Advanced C Concepts

2501ICT/7421ICTNathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2012

Outline

1. Preprocessor Directives
2. Pointers and Memory
   - Pointers, Arrays, and Strings

#include Reviewed

- Includes global or local header files
- Header files are just files that get inserted instead of the `#include` statement
- No protection against multiple inclusion!
  - will cause problems with `#define`, `struct`, ...
- Can be overcome by conditional compilation
  - `#if` / `#ifdef` / `#else` / `#endif`
  - evaluates `#define` macros and selectively passes code to the compiler

Preprocessor conditionals

- `#ifdef` *macro*
  - only includes the subsequent code if *macro* was `#defined`
- `#ifndef` *macro*
  - only includes the subsequent code if *macro* was not `#defined`
- `#if` *expression*
  - only includes the subsequent code if *expression* is true
- `#else`
  - reverses the effects of the previous `#if`
- `#elif` *expression*
  - combines `#else` with the effects of `#if`
- `#endif`
  - ends the conditional block started by `#if` or `#ifdef`
  - each `#if` or `#ifdef` needs exactly one `#endif`

#ifdef / #else Example

Example (What does this code print)

```
#define DEBUG 1                    // turn on debugging

int main(void)
{
#ifdef DEBUG
  printf("debugging is on, DEBUG is %d\n", DEBUG);
#else
  printf("debugging is off, DEBUG is not defined\n");
#endif

  return 0;
}
```

Answer

```
debugging is on, DEBUG is 1
```

#ifdef / #else Example

Example (What does this code print)

```
#define DEBUG 0                    // turn on debugging

int main(void)
{
#ifdef DEBUG
  printf("debugging is on, DEBUG is %d\n", DEBUG);
#else
  printf("debugging is off, DEBUG is not defined\n");
#endif

  return 0;
}
```

Answer

```
debugging is on, DEBUG is 0
```

#if / #elif Example

Example (What does this code print)

```
#define DEBUG_LEVEL 3           // define debug level to be 3

int main(void)
{
#if DEBUG_LEVEL < 1              // test the actual value of DEBUG_LEVEL
  printf("debugging is off\n");
#elif DEBUG_LEVEL == 1
  printf("debugging is on\n");
#else
  printf("debugging is verbose, DEBUG_LEVEL is %d\n", DEBUG_LEVEL);
#endif

  return 0;
}
```

Answer

```
debugging is verbose, DEBUG_LEVEL is 3
```

#include Protection

Example (a protected header file `profit.h`)

```
#ifndef PROFIT_H                    // only if PROFIT_H was not defined yet
#define PROFIT_H                    // now define PROFIT_H for protection

struct Profit                      // definition of a 'Profit' structure
{
        int     year;
        double dollars;
};

#endif                              // PROFIT_H
```

How does this header protection work?

- `PROFIT_H` is not `#define`d to begin with
- `PROFIT_H` gets defined the first time `profit.h` gets `#include`d
- The next time `profit.h` gets `#include`d, everything betweeen the `#ifdef` and `#endif` is ignored!

Copying Strings

- A String is an array of `char`acters
  - one character after the other in memory
- Strings need to be copied character by character
  - → loop that stops when the end of string is reached

String Copying Example

Example

```
int main(void)
{
  char b[8], a[6] = "Hello";     // two character arrays
  int i = 0;                     // index for copying string a to b

  do {                           // loop to copy string a to b
    b[i] = a[i];                 // copy one character at a time
  } while (a[i++] != '\0')       // until we have reached the end of the string

  printf("%s\n", b);             // now we can print the string copy b

  return 0;
}
```

Explanation

- String `a` gets copied to `b` character by character
- Integer `i` counts up the current index into the array
- `'\0'` denotes the end of the string
  - needs to be copied before finishing the loop

Printing Strings Revisited

Example (How does this all work?)

```
int main(void)
{
  char s[6] = "HELLO";          // (1) how much space is needed for this string?

  printf("%s\n", s);             // (2) how does printf print the string s?

  return 0;
}
```

Answer

1. the string `s` needs memory space for 6 characters
2. `printf()` reads the string from the memory location of `s`

Pointer Variables

Example (A Character Pointer)

```
int main(void)
{
  char s[6] = "Hello";

  char *p;                          // a pointer variable

  p = s;                             // store the address of 's' in p

  printf("%s\n", s);
  printf("%s\n", p);                 // the same string as 's' (not a copy!)

  return 0;
}
```

