Computer Architecture

Programming in C

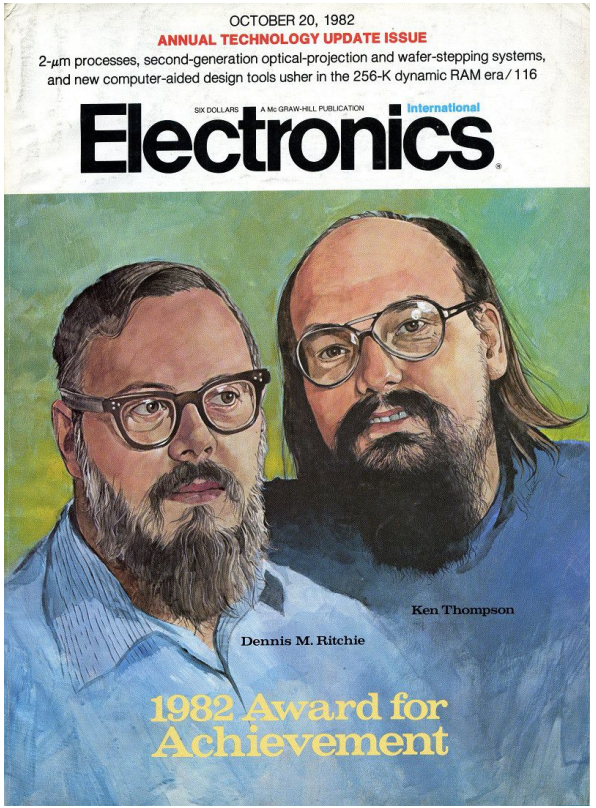
Basic types and control structures

Schedule

- The basics of C
  - types, variables, expressions
  - loops and conditional statements
  - arrays and strings
  - functions
- Advanced
  - pointers
  - memory allocation
  - structures
  - file operations
  - data structures (e.g., linked lists, binary trees)

History

- C was created by Dennis Ritchie and Ken Thompson at Bell labs in 1970s.
- It was developed as a programming language that could be used to write the UNIX operating system.
- It is an efficient, portable, and flexible programming language, and it a foundational language in computer science and software engineering.
- Still being widely used today.



Agenda

- Basic types
  - types
  - syntax
- Control structures
  - loops
  - conditionals

Basic components of the language

A C program is often a combination of the following components:

variables:	access to the memory
operators:	`+ - * < > …`
conditionals:	`if(...) {...} else {...}`

loops:	`for(...) {...}`
	`while(...) {...}`
	`do {...} while (...)`

Input / output:	`printf (...)` `scanf (...)`

My very first program

C my_first_program.c > ...

```c
#include <stdio.h>

// my very first program
int main(){

    printf("Hello World!\n");

    return 0;
}

```

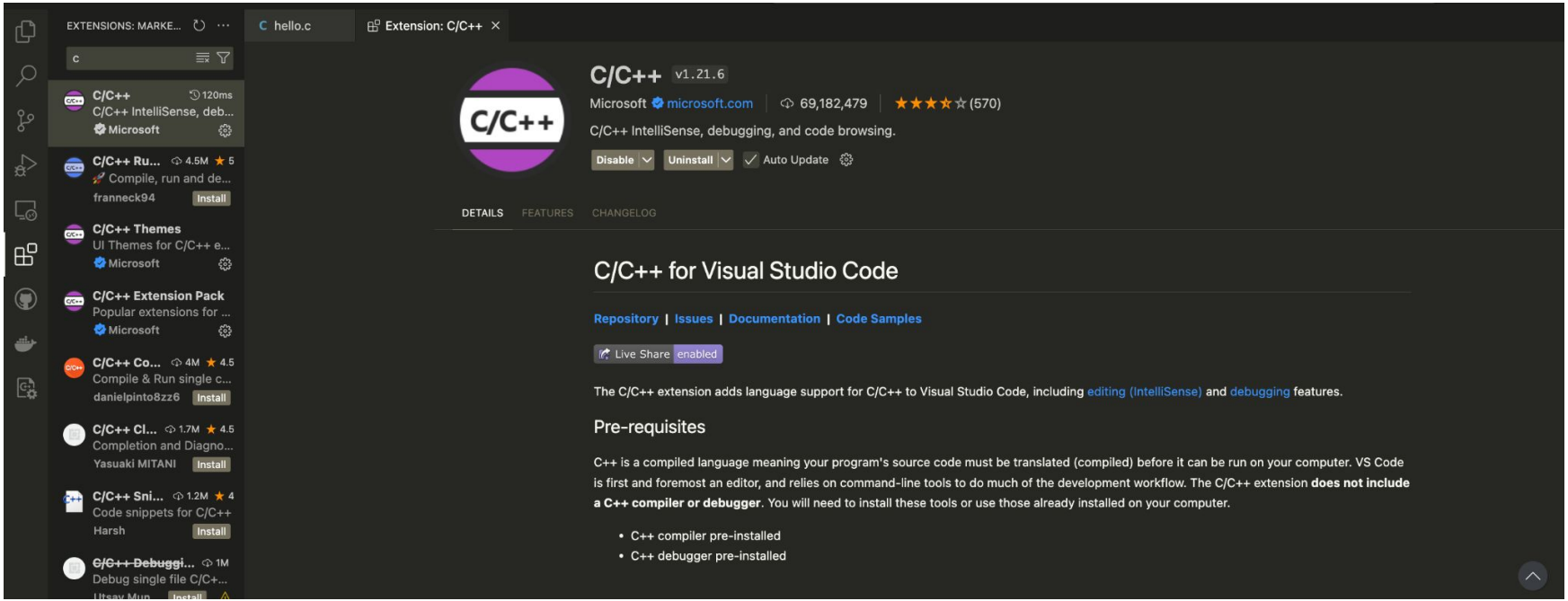
- We can use VS Code to write and run it.

Let's set up the environment first.

