip += 1;
(*ip)++;  // *ip++ means
          //   *(ip++);
iq = &y;
*iq = *ip;
iq = ip;
```


© Yukon Chang 2004

Swap Revisited

- Recall that C passes parameters by value
- This version of swap doesn't work

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

- Now try this

```
void swap(int *px, int *py)
{  int temp;
   temp = *x; *x = *y; *y = temp;
}
```

© Yukon Chang 2004

Swap Revisited

- Call the new version of swap like this:

```
void swap(int *px, int *py)
int i=6, j=12;
swap(&i, &j);
```

- Now observe various addresses in debugger

i		px	
j		py	

© Yukon Chang 2004

Pointers and Arrays

- In C, arrays and pointers are closely related. They can be used almost interchangeably.
- **`int a[5];`**
  declare **`a`** as a constant pointer to **`a[0]`**. The compiler also sets aside space for 5 integers.
- **`a`** can be used just as any pointer, except that its value cannot be changed.



© Yukon Chang 2004

Pointer Arithmetic

- Since a points to **`a[0]`**, naturally **`a+1`** points to **`a[1]`**, **`a+2`** points to **`a[2]`**, etc



- In other words, **`a[i]=*(a+i)`**

© Yukon Chang 2004

Examples

```
int x = 1, y = 2, z[5];
int *ip, *iq;;
ip = &z[0];
*ip = 20;
*(ip+1) = *ip + 10;
ip += 2;
*ip += 2;
iq = z+4; // using array
          // as pointer
iq--;
*iq = *ip;
iq = ip;
a = &x; // WRONG!
a++;    // WRONG!
ip++;   // OK
```

x

y

ip

iq

z



Using Pointer as Array

```
int x = 1, y = 2, z[5];
int *ip, *iq;;
ip = z+1;
*ip = 20;
ip[1] = 3;// using pointer
          // as array
ip[2] = 10;
ip[3] = -7;
ip[-1] = 3;

ip[4] = 9;  // OUT OF BOUND
ip[-2] = 9; // OUT OF BOUND
// NO RUNTIME
// ERROR!
```

x

y

ip

iq

z



What Is This?

```
int z[5];
z[3] = 8;        // OK
4[z] = 13;       // OK, too!
```

- Why? Recall that `a[i]=*(a+i)`
- ```
  4[z] = *(4+z)
       = *(z+4)
       = z[4]
  ```
- Pointer is array
- Array is (constant) pointer

© Yukon Chang 2004

Passing Array as Argument

- C always passes parameters by value
- When passing an array as argument, C passes the address of the first element in the array by value, i.e., it passes a pointer to the first element in the array
- The array itself is not copied
- `int binsearch(int x, int v[],int n)`
  is the equivalent to

  `int binsearch(int x, int *v, int n)`

© Yukon Chang 2004
  ```

Address Arithmetic

- Legal expressions:

  `p + 1`

  `p - 2`

  `p++`

  `p += 3`

  `p - q`, if `p` & `q` point to the same array

  `p < q`, if `p` & `q` point to the same array

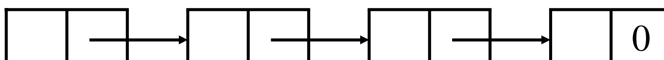
- Other expressions are illegal, such as

  `p * 2`

  `p + q`

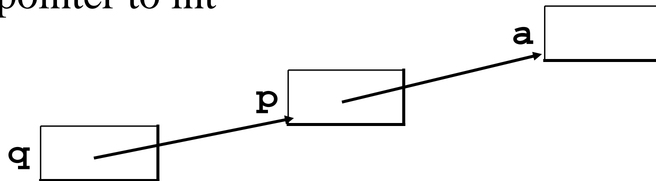
Null Pointer

- A pointer variable may contain NULL, or 0
- Such a pointer is called a null pointer
- It does not really point to the memory address of 0
- Rather, it is a special case that meaning not pointing anywhere at all
- A linked list uses null pointer to indicate the end of the list (wait till next semester)