Explanation

- `char *` is a character pointer type.
- `p` is called a character pointer variable.
  - → stores the memory address of a character
    - (the first character (`'H'`) of the string `"Hello"`)

The Address Operator &

- The ampersand character & is the address operator.
  - It returns the memory address of any variable
- For an array, the name of the array is a shortcut to the memory address of the first element

Example

```
int main(void)
{
  char s[6] = "Hello";          // the same string as in the previous example

  printf("%s\n", s);             // shortcut notation
  printf("%s\n", &s[0]);         // exactly the same as the above!

  return 0;
}
```

Printing Memory Addressess using %p

Example (Printing a Memory Address)

```
int main(void)
{
  char s[6] = "Hello";

  char *p = s;

  printf("%p\n", s);              // while we won't know upfront what the
  printf("%p\n", &s[0]);          // memory address is, all three printf()
  printf("%p\n", p);              // will print the same address

  return 0;
}
```

Explanation

- %p prints a memory address (in hexadecimal notation)
- all three printf()'s are equivalent
  - $\Rightarrow$ print the same address!

More Pointer Examples

Example (Pointers to other types than char)

```
int main(void)
{
  char aCharacter = 'A';                  // some normal variables
  int anInteger   = 12345;
  double aDouble  = 12.45;

  char   *a = &aCharacter;                // pointers to different types
  int    *b = &anInteger;                 // storing the addresses of the
  double *c = &aDouble;                   // corresponding variables above

  printf("%p %p %p\n", a, b, c);          // print the three addresses

  return 0;
}
```

Explanation

- Every variable occupies space in memory
  $\Rightarrow$ pointers can be defined for any type!
- Different variables are stored in different memory locations
  $\Rightarrow$ all addresses printed in the example will be different!

Pointers to Pointers

Example

```
int main(void)
{
  int x = 7;                          // normal integer variable

  int  *a = &x;                       // pointer to the address of x
  int **b = &a;                       // pointer to the address of a

  printf("%p %p\n", a, b);            // a and b are different!

  return 0;
}
```

Explanation

- Like normal variables, pointers occupy memory space as well!
  - $\Rightarrow$ `&a` will return the address of the pointer `a`
- `int **b` is a pointer to a pointer
  - $\rightarrow$ every additional $*$ adds a level of indirection

How to use Pointers – de-referencing using *

- The question is how can memory be accessed using a pointer?
- The asterisk (star) character * is the de-referencing operator.
  - It accesses the content of the memory address pointed to by a pointer.

  → opposite of the & operator!
- Allows to manipulate variables indirectly
  - without knowing the name of the variable at he point where it gets manipulated

Pointer de-referencing example

Example (What does this program print?)

```
int main(void)
{
  int  x = 5;
  int *p = &x;          // p now points to the address of x

  int  y = *p;          // get the value at the address pointed to by p

  *p = 7;               // set the value at the address pointed to by p

  printf("x = %d, y = %d\n", x, y);

  return 0;
}
```

Answer

```
x = 7, y = 5
```

Call-by-reference through Pointers

Example (What does this program print?)

```
void manipulate(int *p)
{
  *p = *p / 2;           // change the memory content pointed to by p
}

int main(void)
{
  int x = 8;

  manipulate(&x);        // pass address of variable x so x can be manipulated

  printf("x = %d\n", x);

  return 0;
}
```

Answer

```
x = 4
```

Pointer Arithmetic

- Pointers store memory addresses
  - just numbers telling the processor which memory cell to access
- Adding `1` to a pointer makes it point to the next memory location
- Subtracting `1` from a pointer makes it point to the previous memory location
- Subtracting two pointers from each other shows how much space is between the memory locations pointed to by the pointers
- Pointers “know” the sizes of the variables they point to
  - adding to an `int` pointer will probably result in a different address than adding to a `char` pointer

Pointers and Arrays

- Arrays store elements of the same kind in adjacent memory addresses
- Pointers can store array locations
- Pointer and array notations are often interchangeable
  - E.g. for `char *p`
  - `p[4]` is the same as `*(p + 4)`
  - `&p[4]` is the same as `(p + 4)`

$\Rightarrow$ Strings can be represented by pointers as well as arrays

Pointer and Array Example: Strings

Example (What does this program print?)

```
void print(char *text)
{
  printf("%s\n", text);          // print the string pointed to by 'text'
}

int main(void)
{
  char  s[10] = "fantastic";    // a string
  char *p = s;                  // a pointer to the same string

  *(p + 3) = '\0';              // manipulate the memory pointed to by p+3

  print(s);                     // print the string s

  return 0;
}
```

