Programming in C

Devendra Tewari

December 05, 2005

Objectives

Course Objectives

- Learn the core elements of ANSI C
- Get to know tools for building C programs on GNU/Linux
- Learn to read and debug C programs
- Build useful utilities to reuse in your own programs

Introduction

History

- Originally designed and implemented by Dennis Ritchie on a DEC PDP-11
- Influenced by B written by Ken Thompson in 1970
- First C standard in 1988 by ANSI (C89)
- Adopted by ISO in 1990 (C90)
- Most recent standard C99 by ISO
- Compiled language
- Source code portable

C program – hello.c

```c
#include <stdio.h>

 /* function main - print hello world */
int
main()
{
    printf("hello world!\n");
    return 0;
}
```

C program structure

- Multi-line comments begin with /* and end with */, these are called delimiters
- # is used to begin pre-processor directives
- Execution of a C program begins at function main
  - main can return an int value to the operating system otherwise it should return void
- Code blocks and function bodies begin with { and end with }
- C statements end with a semicolon ;

Executing hello.c

- Use GCC

  ```
  gcc hello.c -o hello -Wall
  ```

- Without -o option the output is named a.out
- Execute

  ```
  ./hello
  ```

- Output

  ```
  hello world!
  ```

Compilation process

- Compiler produces the executable by performing the following steps
  - Pre-processing
  - Compilation and assembly
  - Linking

Pre-processing

- Conceptual first step in compilation
- Two tasks commonly performed
- File inclusion with `#include` directive

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

- Macro substitution with `#define` directive

```
#define pf printf
pf("hello world!")
```

Compilation and assembly

- Lexical and semantic analysis to generate intermediate code
- Transform the intermediate code to assembly or machine code
- Creating an object file using GCC

  `gcc hello.c -c`

  - The -c option tells GCC not to perform linking
  - A file called `hello.o` is produced

Linking

- Linking combines all the object files and required library code to produce a single executable

```
gcc hello.o -o hello
```

Multiple source files – hello.c

```c
#include "print.h"
/* Function main - Print hello world */
int
main()
{
  print_hello();
}
```

Multiple source files – print.c

```c
#include <stdio.h>
#include "print.h"
void print_hello()
{
  printf("printing: hello world!\n");
}
```

Multiple source files – print.h

```
#ifndef _PRINT_H_
#define _PRINT_H_

extern void print_hello();

#endif //_PRINT_H_
```

- `#ifndef` / `#endif` prevents pre-processor from including same file twice

Simple compilation

- Compile and link using gcc

  ```
  gcc hello.c print.c -o hello
  ```

- Execute

  ```
  ./hello
  ```

- Output

  ```
  printing: hello world!
  ```

Complex Compilation - With Error

- Compile `hello.c` using gcc; produces `hello.c`

```
gcc hello.c -c
```

- Compile `print.c`; produces `print.o`

```
gcc print.c -c
```

- Link using gcc

```
gcc hello.o -o hello
```

```
hello.o(.text+0x27):hello.c: undefined reference to `_p
collect2: ld returned 1 exit status
```

Complex Compilation – Correcting the Error

- Link using gcc

```
gcc hello.o print.o -o hello
```

- Using ld

```
ld -o hello \
/lib/crt0.o -L/opt/gcc.3.3/lib/gcc-lib/i586-pc-interix3
hello.o print.o -lgcc -lc -lpsxdll -v
```

  - Calling gcc with `-v` switch shows how
  - Note path to `libgcc.a` in my Windows SFU installation

Using a Graphical Debugger

- DDD is a graphical debugger for X Windows and it uses gdb, the command line debugger
- Re-compile source code with extra debug information for gdb

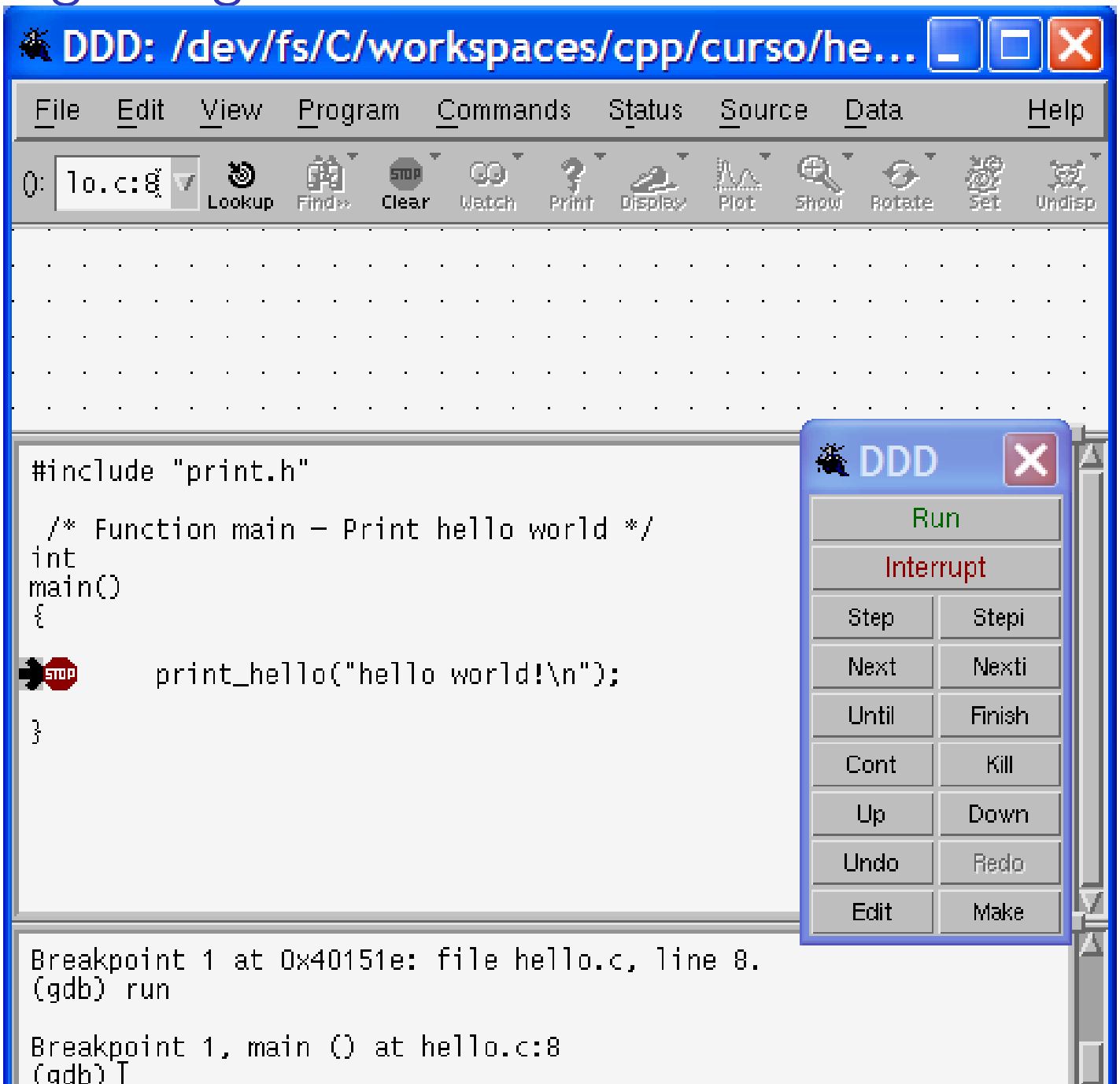
  ```
  gcc -g hello.c print.c -o hello
  ```

