Languages for Informatics

3 – Variables, Types, Operators and Control Flow

Department of Computer Science
University of Pisa
Largo B. Pontecorvo 3
56127 Pisa

Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
  1. Getting started with C Progamming
  2. Variables, Types, Operators and Control Flow
  3. Functions and Libraries
  4. Arrays and Pointers
  5. Structures
  6. Input and Output
- Basic system programming in Linux (10h)

Overview

1. Variables, Datatypes and Operators
    - Variables
    - Primitive data types
    - Operators
    - Data type conversion
    - Booleans
2. Control Flow
    - Loops
    - Hybrid

Motivation

- Most, if not all C programs contain **variables** that can be declared locally or globally.
- Their values are stored in a digital computer with certain accuracy, determined by their **type**
- C has rich variety of **math operators** including $+, -, \times, /, \%, ++$, and logical operators such as $==, !, >, <, ||, \&\&$, to manipulate variables.
- **Control flow** determines the order in which statements and function calls are executed.

Variables

- A **variable** is a name given to a storage area in the system's memory that can be manipulated.
  - For example, `int x=0, y=0; y=x+1`
  - Variables x, y;
  - Operator $+$.
- Rules for naming variables
  - can contain letters, digits and underscore
  - first element must be either letter or underscore
  - case sensitive
  - cannot contain keywords.

Declaration

The syntax to declare a variable is as follows:

```
type name_variable [=init value];
```

- type of the variable;
- name of the variable: name can have characters and digits; always start with a letter. Always keep in mind the general rules for naming variables and functions
- you can define an init value for the variable. It is strongly suggested to always init variables

Examples

- `int while`

Examples

- `int while`
- `int my$number`

Examples

- `int while`
- `int my$number`
- `int 2do`

Examples

- `int while`
- `int my$number`
- `int 2do`
- `int you2`

Examples

- `int while`
- `int my$number`
- `int 2do`
- `int you2`
- `int my_number`

Examples

- `int while` (incorrect due to keyword)
- `int my$number` (ok)
- `int 2do` (incorrect due to initial digit)
- `int you2` (ok)
- `int my_number` (ok)

Primitive Data Types

There a four primitive data types

- Integer `int` $\in \mathbb{Z}$ and its derivative types.
- Floating-point types `double, float` $\in \mathbb{R}$.
- Single characters `char`.

Primitive data types: `int`

`int`: an integer (placeholder `%d`)

- size of values represented by `int` depends on the machine where your code is running.

Primitive data types: `int`

`int`: an integer (placeholder `%d`)

- size of values represented by `int` depends on the machine where your code is running.
- the predefined function `sizeof()` gives the length in bytes of any type of variable in C. For instance:

```
#include <stdio.h>
int main()
{
  printf("%d\n",sizeof(int));
}
```

Primitive data types: `int`

`int`: an integer (placeholder `%d`)

- size of values represented by `int` depends on the machine where your code is running.
- the predefined function `sizeof()` gives the length in bytes of any type of variable in C. For instance:

```
#include <stdio.h>
int main()
{
  printf(”%d\n”,sizeof(int));
}
```

Result

```
4
```

Primitive data types: `int`

`int`: an integer (placeholder `%d`)

- size of values represented by `int` depends on the machine where your code is running.
- the predefined function `sizeof()` gives the length in bytes of any type of variable in C. For instance:

```c
#include <stdio.h>
int main()
{
  printf(”%d\n”,sizeof(int));
}
```

Result

```
4
```

- the modifiers `short`, `long` and `long long` handle integers of different length.

`signed int` vs. `unsigned int`

`signed` and `unsigned` are used for numbers with and with no sign, respectively. According to ISO C docs:

- The `int` data type is signed and has a minimum range of **at least** -32767 through 32767 (on a 16-bit machine). The actual values are given in `limits.h` as `INT_MIN` and `INT_MAX` respectively.
- An `unsigned int` has a minimal range of 0 through 65535 with the actual maximum value being `UINT_MAX` from that same header file.

