Computer Organization and Architecture

C types and expressions

Utah Tech University

Spring 2025

An example

```c
#include <stdio.h>

double circularArea(double r);

int main(void) {
    double radius = 1.0, area = 0.0;
    printf("    Areas of Circles\n\n");
    printf("     Radius          Area\n"
           "-----------------------\n");
    area = circularArea(radius);
    printf("%10.1f    %10.2f\n", radius, area);
    radius = 5.0;
    area = circularArea(radius);
    printf("%10.1f    %10.2f\n", radius, area);
    return 0;
}

double circularArea(double r) {
    const double pi = 3.1415926536;
    return pi * r * r;
}
```

Notes

- Header files
- Preprocessor
- Prototypes and forward/external references
- `main`
- `printf`
- Variable declarations

The most common `printf` verbs:

`%d, %u`	decimal integer
`%x, %X, %o`	hexadecimal, octal
`%f, %e, %g`	floats: 392.65, 3.9265e+2, shortest
`%c`	single characacter
`%s`	string
`%p`	pointer address
`%%`	literal percent sign (no operand)

Compiling

The compilation process:

1. Compile source file 1 (with headers, preprocessor)
   - Creates an *object* file with `.o` extension
2. Compile source file 2 (with headers, preprocessor)
   - Another object file
3. ...
4. Link with standard library (and other libraries)
   - Output is an executable file, default `a.out`
   - Also called a *binary*
   - See also: `/bin` directory

Memory map

```
  Top of memory
+-----------------------------------------------+
| Stack (grows down as functions are called)    |
| ...                                           |
+-----------------------------------------------+
| ...                                           |
| Heap (grows up, arranged by malloc/free)      |
+-----------------------------------------------+
| Data segment (arranged by linker)             |
+-----------------------------------------------+
| Text segment (arranged by linker)             |
+-----------------------------------------------+
| (unused addresses)                            |
+-----------------------------------------------+
  Bottom of memory (address zero)
```

Expression statements

Any expression followed by a semicolon is a statement. This is normally only used when the expression has a side effect.

```
y = x;                          // An assignment
sum = a + b;                    // Calculation and assignment
++x;
printf("Hello, world\n");       // A function call

100;                            // Correct, but not very useful
y < x;

(void) unused_variable;         // suppress warnings about an unused variable
```

A statement can also consist of a semicolon by itself: this is called a *null statement*:

```
for (i = 0; s[i] != '\0'; i++)
    ;
```

Block statements

A *compound statement*, call a *block* for short, groups statements and declarations between braces to form a single statement. Note that it is **not** terminated by a semicolon:

```
{
    double result = 0.0, x = 0.0;           // declarations
    static long status = 0;                 // these variables are scoped
    extern int limit;                       // to the current block

    x++;
    if (status == 0) {                      // new block
        int i = 0;                          // i is scoped to the if block
        while (status == 0 && i < limit) {  // another new block
            // ...
        }
    } else {                                // and another
        // ...
    }
}
```

while loops

A `while` loop executes a statement repeatedly as long as the controlling expression is true:

- `while (`*expression*`)` *statement*

```
#include <stdio.h>

int main(void) {
    double x = 0.0, sum = 0.0;
    int count = 0;
    printf("\t--- Calculate Averages ---\n");
    printf("\nEnter some numbers:\n"
           "(Type a letter to end your input)\n");
    while (scanf("%lf", &x) == 1) {
        sum += x;
        count++;
    }
    if (count == 0)
        printf("No input data!\n");
    else
        printf("The average of your numbers is %.2f\n", sum/count);
    return 0;
}
```

for loops

A `for` loop has more loop logic contained in the statement itself:

- `for (`*expression1*`;`*expression2*`;`*expression3*`)` *statement*
  - *expression1*: **Initialization**. Evaluated only once, before the first evaluation of the controlling expression, to perform any necessary initialization
  - *expression2*: **Controlling expression**. Tested before each iteration. Loop execution ends when this expression evaluates to false.
  - *expression3*: **Adjustment**. An adjustment, such as incrementation of a counter, performed *after* each loop iteration and before *expression2* is tested again.

```
for (i = 0; i < LENGTH; i++)
    arr[i] = 2*i;

for (;;) { ... }

for (; more_to_do(x); ) { }

i = 0;
while (i < LENGTH) {
    arr[i] = 2*i;
    i++;
}
```

```
// starting with C99 you can declare a variable
// in place of expression1
for (int i = 0; i < LENGTH; i++)
    arr[i] = 2*i;

for (int i = 0, j = strlen(str)-1; i < j; i++, j--)
    ch = str[i], str[i] = str[j], str[j] = ch;
```

do while loops

The `do ... while` loop is *bottom driven*

- `do` *statement* `while (`*expression*`);`

The body statement is executed once before the controlling expression is evaluated for the first time, so at least one iteration of the loop is always performed.

```
do {
    int command = getCommand();
    performCommand(command);
} while (command != END);
```

```
char *strcpy(char *s1, const char *s2) {
    int i = 0;
    do
        s1[i] = s2[i];
    while (s2[i++] != '\0');
    return s1;
}
```

Nested loops

A loop body can be any simple or block statement and may include other loop statements.

```
void bubbleSort(float arr[], int len) {
    int isSorted;
    do {
        isSorted = 1;
        len--;
        for (int i = 0; i < len; i++) {
            if (arr[i] > arr[i+1]) {
                isSorted = 0;
                float temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
        }
    } while (!isSorted);
}
```

if statements

An `if` statement has the following form:

- `if (`*expression*`)` *statement1* [`else` *statement2*]

```
double power(double base, unsigned int exp) {
    if (exp == 0) return 1.0;
    else return base * power(base, exp-1);
}
```

If several `if` statements are nested, then an `else` clause always belongs to the last `if` (on the same block nesting level that does not yet have an `else` clause. If there is any doubt or the nesting is complex, use a block and make sure your indentation matches the code.

```
if (n > 0)
    if (n%2 == 0)
        puts("n is positive and even");
    else
        puts("n is positive and odd");
```

```
if (n > 0) {
    if (n%2 == 0)
        puts("n is positive and even");
} else
    puts("n is negative or zero");
```

if statements

To select one of more than two alternative statements, `if` statements can be cascaded in an `else if` chain. Each new `if` statement is simply nested in the `else` clause of the preceding `if` statement:

```
double spec = 10.0, measured = 10.3, diff;
/* ... */
diff = measured - spec;

if (diff >= 0.0 && diff < 0.5)
    printf("Upward deviation: %.2f\n", diff);
else if (diff < 0.0 && diff > -0.5)
    printf("Downward deviation: %.2f\n", diff);
else
    printf("Deviation out of tolerance!\n");
```

switch statements

A `switch` statement causes a jump to one of several statements according to the value of an integer expression. The cases must all be constants:

- `switch (`*expression*`)` *statement*

```
switch (menu()) {   // menu() returns an int
case 'a':
case 'A':
    action1();
    break;
case 'b':
case 'B':
    action2();
    break;
default:
    putchar('\a');
}
```

- Cases *fall through* so you must add a `break` statement to exit the `switch`
- To declare variables in a case you must introduce a nested block

break

The `break` statement can occur only in the body of a loop or a `switch` statement and jumps to the first statement after the loop or `switch` in which it is immediately contained:

- `break;`

```
int getScores(short scores[], int len) {
    puts("Please enter scores between 0 and 100.\n"
         "Press <Q> and <Enter> to quit.\n");
    int i;
    for (i = 0; i < len; i++) {
        printf("Score No. %2d: ", i+1);
        if (scanf("%hd", &scores[i]) != 1)
            break;
        if (scores[i] < 0 || scores[i] > 100) {
            printf("%d: Value out of range.\n", scores[i]);
            break;
        }
    }
    return i;
}
```

continue

The `continue` statement can be used only within the body of a loop, and causes the program to skip over the rest of the current iteration of the loop:

- `continue;`

```
int getScores(short scores[], int len) {
    puts("Please enter scores between 0 and 100.\n"
         "Press <Q> and <Enter> to quit.\n");
    int i;
    while (i < len) {
        printf("Score No. %2d: ", i+1);
        if (scanf("%hd", &scores[i]) != 1)
            break;
        if (scores[i] < 0 || scores[i] > 100) {
            printf("%d: Value out of range.\n", scores[i]);
            continue;   // discard this value and read in another
        }
        i++;
    }
    return i;
}
```

goto

The `goto` statement causes an unconditional jump to another statement in the same function. The destination is specified by the name of a label.

```
bool calculate(double arr[], int len, double *res) {
    bool error = false;
    if (len < 1 || len > MAX_ARR_LENGTH)
        goto error_exit;
    for (int i = 0; i < len; i++) {
        // do stuff that might set error
        if (error)
            goto error_exit;
        // continue calculation and set *res
    }
    return true;

error_exit:
    *res = 0.0;
    return false;
}
```

- You should never use `goto` to jump into a block from outside if the jump skips over declarations or statements that initialize variables
- Code that makes heavy use of `goto` statements is hard to read and should be avoided. The most common uses for `goto` are:
  - Consolidating exits from a function (see example)
  - Simulating a `break` or `continue` from an inner loop directly to an outer loop

return

The `return` statement ends execution of the current function and jumps back to where the function was called:

- `return` [*expression*]`;`

The *return value* is converted to the function's return type if necessary.

```
int min(int a, int b) {
    if (a < b) return a;
    else       return b;
}
```

See also

```
int min(int a, int b) {
    return a < b ? a : b;
}
```

For functions with `void` return type the expression is omitted.

Operators

Precedence	Operators	Associativity
1.	Postfix operators: `[]` `()` `.` `->` `++` `--` (*type name*){*list*}	Left to right
2.	Unary operators: `++` `--` `!` `~` `+` `-` `*` `&` `sizeof`	Right to left
3.	The cast operator: (*type name*)	Right to left
4.	Multiplicative operators: `*` `/` `%`	Left to right
5.	Additive operators: `+` `-`	Left to right
6.	Shift operators: `<<` `>>`	Left to right
7.	Relational operators: `<` `<=` `>` `>=`	Left to right
8.	Equality operators: `==` `!=`	Left to right
9.	Bitwise AND: `&`	Left to right
10.	Bitwise exclusive OR: `^`	Left to right
11.	Bitwise OR: `|`	Left to right
12.	Logical AND: `&&`	Left to right
13.	Logical OR: `||`	Left to right
14.	The conditional operator: `?` `:`	Right to left
15.	Assignment operators: `=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=`	Right to left
16.	The comma operator: `,`	Left to right

Memory addressing operators

Operator	Meaning	Example	Result
`&`	Address of	`&x`	Pointer to `x`
`*`	Indirection operator	`*p`	The object or function that `p` points to
`[]`	Subscripting	`x[y]`	The element with index `y` in the array `x`
`.`	Structure or union member designator	`x.y`	The member named `y` in the structure or union `x`
`->`	Structure or union member designator by reference	`p->y`	The member named `y` in the structure or union that `p` points to

```
float x, *ptr;
ptr = &x;           // OK: Make ptr point to x.
ptr = &(x+1);       // Error: (x+1) is not an lvalue

*ptr = 1.7;         // Assign the value 1.7 to the variable x
++(*ptr);           // and add 1 to it. (note: parentheses are superfluous)
```

Element of arrays

The *subscript operator* `[]` allows you to access individual elements of an array. In its simplest form:

```
myarray[i]  // note: arrays always start with element 0
```

