Operating Systems 2017/2018

C Language

Ricardo Rocha

Department of Computer Science

Faculty of Sciences

University of Porto

Adapted from the slides

'Revisões sobre Programação em C, Sérgio Crisóstomo'

Compilation

```
#include <stdio.h>

int main() {
  printf("Hello World!\n");
  return 0;
}
```

Main compilation flags:

```
  $ gcc -o hello hello.c  // -o: to specify executable name
  $ gcc -Wall hello.c     // -Wall: gives much better warnings
  $ gcc -g hello.c        // -g: to enable debugging with gdb
  $ gcc -O hello.c        // -O: to turn on optimization
```

Data Types: Examples

```
char a = 'A';
char b = 65;
char c = 0x41;

int i = -2343234;
unsigned int ui = 4294967295;
unsigned int uj = ((long) 1 << 32) - 1;
unsigned int uk = -1;

float pi = 3.14;
double long_pi = 0.31415e+1;
```

Integer Data Types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

- There is no explicit boolean data type:
  - Values **0** or **NULL** mean **FALSE**
  - Any other value means **TRUE**

Floating-Point Data Types

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

- To get the exact size of a data type or variable, use the **sizeof()** operator that yields the **storage size of the type or variable in bytes**:

```
int x;
printf("storage size for type int: %d bytes\n", sizeof(int));
printf("storage size for variable x: %d bytes\n", sizeof(x));
```

Void Data Type

- The void data type specifies that **no value is available** and it can be used in three kinds of situations:

1	**Function returns as void** There are various functions in C which do not return value or you can say they return void. A function with no return value has the return type as void. For example **void exit (int status);**
2	**Function arguments as void** There are various functions in C which do not accept any parameter. A function with no parameter can accept as a void. For example, **int rand(void);**
3	**Pointers to void** A pointer of type void * represents the address of an object, but not its type. For example a memory allocation function **void *malloc(size_t size);** returns a pointer to void which can be casted to any data type.

Arithmetic Operators

Operator	Description
+	Adds two operands
-	Subtracts second operand from the first
*	Multiplies both operands
/	Divides numerator by de-numerator
%	Modulus Operator and remainder of after an integer division
++	Increments operator increases integer value by one
--	Decrements operator decreases integer value by one

Relational Operators

Operator	Description
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

Logical Operators

Operator	Description
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.

Bitwise Operators

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

Decision Making Statements

- If statement

```
if (cond) {
  ... // continue here if cond is true
else {
  ... // continue here if cond is false
}
```

- Switch statement

```
switch (val) {
  case const1: ... // continue here if val is const1
       break;
  ...
  default:     ... // continue here if all other cases fail
}
```

Loop Control Statements

- For statement

```
for (pre_cond; cond; post_loop) {
  ... // execute pre_cond once and loop while cond is true
} // at the end of each loop, execute post_loop
```

- While statement

```
while (cond) {
  ... // loop while cond is true
}
```

- Do/while statement

```
do {
  ... // execute once and loop while cond is true
} while (cond);
```

Pointers

- A **pointer** is a variable whose value is a memory address
- The general form of a pointer variable declaration is

```
type *ptrname;
```

where **type** is the pointer's base data type and **ptrname** is the name of the pointer variable:

```
char *ch;    // pointer to a character
int *ip;     // pointer to an integer
float *fp;   // pointer to a float
int **ipp;   // pointer to a pointer to an integer
```

- There are two main pointer operations:
  - **&** to obtain the memory address of something
  - ***** to access the memory address stored in a pointer

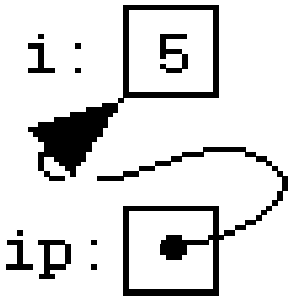
Pointers: Example I

```
int main () {
  int v = 0;
  int *ip;  // initially ip stores an undefined memory address
  ip = &v;  // here ip stores the memory address of variable v
  // print address stored in pointer ip
  printf("value stored in ip variable: %p\n", ip);
  // change v's value using pointer ip
  *ip = 10;
  // print v's value using pointer ip
  printf("value stored in v variable: %d\n", *ip);
  return 0;
}
```