Binary representation

- ISO C uses **two’s bit complement.**
- A 3-bit illustration:

Bits	Unsigned integer	Signed integer
000	0	+0
001	1	+1
010	2	+2
011	3	+3
100	4	-3
101	5	-2
110	6	-1
111	7	-0

Binary representation

- ISO C uses **two's bit complement.**
- A 3-bit illustration:

Bits	Unsigned integer	Signed integer
000	0	+0
001	1	+1
010	2	+2
011	3	+3
100	4	-3
101	5	-2
110	6	-1
111	7	-0

Note

Be aware of underflow and overflow !

Primitive data types: reals

`float`, **`double`**: used to represent real numbers (single and double precision)

```
float x=123.34;
double y=100.1e5; //scientific notation
```

Primitive data types: reals

`float`, **`double`**: used to represent real numbers (single and double precision)

```
float x=123.34;
double y=100.1e5; //scientific notation
```

- placeholders `%f` and `%lf`;
- `sizeof(float)` gives 4 bytes.
- `sizeof(double)` gives 8 bytes.

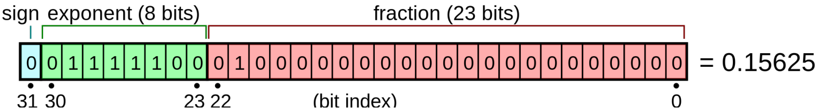


Figure: 32-bit representation according to IEEE 754 (Source:wikipedia)

Primitive data types char

`char`: a single byte representing a character (`ASCII` code). Placeholder `%c`.

```
char a='a'; //chars are single quoted
```

Primitive data types char

`char`: a single byte representing a character (`ASCII` code). Placeholder `%c`.

```
char a='a'; //chars are single quoted
```

- chars are integers in C.

```
int a='a';
printf("%c\n",a);
printf("%d\n",a);
```

Result

```
a
97
```

- Indeed, `97` corresponds to the `ASCII` code of `a`

Primitive data types: char (2)

- Some char constants and their integer values:

Char constant	’a’	’b’	...	’z’
Integer value	97	98	...	122
Char constant	’A’	’B’	...	’Z’
Integer value	65	66	...	90
Char constant	’0’	’1’	...	’9’
Integer value	48	49	...	57

Primitive data types: char (2)

- Some char constants and their integer values:

Char constant	'a'	'b'	...	'z'
Integer value	97	98	...	122
Char constant	'A'	'B'	...	'Z'
Integer value	65	66	...	90
Char constant	'0'	'1'	...	'9'
Integer value	48	49	...	57

Note

There is no relationship between a char constant and its digit counterpart: '2' is not 2.

Data type size

On a typical 32-bit machine:

- `int` is 32 bits
- `long` is 32 bits
- `long long` is 64 bits

On a typical 64-bit architecture:

- `int` is 32 bits
- `long` is 32 or 64 bits
- `long long` is 64 bits

On both:

- `float` is 32 bits
- `double` is 64 bits (always!)
- `char` is 8 bits
- `signed char` is 8 bits

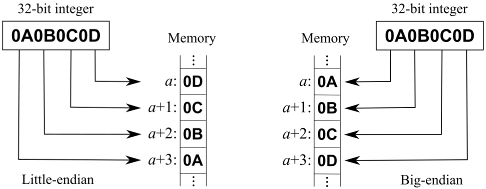
Big endian vs. little endian

In other words,

- `sizeof(char)<sizeof(short)≤sizeof(int)≤sizeof(long)`
- `sizeof(char)<sizeof(short)≤sizeof(float)≤`
  `≤sizeof(double)`
- Numerical data types span multiple bytes. Their order is relevant.

Big endian vs. little endian (2)

- **Little Endian**: The **least** significant byte is stored in the lowest memory address, and increases address for each more significant byte. Typical representation in all x86 (intel) compatible processors.
- **Big endian**: The **most** significant byte occupies the lowest memory address. Typical representation in ARM architectures.



(Source:wikipedia)

Constants

the attribute `const` can be applied to the declaration of any variable, with the effect of stating that its value will not change.

```
const double pi=3.141592;
const int five=5;
```

Note

An attempt to modify constants typically ends up in a compiling error!

Difference between `#define` and `const`????