Answer

```
fan
```

Copying Strings revisited

Example (a more efficient string copy)

```
void string_copy(char *dst, char *src)  // copy a string from src to dst
{
        while (*dst++ = *src++) ;       // copy and test each character
}

int main(void)
{
  char b[8], *a = "Hello";      // destination array and source string

  string_copy(b, a);             // copy a to b

  printf("%s\n", b);             // now we can print the string copy b

  return 0;
}
```

Explanation

- in C each assignment has a value that can be tested
- any non-zero result is treated as TRUE in C
- the end-of-string character `\0` is treated as FALSE

Arrays of Pointers

- A pointer is just another data type
  - ⇒ arrays of pointers can be defined like any other array
- E.g. `int *x[6]`
  - an array of 6 integer pointers
- E.g. `char *a[4]`
  - an array of 4 character pointers
  - ⇒ an array of 4 strings

Array of Strings Example

Example (What does this program print?)

```
int main(void)
{
  char *s[3] = { "one", "two", "three"};

  printf("%s\n", s[1]);

  return 0;
}
```

Answer

two

Passing Command Line Parameters

Example (Command Line Parameters)

```
int main(int argc, char *argv[])        // a main() that takes parameters
{
  int i;

  printf("argc = %d\n", argc);           // print the number of parameters

  for (i = 0; i < argc; i++)                    // loop through all parameters
    printf("argv[%d] = '%s'\n", i, argv[i]);    // and print each one of them

  return 0;
}
```

Points to remember

- Command line parameters are passed as an array of strings (`argv`)
- The first argument (`argc`) contains the number of elements in the array
- `argv[0]` always contains the program name itself

Pointers to Structs

Example (What does this program print?)

```
struct Student
{
        char *name;             // student name
        long  num;              // student ID
};

int main(void)
{
  struct Student s;             // a student variable s
  struct Student *p = &s;       // a pointer to that variable

  (*p).name = "Peter";          // set the name
  (*p).num  = 1234567;          // and student ID

  printf("%s's ID is %ld\n", s.name, s.num);

  return 0;
}
```

Answer

```
Peter's ID is 1234567
```

Shortcut Notation

Example (Shortcut Notation)

```
struct Student
{
        char *name;             // student name
        long  num;              // student ID
};

int main(void)
{
  struct Student s;             // a student variable s
  struct Student *p = &s;       // a pointer to that variable

  p->name = "Peter";              // set the name   -- shortcut notation
  p->num  = 1234567;              // and student ID -- shortcut notation

  printf("%s's ID is %ld\n", s.name, s.num);

  return 0;
}
```

Explanation

`p->x` is a shortcut for `(*p).x`

Pointers to Remember

- Call by Reference can be implemented through Pointers
  - → can save copying lots of data
  - → allows functions to indirectly manipulate data
- Beware of Invalid Pointers!
  - no run-time checking for array boundaries and pointer validity
  - accessing invalid memory may crash your program
  - ⇒ never de-reference uninitialised pointers
  - ⇒ never de-reference NULL pointers
  - ⇒ never de-reference expired pointers

Uninitialised Pointer Error Example

Example (What is Wrong with this Program?)

```
int main(void)
{
  int *p;                  // an unitialised pointer

  *p = 7;                   // ERROR: THE PROGRAM WILL PROBABLY CRASH HERE

  printf("*p = %d\n", *p);

  return 0;
}
```

Explanation

- `p` does not point to a valid address!
- typical errors are `Bus Error` and `Segmentation Fault`

NULL Pointer Error Example

Example (What is Wrong with this Program?)

```
int main(void)
{
  int *p = 0;             // a NULL pointer

  *p = 7;                  // ERROR: THE PROGRAM WILL PROBABLY CRASH HERE

  printf("*p = %d\n", *p);

  return 0;
}
```

Explanation

- `0` (NULL) is not a valid memory address!
- unlike Java, there are no NULL pointer exceptions!
- typical errors are `Bus Error` and `Segmentation Fault`

Expired Pointer Error Example

Example (What is Wrong with this Program?)

```
int *function(void)      // a function that returns an integer pointer
{
  int x = 2;

  return &x;             // THIS IS WRONG: x will expire at the end of 'function'
}

int main(void)
{
  int *p = function();  // assign the return value of function to p

  *p = 7;                // ERROR: THE PROGRAM WILL PROBABLY CRASH HERE

  return 0;
}
```

Explanation

- `x` expires at end of `function()`, memory will be re-used!
- will probably only crash sometimes!
  - → one of the hardest errors to find and correct!