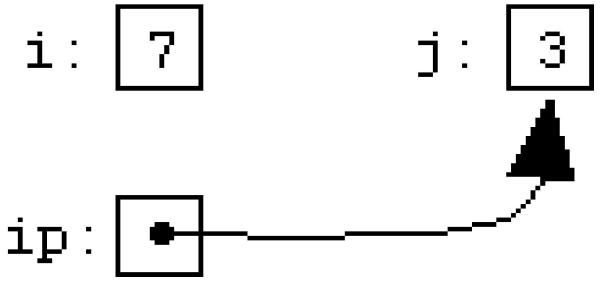
Pointers: Example II

```
int i, j, *ip, *ip2;
// step 1
i = 5;
ip = &i;
// step 2
*ip = 7;
j = 3;
ip = &j;
// step 3
ip2 = ip;
// step 4
ip = &i;
...
```

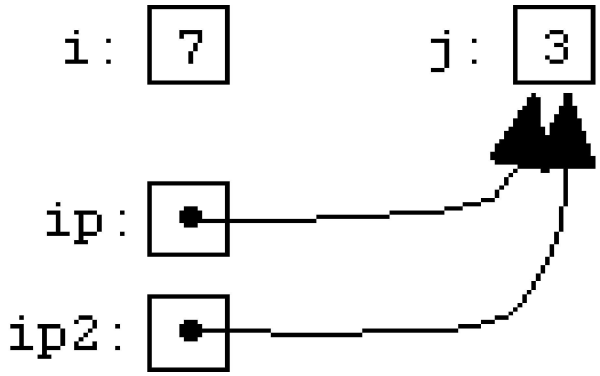
step 1



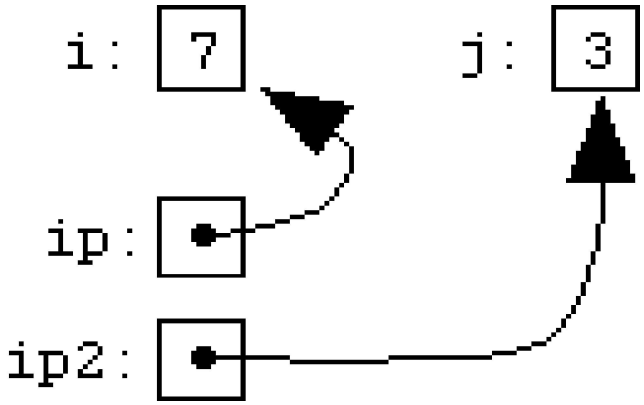
step2



step 3



step 4



Arrays

- Arrays consist of contiguous memory locations used to store a **fixed-size sequential collection of elements of the same type**
- Each specific element in an array is **accessed by an index**
  - The lowest index (0) corresponds to the first element and the highest address to the last element
- The general form of an array variable declaration is

```
type arrayName[size];
```

where **type** is the data type of the elements, **arrayName** is the name of the array variable and **size** is the number of elements in the array:

```
char cs[10];  // an array of 10 characters
int  is[5];   // an array of 5 integers
```

Arrays

- Arrays can also be initialized when declaring them:

```
int is[5] = {1000, 2, 3, 7, 50};
int is[]  = {1000, 2, 3, 7, 50};
```

- When declaring the size of the array, the number of values between braces { } cannot be larger than the number of elements declared for the array between square brackets [ ]

- When omitting the size of the array, an array just big enough to hold the initialization is created

Arrays: Example

```
int main () {
  int i;
  // declare an array of 10 integers
  int n[10];
  // initialize elements of array
  for (i = 0; i < 10; i++)
    n[i] = i + 100;
  // output each array element's value
  for (i = 0; i < 10; i++)
     printf("element[%d] = %d\n", i, n[i]);
  return 0;
}
```

Strings

- A string is an **array of characters terminated by the null character '\0'**

- To hold the null character, the size of the array is one more than the number of characters in the string:

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

- Following the array initialization rule, we can also write:

```
char greeting[] = "Hello";
char *greeting = "Hello";
```

Basic String Functions

Function & Purpose
strcpy(s1, s2); Copies string s2 into string s1.
strcat(s1, s2); Concatenates string s2 onto the end of string s1.
strlen(s1); Returns the length of string s1.
strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