Constants

the attribute `const` can be applied to the declaration of any variable, with the effect of stating that its value will not change.

```
const double pi=3.141592;
const int five=5;
```

Note

An attempt to modify constants typically ends up in a compiling error!

Difference between `#define` and `const`????

- `#define` is a directive of the pre-processor and replaced in the source code before compilation;
- a variable defined as `const` is manipulated from the compiler: it has a type and an address.

Arithmetic operators

In C there are the following operators:

$$+ \quad - \quad * \quad / \quad \%$$

representing the usual arithmetic operations. They are used to modify variables' values.

The value of *a* modulo *b* is the remainder of the division between *a* and *b*: for instance, $5\%3 = 2$.
Modulo operator cannot be applied to `float` and `double` variables.

Arithmetic operators (2)

- Operators obey to precedence and associativity rules, to establish how to evaluate an expression.

Arithmetic operators (2)

- Operators obey to precedence and associativity rules, to establish how to evaluate an expression.
- As usual, $+$ and $-$ have the same precedence, lower than $*$, $/$ and %.

Arithmetic operators (2)

- Operators obey to precedence and associativity rules, to establish how to evaluate an expression.
- As usual, $+$ and $-$ have the same precedence, lower than $*$, $/$ and $\%$.
- Moreover, addition and multiplication are both left and right associative, e.g. $(a \times b) \times c = a \times (b \times c)$ while subtraction and division, as used in conventional math notation, are inherently left-associative.

Arithmetic operators (3)

Other operators:

- **compact operators**: allow to execute an operation on a variable, and assign the result to the same variable. This means that an
  - expression `var op = expr`
  - is equivalent to `var = var op expr`

Arithmetic operators (3)

Other operators:

- **compact operators**: allow to execute an operation on a variable, and assign the result to the same variable. This means that an
  - expression `var op = expr`
  - is equivalent to `var = var op expr`
  - for instance: `j*=i+2` ⇔ `j=j*(i+2)`

Arithmetic operators (3)

Other operators:

- **compact operators**: allow to execute an operation on a variable, and assign the result to the same variable. This means that an
  - expression `var op = expr`
  - is equivalent to `var = var op expr`
  - for instance: `j*=i+2` ⇔ `j=j*(i+2)`
- **unitary increment/decrement operators**: comprising the operators `++` and `--`, respectively. They can be used either as prefix (before the variable: `++n`) or as suffix (after the variable: `n++`). The effect is the same, however:
  - `++n` execute the increment before using the value of `n`;
  - `n++` increments after using the value.

Data type conversion

- Type conversion occurs when the expression has data of mixed types.
- Common problem:

```
double a = 1.2;
int b = 2;
double c = b/a;  /* what is the precision of c? */
```

Data type conversion

- Type conversion occurs when the expression has data of mixed types.
- Common problem:

```
double a = 1.2;
int b = 2;
double c = b/a;  /* what is the precision of c? */
```

- When an operator is applied to values having different types, they are converted to the same type using some automatic rules.
- Data type is promoted **from lower to higher** accuracy.

Type conversion rules

Type conversion rules

- f(`int`,`float`) → f(`float`)
- f(`double`,other) →f(`double`)
- if either operand is unsigned, the other shall be converted to unsigned i.e., f(`unsigned int`,`long`) → f(`unsigned long`)
- Promotion: f(`unsigned char`,`unsigned short`) → f(`unsigned int`)

Example

```
short i = 1;
char ch = ’a’;
printf(”%zu,%zu,%zu\n”,sizeof(i),sizeof(ch),sizeof(ch+i)
    );
```

Example

```
short i = 1;
char ch = 'a';
printf("%zu,%zu,%zu\n",sizeof(i),sizeof(ch),sizeof(ch+i)
    );
```

```
2,1,4
```

Note

The type of `sizeof()` is `size_t` having format %zu.

Forced Type Conversion

- occurs when the value of the **larger** data type is converted to the value of the **smaller** data type
- The result may have **lower** precision.
- Type casting is the preferred method of forced conversation.