- Execute ddd

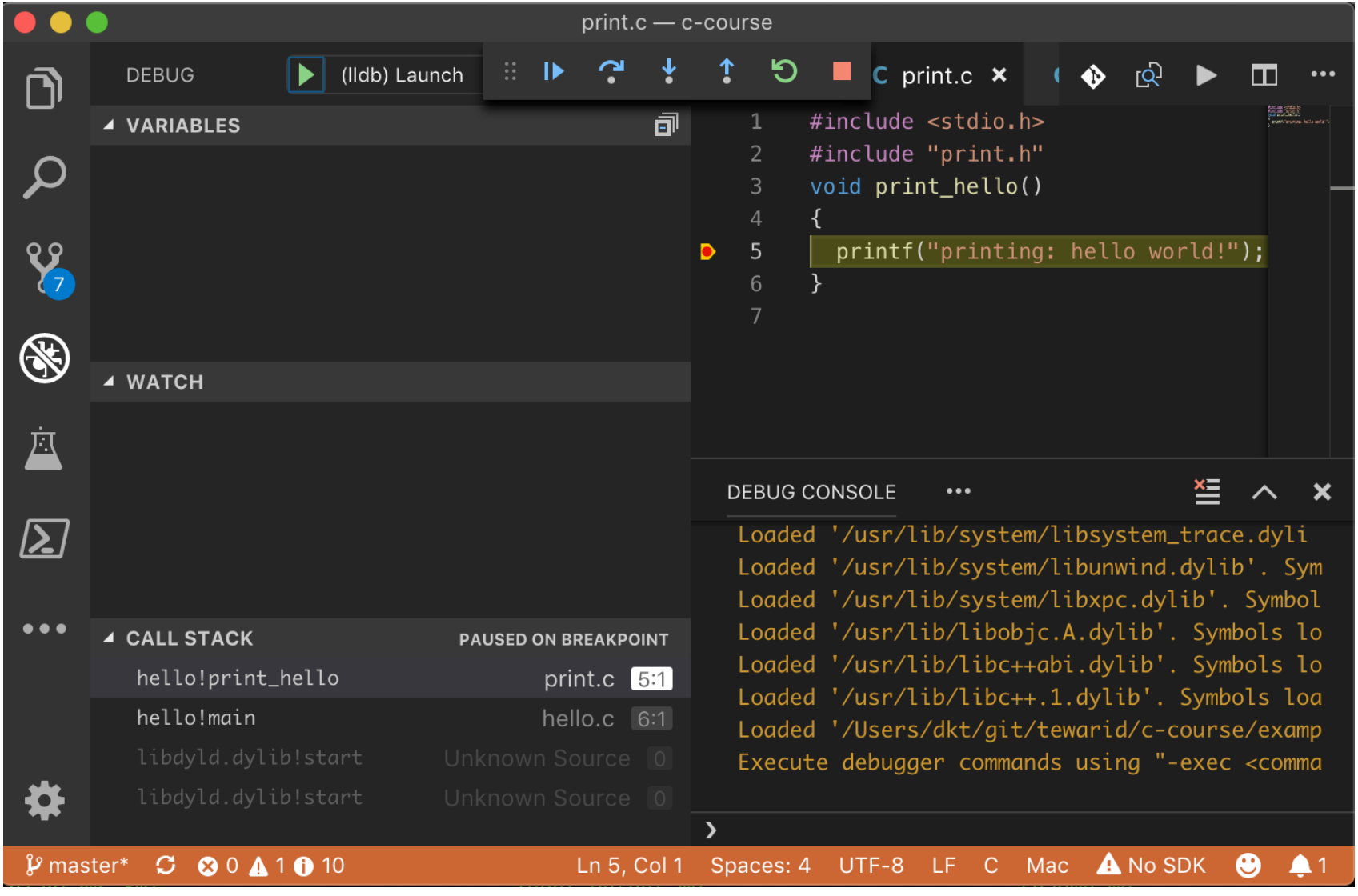
  ```
  ddd hello
  ```

- Try stepping through code and adding watch expressions
- Repeat with VS Code

Debug Using DDD



Debug Using VS Code



Other Topics

- Creating static and shared libraries
- Dynamic linking
- GCC compile, link and optimize options
- Building applications with make
- Using an IDE for C/C++ development

Types, Operators and Expressions

Introduction

- The basic C data objects are
  - Variables
  - Constants
- Operators act on data objects
- Expressions are composed of data objects and operators

Variable Names

- Names are composed of letters and digits
  - Underscore is treated as a letter
- Must start with a letter
- Are case sensitive
- Have a size limit of 31 characters
- As a convention lower case is used for variable names

Keywords

- Symbols reserved by C
- Cannot be used as variable names

auto	default	for	return	switch	while
break	double	goto	short	typedef	
case	else	if	signed	union	
char	enum	int	sizeof	unsigned	
const	extern	long	static	void	
continue	float	register	struct	volatile	

Basic Data Types

- Integer
  - int
  - short int
  - long int
  - char
  - signed and unsigned types
- Floating point
  - float
  - double
  - long double
- Size is implementation dependent
  - `<limits.h>` and `<float.h>` contain constants for sizes

Integer types

- int is normally the natural size for a machine
- unsigned types store negative values in two's complement form
- char meant for holding single byte character code
  - insufficient for Unicode and other codes

	bits	unsigned	signed
long	32	0 to $2^{32}-1$	-2^{31} to $2^{31}-1$
int	32	0 to $2^{32}-1$	-2^{31} to $2^{31}-1$
short	16	0 to $2^{16}-1$	-2^{15} to $2^{15}-1$
char	8	0 to $2^{8}-1$	-2^{7} to $2^{7}-1$

Floating point types

- Maximum and minimum values implementation dependent
- Decimal precision is limited

	Range	Precision
float	1E-37 to 1E+37	6
double	1E-308 to 1E+308	15
long double	1E-308 to 1E+308	15

Constants

- Following types
  - Numeric
  - Character
  - String
  - Enumeration

Numeric constants

- Integer constants
  - A sequence of numbers (4567) is an `int`
  - Signed long integers specified by suffixing `l` or `L` (`4567L`) and unsigned long integers by suffixing `ul` or `UL` (`4567UL`)
  - Octal values indicated with a leading `0`
  - Hexadecimal values indicated with leading `0x`
- Floating-point constants
  - A sequence of numbers with a decimal point (1.23) or an exponent (123E-2) or both (12.3E-1) is a double
  - Long doubles are indicated by suffixing `l` or `L`

Character constants

- A character between single quotes (`'0'`) is a `char`
- A character constant represents the integer value of the character (`'0'` = 48 in ASCII character set)
- A character constant is more portable

String constants

- A sequence of characters delimited by double quotes (`"hello world\n"`)
- Strings constants separated by white-spaces are concatenated at compile time
  - `"hello " "world\n"` becomes `"hello world\n"`
- Internally a string constant is terminated by a `'\0'` (null) character
- Function `strlen(s)` in `<string.h>` returns the size of a string
- A string is actually an array of char (`char []`)

