Thinking in C

Foundations for Java and C++

by Chuck Allison

Introduction

- This course covers what you need to know to move on to either Java or C++
  - It is an effective but not completely rigorous course on C
  - We cover just enough to quickly prepare you to move on to the other languages

About C

- An extremely popular systems programming language, first used to write operating systems at AT&T
- Very portable & efficient
- Also popular for business & scientific applications
- C++ is a superset of C
- Java draws heavily on C++

Course Outline

1. Getting Started
2. Fundamental Data Types
3. Operators
4. Controlling Program Flow
5. Compound Data Types
6. Programming with Functions
7. Pointers 101
8. Pointers 102

1: Getting Started

- A First Look
- C Program Components
  - statements
  - comments
  - include files
  - the **`main()`** function

A First Look

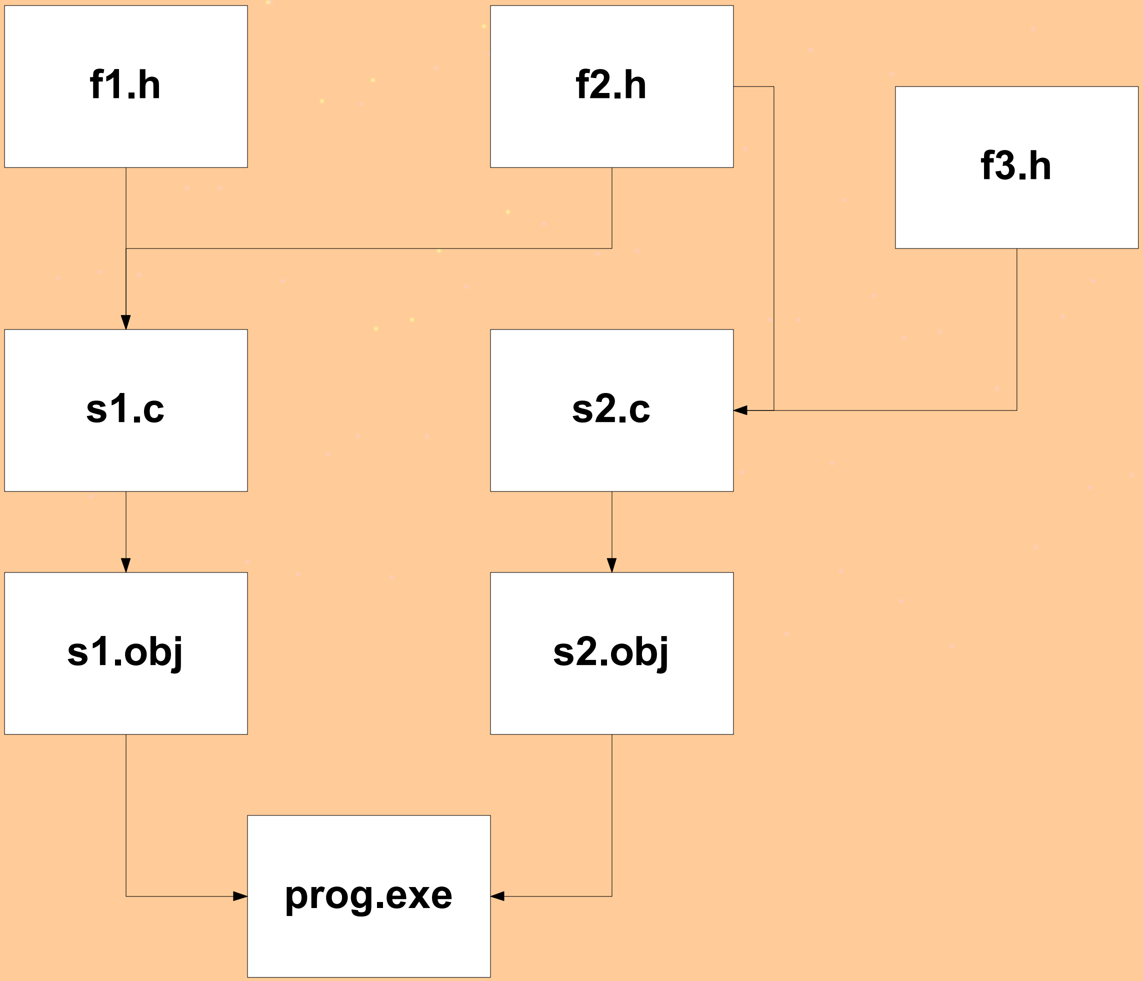
```
/* first.c: A First Program */
#include <stdio.h>

int main()
{
    puts("** Welcome to Thinking in C **");
    puts("(You'll be glad you came!)");
    return 0;
}
```

```
** Welcome to Thinking in C **
(You'll be glad you came!)
```

C Program Components

- A C program is a collection of functions
  - a.k.a. procedures, subroutines
  - and optional global variables
  - programs can span multiple files
- A function is a collection of statements
  - enclosed by **{**braces**}**
  - **`main()`** is special

Building a C Executable



Statements & Comments

- Statements contain one or more expressions
  - function calls, numeric operations, etc.
- End with a semi-colon
- Do not have to be on their own line
  - C uses a free-format syntax
- Comments are delimited by
  `/*  …  */`
  - can also span multiple lines
  - do not nest!

Include Files

- The **#include** directive inserts the text of a file into the compilation stream
  - occurs before the actual compilation
- Used mostly for function declarations and defining constants
  - no code!
- Standard library components have headers
  - using angle brackets (**#include <stdio.h>**)
- You can include any file
  - using quotes (**#include "mydefs.h"**)

Standard I/O

- Provides console, file, and memory I/O
- 3 pre-defined I/O streams:
  - `stdin`
    - "standard input" (keyboard)
  - `stdout`
    - "standard output" (screen)
  - `stderr`
    - "standard error" (screen)
- Many functions implicitly use **stdin** or **stdout**

Keyboard Input

```
/* avg.c: Averages 2 integers */
#include <stdio.h>

int main()
{
    /* Declarations must be at beginning: */
    int num1, num2;
    float sum;

    puts("Enter the 1st number:");
    scanf("%d",&num1);
    puts("Enter the 2nd number:");
    scanf("%d",&num2);

    sum = num1 + num2;
    printf("The average is %f\n", sum/2);
    return 0;
}
```

Sample Output

```
Enter the 1st number:
10
Enter the 2nd number:
23
The average is 16.500000
```

The following statement requests 2-decimal format:

```
printf("The average is %.2f\n", sum/2);
```

```
The average is 16.50
```

Summary

Getting Started

- Program source resides in one or more text files
- Source files can **`#include`** one or more header files
- Source files contain one or more functions
- Functions contains statements
- 3 pre-defined I/O streams

Exercises

Getting Started

- Compile and run the two programs in this section in your development environment.
- Experiment with redirecting the input/output from/to a text file for the second program.