Examples

Example 1:

```
int e = 2.4;
printf("e = %d\n",e);
```

Examples

Example 1:

```
int e = 2.4;
printf("e = %d\n",e);
```

```
e = 2
```

Examples

Example 1:

```
int e = 2.4;
printf("e = %d\n",e);
```

```
e = 2
```

Example 2:

```
float x=12.4,y=8.3,z=4.7;
int result = x*y*z/100;
printf("result = %d\n",result);
```

```
result = 4
```

During evaluation the integers would be **first promoted** to float and so would be the result, but **then occurs a truncation** to `int`.

Type Casting

Beside automatic conversions, it is possible to enforce conversions, by using casting, as follows:

`(type)` expression;

Example:

```
int sum, n;
float avg;
...
avg = sum/n; /* integer division */
avg = (float)sum/n; /* real numbers division*/
```

The cast operator in parentheses has higher precedence, and it associates from right to left. Thus `(float)sum/n` is equivalent to `((float)sum)/n`

Booleans and relational operators

In C there does not exist Boolean type. It is represented through an `int`:

- `0` represents FALSE;
- A value `different from 0` (typically 1) represents TRUE.

Booleans and relational operators

In C there does not exist Boolean type. It is represented through an `int`:

- `0` represents FALSE;
- A value `different from 0` (typically 1) represents TRUE.

Logical operators:

- `!`: NOT (unary operator). Example: `!a`;
- `&&`: AND (binary operator). Example: `a && b`;
- `||`: OR (binary operator). Example: `a || b`;

Returns an integer value: either 0 or 1, depending on the value (false/true) of the expression.

Booleans and relational operators

In C there does not exist Boolean type. It is represented through an `int`:

- `0` represents FALSE;
- A value `different from 0` (typically 1) represents TRUE.

Logical operators:

- `!`: NOT (unary operator). Example: `!a`;
- `&&`: AND (binary operator). Example: `a && b`;
- `||`: OR (binary operator). Example: `a || b`;

Returns an integer value: either 0 or 1, depending on the value (false/true) of the expression.
Other operators on **single bits**: **shift** operators (`<<`, `>>`), AND (`&`), OR (`|`), XOR (`^`) ...

Example

Differences between bitwise and logical AND operators in C

```
int main() {
    int x = 3;  //...0011
    int y = 7;  //...0111

    if (y > 1 && y > x)
      printf("y is greater than 1 AND x\n");

    int z = x & y;   // 0011
    printf ("z = %d", z);
    return 0;
}
```

Output

```
y is greater than 1 AND x
z = 3
```

Relational Operators (2)

Checking for equality is essential in C

- The equality operator == compares primitive types such as char, int, float, etc.
  - e.g. `1==1` results in 1
  - e.g. `'A'=='a'` results in 0
- The inequality operator `!=` returns true if its operands are not equal, false otherwise.
  - e.g. `1!=1` results in 0
  - e.g. `'A'!='a'` results in 1
  - e.g. `0.999!=1` results in 1

Note

C cannot compare floating-point values due to rounding errors

Relational operators (3)

Other relational operators are:

$$< \quad > \quad <= \quad >=$$

they are all binary: they take two expressions, and return a result of type `int` that can be either `0` or `1`.

For instance, the expression `a<b`:

- if `a` is less than `b`, the value `1` (true);
- otherwise, the value is `0` (false).

Relational operators (3)

Other relational operators are:

$$< \quad > \quad <= \quad >=$$

they are all binary: they take two expressions, and return a result of type `int` that can be either `0` or `1`.

For instance, the expression `a<b`:

- if `a` is less than `b`, the value `1` (true);
- otherwise, the value is `0` (false).

Quiz

What is the output of the following code ?

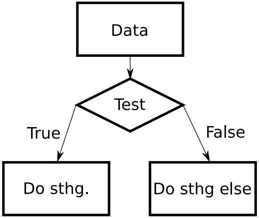
```
#include<stdio.h>
 int main() {
   int const a=5;

   a++;
   printf("a = %d",a);
}
```

1. a = 5
2. a = 6
3. Runtime error
4. Compile error

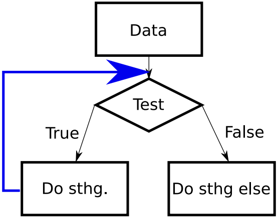
Control Flow

Control flow describes the order in which individual statements, instructions or function calls of our C program are executed.