Escape sequences

- Some characters are hard to represent in character constants and string constants
- Escape sequences are used to represent such characters

`\a`	alert (beep)	`\\`	backslash
`\b`	backspace	`\?`	question mark
`\f`	formfeed	`\'`	single quote
`\n`	newline	`\"`	double quote
`\r`	carriage return	`\nnn`	octal number
`\t`	horizontal tab	`\xhh`	hexadecimal number
`\v`	vertical tab	`\0`	null character

Constant expression

- Expressions involving only constants
- May be evaluated at compile time
- Can be used in the place of a constant

```
#define MAX 100
int i = MAX;
```

Enumeration constants

- A list of constant integers
- Values can be specified or generated

```
enum colors { RED = 'r', BLUE = 'b', GREEN = 'g'};

enum dow { SUN = 1, MON, TUE, WED, THU, FRI, SAT};
```

- Variables of enum types can be declared

```
enum colors c = RED;
```

- DDD shows values of enum variables as symbols

Variable Declarations

- All variables must be declared before use
- A declaration only specifies the nature of a variable (i.e. type)
- A declaration contains a type followed by a list of one or more comma separated names

```
char c, name [50];
```

- A variable may be initialized in its declaration

```
int i = MAX + 1, j = i;
```

- By prefixing `const` to a declaration, a variable can be declared as unchangeable

```
const double pi = 3.14;
```

Arithmetic Operators

- Binary operators

  `+ - * / %`

- Unary operators

  `++ -- + -`

- Postfix / prefix unary operator usage

```
int i = 0, j;
j = i++; /* j = 0, i = 1 */
j = ++i; /* j = 2, i = 2 */
```

Relational operators

- Equal and not equal

  `==` `!=`

- Less than and less than or equal to

  `<` `<=`

- Greater than and greater than or equal to

  `>` `>=`

- No boolean type, a value of 0 represents FALSE, any other value is TRUE

Logic operators

- Logical negation

  !

- Logical AND

  &&

- Logical OR

  ||

Bitwise operators

- Bitwise complement

  `~`

- Bitwise left and right shift

  `<< >>`

- Bitwise AND, OR, and exclusive OR (XOR)

  `& | ^`

Assignment operators

- Normal assignment

  `=`

- Arithmetic

  `+= -= *= /= %=`

- Bitwise assignment operators

  `&= ^= |= <<= >>=`

Expressions

- Most expressions are assignments or functions calls
- If an expression is missing the statement is called a null statement
  - Can be used to supply an empty body for an iteration or loop
- All side effects from the expressions are completed before the next statement

Side effects

- Side effects are unpredictable assignment to variables resulting from undefined order of evaluation of an expression
  - In function calls

    ```
    printf("%d, %d", i + 1, i = j + 2 );
    printf("%d, %d, %d", i++, i++, i++);
    ```

  - Nested assignments

    ```
    c = getchar() != EOF
    ```

  - Increment and decrement operators

    ```
    a[i] = i++;
    ```

- Avoid side effects, don't depend on the results from your compiler

Operator Precedence

`() [] -> .`	left to right
`! ~ ++ -- + - * & (type) sizeof`	right to left
`* / %`	left to right
`+ -`	left to right
`<< >>`	left to right
`< <= > >=`	left to right
`== !=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`= += -= *= /= %= &= ^= |= <<= >>=`	right to left
`,`	left to right

Conditional expressions

- `lvalue = expr1 ? expr2 : expr3`

  `lvalue` is the value of the expression
  `expr1` is evaluated first
  `expr2` is evaluated if `expr1` is not 0 (i.e. true)
  `expr3` is evaluated if `expr1` is 0 (i.e. false)

- This is equivalent to

```
if (expr1)
  lvalue = expr2;
else
  lvalue = expr3;
```

Automatic type conversion

- Wider conversions are automatic, e.g. `char` to `int`
- Beware mixing signed and unsigned values, `-1L > 1UL`

  `1UL`
  `00000000000000000000000000000001`

  `1L` as two's complement of `1L` (calculated as `~1+1` or $2^{32} - 1$)
  `11111111111111111111111111111111`

Automatic type conversion rules

For an operator that takes two operands

- If one is `long double`, convert other to `long double`
- Else, if one is `double`, convert other to `double`
- Else, if one is `float`, convert other to `float`
- Else, if one is `unsigned long int`, convert other to `unsigned long int`
- Else, if one is `long int` and other is `unsigned int`
  - If a `long int` can represent all values of an `unsigned int`, convert both to `long int`
  - Else convert both to `unsigned long int`
- Else, if one is `long int`, convert other to `long int`
- Else, if one is `unsigned int`, convert other to `unsigned int`
- Else, convert both to `int`

Type casting

- Forced type conversions (coercion)

  (*type-name*) *expression*
- Required for *narrow* conversions
- Can result in loss of data
  - Wide integers to narrow integers
  - Float or double to integers

Control Flow

if-else

- `if-else` expresses decisions
- `else` part is optional

```
if (expression)
statement1
else
statement2
```

- `expression` returns a numerical value
- `0` is considered FALSE, any other value is TRUE

if-else ambiguities

- By default, the `else` is associated with the inner `if`

```
if (i >= 0)
  if (i < 5)
    a = b;
  else
    a = c;
```

- Use braces to remove ambiguity

```
if (i >= 0) {
  if (i < 5)
    a = b;
} else
  a = c;
```

else-if

- Useful for expressing multi-way decisions

```
if (expression)
  statement
else if (expression)
  statement
else if (expression)
  statement
else
  statement
```

switch

- Used to express multi-way decision
- Matches the result of an expression to one of several integer constants

```
switch (expression) {
  case const-expr:
    statements
  case const-expr:
    statements
  default:
    statements
}
```

- a `break` statement causes exit from the `switch`
- without a `break` all statements after the matching `case` are executed till the end of the `switch` block

while

- The while loop executes as long as `expression` is TRUE (not 0)

```
while (expression) {
  statements
}
```

for

- The loop has three parts

```
for (expr1; expr2; expr3) {
  statements
}
```

  - `expr1` is an initialization expression
  - `expr2` is a relational expression, and
  - `expr3` is the increment expression

- The loop executes as long as `expr2` is TRUE (not 0)
- All three expressions can be empty which leads to an infinite `for` loop

do-while

- The `do` loop executes at least once before `expression` is evaluated

```
do {
  statements
} while (expression);
```

- The loop executes as long as `expression` is TRUE (not `0`)

break

- Using the break statement causes immediate exit from a loop (`for`, `while` or `do-while`) or `switch` block

continue

- The `continue` statement causes a loop to begin the next iteration, the statements following `continue` are not executed

goto

- the `goto` statement causes execution to jump to the statements after the `label`

```
goto label;
statements
label:
statements
```

- `goto` is not recommended as it results in spaghetti code

Exercise

- Write a program that converts 1 to 50 mile(s) into kilometers

  NOTE 1 mile $=$ 1.609344 kilometers

- Print the result in tabular form

```
01 mile(s) = 01.609344 km    02 mile(s) = 03.218688 km
03 mile(s) = 04.828032 km    04 mile(s) = 06.437376 km
```

Functions and Advanced Program Structure

Introduction to Functions

- Useful for program structuring
- Make program more modular
- Should be as generally applicable as possible
- Should encapsulate implementation as best as possible
- Cannot be nested (unlike in Pascal)