2: Fundamental Data Types

- Integers
- Characters
  - just small integers
- Floating-point numbers
- Literals and Constants
- Arithmetic
  - integer vs. floating-point
- Conversions and Casts

C Types

- Data Objects
  - Fundamental (scalar)
    - are all numeric
  - Compound (composite)
    - Section 5
- Functions
  - Section 6
- All declarations must specify a type

Integers

- Come in different sizes:
  - not necessarily distinct!
  - are signed by default (char special)
- **`int`**
  - your machine's word size (at least 16 bits)
- **`short int`**
  - usually the same as **`int`** on 16-bit platforms
- **`long int`**
  - at least 32 bits
  - usually the same as **`int`** on 32-bit platforms

Characters

- Are really just integers
  - character encoding is platform-specific
  - ASCII, EBCDIC, ISO 8859, Unicode
- **`char`**
  - at least 8 bits (“byte”)
- **`wchar_t`**
  - “wide character”
  - usually the same as **`unsigned int`** (Unicode)

Numeric Limits

- Relevant values for each type
  - minimum, maximum, etc.
- Integral limits are in **<limits.h>**
- Floating-point limits are in **<float.h>**

Sample Integral Limits

```
/* limits.c: Illustrates integral limits */
#include <stdio.h>
#include <limits.h>

int main()
{
    printf("char: [%d, %d]\n", CHAR_MIN, CHAR_MAX);
    printf("short: [%d, %d]\n", SHRT_MIN, SHRT_MAX);
    printf("int: [%d, %d]\n", INT_MIN,  INT_MAX);
    printf("long: [%ld, %ld]\n", LONG_MIN, LONG_MAX);
    return 0;
}
```

```
char: [-128, 127]
short: [-32768, 32767]
int: [-2147483648, 2147483647]
long: [-2147483648, 2147483647]
```

Floating-point Types

- `float`
  - "single precision"
- `double`
  - "double precision"
  - the default
- `long double`
  - "extended precision"
  - could be same as `double`

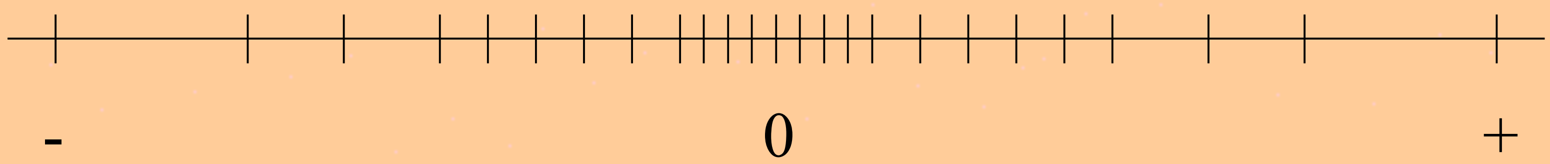
Sample Floating-point Limits

```
/* float.c: Illustrates floating-pt. limits */
#include <stdio.h>
#include <float.h>

int main()
{
    printf("radix: %d\n", FLT_RADIX);
    printf("float: %d radix digits\n",
           FLT_MANT_DIG);
    printf("\t[%g, %g]\n", FLT_MIN, FLT_MAX);
    printf("double: %d radix digits\n",
           DBL_MANT_DIG);
    printf("\t[%g, %g]\n", DBL_MIN, DBL_MAX);
    printf("long double: %d radix digits\n",
           LDBL_MANT_DIG);
    printf("\t[%Lg, %Lg]\n", LDBL_MIN,
           LDBL_MAX);
    return 0;
}
```

```
radix: 2
float: 24 radix digits
      [1.17549e-38, 3.40282e+38]
double: 53 radix digits
      [2.22507e-308, 1.79769e+308]
long double: 64 radix digits
      [3.3621e-4932, 1.18973e+4932]
```

Non-uniform Distribution

- $\pm 0.d_1d_2\ldots d_n \times \beta^e,\ d_1 \mathrel{!=} 0$
- Dense near zero
- Sparse away from zero



Missing Numbers

```
/* missing.c */
#include <stdio.h>
#include <limits.h>

main()
{
    float x = ULONG_MAX;     /* 4,294,967,295 */
    double y = ULONG_MAX;
    long double z = ULONG_MAX;

    printf("%f\n%f\n%Lf\n",x,y,z);
}
```

```
4294967296.000000 /* Oops! */
4294967295.000000
4294967295.000000
```

Literals

- `int i = 9, j = 017,`
  `k = 0x7f;`
- `char c = 'a', c2 = 97;`
- `long n = 1234567L;`
- `float x = 1.0F; // or 1.0f`
- `double y = 2.3;`
- `long double z = 4.5L; //4.5l`
- `char string[] = "hello";`

Special Character Literals

`'\n'`	newline
`'\t'`	tab
`'\0'`	null byte (ASCII 0)
`'\\'`	backslash (\)
`'\b'`	backspace
`'\r'`	carriage return
`'\f'`	form feed
`'\ddd'`	octal bit pattern
`'\xdd'`	hexadecimal

Constants

- Variables that cannot be modified
- **`const`** keyword
- Initialized with a literal:

```
const int i = 7; /* compile-time */
```

- Cannot be used as array dimensions in C
  - but can in C++!

Macro Substitution

- Using the **`#define`** directive
- Text substitution by the preprocessor
- An alternative for defining constants
  - not used much in C++
- *Can* be used as array dimensions in C

```
#define SIZE 100
int a[SIZE]; /* "int a[100];" */
```

Arithmetic

- Integer vs. floating-point
- Integer arithmetic truncates any fraction in the result:

```
int i = 2; int j = 3;
int k = i/j;        /* k == 0! */
```

- Floating-point arithmetic suffers from *roundoff error:*
  - the result may not be in the set of machine numbers
  - how it rounds is platform-specific

Promotions & Conversions

- All integral operations use either **`int`** or **`long`** arithmetic
- A numeric operation assumes the precision of its largest type operand
  - smaller operands are temporarily "widened" (or "promoted") automatically
- Beware narrowing conversions:
  - if the value is not in the range of the receiving type, the result is undefined

Casts

- User-defined explicit conversion
- Precede the expression with the target type in parentheses:

```
int i = (int)x;
```

- To force the precision of an operation, for example:

```
/* The following keeps the
   fraction: */
float x = (float)i / j;
```

Summary

Fundamental Data Types

- Built-in C data types are numeric
  - integer and floating-point
- Integer arithmetic truncates fractions
- Floating-point arithmetic is inexact
- Operands are widened as necessary
- You can force conversions with a cast

Exercise

Fundamental Data Types

- Write a program that reads a real number (with a nonzero fractional part) from the keyboard and rounds it to the nearest integer. Print out the original number and the rounded result.

3: Operators

- Mathematical
- Relational
- Logical
- Bitwise
- Assignment
- Operator Associativity and Precedence

Operator Cardinality

- All operators are either *unary* or *binary*
- Exception:
  - the “conditional” operator is *ternary:*

```
max = x > y ? x : y;
```

Mathematical Operators

- Additive: `+, -, ++, --`

```
i = j++;  /* i = j; j = j + 1; */
i = ++j;  /* j = j + 1; i = j; */
```

- Multiplicative: `*, /, %`

```
i = 10 % 3; /* = 1 */
```

Relational Operators

- Equality: `==`

  ```
  if (i == j) …;
  ```

  not

  ```
  if (i = j) …; /* error! */
  ```

- Inequality: `!=`
- Greater-than: `>, >=`
- Less-than: `<, <=`

Boolean Expressions

- Truth values
- There is no Boolean type in C
  - but Java and C++ have them
- Zero is false
- Non-zero is true

Logical Operators

- AND: `&&`

```
if (i < n && a[i] == 99) …
```

- OR: `||`

```
if (i == 2 || i == 3) …
```

- NOT: `!`

```
if (!(i == 2 || i == 3)) …
```

Bitwise Operators

- AND: &
- OR: |
- XOR: ^
- 1's Complement: ~
- Shift left: <<
- Shift right: >>

Bitwise Example

```
/* bitwise.c: Illustrates bitwise ops */
#include <stdio.h>

int main()
{
    short int n = 0x00a4;   /* 00000000 10100100 */
    short int m = 0x00b7;   /* 00000000 10110111 */

    printf("n & m == %04x\n", n & m);
    printf("n | m == %04x\n", n | m);
    printf("n ^ m == %04x\n", n ^ m);
    printf("~n == %04x\n", ~n);
    printf("n << 3 == %04x\n", n << 3);
    printf("n >> 3 == %04x\n", n >> 3);
    return 0;
}
```