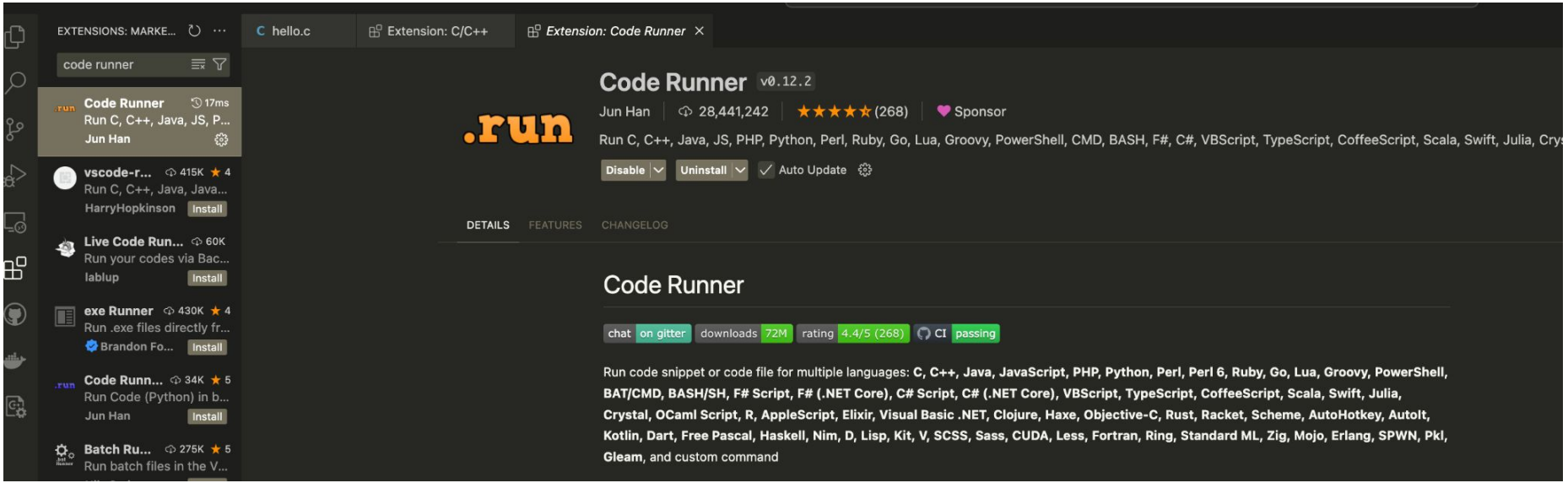
Set up your IDE

- We will use VS Code (https://code.visualstudio.com/)
- Two extensions are needed:
  - C/C++ by microsoft for debugging, code browsing, …
  - code runner for running the code with VS Code
- We also need to install a C compiler
  - macOS:
    - open a terminal, input `clang --version` to check if the compiler has been installed; if installed it will show you the version of it. Otherwise, there will be some errors, and you need to install clang.
    - To install clang, in the terminal, input `xcode-select --install`
  - Windows: you may follow this to install minGW-w64: https://code.visualstudio.com/docs/cpp/config-mingw#_prerequisites
    - For windows users, you may also use this IDE: CodeBlocks, https://www.codeblocks.org/; remember to install the mingw-setup

Extension


EXTENSIONS: MARKE...
c
C/C++ 120ms
C/C++ IntelliSense, deb...
Microsoft
C/C++ Ru... 4.5M ★ 5
Compile, run and de...
franneck94
Install
C/C++ Themes
UI Themes for C/C++ e...
Microsoft
C/C++ Extension Pack
Popular extensions for ...
Microsoft
C/C++ Co... 4M ★ 4.5
Compile & Run single c...
danielpinto8zz6
Install
C/C++ Cl... 1.7M ★ 4.5
Completion and Diagno...
Yasuaki MITANI
Install
C/C++ Sni... 1.2M ★ 4
Code snippets for C/C++
Harsh
Install
C/C++ Debuggi... 1M
Debug single file C/C+...
Install
hello.c
Extension: C/C++
C/C++
v1.21.6
Microsoft
microsoft.com
69,182,479
(570)
C/C++ IntelliSense, debugging, and code browsing.
Disable
Uninstall
Auto Update
DETAILS
FEATURES
CHANGELOG
C/C++ for Visual Studio Code
Repository | Issues | Documentation | Code Samples
Live Share enabled
The C/C++ extension adds language support for C/C++ to Visual Studio Code, including editing (IntelliSense) and debugging features.
Pre-requisites
C++ is a compiled language meaning your program's source code must be translated (compiled) before it can be run on your computer. VS Code is first and foremost an editor, and relies on command-line tools to do much of the development workflow. The C/C++ extension does not include a C++ compiler or debugger. You will need to install these tools or use those already installed on your computer.
C++ compiler pre-installed
C++ debugger pre-installed

Extension


EXTENSIONS: MARKE...
code runner
Code Runner 17ms
Run C, C++, Java, JS, P...
Jun Han
vscode-r... 415K ★ 4
Run C, C++, Java, Java...
HarryHopkinson Install
Live Code Run... 60K
Run your codes via Bac...
lablup Install
exe Runner 430K ★ 4
Run .exe files directly fr...
Brandon Fo... Install
Code Runn... 34K ★ 5
Run Code (Python) in b...
Jun Han Install
Batch Ru... 275K ★ 5
Run batch files in the V...
hello.c
Extension: C/C++
Extension: Code Runner
Code Runner v0.12.2
.run
Jun Han 28,441,242 ★★★★★ (268) Sponsor
Run C, C++, Java, JS, PHP, Python, Perl, Ruby, Go, Lua, Groovy, PowerShell, CMD, BASH, F#, C#, VBScript, TypeScript, CoffeeScript, Scala, Swift, Julia, Crys
Disable Uninstall Auto Update
DETAILS FEATURES CHANGELOG
Code Runner
chat on gitter downloads 72M rating 4.4/5 (268) CI passing
Run code snippet or code file for multiple languages: C, C++, Java, JavaScript, PHP, Python, Perl, Perl 6, Ruby, Go, Lua, Groovy, PowerShell, BAT/CMD, BASH/SH, F# Script, F# (.NET Core), C# Script, C# (.NET Core), VBScript, TypeScript, CoffeeScript, Scala, Swift, Julia, Crystal, OCaml Script, R, AppleScript, Elixir, Visual Basic .NET, Clojure, Haxe, Objective-C, Rust, Racket, Scheme, AutoHotkey, AutoIt, Kotlin, Dart, Free Pascal, Haskell, Nim, D, Lisp, Kit, V, SCSS, Sass, CUDA, Less, Fortran, Ring, Standard ML, Zig, Mojo, Erlang, SPWN, Pkl, Gleam, and custom command