Function Declaration

- Functions need to be declared before use
- The compiler matches the declaration with the syntax of usage and definition to see if they match
  - the return type should be the same
  - the parameters should be the same type (not name)

```
return-type function-name (argument declarations);
```

Function Definition

```
return-type function-name (argument declarations) {
  declarations and statements
  return statement returns a value of type return-type
}
```

- The return-type can be void or any other type; if not specified, it defaults to `int`
- A `return` statement is optional and can be used to return a value to the caller, the caller may ignore this value

```
return expression;
```

Variable Declaration

- Anywhere in a C source file
- Inside a function

```
int main() {
  int a, b;
}
```

- Inside any code block

```
{
  int a, b;
}
```

Variable Declaration Example

```c
int a;

int main() {
  int a = 10; // "a" is local to main
  print(); // prints "a: 0"
}

int print() {
  printf("a: %d\n", a);
}
```

External variables

- Variable defined outside functions or in other source files are external
  - *Definition* indicates the place where a variable is created or assigned storage
- A variable defined before the function definition in a source file is visible to the function, as seen in previous example
- Remember multiple source files example from Introduction?

extern keyword

- The extern keyword is used to declare variables defined outside the current function or source file

```
int a;

int main() {
  int a = 10; // "a" is local to main
  print(); // prints "a: 0"
}

int print() {
  extern int a;
  printf("a: %d\n", a);
}
```

auto keyword

- Variables within functions or code blocks that are not declared as extern are auto (for automatic)

```
int a;

int main() {
  auto int a = 10; // "a" is local to main
  print(); // prints "a: 0"
}

int print() {
  extern int a;
  printf("a: %d\n", a);
}
```

static variables

- A variable declared with the keyword `static` within a function or code block retains its value till the program ends

```
int main() {
  print(); // prints "a: 0"
  print(); // prints "a: 1"
}

int print() {
  static int a;
  printf("a: %d\n", a++);
}
```

- A static variable anywhere else in the source file is considered local to that file

register variables

- Useful for advising a compiler to retain a heavily used variable in a CPU register
- Examples

```
register int i;
register char c;
```

Variable initialization

- External and static variables
  - Are guaranteed to be initialized to zero
  - Any values assigned must be constant expressions
- Automatic and register variables
  - Contain garbage unless initialized
  - Can be initialized by specifying expressions containing constants and variables defined earlier

Recursion

- A function can call itself
- The local automatic variables are stored in the stack
- Function parameters are passed using the stack
- Prone to stack overflow
- There is always a danger of creating an infinite loop if the exit criteria is not clear

Recursion – Example

```
int main() {
  print(1);
}

int print(int i) {
  printf("i:%d\n", i++);
  if (i > 5) return;
  else print(i);
}
```

Header files

- Used to include external variable and function definitions
- Allow applications to be compiled in parts
- The remaining parts are resolved during linking from statically or dynamically linked libraries
- Remember the example from Introduction?

Macro definition and substitution

- A macro definition takes the form

  ```
  #define name replacement-text
  ```

- Token name has the same syntax as a variable name
- Everywhere in the source file, where the token name occurs, it is substituted by `replacement-text`
- `replacement-text` is any arbitrary text and it can span several lines by ending each line with a `\`
- A macro can also be defined or redefined by using the `-D` compiler option

  ```
  gcc -Dname=value
  ```

Un-define macros

- To un-define a macro called `name`

  ```
  #undef name
  ```

- A macro defined in a program can also be undefined by using the `-U` compiler option

  ```
  gcc -Uname
  ```

- where `name` is the name of the macro you want to undefine

Macro with arguments

- Look like functions but result in inline code
- Macro with arguments are applicable to arbitrary types

```
#define MAX(A,B) ((A) > (B) ? (A) : (B))

MAX(1.5,2.9) // ((1.5) > (2.9) ? (1.5) : (2.9))

MAX(a+b, c+d) // ((a+b) > (c+d) ? (a+b) : (c+d))
```

- The parentheses are required to maintain proper expression semantics after substitution

Macro with arguments – additional syntax

```
#define debug_print(expression) printf(\
  #expression " = %g\n", expression)

debug_print(x); //printf("x" " = %g\n", x);


#define concat(prefix, suffix) prefix ## suffix

concat(name, 1); //name1;
```

Conditional inclusion

- Preprocessing provides for means to insert code conditionally
- This can useful to
  - Enable or disable tracing statements
  - Include OS specific code
  - Include a header file just once

Enable and disable tracing

- Only integer constants and the following operators can be used in the expression following `#if`
  - `&&`, `||`, `<`, `>`, `<=`, `>=`, `!`, and `==`

```
#define TRACE_NONE 0
#define TRACE_DEBUG 1
#define TRACE_ALL 2

#define TRACE_LEVEL TRACE_DEBUG

int main() {
#if TRACE_LEVEL == TRACE_ALL || TRACE_LEVEL == \
    TRACE_DEBUG
  printf("within main\n");
#endif
  return 0;
}
```

OS specific codes

```
#if !defined(OSNAME)
  #error OSNAME not specified
#endif

int main() {
#if OSNAME == LINUX
  printf("Linux\n");
#else
  printf("Windows\n");
#endif
  return 0;
}
```

- Compile program

  ```
  gcc -DOSNAME -DLINUX macro.c
  ```

Include header file just once

```
#ifndef _HDR_H_
#define _HDR_H_
  declarations
#endif
```

Exercise

- A factorial of a number `n`, denoted as `n!`, is calculated as:

  `n * (n-1) * (n-2)... 3 * 2 * 1`

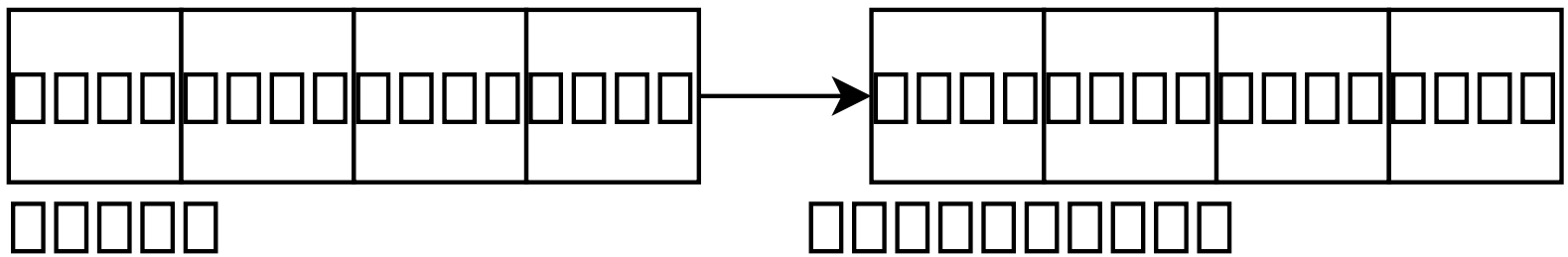
  Thus, `5!`=120 and `10!`=3628800
- Write a recursive function to calculate factorial for any number `n`

Pointers and Arrays

Introduction

- Pointers are variables that store memory addresses
- They store the address of a memory region that stores a particular type of data
- The size of a pointer is determined by the address size of the CPU

```
int* p;
int i = 10;
p = &i;
```



Pointer declaration

- A pointer is declared using the `*` operator

```
int* p;
```

- `*` is called the dereferencing operator because `*p` returns the value of the value `p` points to
- The `&` operator is used to recover the memory address of a variable; it cannot be applied to expressions, constants, or register variables

```
p = &i;
```

Pointer assignment and usage

- Pointers of the same type can be assigned to one another

```
int i = 10;
int* ip = &i;
int* iq = ip; // iq now points to i
```

- Operator precedence in usage of `*` operator

```
*ip += 1;
++*ip;
(*ip)++;
```

  - increment value pointed to by `ip`
  - `*ip++;` would be incorrect; why?