Output

```
n & m == 00a4                  (0000000010100100)
n | m == 00b7                  (0000000010110111)
n ^ m == 0013                  (0000000000010011)
~n == ffffff5b           …     (1111111101011011)
n << 3 == 0520                 (0000010100100000)
n >> 3 == 0014                 (0000000000010100)
```

The following format drops the high-byte of ~n:

```
printf("~n == %04hx\n", ~n);
```

```
~n == ff5b
```

Assignment Operators

- Assignment can be combined with other binary operators:

```
+=, -=, *=, /=, %=, >>=,
<<=, &=, ^=, !=
```

```
i += 5;  /* same as i = i + 5; */
```

Operator Associativity

- Governs the order in which adjacent operations execute
- Right-to-left:
  - unary and assignment operators

  `i = j = k` same as `i = (j = k)`
- Left-to-right:
  - everything else

  `i + j + k` same as `(i + j) + k`

Operator Precedence

- Follows Mathematical Intuition (mostly)
  - unary, then multiplicative, then additive
- Unary operators are high priority
- Assignment is last (almost -- except for the comma operator)
- Beware bitwise operators!
  - When in doubt, use parentheses
- Any C book should have a precedence table

Summary

Operators

- The modulus op (%) gives the remainder from integer division
- The equality op has 2 equal signs (= =)
- In boolean contexts, 0 is false, non-zero true
- Unary and assignment ops group right-to-left, all others left-to-right
- Beware the precedence of bitwise ops

Exercise

Operators

- Write a program that reads three integers from the keyboard and prints out the sum of all the even numbers and the sum of all the odds. For example, if the numbers are 1, 2, and 3, the output is:

```
Sum of evens: 2
Sum of odds: 4
```

4: Controlling Program Flow

- Structured Programming
- Decision-making
- Repetition
- Branching

Structured Programming

- In theory, all processes can be expressed via three constructs:
  - sequences of statements
  - selection (aka alternation)
  - repetition
  - along with an arbitrary number of boolean flags
- Most languages add some sort of direct branching capability as well (e.g., goto)

Decision Making

- “Selection”
- **if-then-else**
- **case** statement
  - special case of **if-then-else**
  - selects from a set of integers

The `if` Statement

```
if (<boolean expression>)
    <statement-1>;
else
    <statement-2>;
```

- Enclose compound statements in {braces}

```
/* age.c: Comments on your age  */
#include <stdio.h>

int main()
{
    int age;
    puts("Enter your age:");
    scanf("%d", &age);
    if (age < 20)        /* no semi-colon here! */
        puts("youth");
    else if (age < 40)
        puts("prime");
    else if (age < 60)
        puts("aches and pains");
    else if (age < 80)
        puts("golden");
    else
    {
        char really;
        printf("Are you really %d?\n", age);
        scanf(" %c", &really);   /* note space! */
        if (really == 'Y' || really == 'y')
            puts("Congratulations!");
        else
            puts("I didn't think so!");
    }
    return 0;
}
```

The **`switch`** Statement

- Selects from a set of integral values
- Each case can be delimited by a **`break;`**
  - otherwise you fall through to the next case
- Don't define variables inside a **`switch`**
- The optional **`default`** case executes if none are selected.

```c
/* age2.c: Uses a switch */
#include <stdio.h>

int main()
{
    int age;
    char really;    /* note position! */
    puts("Enter your age:");
    scanf("%d", &age);
    switch(age/20)
    {
    case 0:
        puts("youth");
        break;
    case 1:
        puts("prime");
        break;
    case 2:
        puts("aches and pains");
        break;
    case 3:
        puts("golden");
        break;
    default:
        printf("Are you really %d?\n", age);
        scanf(" %c", &really);
        if (really == 'Y' || really == 'y')
            puts("Congratulations!");
        else
            puts("I didn't think so!");
    }
    return 0;
}
```

Repetition

- 3 types of loops:
  - **while (<cond>) <statement>**
  - **do <statement> while (<cond>);**
  - **for (<init>; <cond>; <iterate>) <statement>**

while Loop

```
/* Count from 1 to n */
i = 1;
while (i <= n)
{
    printf("%d ", i);
    i += 1;
}

/* A shorter version */
i = 1;
while (i <= n)
    printf("%d ", i++);
```

do-while Loop

```
/* Count from 1 to n */
i = 1;
do
    printf("%d ", i++);
while (i <= n);
```

for Loop

```
/* Count from 1 to n */
for (i = 1; i <= n; i++)
    printf("%d ", i);
```

Branching

- **`break`**
  - exits the innermost enclosing loop or **`switch`**
- **`continue`**
  - cycles a loop (i.e., jumps to test)
- **`goto`**
  - jumps to a label
  - useful for exiting nested loops & switches
  - not covered!

```c
/* branch.c: Illustrates branching */
/* Finds an odd number whose digits
   add to 7. Assumes 2 digits only. */

#include <stdio.h>
#define SIZE 5

int main()
{
    int nums[SIZE] = {10,21,32,43,54};
    int i;
    for (i = 0; i < SIZE; ++i)
    {
        int dig1, dig2;
        if (nums[i]%2 == 0)
            continue;   /* skip evens */
        dig2 = nums[i]%10;
        dig1 = nums[i]/10%10;
        if (dig1 + dig2 == 7)
        {
            printf("found %d\n", nums[i]); /* 43 */
            break;
        }
    }
    return 0;
}
```

Summary

Controlling Program Flow

- All algorithms can be expressed with simple statements, decisions, and loops
  - plus some flags, maybe
- Use branching sparingly
- Most loops are **`while`** or **`for`** loops
  - you usually test first

Exercise

Controlling Program Flow

- Rewrite the odd/even number program from the previous section to process an arbitrary number of input integers. Keep reading until the user enters a 0. Use a **`switch`** statement to determine the number's parity.

5: Compound Data Types

- Arrays
- Strings
- Structures

Arrays

- The quintessential data structure!
- Homogeneous, fixed-length sequences
  – of any type whatsoever
- Random access via the indexing operator (`[]`)
- Indexing starts at 0
  – ends at `n-1`

```c
/* reverse.c: Prints an input sequence backwards */
#include <stdio.h>
#define SIZE 20

int main()
{
    int i, n;
    int nums[SIZE];

    /* Read numbers into array.
       Stop when 0 is found */
    for (n = 0; n < SIZE; ++n)
    {
        int input;
        scanf("%d", &input);
        if (input == 0)
            break;
        nums[n] = input;
    }

    for (i = n-1; i >= 0; --i)
        printf("%d ", nums[i]);
    return 0;
}
```

Array Initialization

- Can use an *initializer list*:

```
int a[] = {10,20,30,40,50};
```

- The dimension is optional
  - if provided, it must be **>=** the number of initializers
    - those remaining are *zero-initialized*
  - if omitted, it is the same as if you entered the number of initializers for the dimension
  - for multi-dimensional arrays, must specify all but first dimension

Multi-dimensional Arrays

- Don’t really exist in C!
- C, C++, and Java allow “arrays of arrays”
  - easier to visualize than hyper-tables
- You use one set of brackets for each dimension
- Can initialize with nested initializer lists

Example

2-dimensional Array

- Consider the following 3 x 2 array:

  1 2

  3 4

  5 6

- C stores this linearly: 1 2 3 4 5 6
- The compiler interprets it as an array of 3 "arrays of 2 ints"

Example
continued

```
/* 2d.c: Illustrates a 2-d array */
#include <stdio.h>

int main()
{
    int a[][2] = {{1,2}, {3,4}, {5,6}};
    int i, j;

    for (i = 0; i < 3; ++i)
    {
        for (j = 0; j < 2; ++j)
            printf("%d ", a[i][j]);
        putchar('\n');
    }
    return 0;
}
```