An expression of the form `x[y]` is equivalent to

```
(*((x)+(y)))
```

Either `x` or `y` must have a type that is a pointer to an object type, and the other must have an integer type. This follows the rules of *pointer arithmetic* and means that `x[y]` and `y[x]` are equivalent.

```
#include <stdlib.h>
#define ARRAY_SIZE 100
// ...
    double *pArray = NULL; int i = 0;
    pArray = malloc(ARRAY_SIZE * sizeof(double));
    if (pArray != NULL) {
        for (i = 0; i < ARRAY_SIZE; i++)
            pArray[i] = (double) rand() / RAND_MAX;
    }
// note: pArray[i] or i[pArray] or *(pArray + i)
```

Compound literals

A *compound literal* lets you define literals with any object type and consiste of an object type in parentheses followed by an initialization list in braces:

```
float *fPtr = (float []){ -0.5, 0.0, +0.5 };
```

```
#include "database.h"
// includes: struct Pair { long key; char value[32]; };
insertPair(&db, &(struct Pair){ 1000L, "New York JFK Airport" });
```

Integer types

The basic signed integer types

Type	Synonyms
signed char	
int	signed, signed int
short	short int, signed short, signed short int
long	long int, signed long, signed long int
long long	long long int, signed long long, signed long long int

For each signed type, there is a corresponding unsigned type of the same memory size and alignment

Type	Synonyms
_Bool	bool (defined in stdbool.h)
unsigned char	
unsigned int	unsigned, unsigned int
unsigned short	unsigned short int
unsigned long	unsigned long int
unsigned long long	unsigned long long int

Note: `char` can be signed or unsigned. Be explicit if it matters.

Integers with exact width

The header `stdint.h` defines some aliases for when *width* matters

Type	Meaning	Implementation
`intN_t`, `uintN_t`	An integer type whose width is exactly *N* bits	Optional
`int_leastN_t`, `uint_leastN_t`	An integer type whose width is a least *N* bits	Required for *N*=8, 16, 32, 64
`int_fastN_t`, `uint_fastN_t`	The fastest process with width at least *N* bits	Required for *N*=8, 16, 32, 64
`intmax_t`, `uintmax_t`	The widest integer type implemented	Required
`intptr_t`, `uintptr_t`	An integer type wide enough to store a pointer	Optional

Floating-point types

Floating-point types:

- `float`: 32 bits, ±3.4×10^38, 6–7 digits of precision
- `double`: 64 bits, ±1.7×10^308, 15–16 digits of precision
- `long double`: 80 bits (we will mostly ignore these)

Enumerated types

A special type of integer where you name and list all of the defined values:

```
enum color { black, red, green, yellow, blue, white=7, gray };
```

Here `color` is the *tag* and the color names are the *enumeration constants*.

To use it:

```
enum color bgColor = blue, fgColor = yellow;
void setFgColor(enum color fgc);
```

Different constants may have the same value:

```
enum { OFF, ON, STOP=0, GO=1, CLOSED=0, OPEN=1 };
```

Note there is no tag. This is useful when you just want to define some constants, but not necessarily a new type to go with them.

The void type

The `void` type specifier means no type is available. This is useful:

```c
// in function declarations
void perror(const char *);

// to explicitly discard a value
(void) printf("I don't need the return value\n");

// pointers to void
void *malloc(size_t size);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Literals

A *literal* value is a value that is written directly, as opposed to one that is computed by an expression.

Integer literals

Integers can be written in:

- decimal (starts with a non-zero digit)
- hexadecimal (starts with `0x` or `0X`
- octal (starts with a `0`)

Constants usually default to `int`, but if the value is too big the compiler will choose a bigger size. You can also be explicit with suffixes:

Integer constant	Type
`0x200`	int
`512U`	unsigned int
`0L`	long
`0Xf0fUL`	unsigned long
`0777LL`	long long
`0xAAAllu`	unsigned long long

Float literals

A float literal consists of a sequence of decimal digits with a decimal point, optionally with an exponent:

Floating-point constant	Value
`10.0`	10
`2.32E5`	2.34 × 10^5
`67e-12`	67.0 × 10^-12

The default size of a constant is a `double`. You can append `f` or `F` to get a single-precision float instead.

Character literals

A character constant is written inside single-quote marks:

```
'a'   '0'   '*'
```

A few characters are written using a backslash escape sequence:

```
'\''  '\\'  '\n'  '\t'
```

A character constant has type `int`, and the value is the ASCII code of the character (there are some other cases that we will ignore here).