Function arguments

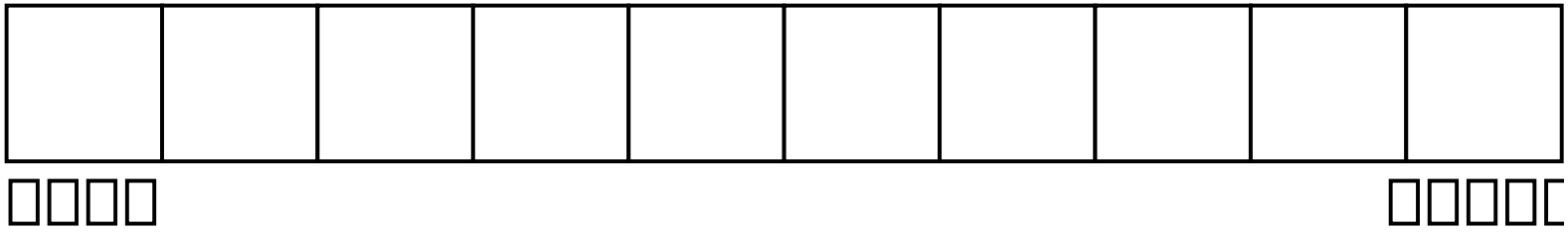
- Arguments are passed to a function by value, even pointer arguments
- Pointers provide a mechanism for functions to alter the value of referenced variables
- Exercise: write a function that swaps the value of its arguments

Arrays

- Arrays provide contiguous storage to multiple elements of the same type

```
int a[10]; // declares an array of 10 integers
```

- Elements of arrays declared as `extern`, `static` and `auto` are initialized to zero
- The array index starts at zero



Array initialization

- Arrays can be initialized during declaration

```
int days[] = {5, 10, 15, 25, 30};
```

  - compiler fills in the size and fills the array

```
char name[] = "name";
```

  - right-hand side is a string constant

```
char name[] = {'n', 'a', 'm', 'e'};
```

- Arrays can be initialized using assignment statements or using loops

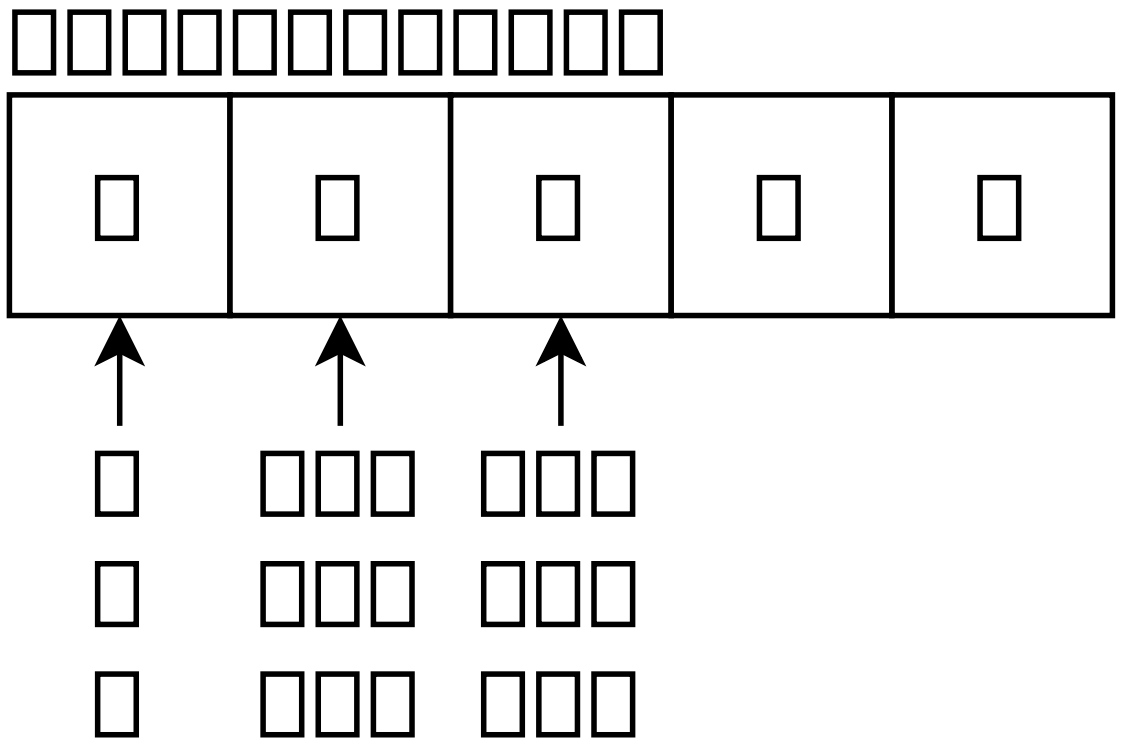
```
int days[5];
days[0] = 5;
```

Pointers vs Arrays

- Arrays and pointers are related

```
int a[5] = {0 , 1, 2, 3, 4};
int* p = &a[0];
int* q = a;
```

  - a always points to the start of the array and cannot be changed



Pointer operations

- Pointers can be incremented in integer steps
- `p++` points to the next element
  - what does `*p++ = 10` do? (hint – see operator precedence table)
- `p--` points to the previous element
  - what does `*--p = 10` do? (hint – see operator precedence table)
- `p+=i` points to `i` elements beyond the current position
- `p-=i` points to `i` elements before the current position
- `p = 0` or `p = NULL` makes `p` a null pointer; a valid pointer that does not point to anything in particular

Strings

- Strings constants are arrays of `char`

  ```
  char name [] = "name";
  ```

- Since an array of `char` can be assigned to a pointer to `char`, a pointer to `char` can refer to a string constant

  ```
  char* name = "name";
  ```

- `strlen(s)` can be used to calculate length of a string

  `strlen(name) returns 4`

- A string is internally padded with NULL character or `'\0'`; `name` is thus internally 5 characters long