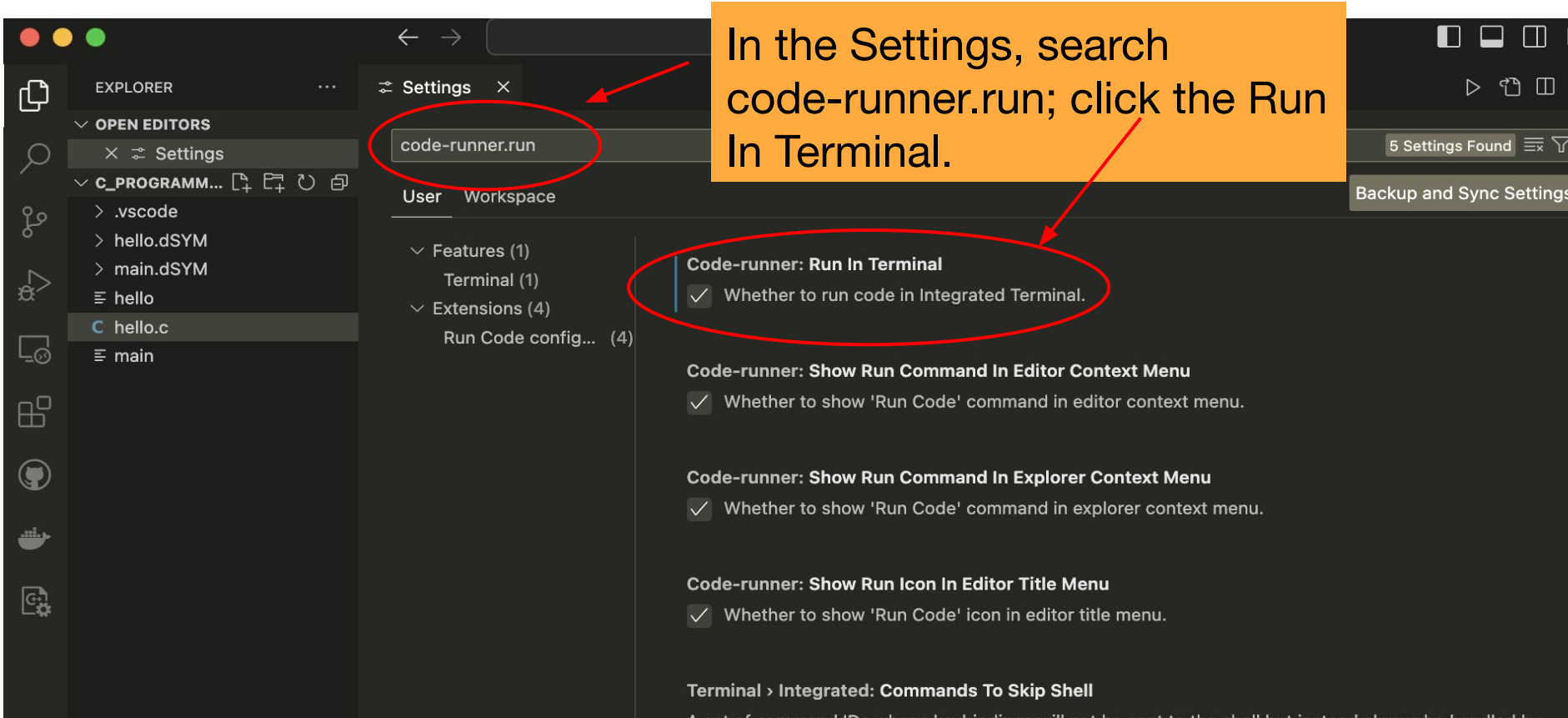
Install clang

bing — -zsh — 80×24

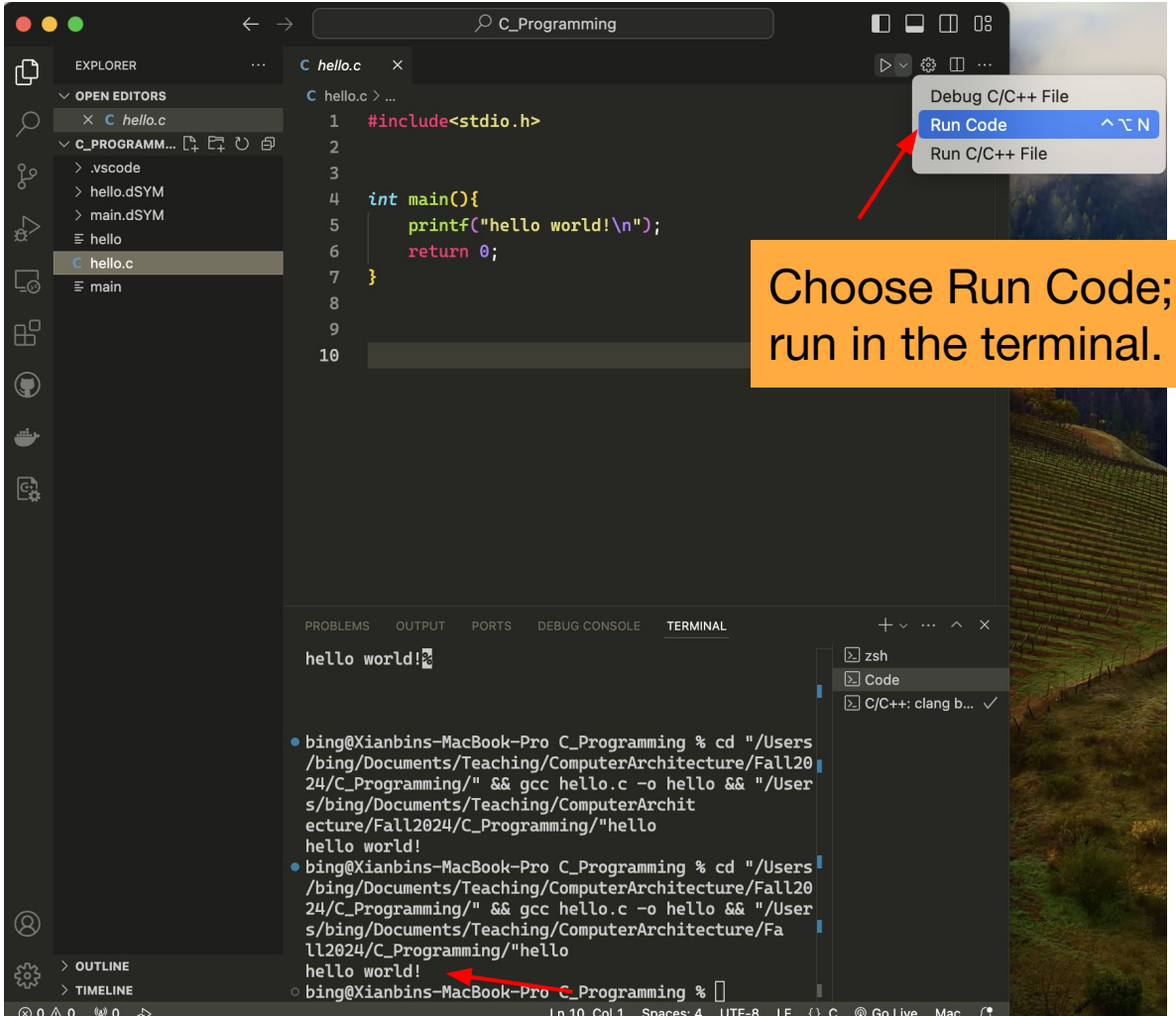
```
Last login: Fri Aug 23 16:01:16 on ttys003
bing@Xianbins-MacBook-Pro ~ % clang --version
Apple clang version 15.0.0 (clang-1500.3.9.4)
Target: arm64-apple-darwin23.6.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault
.xctoolchain/usr/bin
bing@Xianbins-MacBook-Pro ~ % clang --version
Apple clang version 15.0.0 (clang-1500.3.9.4)
Target: arm64-apple-darwin23.6.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault
.xctoolchain/usr/bin
bing@Xianbins-MacBook-Pro ~ % clang --version
Apple clang version 15.0.0 (clang-1500.3.9.4)
Target: arm64-apple-darwin23.6.0
Thread model: posix
InstalledDir: /Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault
.xctoolchain/usr/bin
bing@Xianbins-MacBook-Pro ~ % xcode-select --install
```

If no version information shown, please install clang

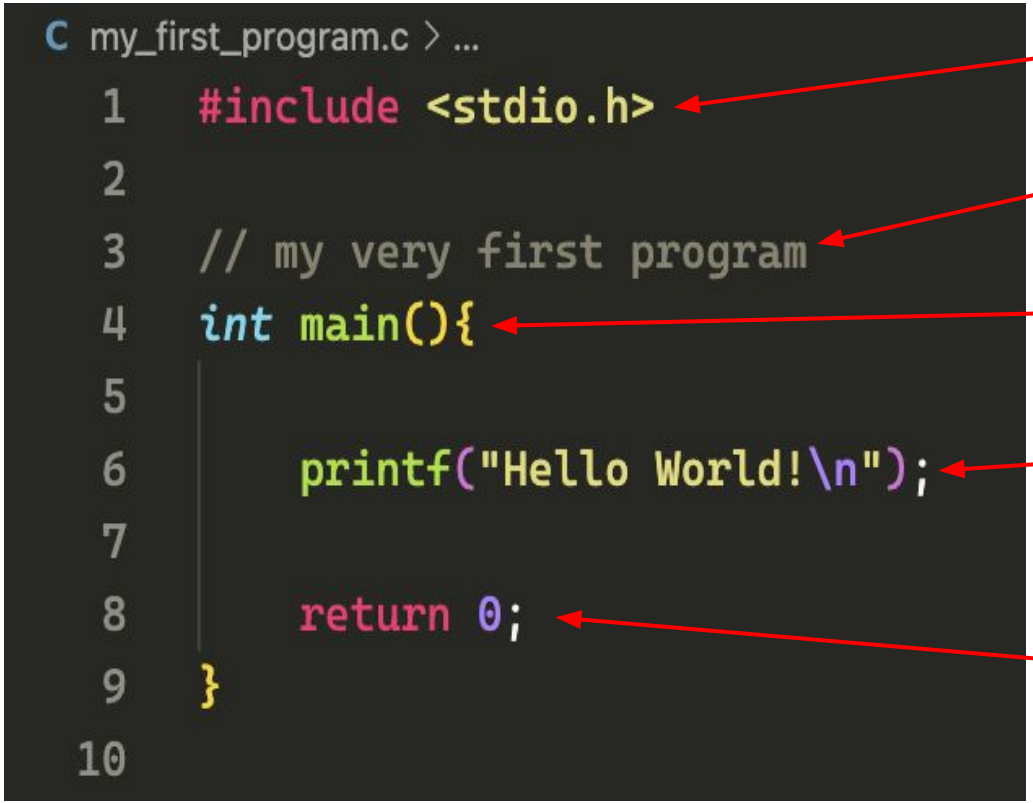