1 2
3 4
5 6

3d Array Example

- The program on the next slide initializes and traverses the following 3d array:

```
1 2
3 4    == a[0]
5 6

7 8
9 0    == a[1]
1 2
```

```c
/* 3d.c: Illustrates a 3-d array */
#include <stdio.h>

int main()
{
    int a[][3][2] = {{{1,2}, {3,4}, {5,6}},
                     {{7,8}, {9,0}, {1,2}}};
    int i, j, k;

    for (i = 0; i < 2; ++i)
    {
        for (j = 0; j < 3; ++j)
        {
            for (k = 0; k < 2; ++k)
                printf("%d ", a[i][j][k]);
            putchar('\n');
        }
        putchar('\n');
    }
    return 0;
}
```

Strings

- Arrays of characters
- End with a null byte (`'\0'`) by convention
- C++ and Java have better string capabilities
- String literals implicitly provide the null terminator:

```
"hello"  becomes:
   'h' 'e' 'l' 'l' 'o' '\0'
```

String Example

```
/* strings.c: Illustrates C strings */
#include <stdio.h>
#include <string.h>

int main()
{
    char last[] = {'f','r','o','s','t','\0'};
    char first[] = "robert";
    printf("last == %s\n", last);
    printf("first == %s\n", first);
    printf("last has %d chars\n", strlen(last));
    printf("first has %d chars\n", strlen(first));
    return 0;
}

last == frost
first == robert
last has 5 chars
first has 6 chars
```

<string.h>

- Contains a number of text processing functions
- Most assume null-terminated char-arrays
- **`strcpy, strcat, memcpy`**
- **`strcmp, memcmp`**
- **`strchr, memchr, strrchr, strstr, strtok`**

In-core Formatting

```
/* incore.c: Illustrates sprintf/sscanf */
#include <stdio.h>

int main()
{
    int n = 1;
    float x = 2.0;
    char s[] = "hello";
    char string[BUFSIZ];

    sprintf(string, "%d %f j%s", n+1, x+2, s+1);
    puts(string);
    sscanf(string, "%d %f %s", &n, &x, s);
    printf("n == %d, x == %f, s == %s\n", n, x, s);
    return 0;
}
```

```
2 4.000000 jello
n == 2, x == 4.000000, s == jello
```

Structures

- Data record
- Uses the **`struct`** keyword
- Ordered Collection of arbitrary variables
  - “members”
- Access members via the `'.'` operator
- The key to objects and data abstraction!
  - data members are object attributes

Structure Example

```
/* struct.c: Illustrates structures */
#include <stdio.h>
#include <string.h>

struct Hitter
{
    char last[16];          /* 15 + 1 */
    char first[11];         /* 10 + 1 */
    int home_runs;
};                          /* Don't forget ';' !!! */

int main()
{
    struct Hitter h1 = {"McGwire", "Mark", 70};
    struct Hitter h2;
    strcpy(h2.last, "Sosa");
    strcpy(h2.first, "Sammy");
    h2.home_runs = h1.home_runs - 4;
```

```
    printf("#1 == {%s, %s, %d}\n",
           h1.last, h1.first, h1.home_runs);
    printf("#2 == {%s, %s, %d}\n",
           h2.last, h2.first, h2.home_runs);
    return 0;
}
```

```
#1 == {McGwire, Mark, 70}
#2 == {Sosa, Sammy, 66}
```

Another Structure Example

- "Hall of Fame" **`struct`**
- **`struct`** members can be any type
- Shows a **`struct`** within a **`struct`**

```
/* struct2.c: Illustrates nested structures */
#include <stdio.h>
#include <string.h>

struct Hitter
{
    char last[16];
    char first[11];
    int home_runs;
    int year;           /* new member */
};

struct HallOfFame
{
    struct Hitter players[10];
    int nPlayers;
};
```

```
int main()
{
    struct HallOfFame hr;
    int i;
    hr.nPlayers = 0;

    /* Insert first player */
    strcpy(hr.players[hr.nPlayers].last, "Ruth");
    strcpy(hr.players[hr.nPlayers].first, "Babe");
    hr.players[hr.nPlayers].home_runs = 60;
    hr.players[hr.nPlayers++].year = 1927;

    /* Insert next player */
    strcpy(hr.players[hr.nPlayers].last, "Maris");
    strcpy(hr.players[hr.nPlayers].first, "Roger");
    hr.players[hr.nPlayers].home_runs = 61;
    hr.players[hr.nPlayers++].year = 1961;
```

```
    /* Print players in hr: */
    for (i = 0; i < hr.nPlayers; ++i)
        printf("%d: {%s, %s, %d}\n",
               hr.players[i].year,
               hr.players[i].last,
               hr.players[i].first,
               hr.players[i].home_runs);
    return 0;
}
```

1927: {Ruth, Babe, 60}
1961: {Maris, Roger, 61}

Summary

Compound Data Types

- Arrays start indexing at 0
- Multi-dim. arrays are “arrays of arrays”
- Strings are arrays of **`char`**
  - delimited by a null byte
- Structure are collections of data members
- Arrays and structures support brace-delimited initializer lists

Exercise

Compound Data Types

- Define an Employee structure that has members last name, first name, title, and salary.
- Write a program that prompts the user for an arbitrary number of Employees, and stores them in an array of Employee. When the user enters an empty string for the last name, print out the list of Employees.

6: Programming with Functions

- Procedural Programming
- Value Semantics
- Function Prototypes
- Scope
- Automatic, static, and global variables

Procedural Programming

- Uses the procedure (or function) as the basic unit of program architecture
  - a program is just a collection of functions
  - a 1950's technology
- Functional Decomposition
  - partitions a system into its key processes
  - a 1970's technology
  - precursor to object-oriented programming

Functions in C

- A Collection of Statements
  - defined at file scope
  - each has a unique name
  - enclosed in {braces}
  - performs some well-defined operation
  - may take arguments
  - may return a value

```c
/* fun1.c: Illustrates a C function */
#include <stdio.h>

float avg(int n, int m)
{
    return (n + m) / 2.0;
}

int main()
{
    int x, y;

    puts("Enter the first number:");
    scanf("%d", &x);
    puts("Enter the second number:");
    scanf("%d", &y);
    printf("The average is %.2f\n", avg(x,y));
    return 0;
}
```

```
Enter the 1st number:
11
Enter the 2nd number:
12
The average is 11.50
```

Value Semantics

- Arguments are passed by value
  - each formal parameter gets a *copy* of its argument's value
  - the calling argument is not affected
- The result is returned by value
  - the calling expression gets a temporary:

```
a = b + avg(c,d);
```

The `void` keyword

- Used to indicate the absence of a return value
  - i.e., the function is a *procedure:*

    ```
    void f(int x, float y) {…}
    ```

- Or to disallow arguments:

```
void title(void){
    printf("Welcome to …");
}
…
title();
```

```c
/* fun2.c: Shows return with void */
/* Also illustrates an array parameter */
#include <stdio.h>

void print_ints(int nums[], int n)
{
    int i;
    if (n <= 0)
        return;
    for (i = 0; i < n; ++i)
    {
        if (i > 0)
            putchar(',');
        printf("%d", nums[i]);
    }
}

int main()
{
    int a[] = {9,0,2,1,0};
    print_ints(a, 5);
    return 0;
}
```

9,0,2,1,0

Using Functions

- The number and types of the calling arguments should match a function's formal parameters
- The compiler can detect usage errors at compile time
- Guideline: either *define* or *declare* a function before its first use
  - required in C++ and Java