- Exercise: write a function to replace `strcpy`

Multi-dimensional arrays

- Declaration

```
int a[10][20];
```

  - 10 rows and 20 columns, *contiguous* storage for 200 integers

- Initialization

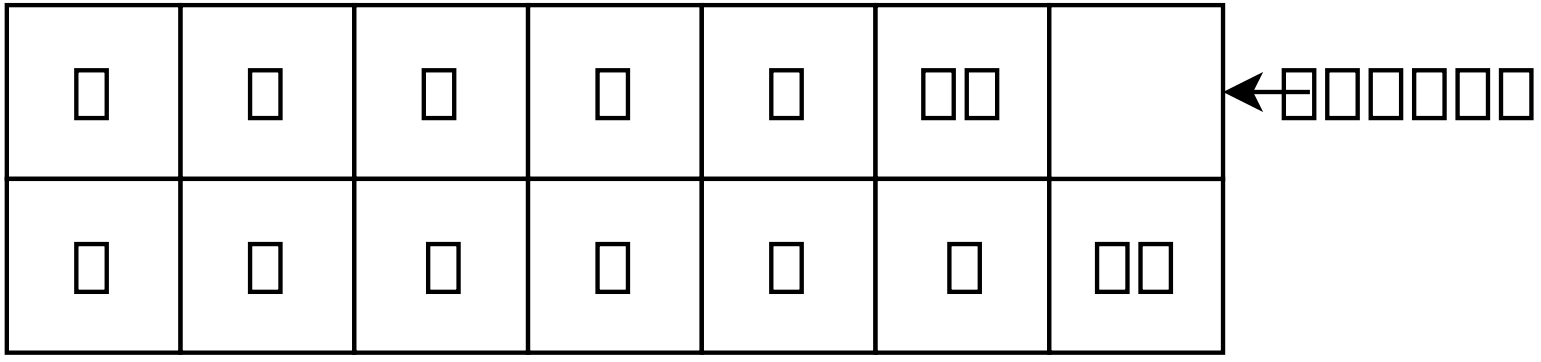
```
int a[][2] = { {1,2}, {3} };
int (b[])[2] = { {1,2}, {3} };
```

  - The number of columns (length of each row) needs to be known beforehand; try printing `a[1][1]`, what do you get?

Wasted space with multi-dimensional array

- This is how you would construct an array of string constants

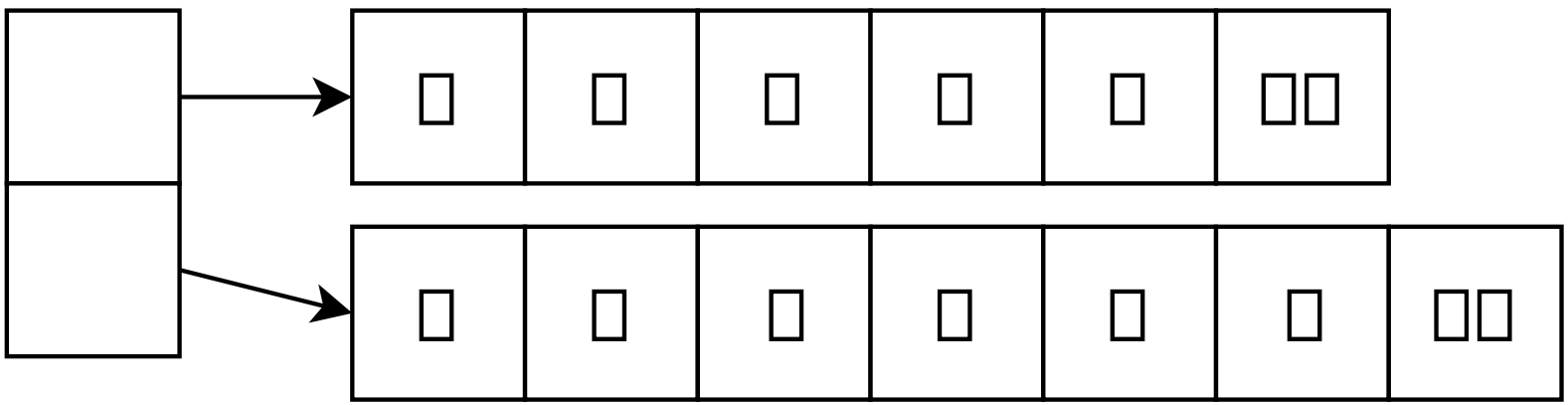
```
char a[][7] = {"hello", "world!"};
printf("%s %s\n", a[0], a[1]);
```



Array of pointers instead of multi-dimensional array

- The multi-dimensional array shown earlier may be substituted by an array of pointers to `char`

```
char* a[] = {"hello", "world!"};
printf("%s %s\n", a[0], a[1]);
```

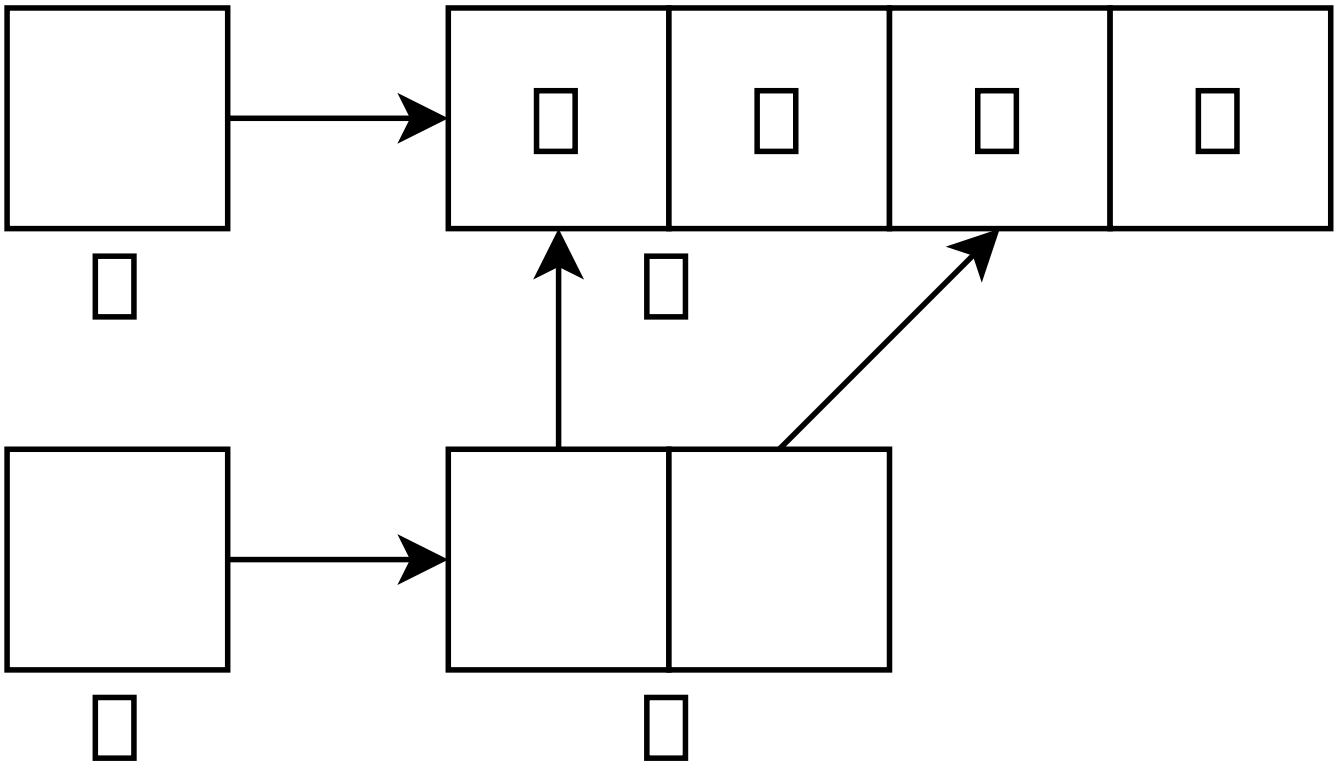


Pointers to multi-dimensional array

- Multi-dimensional arrays can be assigned to pointers

```
int a[2][2] = { {1,2}, {3,4} };
int *b[2], **c, *d;
b[0] = a[0]; b[1] = a[1];
c = b; d = (int *)a;
printf("%d\n", a[1][1]);
printf("%d\n", *(*(a + 1) + 1));
printf("%d\n", b[1][1]);
printf("%d\n", *(*(b + 1) + 1));
printf("%d\n", c[1][1]);
printf("%d\n", *(*(c + 1) + 1));
printf("%d\n", d[3]);
printf("%d\n", *(d + 3));
```