Pointers to Pointers

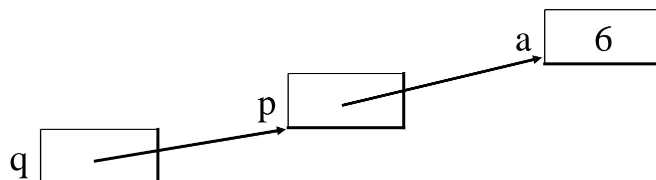
- Every storage has address, so do pointers themselves
- **`int *p = &a;`** defines **`p`** as a pointer to integer **`a`**
- **`int **q = &p;`** defines **`q`** as a pointer to **`p`**, a pointer to int; therefore **`q`** is a pointer to pointer to int



© Yukon Chang 2004

Pointers to Pointers

```
int a = 6, *p = &a, **q = &p;
(*p)++;
p++; *p = 100; // logical error
(**q)++;
(*q)++;
q++; *q = &a;  // logical error
```



© Yukon Chang 2004

Character Pointers

- Recall that strings in C are implemented by character arrays
- **`"hello, world"`** is a string literal: it has no name and is not supposed to be modified by program
- **`char s[]= "hello, world";`** is a named array and can be modified
- **`char s[10]`** is an uninitialized array and must be initialized before use

Character Pointers

- **`char *p;`** is a character pointer, it can be used to point to a string but no space is reserved for the string yet
- **`p= "hello, world";`** directs s to point to a string literal
- The same effect can be achieved in one line: **`char *p= "hello, world";`**
- Any difference? **`s`** is a constant pointer; **`p`** is a nonconstant (i.e., modifiable) pointer
- Refer to figure on p. 104 of [K&R]

String Functions

- The "string library" in C provides various functions for string manipulations:
  - **strcpy(char *s, char *t)**, copy **t** to **s**
  - **strcmp(char *s, char *t)**, compare **s** and **t**
  - **strlen(char *s, char *t)**, length of **s**
  - **strcat(char *s, char *t)**, append **t** after **s**
  - and more
- K&R provides code examples for some of these functions (with non-standard return values)

© Yukon Chang 2004

strcpy( ), version 1

```
void strcpy(char *s, char *t) // version 1
{
    int i = 0;
    while ((s[i] = t[i]) != '\0')
        i++;
}

main() { // test driver
    char x[20], y="hello, world";
    strcpy(x, y);
}
```

© Yukon Chang 2004

strcpy( ), version 2

```
void strcpy(char *s, char *t) // version 2
{
   int i = 0;
   while ((*s = *t) != '\0') {
      s++;
      t++;
   }
}
main() { // test driver
   char x[20], y="hello, world";
   strcpy(x, y);
}
```

© Yukon Chang 2004

strcpy( ), versions 3 & 4

- Version 3:

```
void strcpy(char *s, char *t) // version 3
{
   while ((*s++ = *t++) != '\0')
      ;
}
```

- Version 4:

```
void strcpy(char *s, char *t) // version 4
{
   while (*s++ = *t++)
      ;
}
```

© Yukon Chang 2004

strcmp( ), array version

```
// returns <0 if s<t, 0 if s=t, >0 is s>t
int strcmp(char *s, char *t)
{
   int i;
   for (i = 0; s[i] == t[i]; i++)
      if (s[i] == '\0')
         return 0;
   return s[i] - t[i];
}
```

© Yukon Chang 2004

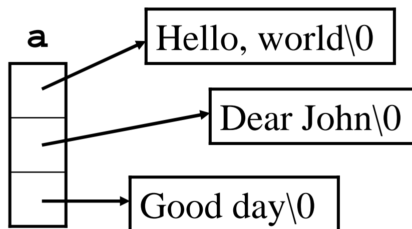
strcmp( ), pointer version

```
// returns <0 if s<t, 0 if s=t, >0 is s>t
int strcmp(char *s, char *t)
{
   for ( ; *s == *t; s++, t++)
      if (*s == '\0')
         return 0;
   return *s - *t;
}
```

© Yukon Chang 2004

Pointer Arrays

- `int a[3];` declares an array of integers
- `char a[3];` declares an array of characters
- `char *a[3];` declares an array of pointers to characters, i.e., each `a[i]` may point to a string, like this:



© Yukon Chang 2004

A Larger Example

- [K&R] provides a larger example using pointer arrays on pp. 108-110
- The example make use of earlier example code such as getline() and alloc()
- I've collected all necessary code into a program and put it on our web server (http://freefall.csie.isu.edu.tw/~c04/programs/KR108_stringquicksort.c)
- Give it a shot

© Yukon Chang 2004