Function Prototypes

- A function declaration
  - signature (name + parameter types)
  - return type
- Lets you define a function in a separate file from where it's called
  - the basis for reusable libraries

```c
/* fun3.c: Illustrates a function prototype */
#include <stdio.h>

float avg(int, int);    /* Prototype */

int main()
{
    int x, y;

    puts("Enter the first number:");
    scanf("%d", &x);
    puts("Enter the second number:");
    scanf("%d", &y);
    printf("The average is %.2f\n", avg(x,y));
    return 0;
}

float avg(int n, int m)
{
    return (n + m) / 2.0;
}
```

Function Libraries

- Prototypes in header files
- Implementation in object code files
- A poor man's module mechanism
  - import declarations with **`#include`**
  - linker binds the needed code
- That's how the standard C library works!
- C++ uses "namespaces"
- Java uses "packages"

```
/* file mystuff.h */
float avg(int, int);
…
•

/* file mystuff.c */
float avg(int n, int m)
{
    return (n + m) / 2.0;
}
…
•

/* file fun3.c */
#include <stdio.h>
#include "mystuff.h"

int main()
{
    …
    … avg(x, y) …
    …
    return 0;
}
```

Scope

- Where an identifier is visible
- Three basic types:
  - local (or “block”) scope
  - file scope
  - program scope
  - (there are others, but they’re beyond the “scope” of this course:-)

Local Scope

- Within a block (i.e., a set of {braces})
  - functions
  - loops and if-statements
- Visible from the point of declaration until the end of the block

File Scope

- The region outside of any function
- Visible from declaration to end of file

```c
/* scope.c */
#include <stdio.h>

int i = 3;      /* A "global" variable */

int main()
{
    int j;
    printf("%d\n",i);
    for (j = 0; j < i; ++j)
    {
        int i = 99;
        printf("%d\n",i);
        /* New block follows: */
        {
            int i = j;
            printf("%d\n",i);
        }
    }
    return 0;
}
```

3
99
0
99
1
99
2

Automatic Variables

- Allocated on the program stack
- Any variable defined in a block
  - without the `static` keyword
- Reinitialized each time execution enters its block

Example

Automatic Storage

```
int min(int nums[], int size)    /* auto variables */
{
    int i, small = nums[0];      /* auto variables */
    for (i = 1; i < size; ++i)
        if (nums[i] < small)
            small = nums[i];
    return small;
}
```

Static Variables

- Reside in a special data area
  - (static) data segment
- Any variable defined outside a block
  - "file scope"
- Any variable declared inside a block with the `static` keyword
- Initialized only once
  - at program startup

Example

Static Storage

```
/* static.c */
#include <stdio.h>

int count()
{
    static int n = 0;
    return ++n;
}

int main()
{
    int i;
    for (i = 0; i < 5; ++i)
        printf("%d ", count());
    return 0;
}
```

1 2 3 4 5

Global Variables

- Have "program scope"
- Accessible outside their source file
- Defined at file scope
  - without the **`static`** keyword
- Use the **`extern`** keyword to refer to global variables in other files
- Static + file scope = private to its file

```
/* file1.c */

int i = 10;            /* global */
static int j = 20;     /* private */

int get_j(void)
{
    return j;
}


/* file2.c */
#include <stdio.h>

int main()
{
    extern int i;
    /* extern optional for functions: */
    int get_j(void);

    printf("i == %d\n", ++i);        /* i == 11 */
    printf("j == %d\n", get_j());    /* j == 20 */
    return 0;
}
```

Information Hiding

- Using file statics in C
- Protects data
- Separates interface from implementation
- Fundamental principle of object-oriented programming
- C++ and Java support it much better

Stack Example

- **stack.h**
  - user function declarations
- **stack.c**
  - function implementation
  - private definitions
- **tstack.c**
  - a test program

```c
/* stack.h: Declarations for a stack of ints */

#define STK_ERROR -32767

void stk_push(int);
int  stk_pop(void);
int  stk_top(void);
int  stk_size(void);
int  stk_error(void);
```

```c
/* stack.c: implementation */
#include "stack.h"

/* Private data: */
#define MAX 10              /* stack limit */
static int error = 0;       /* error flag */
static int data[MAX];       /* the stack */
static int ptr = 0;         /* stack pointer */

/* Function implementation */
void stk_push(int x)
{
    if (ptr < MAX)
    {
        data[ptr++] = x;
        error = 0;
    }
    else
        error = 1;
}
```

```
int stk_pop(void)
{
    if (ptr > 0)
    {
        int x = data[--ptr];
        error = 0;
        return x;
    }
    else
    {
        error = 1;
        return STK_ERROR;
    }
}

int stk_size(void)
{
    return ptr;
}
```

```
int stk_top(void)
{
    if (ptr > 0)
    {
        error = 0;
        return data[ptr-1];
    }
    else
    {
        error = 1;
        return STK_ERROR;
    }
}

int stk_error(void)
{
    return error;
}
```

```
/* tstack.c: Tests the stack of ints */
#include "stack.h"
#include <stdio.h>

int main()
{
    int i;

    /* Populate stack */
    for (i = 0; i < 11; ++i)
        stk_push(i);
    if (stk_error())
        puts("stack error");
    printf("The last element pushed was %d\n",
           stk_top());

    /* Pop/print stack */
    while (stk_size() > 0)
        printf("%d ", stk_pop());
    putchar('\n');
    if (!stk_error())
        puts("no stack error");
    return 0;
}
```

```
stack error
The last element pushed was 9
9 8 7 6 5 4 3 2 1 0
no stack error
```

Summary

Programming with Functions

- A function is a named collection of statements
  - may take arguments
  - may return a value
- C functions use value semantics
- Scope is a variable’s visible region
  - local, file, program
- You hide variables with file statics

Exercise

Programming with Functions

- In this exercise you will split the Employee exercise from the last section into 3 files: **employee.h**, **employee.c**, **lab6.c** (similar to the stack example). **Employee.h** declares 3 functions (see the next slide). You need to provide **employee.c**, which will contain the Employee structure definition and any needed private data, and the implementation of the functions declared in **employee.h**. Use the **lab6.c** provided on the subsequent slide.

Exercise

employee.h

```
/* employee.h */

/* addEmployee reads each field from standard
 * input into the next available Employee slot,
 * as in the exercise in the previous section.
 * It returns the index of the Employee
 * just added, or -1 if the array is full */
int addEmployee(void);

/* printEmployee also returns the index of the
 * Employee just printed, or -1 if the index i
 * is invalid */
int printEmployee(int i);

/* Does what it says: */
int numEmployees(void);
```

Exercise

lab6.c

```
/* lab6.c */
#include "employee.h"
#include <stdio.h>

int main() {
    int i;

    /* Fill Employee array: */
    while (addEmployee() != -1)
        ;

    /* Print each Employee: */
    for (i = 0; i < numEmployees(); ++i) {
        printEmployee(i);
        putchar('\n');
    }
    return 0;
}
```

7: Pointers 101

- Indirection
- Simulating Call-by-reference
- Command-line Arguments
- Heap variables

Pointers

- A pointer, like an integer, holds a number
  - interpreted as the address of another object
- Must be declared with its associated type:
  - e.g., “pointer to integer”, “pointer to character”, etc.
- Useful for:
  - dynamic objects (allocated from the heap)
  - for direct machine access (such as mapped memory)