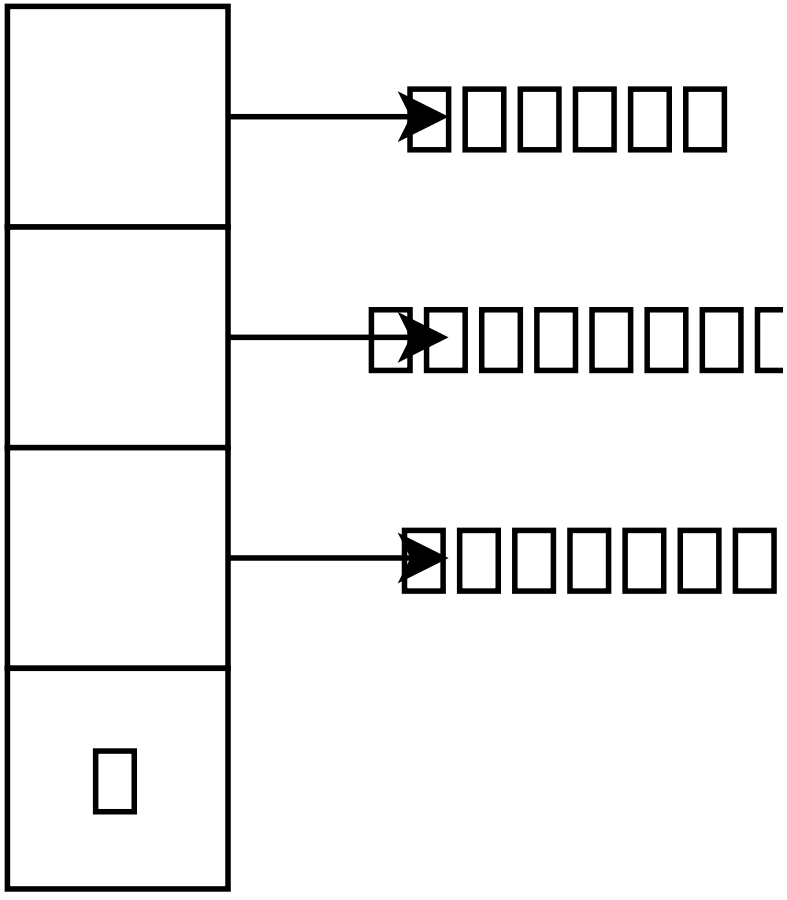
Visualizing pointers to multi-dimensional array



Main function

- `main` function syntax

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



Command line arguments

- `argc` is the number of arguments in the command-line that invoked the program
  - always at least `1` because the program name is itself an argument
- `argv` is an array of pointers to `char`, each element points to a string
- `argv[argc]` required to be a `NULL` pointer

Pointers to Functions

- Pointers can point to functions
- Functions are very different from variables, but have an address where they start
- Declare a pointer to a function

```
int (*p)(int* a, int* b)
```

- Assign a function

```
p = add;
```

- Call the function

```
int a = b = 2;
(*p)(&a, &b);
```

void pointer

- Any pointer type can be assigned to, or passed to a function as, a void pointer

```
int* ip;
void* vp = ip;
```

- void pointer can be cast to any pointer type

```
char* cp = (char*) vp;
```

- Useful for making generic functions that apply to various types
- Be careful when casting `void*` to another type; know what you are doing

Dynamic memory allocation

- Pointers not yet initialized are dangerous if they are not NULL pointers
- Pointers can be initialized to point to storage dynamically allocated using `malloc` or `calloc`
- `free` must be used to release the allocated memory

malloc

- Allocates `n` bytes of storage and returns a void pointer to it

```
void* malloc(size_t n)
```

- Example

```
int* ip = (int*)malloc(10 * sizeof(int));
free(ip);
```

  - `sizeof` *operator* returns the size of the object or type specified

calloc

- Allocates memory for `n` objects of size `size` and returns a void pointer to it

```
void* calloc(size_t n, size_t size)
```

- The memory assigned is initialized to zeros
- Example

```
int* ip = (int*)calloc(10, sizeof(int));
free(ip);
```

Memory allocation problems

- Using an uninitialized pointer
- Writing to memory outside the allocated region (buffer overflow)
- Freeing memory not allocated using malloc or calloc
- Not freeing memory allocated using malloc and calloc (memory leak)

Using memwatch

- memwatch is distributed as a single source file `memwatch.c` and its accompanying header file `memwatch.h`
- Source files you want to watch for memory problems must include `memwatch.h` and be recompiled using the following compiler options

  `-DMEMWATCH -DMW_STDIO`

- memwatch prints an error message in the standard output and produces a detailed log file listing the memory problems it encounters

Exercise

- Write a program that sorts an array of strings. Use your favorite sorting algorithm (bubble sort, insertion sort, etc). Write your own replacement for `strcmp` to compare the strings. Write a generic sort function that can work with arrays of other types

Structures

Introduction

- A C structure is a collection of one or more variables of the same or different types
- Structures permit convenient handling of complicated data as a single unit
- Similar to records in Pascal
- Copying, assigning to, recovering address using & and accessing members are all legal operations on structures

Creating a structure

- The `struct` keyword is used to create structures

```
struct address {
  char * street;
  char * city;
  int zip;
} a, b;
```

- The tag `address` is optional but useful for identifying the `struct` so new variables can be created

```
struct address a, b;
```

Initializing structures

- Structures can be initialized just like arrays

```
struct address a = {"street", "recife", 123456}, b;
```

- An automatic structure can also be initialized by assignment or by calling a function that returns the structure of the right type

```
a.zip = 123456;
a.street = "street";
a.city = "recife";
```

Copying and assigning to structures

- Example of an address structure being assigned to another

```
struct address a, b;
a.zip = 123456;
a.street = "street";
a.city = "recife";
b = a;
b.zip = 654321;
printf("%d, %s, %s\n", a.zip, a.street, a.city);
```

- Output

```
123456, street, recife
```

  - Note that changing the copy has not affected the original structure

Structures and functions

- Structures can be passed as parameters to a function
- Structures are passed by value i.e. copying their content
- Large structures should be passed by reference, by passing their pointers as parameters to functions

Pointers to structures

- Example of structure manipulation using its pointer

```
struct address* b;
b = (struct address*) malloc(sizeof(struct address));
b->street = "street";
b->city = "recife";
(*b).zip = 654321;
printf("%d, %s, %s\n", (*b).zip, b->street, b->city);
```

- The . operator has higher precedence than the * operator
- C provides the -> operator to facilitate accessing members of structures through their pointers
- A structure can point to itself e.g. in a tree structure