Basic String Functions: Example

```
int main () {
  char str1[100] = "Hello";
  char str2[100] = "World";
  // concatenate str2 onto the end of str1
  strcat(str1, str2);
  // copy str1 into str2
  strcpy(str2, str1);
  // print length of str1
  printf("str1 length: %d\n", strlen(str1));
  return 0;
}
```

Functions

- While calling a function, there are two ways that arguments can be passed to a function:
  - **Call by value** copies the actual value of an argument into the formal parameter of the function, meaning that changes made to the parameter inside the function have no effect on the argument
  - **Call by reference** copies the address of an argument into the formal parameter and, inside the function, the address is used to access the actual argument used in the call, meaning that changes made to the parameter affect the argument
- By default, **C uses call by value to pass arguments**
- However, if we use **pointer arguments** in a function, we can implement a kind of **call by reference for the variables pointed by the argument pointers** and thus alter those variables inside the function

Call by Value: Example

```
void swap_call_by_value(int n1, int n2) {
  int temp = n1;
  n1 = n2;
  n2 = temp;
}

int main() {
  int n1 = 1;
  int n2 = 2;
  swap_call_by_value(n1, n2);
  printf("n1: %d n2: %d\n", n1, n2);  // prints ‘n1: 1 n2: 2’
  return 0;
}
```

Call by Reference: Example

```
void swap_call_by_reference(int *p1, int *p2) {
  int temp = *p1;
  *p1 = *p2;
  *p2 = temp;
}

int main() {
  int n1 = 1;
  int n2 = 2;
  swap_call_by_reference(&n1, &n2);
  printf("n1: %d n2: %d\n", n1, n2);  // prints ‘n1: 2 n2: 1’
  return 0;
}
```

Structures

- A structure is a user defined data type which allows to **combine data items of different kinds**
- The **struct** statement defines a new data type which can contain several different item members:

```
struct [structure tag] {
  type1 item1;
  type2 item2;
  ...
  typeN itemN;
} [one or more structure variables];
```

- Similarly to arrays, structures can also be initialized when declaring them:

```
struct list_elem le = {10, NULL};
```

Structures: Example

```
struct list_elem {
  int data;
  struct list_elem *next;
};

int main() {
  struct list_elem le = {10, NULL};
  struct list_elem *lep = &le;
  printf("data %d next %x\n", le.data, le.next);
  printf("data %d next %x\n", (*lep).data, (*lep).next);
  printf("data %d next %x\n", lep->data, lep->next);
  return 0;
}
```

Typedefs

- The keyword **typedef** allows to **give a type a new identifier**:

```
typedef unsigned char BYTE;
```

- The new identifier can then be used as an abbreviation for the type:

```
BYTE b1, b2;
```

- The typedef keyword can also be used to give a name to user defined data types.

```
typedef struct {
  int data;
  struct list_elem *next;
} list_elem;
...
list_elem le = {10, NULL};
```

Dynamic Memory Allocation

Function and Description
void *calloc(int num, int size); This function allocates an array of **num** elements each of which size in bytes will be **size**.
void free(void *address); This function release a block of memory block specified by address.
void *malloc(int num); This function allocates an array of **num** bytes and leave them initialized.
void *realloc(void *address, int newsize); This function re-allocates memory extending it upto **newsize**.

Dynamic Memory Allocation: Example

```
int main() {
  // allocate memory dynamically
  char *name = (char *) malloc(20 * sizeof(char));
  if (name == NULL) { /* malloc error */ }
  else {
    strcpy(name, "Hello world!");
    printf("%s\n", name);
  }
  // free the memory space previously allocated
  free(name);
  return 0;
}
```

Command Line Arguments

```
int main(int argc, char *argv[]) {
  if (argc == 1)
    printf("Hello %s!\n", argv[1]);
  else
    printf("Hello World!\n");
  return 0;
}
```

Running example:

```
$ gcc -o hello hello.c
$ hello
Hello World
$ hello Ricardo
Hello Ricardo
```

Preprocessor Directives

- The **C Preprocessor** is not part of the compiler, but a separate step in the compilation process that works like a text substitution tool

Directive	Description
#define	Substitutes a preprocessor macro
#include	Inserts a particular header from another file
#undef	Undefines a preprocessor macro
#ifdef	Returns true if this macro is defined
#ifndef	Returns true if this macro is not defined
#if	Tests if a compile time condition is true
#else	The alternative for #if
#elif	#else an #if in one statement
#endif	Ends preprocessor conditional
#error	Prints error message on stderr