Run the code in terminal



Finally


C_Programming
EXPLORER
OPEN EDITORS
hello.c
C_PROGRAMM...
.vscode
hello.dSYM
main.dSYM
hello
hello.c
main
OUTLINE
TIMELINE
hello.c
hello.c > ...
#include<stdio.h>
int main(){
printf("hello world!\n");
return 0;
}
Debug C/C++ File
Run Code ^⌥N
Run C/C++ File
PROBLEMS
OUTPUT
PORTS
DEBUG CONSOLE
TERMINAL
hello world!
bing@Xianbins-MacBook-Pro C_Programming % cd "/Users/bing/Documents/Teaching/ComputerArchitecture/Fall2024/C_Programming/" && gcc hello.c -o hello && "/Users/bing/Documents/Teaching/ComputerArchitecture/Fall2024/C_Programming/"hello
hello world!
bing@Xianbins-MacBook-Pro C_Programming % cd "/Users/bing/Documents/Teaching/ComputerArchitecture/Fall2024/C_Programming/" && gcc hello.c -o hello && "/Users/bing/Documents/Teaching/ComputerArchitecture/Fall2024/C_Programming/"hello
hello world!
bing@Xianbins-MacBook-Pro C_Programming %
zsh
Code
C/C++: clang b...
Choose Run Code; it will run in the terminal.