```
int c = getchar();
if (c != EOF && c >= '0' && c <= '9') {
    // the user entered a digit
}
```

You can also enter an explicit numeric code:

```
'\xA3'   // the character with value 163
```

Escape sequences

Escape sequence	Value	Action when printed
`\'`	single quotation mark (`'`)	
`\"`	double quotation mark (`"`)	
`\?`	question mark (`?`)	
`\\`	backslash character (`\`)	
`\a`	alert	beep or other signal
`\b`	backspace	move left one character
`\f`	form feed	move to next page/clear screen
`\n`	line feed (newline)	move to beginning of next line
`\r`	carriage return	move to beginning of current line
`\t`	horizontal tab	move to next horizontal tab stop
`\v`	vertical tab	move to next vertical tab stop
`\o`, `\oo`, or `\ooo` (octal digits)	character with given octal value	
`\xh[h...]` (hex digits)	character with given hex value	
`\uhhhh`, `\Uhhhhhhhh`	character with given universal char name	

String literals

A *string literal* consists of a sequence of characters (and/or escape sequences) enclosed in double quotation marks:

```
"Hello, world!\n"
```

A string literal can be used to initialize a character array:

```
char msg1[100] = "the array will have space for 100 characters";
char msg2[] = "this array will be just the right size";
```

It can also be used to initialize a character pointer:

```
char *msg3 = "this is a string constant and msg3 points to it";
```

Multiple string literals in the source will be contatenated at compile time into a single string:

```
printf("ID    | Name\n"
       "------|-------\n");

#define MY_EMAIL_ADDRESS "jdoe@example.com"
printf("Email me at " MY_EMAIL_ADDRESS " with suggestions\n");
```

Type conversions

An expression may involve values of different types:

```
double dVar = 2.5;
dVar *= 3;
if (dVar < 10L) { ... }
```

At each step, the compiler converts the two values into one compatible type before performing the operation. This only works for scalar types (integers, floats, booleans, pointers) not structs.

You can explicitly *type cast* a value:

```
int sum = 22, count = 5;
double mean = (double) sum / count;
```

Type casting has the same precedence level as most unary operators (which beats most binary operators) so `sum` is first converted to a `double` and then the division is performed. A few basic rules help understand what happens next:

- Operations between two integers will always yield an integer
- Operations between two floats will always yield a float
- Operations that mix an integer and a float will yield a float

Implicit type conversion

The hierarchy of types:

- Any two unsigned integer types have different conversion ranks. If one is wider than the other, then it has a higher rank.
- Each signed integer type has the same rank as the corresponding unsigned type. The type `char` has the same rank as `signed char` and `unsigned char`.
- The standard integer types are ranked in the order:

  ```
  _Bool < char < short < int < long < long long
  ```

- Every enumerated type has the same rank as its corresponding integer type
- The floating-point types are ranked in the following order:

  ```
  float < double < long double
  ```

- The lowest-ranked floating-point type, `float`, has a higher rank than any integer type.

Implicit type conversion

There are many details, but the general goal is to pick the type that can represent a wider range of values. A couple exceptions:

- Converting to a float can lose precision for large numbers
- Converting to an unsigned can lose negative numbers

A few other special cases. The common theme is that data must fit in its container:

- Assignments and initializations: the value of the right operand is always converted to the type of the left operand
- Function calls: arguments are converted to the types of the formal parameters
- Return statements: value is converted to the function's return type