The C Language : Common Programming Concepts

David Bouchet

david.bouchet.epita@gmail.com

Integer Types

Signed Integers

```
(signed) char              // 8 bits
(signed) short             // 16 bits
(signed) int               // 32 bits
(signed) long              // 64 bits
(signed) long long         // 64 bits
```

Sizes are given for 64-bit architectures

LP64 data model on Linux

Integer Types

Unsigned Integers

```
unsigned char          // 8 bits
unsigned short         // 16 bits
unsigned int           // 32 bits
unsigned long          // 64 bits
unsigned long long     // 64 bits
```

Sizes are given for 64-bit architectures

LP64 data model on Linux

Floating-Point Types

IEEE 754 Standard

```
float     // 32 bits (single precision)
double    // 64 bits (double precision)
```

Other Types

`size_t`

- 64-bit unsigned integer (LP64)
- Used for size measurement
- Used for array indexes
- Defined in most header files of the standard library (e.g. <stddef.h>)

Other Types

`ssize_t`

- 64-bit signed integer (LP64)
- Signed version of size_t
- Accepts negative values (for errors and differences)
- Defined in some header files of the standard library (e.g. <stdlib.h>)

Other Types

```
void
```

Absence of Type

- Used as function return type when no return value is expected.
- Can be used as function parameter type when no parameters are passed into the function.

The *main()* Function

The *main()* function is the entry point of the program.

It should return an int value.

- If no error occurred
  → Should return 0
- If any error occurred
  → Should return a value different from 0

The *main()* Function

We can use constants defined in *<stdlib.h>*:

- EXIT_SUCCESS
- EXIT_FAILURE

```
#include <stdlib.h>

int main()
{
    // Some instructions
    // No error occurred

    return EXIT_SUCCESS;
}
```

```
#include <stdlib.h>

int main()
{
    // Some instructions
    // An error occurred

    return EXIT_FAILURE;
}
```

Variables

General syntax

```
<type> <identifier> = <value>;
```

or

```
<type> <identifier>;
<identifier = <value>;
```

Example

```
int a = 5;
int b;
b = 5;
```

The *sizeof()* Operator

The *sizeof()* operator returns the size in bytes of an expression or a type.

The return type is size_t.

```
unsigned long a = 5;
size_t l1 = sizeof(a);      // l1 = 8
size_t l2 = sizeof(short); // l2 = 2
```

Signed Integers

```
char c = 'A';
short h = 100;
int i = 200;
long l = 300;

printf("c = %c\n", c);
printf("c = %hhi\n", c);
printf("h = %hi\n", h);
printf("i = %i\n", i);
printf("l = %li\n", l);
```

```
c = A
c = 65
h = 100
i = 200
l = 300
```

Unsigned Integers

```
unsigned char c = 'A';
unsigned short h = 100;
unsigned int i = 200;
unsigned long l = 300;

printf("c = %c\n", c);
printf("c = %hhu\n", c);
printf("c = %hhx\n", c);
printf("h = %hu\n", h);
printf("i = %u\n", i);
printf("l = %lu\n", l);
```

```
c = A
c = 65
c = 41
h = 100
i = 200
l = 300
```

The printf() Function

size_t and ssize_t

```
size_t l1 = 42;
ssize_t l2 = -l1;

printf("l1 = %zu\n", l1);
printf("l2 = %zi\n", l2);
```

```
l1 = 42
l2 = -42
```

The printf() Function

float and double

```
float f = 42.0;
double d = 72.0;

printf("f = %f\n", f);
printf("d = %f\n", d);
```

```
f = 42.000000
d = 72.000000
```

Conditions

No Boolean Type!

Conditions use integers

- 0 is equivalent to FALSE
- ≠ 0 is equivalent to TRUE

Relational Operators

```
char a = 5;
char b = 0;
```

```
a == 5 => 1
a != 5 => 0
a  > 5 => 0
a >= 5 => 1
a  < 5 => 0
a <= 5 => 1
    !a => 0
    !b => 1
```

The *if*, *else if* and *else* Statements

General Syntax

```
if (condition)
{
    // ...
}
else if (condition)
{
    // ...
}
else
{
    // ...
}
```

The *else* and *else if* statements are optional.

The *if*, *else if* and *else* Statements

Common Conditions

```
if (a)
{
    // ...
}
if (!a)
{
    // ...
}
```

```
if (a != 0)
{
    // ...
}
if (a == 0)
{
    // ...
}
```

Shorter

More explicit

The *for* Statement

General Syntax

```
for (init; condition; post)
{
    // ...
}
```

Example

```
for (int n = 0; n < 5; n++)
{
    printf("n = %i\n", n);
}
```

```
n = 0
n = 1
n = 2
n = 3
n = 4
```

The *while* Statement

General Syntax:

```
while (condition)
{
    // ...
}
```

Example

```
short n = 0;
while (n < 5)
{
    printf("n = %hi\n", n);
    n++;
}
```

```
n = 0
n = 1
n = 2
n = 3
n = 4
```

The *do...while* Statement

General Syntax:

```
do
{
    // ...
} while (condition);
```

Example

```
short n = 0;
do
{
    printf("n = %hi\n", n);
    n++;
} while (n < 5);
```

```
n = 0
n = 1
n = 2
n = 3
n = 4
```

The *break* and *continue* Statements

The ***break*** and ***continue*** statements can be used in loop bodies (e.g. ***for***, ***while***, ***do...while***)

- ***break***: Terminates the loop.
- ***continue***: Goes to the next iteration.