My very first program



import files

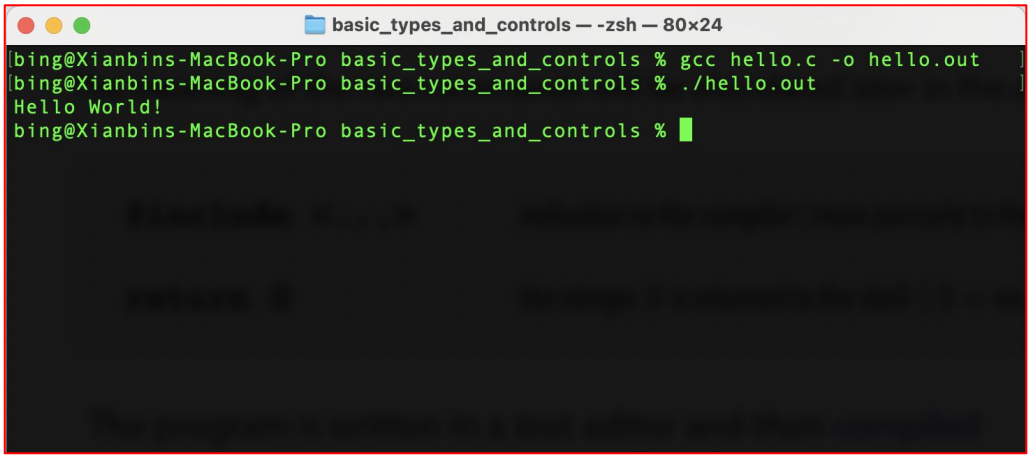
commentary (double slash + text)

defines a function

prints a chain of characters

the integer 0 is returned to the shell (if no errors)

To run the program in a shell



- `gcc hello.c` ⇒ compile hello.c with gcc
- `gcc hello.c -o hello.out` ⇒ compile hello.c with gcc and name the compiled file as hello.out
- `./hello.out` ⇒ run hello.out

Briefly, it needs to steps to run a C program in your machine.

1. compiling: translate the code into machine code
2. run the machine code

General structure of a C program

A program is constructed as a sequence of functions

- In C, the body of the function should beginning and end of the block `{ … }`
- Each instruction ends with a semicolon `;`
- The principal function `main` is launched at the start of the program.
  - a C program should have a main function.

```
name of function (...) {
                ⇣
    sequence of instructions
                ⇣
}
```

Example:

```
4   int main(){
5       printf("hello world!\n");
6       return 0;
7   }
```

Declaring variables, computing and printing

```c
#include<stdio.h>
int main() {
    int x;      //new variable of integer type
    int y;      //new variable of integer type
    int z;      //new variable of integer type

    x = 10;     //store 10 in the variable x
    y = 20;     //store 20 in the variable y
    z = x+y;    //store in z the sum of x and y

    // print the sum of x, y, and z
    printf("the value computed is equal to %d", x+y+z);

    return 0;
}
```

To declare a variable, we should specify the type.

```
type name_of_variable;
```

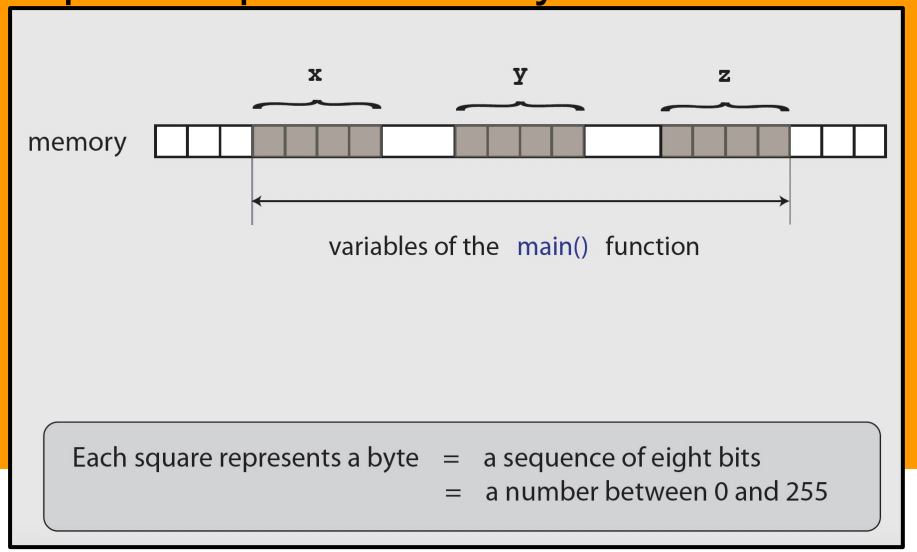
Declaring variables, computing and printing

- `int x;`
  - reverses (or allocates) some piece of the memory sufficiently large to store an integer (int).
  - calls this location `x`, i.e., the variable x will be then used to refer to this specific memory location.
- `int` is the type specifies the nature of the values stored at location `x`
  - many other types:
    - characters
    - floating point
    - array of integers, etc…
- To declare a variable in C, we have to specify the type

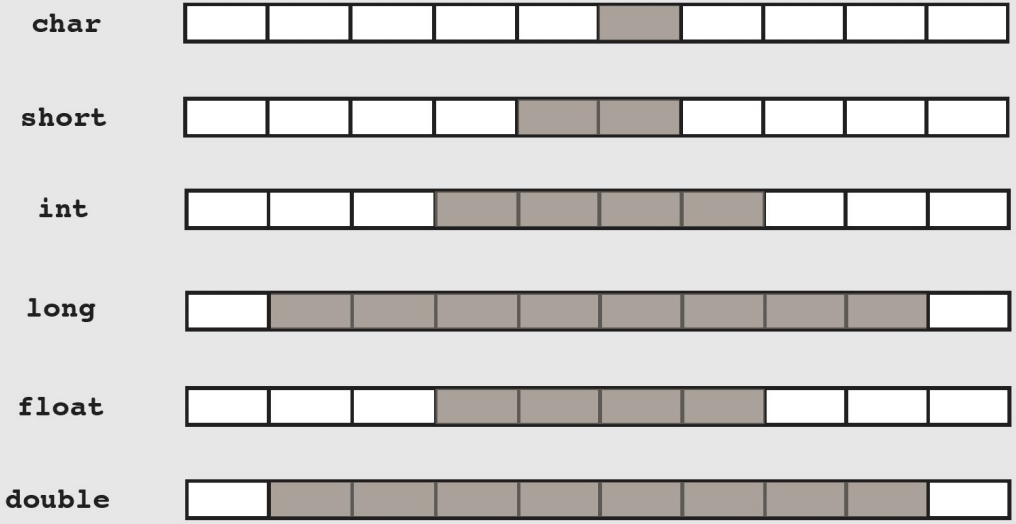
Variables allocated in memory

In a computer, everything is a binary storing in memory.