- For example, $\min_u \sum_{i=1}^{10} x_i(u) \neq \sum_{i=1}^{10} \min_u x_i(u)$.
- C provides two styles of flow control



Branching



Looping

Branching and Looping

- **Branching**: `if`, `else` and `else if`, `switch`, `break` and `continue`
- **Looping**: `while`, `for`, `do-while`
- **Hybrid**: `goto` (branching or looping).

The `If` Statement

```
if (test_condition_is_TRUE) {
/* Do some stuff */
}
```

- test the condition
- if `TRUE`, evaluate body
- Otherwise, do nothing,

The `If` Statement

```
if (test_condition_is_TRUE) {
/* Do some stuff */
}
```

- test the condition
- if `TRUE`, evaluate body
- Otherwise, do nothing,

Example:

```
int x = 3;
if (x%2) /* if condition is true */
  printf("The number is odd.");
```

```
The number is odd.
```

The `Else` Keyword

```
if (test_condition_is_TRUE) {
/* Do some stuff */
}
else { /*  test condition is FALSE */
}
```

- Optional
- test expression is `FALSE`
  - statement inside `if` body is **skipped**
  - statement inside `else` body is **executed**

The `Else` Keyword

```
if (test_condition_is_TRUE) {
/* Do some stuff */
}
else { /*  test condition is FALSE */
}
```

- Optional
- test expression is `FALSE`
  - statement inside `if` body is **skipped**
  - statement inside `else` body is **executed**

Example:

```
int x = 2;
if (x%2) printf("The number is odd.");
else printf("The number is even.");
```

```
The number is even.
```

The `Else if` Keyword

- Additional alternative control path

```
if (test_condition_1_is_TRUE)  /* Do some stuff */
else if (test_condition_2_is_TRUE) /* Do sthg else */
else /* Do something else if all above false */
}
```

The `Else if` Keyword

- Additional alternative control path

```
if (test_condition_1_is_TRUE)  /* Do some stuff */
else if (test_condition_2_is_TRUE) /* Do sthg else */
else /* Do something else if all above false */
}
```

Example:

```
int i=0;
if (i==0) printf("The number is zero.\n");
else if (i%2)  printf("The number is odd.\n");
else printf("The number is non-zero and even.\n");
```

```
The number is zero.
```

The `switch` statement

The `switch` statement is alternative conditional.

The `switch` statement

The `switch` statement is alternative conditional.
Syntax:

```
switch (argument) {
     case label_1: instructions_1
          break;
     ...
     case label_n: instructions_n
          break;
     default : instructions_default
}
```

The `switch` statement (cont'd)

Semantic:

- Input must be `int` or `char`
- The `argument` is evaluated and compared against the different (constant) case labels;
- when argument corresponds to some case label, the respective instructions are executed, followed by a `break` to the next line following the `switch` statement;
- otherwise, (optional) `default` is executed.

The `switch` statement (cont'd)

Example:

```
int day;
...
switch (day) {
      case 1: printf("Monday\n");
              break;        /*  exit statement */
      case 2: printf("Tuesday\n");
              break;
      case 3: printf("Wednesday\n");
              break;
      case 4: printf("Thursday\n");
              break;
      case 5: printf("Friday\n");
              break;
      default: printf("Weekend\n");
}
```

The `switch` statement (cont'd)

Multiple cases:

```
int day = 5;
...
switch (day) {
    case 1:          /*  break removed otherwise! */
    case 3:
    case 5:
    case 7: printf(”Odd day\n”);
            break;
    case 2:
    case 4:
    case 6: printf(”Even day\n”);
            break:
    default: printf(”Invalid day\n”);
}
```

```
Odd day
```

The `break` and `continue` keywords

- The `break` keyword provides an early exit from `for`, `while` and `do`, just as from `switch`

```
#include <stdio.h>
int main () {
   char c;
   while(1) {    /* infinite loop */
      printf("Shall we make a break? (y/n) ");
      c = getchar();
      if( c == 'y') break;
   }
   return 0;
}
```

Note

Break works fine but `Shall we make a break? (y/n)` will be printed **2x**. Why?