The *switch...case* Statement

```
switch (value)
{
    case const_1:
        // ...
        break;

    case const_2:
        // ...
        break;

    // etc.

    default:
        // ...
}
```

```
int a = 10;

switch (a)
{
    case 0:
        printf("a is null.");
        break;

    case 100:
        printf("a is one hundred.\n");
        break;

    default:
        printf("a is not null.\n");
        printf("a is not one hundred.\n");
}
```

```
a is not null.
a is not one hundred.
```

Enumerations

Declaration

```
enum <enum_name>
{
    const_1,
    const_2,
    // ...
    const_N,
};
```

```
enum <enum_name>
{
    const_1 = 0,
    const_2 = 15,
    // ...
    const_N = 3,
};
```

Example

```
int main()
{
    enum color
    {
        red,        // 0
        green,      // 1
        blue,       // 2
    };

    enum color c1 = red;
    enum color c2 = green;
    enum color c3 = blue;

    return 0;
}
```

Ternary Operator

```
if (a)
    x = b;
else
    x = c;
```

```
x = a ? b : c;
```

Variable Scope

```
int a = 10;

{
    int b = 42;
}

printf("a = %i\n", a);
printf("b = %i\n", b);
```

Scope of ‘b’

Scope of ‘a’

Error: ‘b’ is not declared

Variable Scope

Assuming that ‘a’ is declared and initialized

```
if (a)
{
    int b = 42;
}
else
{
    int b = -42;
}
printf("b = %i\n", b);
```

Variable Scope

Assuming that ‘a’ is declared and initialized

```
if (a)
{
    int b = 42;
}
else
{
    int b = -42;
}
printf("b = %i\n", b);
```

Scope of ‘b’

Scope of ‘b’

Error: ‘b’ is not declared

Variable Scope

Assuming that ‘a’ is declared and initialized

```
int b;
if (a)
{
    b = 42;
}
else
{
    b = -42;
}
printf("b = %i\n", b);
```

Scope of ‘b’

Right

Variable Scope

```
for (int i = 0; i < 10; i++)
    printf("i = %i\n", i);

printf("i = %i\n", i);
```

Variable Scope

```
for (int i = 0; i < 10; i++)
    printf("i = %i\n", i);

printf("i = %i\n", i);
```

Scope of ‘i’

Error: ‘i’ is not declared

Global Variables

```c
#include <stdio.h>

int x = 42;

void inc_x()
{
    x++;
}

int main()
{
    printf("x = %i\n", x);
    inc_x();
    printf("x = %i\n", x);
    return 0;
}
```

```
x = 42
x = 43
```

Global Variables – Multiple Files

main.c

```c
#include <stdio.h>

extern int x;
int inc_x();

int main()
{
    printf("x = %i\n", x);
    inc_x();
    printf("x = %i\n", x);
    return 0;
}
```

other.c

```c
int x = 42;

void inc_x()
{
    x++;
}
```

```
x = 42
x = 43
```

Constants – *#define*

General syntax

```
#define <LABEL> <value>
```

Example

```
#define PI 3.14

int main()
{
    float diameter = 50.0;
    float circumference = diameter * PI;
    // ...
}
```

Constants – *const*

General syntax

```
const <type> <identifier> = <value>;
```

Example

```
const float PI = 3.14;

int main()
{
    float diameter = 50.0;
    float circumference = diameter * PI;
    // ...
}
```

Implicit Type Casting

```
float f = 42.75;
int i = f ;
printf("i = %i\n", i);

f = i;
printf("f = %f\n", f);
```

```
i = 42
f = 42.000000
```

Explicit Type Casting

```
float f = 42.75;
int i = (int)f ;
printf("i = %i\n", i);

f = (float)i;
printf("f = %f\n", f);
```

```
i = 42
f = 42.000000
```

Overflow

```
unsigned char c = 250;
c += 10;
printf("c = %hhu\n", c);
```

```
c = ?
```

Overflow

```
unsigned char c = 250;
c += 10;
printf("c = %hhu\n", c);
```

```
c = 4
```

Overflow

```
unsigned char c = 250;
c += 10;
printf("c = %hhu\n", c);
```

```
c = 4
```

$$260_{10} = 1\ \underbrace{0000\ 0100}_{8\text{ bits} \Rightarrow 4}{}_{2}$$

Overflow

```
short i = 129;
char c = i;
printf("c = %hhi\n", c);
```

```
c = ?
```

Overflow

```
short i = 129;
char c = i;
printf("c = %hhi\n", c);
```

```
c = -127
```

Overflow

```
short i = 129;
char c = i;
printf("c = %hhi\n", c);
```

```
c = -127
```

129_{10} => $\underset{+}{0}000\ 0000\ \underbrace{1000\ 0001}_{2}$

$\underset{-}{1}000\ 0001_2$ => 2'c = $\underbrace{0111111}_{127}$

The exit() Function

It causes normal process termination.

```
#include <stdlib.h>

void f()
{
    exit(EXIT_FAILURE);
}

int main()
{
    f();
    return 0;
}
```

```
$ ./a.out
$ echo $?
1
```

Never executed

The errx() Function

Similar to *exit()* with extra features:

- It sends an error message to the standard error.
- Printf-like syntax().
- It appends a newline character to the message.

The errx() Function

```
#include <err.h>

void f(int code)
{
    errx(code, "Error code: %i", code);
}

int main()
{
    f(42);
    return 0;
}
```

Never executed

```
$ ./a.out
a.out: Error code: 42
$ echo $?
42
```