- The atom of memory is a **bit**.
- Usually, the smallest unit of memory is **8 bits**, also called a **byte**.
- In the following figure, each square represents a byte.



Each data type corresponds to a different amount of memory



Each block here represents a byte = a sequence of eight bits = a number between 0 and 255

Different data type needs different amount of memory.

Test yourself the size of your data types!

```c
#include <stdio.h>

int main(){

    printf("size of char = %zu\n", sizeof(char));
    printf("size of short = %zu\n", sizeof(short));
    printf("size of int = %zu\n", sizeof(int));
    printf("size of unsigned int = %zu\n", sizeof(unsigned int));
    printf("size of long = %zu\n", sizeof(long));
    printf("size of unsigned long = %zu\n", sizeof(unsigned long));
    printf("size of float = %zu\n", sizeof(float));
    printf("size of double = %zu\n", sizeof(double));
    return 0;

}
```

Now, create a file named size.c, and input the code.

Test yourself the size of your data types!

If everything goes well, you will have the following printed in the terminal.

```
size of char = 1
size of short = 2
size of int = 4
size of unsigned int = 4
size of long = 8
size of unsigned long = 8
size of float = 4
size of double = 8
```

Note:

- the unit of the size is byte (i.e., 8 bits)

Q: How many bits does a float type variable take?

printf format identifiers

`%d %i`	Decimal signed integer
`%o`	Octal integer
`%x %X`	Hex integer
`%u`	Unsigned integer
`%ld`	Long decimal signed integer
`%lu`	Long unsigned integer
`%c`	Character
`%s`	String
`%f`	Double
`%p`	Pointer
`%zu`	Size of a type

In the `printf`, we should specify the type of the variable we want to print.

- Identifiers are the symbols for the types.

`printf("x is %d", x)` //print the value of x in the format of the decimal signed integer

Example: printf format identifiers

Try different identifiers for `sizeof(char)`, what do you get?

```
3   int main(){
4
5       printf("size of char = %zu\n", sizeof(char));
6       printf("size of char = %d\n", sizeof(char));
7       printf("size of char = %f\n", sizeof(char));
8       printf("size of char = %c\n", sizeof(char));
9       return 0;
10  }
```

Basic syntactic elements

- Variable declaration
  - In C, the variable should be declared before being used.
  - To declare a variable, it **always** needs to indicate its type.
- It is possible to declare several variables of the same type:

```
int x, y, z;
```

- In practice, the variable declarations of a function are done at the very beginning, followed by the instructions.

Basic syntactic elements

- Variable declaration and immediate initialization:

`int x = 10; //`declare a variable x of int and initialize it with 10.

- We can mix declarations and initializations:

`int x=10; y=20; z=x+y;//`the declarations and initializations are performed sequentially. So, we will have `x=10, y=20,` and `z=30`.

Basic syntactic elements

Note: variables that have not been initialized may contain any value!

```c
#include <stdio.h>

int main(){
    int x;
    printf("the value in x is %d\n", x);
    return 0;
}
```

Print an uninitialized variable , what will you get?

Basic syntactic elements

The assignment operator: = (it is like a copy, but it is not retroactive)

```
int x;   //x undefined
int y;   //y undefined
x = 10; //x is equal to 10
y = x;   //value of x read and stored in  y, so, x and y have the same value 10
x=20;    //x has value 20, but y has still value 10
```

Increment a variable: two ways

Option 1:

```
int x;  //x is undefined
x = 1; //x is equal to 1
x = x+1; //x is incremented to 2
```

Option 2:

```
int x;  //x is undefined
x = 1; //x is equal to 1
x++; //x is incremented to 2
```

Can you make a guess on the ways to decrement a variable?

Computing and printing

```
printf("the value computed is equal to %d", x+y+z);
```

- Computes the sum of the current values of `x, y,` and `z`
- Prints the chain in quotation marks where `%d` is replaced by the result. (Please try it, the expected output is 60)
- `printf` also can print several integers:

```
printf("the sum of %d and % is equal to %d",x,y,x+y);
```

- It will print `the sum of 10 and 20 is equal to 30`

Reading on the keyboard

- `scanf`: built-in function for getting input from the keyboard

```c
#include <stdio.h>
int main(){
    int n;
    int m;

    printf("enter a first number: \n");
    scanf("%d", &n); //reading n on the keyboard

    printf("enter a second number: \n");
    scanf("%d", &m);//reading m on the keyboard

    printf("their sum is equal to %d", n+m);
    return 0;
}
```

Note: `&n` and `&m` are the memory addresses of `n` and `m`.

Reading on the keyboard

If you need to input two values for two variables, you may use the following way:

```c
#include<stdio.h>

int main() {
    char c, d;
    printf("Please input two chars: \n");
    scanf("%c %c", &c, &d);
    printf("The chars you input are %c and %c\n", c, d);
}
```

A space is needed.

The most usual types of C: double

The type `double` for floating point numbers:

```
double x = 3.14;
```

Note: the computations on floating point numbers are necessarily imprecise.

```
printf("enter a number: ");

scanf("%lf", &x);

printf("The input value is %lf", x);
```

- In order to print and read these numbers, one uses `%lf`

The most usual types of C: double

- It is possible to store an `int` in a variable of type `double` but **not** the inverse without rounding up, or even an undefined result:

```
int n=1;

double pi=1;      //pi is equal to 1.0

pi = pi + 2.14;   //pi becomes 3.14

n = pi;           //compiles but n becomes 3
```

Note: the value of `n` would be undefined if the `pi` were larger.