Pointer Indirection

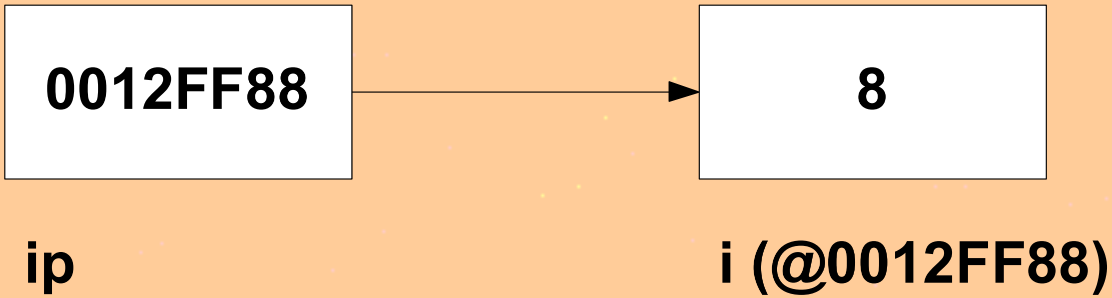
- Accessing an object through a pointer is called *indirection*
- The “address-of” operator (`&`) obtains an object’s address
- The “de-referencing” operator (`*`) refers to the object pointed at

Indirection Example

```
/* indirect.c: Illustrate indirection */
#include <stdio.h>

int main()
{
    int  i = 7;
    int* ip = &i;

    printf("Address %p contains %d\n", ip, *ip);
    *ip = 8;
    printf("Now address %p contains %d\n", ip, *ip);
    return 0;
}
```

```
Address 0012FF88 contains 7
Now address 0012FF88 contains 8
```

Pointer Diagram



Reference Semantics

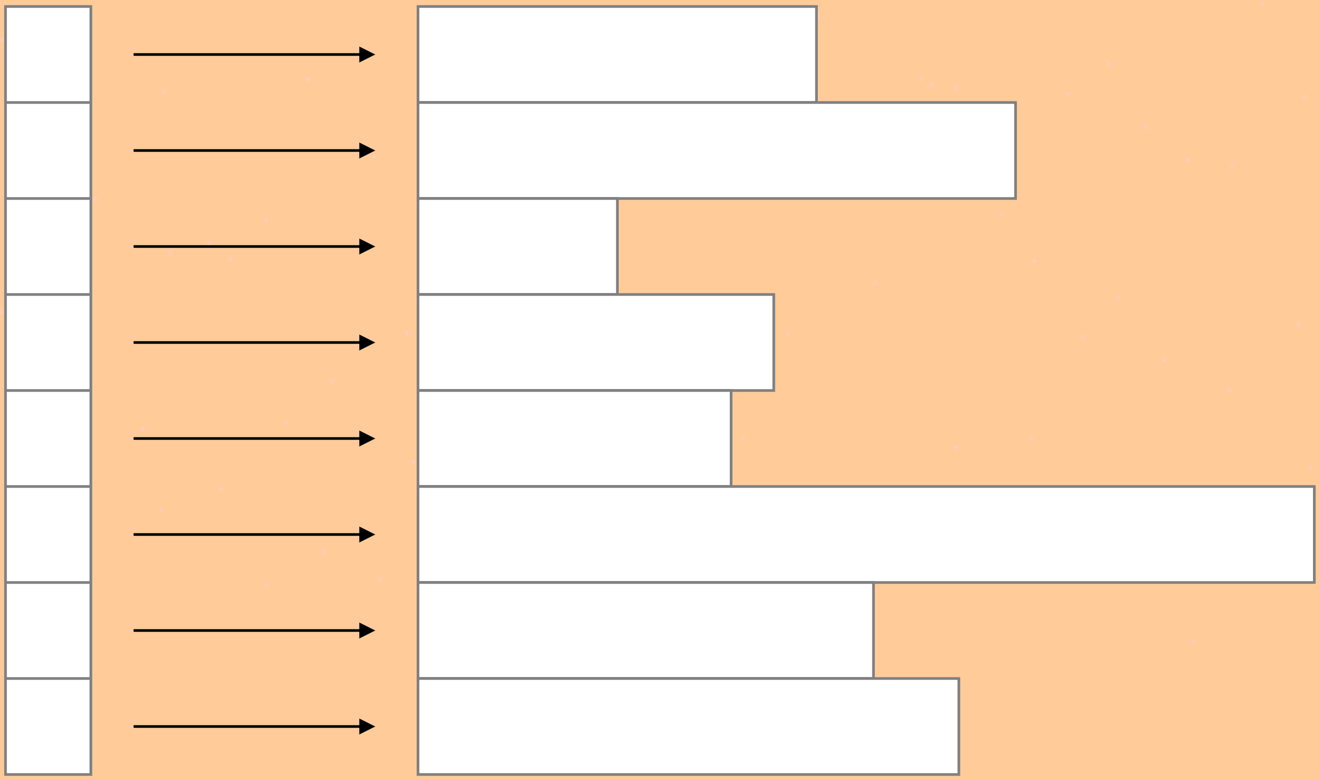
- Where a formal parameter is just an alias for the calling argument
  - changing the parameter changes the original argument
- Not supported directly in C
  - nor in Java
  - but C++ supports it
- In C we fake it with pointers

Simulating Reference Semantics

```
/* swap.c */
#include <stdio.h>

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

int main()
{
     int i = 1, j = 2;
     swap(&i, &j);
     printf("i == %d, j == %d\n", i, j);
     return 0;
}
```

i == 2, j == 1

Ragged Arrays



Command-line Arguments

- Optional arguments to main

```
int main(int argc, char* argv[]) {
    …
}
```

- **argv** is a ragged array

Example

Command-line Arguments

```
/* echo.c: Echoes command-line args */
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i;
    for (i = 0; i < argc; ++i)
        puts(argv[i]);
    return 0;
}
```

```
D:\TIC> .\echo hello there
D:\TIC\echo.exe
hello
there
```

The **`NULL`** Pointer

- A special value
  - the address 0
- Doesn't point anywhere
- Can use to compare to other pointers
  - e.g., as a sentinel
  - returned by selected library functions
- Cannot de-reference **`NULL`**

Heap Variables

- Accessed indirectly through a pointer
  - returned by `malloc()`
- Reside in a unique data area
  - often called the "heap" or "dynamic storage"
- You give the memory back when done
  - via `free()`
- Useful when you don't know everything ahead of time
  - like how big an array needs to be

C Heap Functions

- Defined in `<stdlib.h>`:

```
void* malloc(size_t size);
void  free(void *ptr);
void* calloc(size_t nelems,
             size_t elem_size);
void* realloc(void *ptr, size_t size);
```

```
/* reverse2.c: Prints lines in reverse
 *             order from input
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXWIDTH 81
#define MAXLINES 100

int main()
{
    char* lines[MAXLINES];
    char line[MAXWIDTH];
    int i, n;

    /* Store in a ragged array */
    for(n = 0;
        n < MAXLINES && gets(line) != NULL; ++n)
    {
        if ((lines[n] = malloc(strlen(line)+1))
             == NULL)
           exit(1);
        strcpy(lines[n], line);
    }
```

```
    /* Print in reverse order */
    for (i = 0; i < n; ++i)
    {
        puts(lines[n-i-1]);
        free(lines[n-i-1]);
    }
    return 0;
}
```

The `sizeof` operator

- Gives the size of a variable or type in bytes
- A compile-time operator
- Array size idiom:

```
int n = sizeof a / sizeof a[0];
```

- Must use parentheses with types:

```
float* p =
  malloc(sizeof(float));
```

structs on the Heap

- Use **`sizeof`**, as usual:

```
struct Employee* p =
   malloc(sizeof(struct Employee));
```

- Accessing members uses a messy syntax:

```
(*p).age = 47;
```

- Alternate syntax (the `->` operator):

```
p->age = 47;
```

structs as Arguments

- You usually pass a struct's address
  - into a pointer parameter, of course
  - saves time and space
- Equivalent to Java semantics
  - Java never passes objects by value
  - passes a pointer instead
  - but it's all invisible to you

```
/* structarg.c: Passes a struct by address */
#include <stdio.h>

struct Date
{
    int year;
    int month;
    int day;
};

void printDate(struct Date* p)
{
    printf("%2d/%2d/%02d", p->month, p->day,
                           p->year);
}

int main()
{
    struct Date d = {98, 10, 2};
    printDate(&d);
    return 0;
}
```

10/ 2/98

Arrays as Arguments

- Inefficient to pass entire array
- A pointer to the first element is passed
- **`char*`** and **`char[]`** mean the same as a function parameter
- More on this for C++ programmers in Section 8b.