The `break` and `continue` keywords

- The `continue` keyword skips rest of `for`, `while` and `do`-loop.

```
#include <stdio.h>
int main () {
   char c = 'n';
   while(1) {    /* infinite loop */
      puts("Shall we make a break? (y/n) ");
      scanf(" %c",&c);
      if (c == 'n') continue;
      if (c == 'y') break;
      printf("Your answer is unclear. ");
   }
   return 0;
}
```

Loops: The `while` loop

- A loop that executes a block of statements over and over again until a given condition returns `FALSE`.

```
while (test_condition_is_TRUE)
{
/* sequence of statements */
}
```

Example (1)

```c
#include <stdio.h>
int main () {
  char c = 'y';  /* Initialize to a value as true in
    while*/
  while(c=='y') {
    printf("Keep going ? (y/n) ");
    scanf(" %c",&c);
  }
  return 0;
}
```

Output

```
Keep going ? (y/n) y
Keep going ? (y/n) n
$
```

Example (2)

```
#include <stdio.h>
int main () {
  char c;
  while(1) {  /* condition always true */
    printf("Gimme a char: ");
    scanf(" %c",&c);
  }              /* do forever */
  return 0;
}
```

Output

```
Gimme a char: g
Gimme a char: f
...
Gimme a char: z
^C
```

The `for` loop

- A **counting** loop that executes a block of statements over and over again until a given condition returns `FALSE`.
- Internal counter in contrast to `while`-loop.

```
for (initialization; test_condition;
    increment_or_decrement_counter)
{
/* sequence of statements */
}
```

- Internal counter is only updated **after** the block of statements
  - true for both pre/post counter (++counter/counter++).

The `for` loop

- Some arguments of `for` function can be **empty.**

```
for (int i=0;;i++) { /* infinite loop */
}

for (int i=3;;) { /* keeps at i=3 */
}
```

- Multiple declarations are separated by **comma.**
- Expressions are evaluated left-to-right

Example

```
for (i=1,j=1;i<10;j*=i, i++)
{
/* first j=j*i then i=i+1 */
}
```

```
for (i=1,j=1;i<10;i++,j*=i)
{
/* first i=i+1 then j=j*i */
}
```

Counter variables	Counter variables
j= 1, i= 1	i= 1, j= 1
j= 1, i= 2	i= 2, j= 2
j= 2, i= 3	i= 3, j= 6
j= 6, i= 4	i= 4, j= 24
j= 24, i= 5	i= 5, j= 120
j= 120, i= 6	i= 6, j= 720
j= 720, i= 7	i= 7, j= 5040
j= 5040, i= 8	i= 8, j= 40320
j= 40320, i= 9	i= 9, j= 362880

The `do-while` loop

- A `do-while` loop executes the body of the loop and only **then** tests some condition.
    - will be executed at least once, even if the condition is `FALSE`.

```
do { /* execute statements */
} while (test_condition_is_TRUE);
```

Example

```
#include <stdio.h>
int main () {
  int i=4;
  do {  /* in any case */
    printf("My integer: %d \n",i);
    i++;
  } while (i<5);
  return 0;
}
```

Output

```
My integer: 4
$
```

Hybrid: The `goto` statement

- `goto` jumps unconditionally to a named location in the code, i.e.
- a **label** followed by a colon ":", that
- can be placed anywhere (within the same function).

```
#include <stdio.h>
int main() {
  int a = 1;
 LOOP:do {
    if( a == 3) {
      a = a + 1;  /* skip iterating */
      goto LOOP;
    }
    printf("value of a: %d\n", a);
    a++;
  }while( a < 5 );
return 0;
}
```

The `goto` statement

Compiling and executing the program, we obtain

```
$ gcc -Wall -o myprogram *.c
$ ./myprogram
value of a:  1
value of a:  2
value of a:  4
```

Quiz

What is the output of the code?

```
#include<stdio.h>
int main() {
  int a=0,i=0;
  for (i=0;i<3;i++) {
  a++
  continue;
  }
printf(”a = %d\n”,a);
}
```

Quiz (2)

The keyword getting out of recursion is:

1. `break`
2. `return`
3. `exit`
4. Both 1) and 2)