The most usual types of C: char

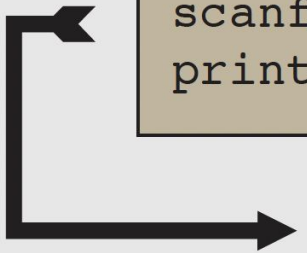
- The type `char` is the type used to represent characters.
- The characters are represented in memory by **numbers** but the variables of type char can also be written as ‘a’, ‘b’, …
  - The number associated to a character is its **ASCII code** (ASCII table).
  - It is possible to do arithmetic calculations on these characters

```
char c = ‘a’;    //in memory: 97
int n = c;       //n becomes equal to 97
c = c+1          //c becomes 98, that is ‘b’
```

The most usual types of C: char

- For reading and printing values of char as characters, one uses `%c`.
- Using `%d` for printing characters as numbers.

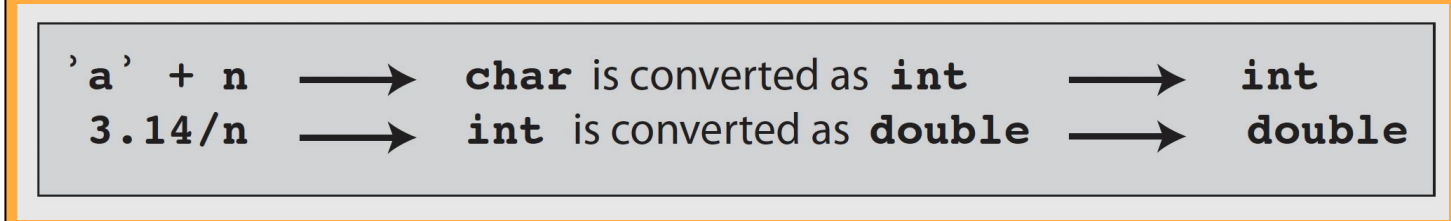
```
char c;
scanf(" %c", &c);  //   input
printf("the integer associated to %c is %d", c, c);
```



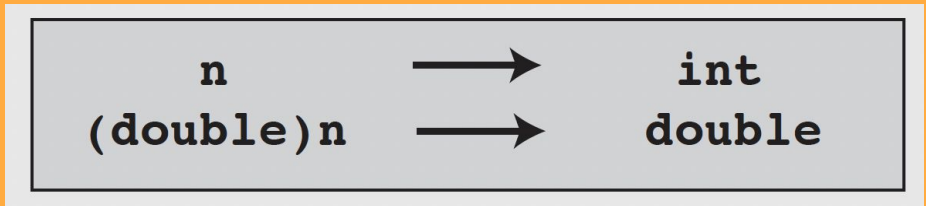
```
the integer associated to a is 96
```

Typing of expressions and rules of conversion

- In simple words, small size types are coerced into larger size types without loss of information whenever it is necessary.



- We can force the conversion in some cases:



Control structures

What we are going to see:

- The structure `if-else`
  - The conditional evaluation
- The loops `for, while, do-while`
- The structure `switch`
- The control instructions `break` and `continue`

The if-else control structure

```
3   int main() {
4       int n, m, max;
5
6       printf("enter two numbers: \n");
7       scanf("%d%d", &n, &m);
8
9       if (n>m) {
10          max=n;
11      }
12      else{
13          max=m;
14      }
15      printf("the larger one is %d\n", max);
16      return 0;
17  }
```

if-else general form:

```
if (boolean condition) {
        …
     Body of the if
        …
}
else {
        …
    Body of the else
        …
}
```

Note: a block reduced to a **single** instruction may be written without any `{...}`

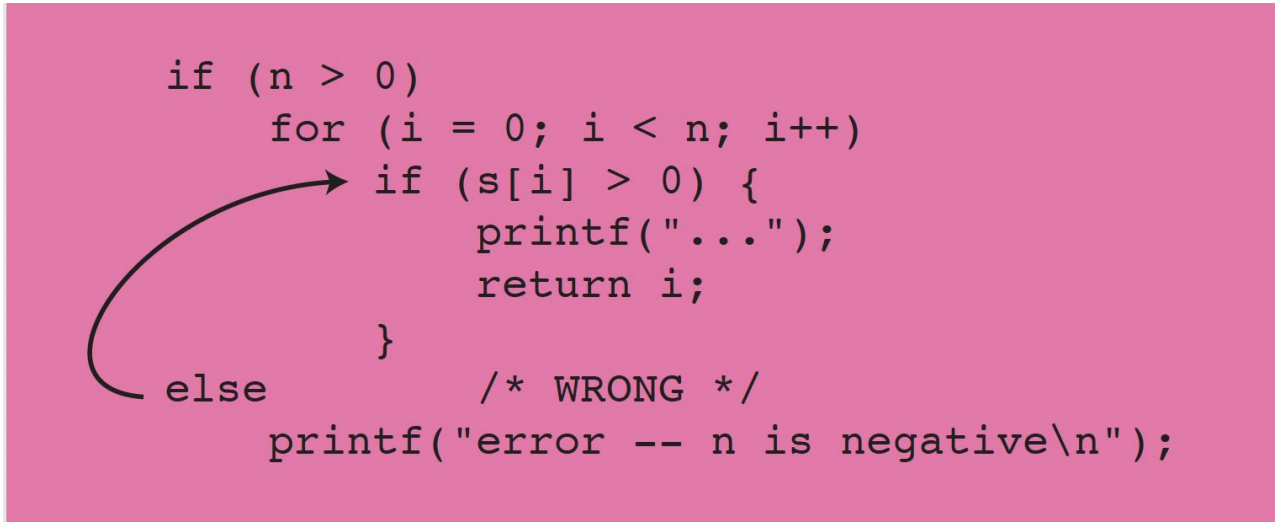
More details of if-else

- The contents of two bodies of the if-else are arbitrary and can be freely interchanged.
- One can write an `if` without any `else` (but not the contrary!)
- An `else` is always associated to the last `if` of the same depth and not yet associated to any previous else.