Exercise

Pointers

- As in the previous section, a header file (**employ2.h**) and a test file (**lab7.c**) follow this slide. The test program creates **Employee** objects and exercises the various functions declared in **employ2.h**. Provide the implementation for these functions in a file named **employ2.c**. The function **createEmployee(…)** allocates an **Employee struct** on the heap and initializes it with its arguments, and returns the pointer returned from **malloc( )**.

Exercise

employ2.h

```
/* employ2.h */
#ifndef EMPLOYEE_H
#define EMPLOYEE_H

struct Employee
{
    char last[16];
    char first[11];
    char title[16];
    int salary;
};

struct Employee*
    createEmployee(char*,char*,char*,int);
char* getLast(struct Employee*);
char* getFirst(struct Employee*);
char* getTitle(struct Employee*);
int getSalary(struct Employee*);
void setLast(struct Employee*, char*);
void setFirst(struct Employee*, char*);
void setTitle(struct Employee*, char*);
void setSalary(struct Employee*, int);
void printEmployee(struct Employee*);

#endif
```

Exercise

lab7.c

```
/* lab7.c */
#include "employ2.h"
#include <stdio.h>
#include <stdlib.h>

#define MAXEMPS 5

int main()
{
    struct Employee* emps[MAXEMPS];
    struct Employee* p;
    int i, nemps = 0;

    emps[nemps++] = createEmployee("Mantle", "Mickey",
                                   "Outfielder", 58);
    emps[nemps++] = createEmployee("Maris", "Roger",
                                   "Shortstop", 60);
    if (emps[nemps-1]->salary != 61)
        emps[nemps-1]->salary = 61;
```

```
    p = createEmployee("", "", "", 0);
    setLast(p, "Kaline");
    setFirst(p, "Al");
    setTitle(p, "Outfielder");
    setSalary(p, 52);
    emps[nemps++] = p;

    for (i = 0; i < nemps; ++i)
    {
        printEmployee(emps[i]);
        putchar('\n');
        free(emps[i]);
    }
    return 0;
}
```

```
{Mantle,Mickey,Outfielder,58}
{Maris,Roger,Shortstop,61}
{Kaline,Al,Outfielder,52}
```

8: Pointers 102

- Odds and ends
- Pointers to Pointers
- Pointer Arithmetic
- Pointers and Arrays
- Pointers and **`const`**
- Generic Pointers (**`void*`**)
- Pointers to functions
- Incomplete Types

Odds and Ends

- Unsigned integers
- **`typedef`**

Unsigned Integers

- Each integral type has an unsigned version
  - **`unsigned`** keyword
- Holds only non-negative numbers
- Uses sign bit in mantissa
  - max is twice as large as signed version
  - min is zero
- Often used as array indices or to hold sizes
  - **`size_t`**

```c
/* ulimits.c: Illustrates unsigned limits */
#include <stdio.h>
#include <limits.h>

int main()
{
    printf("char: [0, %u]\n", UCHAR_MAX);
    printf("short: [0, %u]\n", USHRT_MAX);
    printf("int: [0, %u]\n", UINT_MAX);
    printf("long: [0, %lu]\n", ULONG_MAX);
    return 0;
}
```

```
char: [0, 255]
short: [0, 65535]
int: [0, 4294967295]
long: [0, 4294967295]
```

Sign Extension

```
/* bitwise2.c: Illustrates sign extension */
#include <stdio.h>

int main()
{
    unsigned int n = 0x00a4; /* 000... 10100100 */
    int m = 0x00b7;          /* 000... 10110111 */

    printf("~n == %08x\n", ~n);
    printf("~m == %08x\n", ~m);
    printf("(~n) >> 4 == %08x\n", (~n) >> 4);
    printf("(~m) >> 4 == %08x\n", (~m) >> 4);
    return 0;
}

~n == ffffff5b
~m == ffffff48
(~n) >> 4 == 0ffffff5
(~m) >> 4 == fffffff4
```

typedef

- “Type definition” facility
- Defines *synonyms* for other types
  - an abstraction mechanism:

    `typedef unsigned int size_t;`
  - a shorthand
    - e.g., eliminates proliferation of **`struct`**
    - implicit in C++

Structure Tag `typedef` Idiom

```
struct Foo
{
    int x;
    int y;
};

typedef struct Foo Foo;

Foo f;    /* struct not needed */
```

Pointers to Pointers

- A pointer can point to any type
- Including another pointer type
- Can nest to 12 levels of indirection
  - 2 is the practical limit
  - less common in C++

```
/* pptr.c: Illustrates pointers to pointers */
#include <stdio.h>

int main()
{
    int i = 7, *ip = &i, **ipp = &ip;
    printf("Address %p contains %d\n", ip, *ip);
    printf("Address %p contains %p\n", ipp, *ipp);
    printf("**ipp == %d\n", **ipp);
    return 0;
}
```

```
Address 0012FF88 contains 7
Address 0012FF84 contains 0012FF88
**ipp == 7
```

Pointer Arithmetic

- You can add/subtract an integer to/from a pointer
- The pointer advances/retreats that number of *elements*
  - not bytes
- Subtracting two pointers yields the number of *elements* between them

Pointer Arithmetic Example

```
/* parith.c: Illustrates pointer arithmetic */
#include <stdio.h>
#include <stddef.h>     /* for ptrdiff_t */

int main()
{
    float a[] = {1.0, 2.0, 3.0}, *p = &a[0];
    ptrdiff_t diff;

    printf("sizeof(float) == %u\n",sizeof(float));
    printf("p == %p, *p == %f\n", p, *p);
    p += 2;
    printf("p == %p, *p == %f\n", p, *p);
```

```
sizeof(float) == 4
p == 0012FF80, *p == 1
p == 0012FF88, *p == 3
```

```
    diff = p - a;      /* a == &a[0] */
    printf("diff == %ld\n", diff);
    diff = (char*)p - (char*)a;
    printf("diff == %ld\n", diff);
    return 0;
}
```

diff == 2
diff == 8

Pointers and Arrays

- The name of an array becomes a pointer to its `1`st element in most expressions
- In other words: **`a`** is the same as **`&a[0]`**
- Or, in other words: **`*a == a[0]`**
- More generally: **`a + i == &a[i]`**
- Or: **`*(a + i) == a[i]`**
- You can even say **`i[a]`**!
  - but don't!

Pointer and Arrays Example

```
/* parray.c */
#include <stdio.h>

int min(int* nums, int size)    /* or int nums[] */
{
    int* end = nums + size;     /* past the end */
    int small = *nums;
    while (++nums < end)
        if (*nums < small)
            small = *nums;
    return small;
}

main()
{
    int a[] = {56,34,89,12,9};
    printf("%d\n", min(a, 5)); /* 9 */
}
```

Question: What is **`sizeof a`**? What is **`sizeof nums`**?