Arrays of structures

- Arrays can store structure types

```
struct address a[10];
```

- Arrays can be initialized at time of declaration using literal values and variables

```
struct address a[] = {"street1", "recife", 4123456,
  "street2", "salvador", 654321};
// or
struct address a[] = { {"street1", "recife"},
  {"street2", "salvador", 654321} };
```

Typedef

- New data types can be created using the `typedef` keyword

```
typedef unsigned short UCHAR;
```

- New complex types can be created using structures

```
typedef struct address {
  char * street;
  char * city;
  int zip;
} Address;

b = (Address*)malloc(sizeof(Address));
b->street = "street";
b->city = "recife";
b->zip = 654321;
```

Unions

- Look like a structure but store only one type at any given time

```
union number {
  int ival;
  float fval;
} n;
```

- The compiler assigns a union a size large enough to store the widest type
- Unions can be nested within structures
- Unions support the same operations as structures

Bit fields

- Useful for conveniently handling several option flags as a single entity
- Each flag field can only be an `int`
- The fields cannot be arrays, nor be pointed to; thus the `&` operator cannot be applied to them

```
struct bit_fields {
  unsigned int is_keyword : 1;
  unsigned int is_extern : 1;
  unsigned int is_static : 1;
} f;
```

Exercise

- Write a program to count the occurrence of each word in a given string. Use a binary search tree to store the words along with their counts.
- Print the words with their count to standard output in an ascending order by traversing the binary search tree in-order.

Input and Output

Introduction

- Input and Output in C is provided by several standard library functions and not by the core language itself
- The standard library functions operate on streams (source or destination of data)
- The state of a stream is stored in the FILE structure declared in `<stdio.h>`
- A FILE structure variable should not be created by the programmer

Standard streams

- The C library provides three standard streams in `<stdio.h>`
  - Standard input – `stdin`
  - Standard output – `stdout`
  - Standard error – `stderr`

Standard Input

- Stream that represents the standard input; usually this is the keyboard
- The `stdin` object is the standard input stream; it can be redirected by using the `freopen` function
- It can also be redirected at the command line

  `executable < filename`

  - content of filename available in `stdin` of program

  `executable1 | executable2`

  - output of `executable1` available in `stdin` of `executable2`

Standard Output

- Stream that represents the standard output; usually this is the display
- The `stdout` object is the standard output stream; it can be redirected using the `freopen` function
- It can also be redirected at the command line

  `executable > filename`

  - redirects `stdout` of program to file `filename`

  `executable1 | executable2`

  - redirects `stdout` of `executable1` to `stdin` of `executable2`

Standard Error

- Stream that represents the standard error; usually this is the display
- The `stdout` object is the standard output stream; it can be redirected using the `freopen` function
- Program errors should be sent to this stream; this way, when the standard output is redirected to some other file, the error messages will continue to appear to the user on the display
- It can also be redirected at the command line

```
executable 2> filename
```

File Access

- Open a file stream

```
FILE* fp;
fp = fopen(name, mode);
```

  - `name` is a relative or absolute file name and path
  - mode can be `"r"`, `"w"`, `"a"`, `"rt"`, `"wt"`, `"at"`, `"rb"`, `"wb"`, `"ab"`, `"r+t"`, `"w+t"`, `"a+t"`, `"r+b"`, `"w+b"`, or `"a+b"`

- Use the functions in `<stdio.h>` to manipulate the content of the file stream
- Call `fclose` to close the file stream

Formatted Output

- Function `printf` prints text to stdout

  ```
  int printf(char* format, ...)
  ```

- Function `fprintf` prints to any open stream

  ```
  int fprintf(FILE* fp, char* format, ...)
  ```

- `format` is the format string; it contains the text to be printed, interspersed with conversion specifications that are used to convert and print the arguments that follow

Print conversion specification

`%[flags][width][.precision][modifiers]type`

- `flags` can be `-`, `+`, space, `0`, `#`
- `width` specifies the minimum field width
- `precision` specifies different things depending on the `type`
- `modifiers` can be `h`, `l` or `L`
- `type` can be `d`, `i`, `o`, `x`, `X`, `u`, `c`, `s`, `f`, `e`, `E`, `g`, `G`, `p`, `n`, or `%`

Formatted Input

- Function `scanf` reads formatted input from stdin

  ```
  int scanf(char *format, ...)
  ```

- Function `fscanf` reads formatted input from any stream

  ```
  int fscanf(FILE *fp; char *format, ...)
  ```

- `format` is the format string; this contains the text to be matched against the input, interspersed with conversion specifications that are used to convert and read values into the following arguments
- Blanks and tabs in the format string are ignored
- White-space characters in the input stream act as field separators

Input conversion specification

`%[*][width][modifiers]type`

- `*` specifies assignment suppression
- `width` specifies the maximum width
- `modifiers` can be `h` or `l`
- `type` can be `d`, `i`, `o`, `u`, `x`, `c`, `s`, `e`, `f`, `g`, `p`, `n`, `[...]`, `[^...]`, or `%`

```
scanf("%d/%d/%d", &day, &month, &year)
```

Variable length argument lists

- A function may contain a variable length argument list

```
int printf(const char*, ...)
```

- Header <stdarg.h> contains macro definitions that define how to read the argument list
  - Declare a variable ap of type va_list
  - Call va_start(ap, lastarg) to initialize ap; lastarg is the last argument before ...
  - Call va_arg(ap, type) to read next argument
  - Call va_end(ap) to clean up

Character input and output

`int getc (FILE* fp)`

- returns next character from stream `fp`, or `EOF`

`int putc(int c, FILE* fp)`

- write character `c` to stream `fp`
- returns character written, or `EOF` on error

`getchar()`

- same as `getc(stdin)`

`putchar(c)`

- same as `putc(c, stdout)`

Line input and output

```
char* fgets(char* line, int maxline, FILE* fp)
```

- reads at most `(maxline - 1)` characters from file stream `fp`
- returns line, `NULL` on error, or `EOF`

```
int fputs(char* line, FILE* fp)
```

- writes the string in `line` to the file stream `fp`
- returns zero or `EOF` if an error occurs

File positioning

`fseek(FILE* stream, long offset, int origin)`

- sets the file position for the stream; offset may be `SEEK_SET`, `SEEK_CUR` or `SEEK_END`

`long ftell(FILE* stream)`

- returns the current file position or `-1L` on error

`void rewind(FILE* stream)`

- sets the file position to the beginning, this is same as calling `fseek(fp, 0L, SEEK_SET)`

Error handling

`void clearerr(FILE* stream)`

- clears end of file and error indicators

`int feof(FILE* stream)`

- returns non-zero if end of file indicator is set

`int ferror(FILE* stream)`

- returns non-zero if error indicator is set

`void perror(const char*)`

- prints error message

Listing directories

- For BSD / Linux compatible library functions in `<dirent.h>`
- File information function stat from `<sys/stat.h>`

```
DIR* dir;
struct dirent* item;
struct stat statbuf;
dir = opendir(".");
item = readdir(dir);
while(item != NULL) {
  stat(item->d_name, &statbuf);
  if(S_ISDIR(statbuf.st_mode)) {
    //...
  }
  item = readdir(dir);
}
```

Exercise

- Write a program that functions like the Unix tar command. The program should pack all files in the current directory into a single file whose name is specified at the command line. If the program receives the `-u` flag followed by a file name, it should unpack the content of the file to the current directory

  ```
  pack [-u] filename
  ```