Pointers and Multi-dimensional Arrays

- Consider **`int a[3][4]`**:
  - **`a`** is an array of 3 arrays of 4 ints
  - **`a[i]`** is an array of 4 ints
  - **`sizeof a[i] == 16 (4*sizeof(int))`**
  - How would you declare a pointer to **`a[0]`**?

```c
#include <stdio.h>

int main()
{
    int a[][4] = {{0,1,2,3},{4,5,6,7},{8,9,0,1}};
    /* Pointer to array of 4 ints: */
    int (*p)[4] = a;
    int i;
    size_t nrows = sizeof a / sizeof a[0];
    size_t ncols = sizeof a[0] / sizeof a[0][0];

    printf("sizeof(*p) == %d\n",sizeof *p);
    for (i = 0; i < nrows; ++i)
    {
        int j;
        for (j = 0; j < ncols; ++j)
            printf("%d ",p[i][j]);
        putchar('\n');
    }
    return 0;
}
```

```
sizeof(*p) == 16
0 1 2 3
4 5 6 7
8 9 0 1
```

```
/* 3d version */
#include <stdio.h>

int main()
{
    int a[][3][4] = {{{0,1,2,3},{4,5,6,7},{8,9,0,1}},
                     {{2,3,4,5},{6,7,8,9},{0,1,2,3}}};
    int (*p)[3][4] = a;
    int i;
    size_t ntables = sizeof a / sizeof a[0];
    size_t nrows   = sizeof a[0] / sizeof a[0][0];
    size_t ncols   = sizeof a[0][0] / sizeof a[0][0][0];

    printf("sizeof(*p) == %d\n",sizeof *p);
    for (i = 0; i < ntables; ++i)
    {
        int j;
        for (j = 0; j < nrows; ++j)
        {
            int k;
            for (k = 0; k < ncols; ++k)
                printf("%d ",p[i][j][k]);
            putchar('\n');
        }
        putchar('\n');
    }
    return 0;
}
```

```
sizeof(*p) == 48
0 1 2 3
4 5 6 7
8 9 0 1

2 3 4 5
6 7 8 9
0 1 2 3
```

Can you see a pattern here?

Pointers and `const`

- Prevents changing a pointer or its contents
- **`const`** *before* the asterisk means that the *contents* is **`const`**
- **`const`** *after* the asterisk means that the *pointer* is **`const`**
- **`volatile`** observes the same syntax

```
const char* p1;         /* *p1 = 'c' illegal; ++p1 OK */
char* const p2;         /* *p2 = 'c' OK; ++p2 illegal */
const char* const p3;   /* no changes at all allowed */
```

Generic Pointers

- "Pointer to void" (**`void*`**)
- Can assign any pointer to or from a **`void*`**
  - Need to cast back to use the item pointed at in C++
  - cannot de-reference a **`void*`**
- Useful for:
  - treating any object as a sequence of bytes
  - implementing generic containers

Generic Pointer Example

```
void* memcpy(void* target, const void* source,
             size_t n)
{
    /* Copy any object to another */
    char* targetp = (char*) target;
    const char* sourcep = (const char*) source;
    while (n--)
        *targetp++ = *sourcep++;
    return target;
}

int main()
{
    float x = 1.0, y = 2.0;
    memcpy(&x, &y, sizeof x);
    printf("%d\n", x);
    return 0;
}
```

2

```
/* qsort.c: Illustrates qsort */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int comp(const void*, const void*);

int main()
{
    char* strings[] =
        {"how", "now", "brown", "cow"};
    const unsigned int nstr =
        sizeof strings / sizeof strings[0];
    unsigned int i;

    qsort(strings, nstr, sizeof strings[0], comp);
    for (i = 0; i < nstr; ++i)
        puts(strings[i]);
    return 0;
}
```

```
int comp(const void* p1, const void* p2)
{
    char* a = *(char **) p1;
    char* b = *(char **) p2;
    return strcmp(a,b);
}
```

```
brown
cow
how
now
```

Pointers to Functions

- A function name without its argument list becomes a pointer to the function
- Allows passing pointers as arguments
  - like **`comp`** in the call to **`qsort`**
- Explicit indirection not required
  - don't need to use **`&`** or **`*`** operators

Function Pointer Syntax

```
/* fptr.c */
#include <stdio.h>

int main()
{
    int i = 1;
    int (*fp)(const char*, ...) = printf;
    fp("i == %d\n", i);
    /* or (*fp)(" i == ...); */
    return 0;
}
```

i == 1

Using typedef

```
/* fptr2.c */
#include <stdio.h>

int main()
{
    typedef void (*ftype)(const char*, ...);
    int i = 1;
    ftype fp = (ftype) printf;
    fp("i == %d\n", i);
    return 0;
}
```

Functions and Menu Choices

```
1) Retrieve
2) Insert
3) Update
4) Quit
```

```c
/* menu.c:  Illustrates an
   array of function ptrs */
#include <stdio.h>

/* You must provide definitions for these: */
extern void retrieve(void);
extern void insert(void);
extern void update(void);
/* Returns keypress: */
extern int show_menu(void);
int main()
{
    int choice;
    void (*farray[])(void) =
        {retrieve,insert,update};
    for (;;)
    {
        choice = show_menu();
        if (choice >= 1 && choice <= 3)
            /* Process request: */
            farray[choice-1]();
        else if (choice == 4)
            break;
    }
    return 0;
}
```

Incomplete Types

- Type whose size is undetermined at point of declaration
- Array of unknown size:
  - `extern int a[]; /* Not a parameter */`
- Structure that hasn't been fully declared:
  - `struct foo; /* Forward declaration */`
- Useful for information hiding

Canonical Example

- A Stack
- User only sees incomplete type and interface functions
  - as defined in **`stack8b.h`**
- Implementation is totally hidden
  - in **`stack8b.c`**

```c
/* stack8b.h: Declarations
   for a stack of ints */

#define STK_ERROR -32767

/* Incomplete type */
typedef struct Stack Stack;

Stack* stk_create(int);
void stk_push(Stack*, int);
int  stk_pop(Stack*);
int  stk_top(Stack*);
int  stk_size(Stack*);
int  stk_error(Stack*);
void stk_destroy(Stack*);
```

```c
/* stack8b.c: implementation */
#include "stack8b.h"
#include <stdlib.h>

/* Complete the Stack type */
struct Stack
{
    int *data;
    int size;
    int ptr;
    int error;
};

Stack* stk_create(int stk_size)
{
    Stack* stkp = malloc(sizeof(Stack));
    stkp->size = stk_size;
    stkp->data = malloc(stkp->size*sizeof(int));
    stkp->ptr = stkp->error = 0;
    return stkp;
};
```

(The rest is on the CD ...)

Summary

Pointers 102

- Pointers can refer to any C type
- An array name decays into a pointer to its first element
  - – except when used with **`sizeof`**
- **`const`** can modify a pointer or its referent
- **`void*`** supports generic function arguments
- Function pointers allow functions as arguments
- Incomplete types support information hiding

Exercise 1

Pointers 102

- Write a function:

```
void inspect(const void*, size_t);
```

which interprets its first argument as an array of bytes,and the second is the size of the first argument in bytes. This function prints the contents of each byte of its first argument in hexadecimal. Test it on several different types of arguments (**`int`**, **`float`**, etc.)

Exercise 2

Pointers 102

- Move the definition of the **`Employee struct`** in the exercise of Section 7 to the implementation file (**employ2.c**), leaving **`Employee`** as an incomplete type in **employ2.h**. Use a **`typedef`** to eliminate the need for repetition of the **`struct`** keyword. You'll have to use access functions in lieu of referencing salary directly in the test file.

Exercise 3

Pointers 102

- *Optional.* Extend the 3d pointer-to-array example in this section to 4 dimensions!

Congratulations!

You’re ready to tackle C++