Question:
what is the `else` associated with the first `if`?

```
if (expression)
    statement
else if (expression)
    statement
else if (expression)
    statement
else
    statement
```

(Python users:) Be careful when nesting if



```
if (n > 0)
    for (i = 0; i < n; i++)
        if (s[i] > 0) {
            printf("...");
            return i;
        }
else            /* WRONG */
    printf("error -- n is negative\n");
```

- The intention is to print an error message when `n` is negative, but the `else` is associated to a improper `if`.
- This kind of bugs is very difficult to detect…
- It is thus a good practice to put braces `{...}` when there are nested `if`s!!!

The syntax of conditions

- Comparison operators
  - The usual form: comparison of two expressions

    expression `op` expression

  - The op is one of the comparison operators as below:
    - `<, <=, >=, >, ==, !=`
    - The comparison operators are valid for all numerical types

The syntax of conditions

- It is also **possible** to compare expressions with different types:

```
char c; int n; double d;
// ... initialization of the variables ...

if (c <= n*2) {...}  // c promoted into int
if (n+1 <= d) {...}  // n+1 promoted into double
if (n != c+d) {...}  // n promoted into double
                     // c promoted into double
```

Combining comparison combinators

- Two conditions can be combined into a single one using:
  - `&&` the logical conjunction (i.e., and), true when it two components are true
  - `||` the logical disjunction (i.e., or), true when one of its components is true

```
if (n <= 1 || x > 3) {...}
```

- The inverse of a condition can be constructed using:
  - `!` the logical negation, true when its unique component is false

```
if (!(x==0)) {...}
```
is the same as
```
if (x!=0) {...}
```

No boolean type in C

- There is **no** boolean type in C.
- The type `int` is thus used to represent the truth values…
- The conditions are values of type `int` like any other integer value.

So, a comparison evaluates as:

- 1 when it is satisfied
- 0 otherwise

Illustration :

1 > 2 evaluates to **0**
2 > 0 evaluates to **1**

For the expression `v = n > 2;` to ask that `v` receives the value `0` or `1` depending on the value of `n`.

The evaluation of conditions

From the point of view of an if (of a for, of a while, etc…)

- all the expressions of non zero value are considered true
- all the expressions of zero value are considered false

```
if (x + y){...}     is the same as  if (x + y != 0) {...}
if (!(x + y)){...}  is the same as  if (x + y == 0) {...}
```

Probably useless but does compile properly:

- if (42) … the condition is always satisfied
- if (0) … the condition is never satisfied

Beware: a common mistake!

A typical bug in C is the following one: (may not only be in C)

```
if (n = 0){ /* instead of n == 0 */
    ...
    ...
}
```

Be warned of its dangers:

- This code will compile well and will produce an executable code
- It will force the value of n to be zero
- It will never branch in the block if whatever the original value of n tested by the condition.

Assignment instructions have values!

- `n = 0`
  - an instruction which give value 0 to n
  - used here as an expression
  - the value of the expression is the value received by n
- Every assignment instruction can be used in that way:
  - `x = y = 42` can be read as `x = (y = 42)`
    - the variable y receives the value 42
    - this determines the value 42 of the expression `(y=42)`
    - the variable x receives the value 42

Assignment instructions have values!!!

```
if (n = 0) { ... }
```

- the variable n receives the value 0
- the expression `(n=0)` has thus value 0
- as such, it is considered false
- the conditional thus branches on the `else` block

```
if (n = 42) { ... }
```

- the variable n receives the value 42
- the expression `(n=42)` has thus value 42
- as such, it is considered true (because not zero)
- the conditional thus branches on the `if` block

Assignment instructions have values!!!

Q: What is the point of treating assignments as expressions?

A: This enables one to write more concise and clearer code in many situations of interest, especially in file operations, like the following one:

```
int c;

if ((c=getchar()) != EOF){
/* case when the character c has been read */
/* from the standard input */
}
else {
/* case when the program has reached the end */
/* of the standard input */
}
```

Note:

- EOF is a signal End-Of-File emitted by the system when the end of the standard input has been reached.
- `getchar()` is a built-in function that reads the next character of the input.
- Press Ctrl + d to input an EOF.

Assignment instructions have values!!!

- Difference between `x = x+1` and `x++`:

```
4   int main(){
5       int x=0;
6
7       printf("x = %d\n",  x);
8       printf("increment x by x+=1: %d\n",  x += 1);
9       printfprintf("increment x by x++: %d\n",  x ++);
10
11      //x += 1 (or x=x+1) directly updates x and returns the new value.
12      //x++ updates x but returns the original value of x before the increment.
13      return 0;
14  }
```

Conditional evaluation

```
int n, m;

printf("enter two numbers :\n");
scanf("%d%d",&n,&m);

printf("the largest one is %d", (n > m) ? n : m);
```

Ternary operator: a variant of the construction `if-else`: select an expression among two possibilities depending on the boolean condition.

The for loop example

```
// computing the first ten square numbers

int i,square;

for (i = 1; i <= 10 ; i=i+1){
    square = i * i;
    printf("The square of %d is equal to %d\n",i,square);
}
```

for loop: general form

```
for (i = 1; i <= 10 ; i=i+1){ body of the loop };
```

where the variable **i** is called the counter of the loop

- `i = 1` ⇒ initialization instruction; the initial value of the counter is 1.
- `i <= 10` ⇒ boolean condition; the loop is performed as long as this condition is true.
- `i = i + 1`⇒ incrementation instruction; the counter is incremented each time the body of the loop is executed.

for loop: general form

The method of using a counter in a loop is well-know, general and safe.

```
for (i = 1; i <= 10 ; i = i + 1){ body of the loop };
```

```
for (i = 1; i < 10 ; i = i + 1){ body of the loop };
```

```
for (i = 1; i < 12 ; i = i + 2){ body of the loop };
```

```
for (i = 10; i >= 0 ; i = i - 1){ body of the loop };
```

The for loop

The interval of the counter is not necessary known before execution:

```
// computing the sum of the n first numbers

int i,n,sum;

printf("enter a positive number : ");
scanf("%d", &n);

sum = 0;
for (i = 1 ; i <= n ; i = i + 1){
    sum = sum + i;
}

printf("The sum of the %d first numbers is %d\n",n,sum);
```

Question:
What happens when the integer `n` is negative?

The for loop

Advice: avoid to alter the counter inside the body of the for loop.

- The following program is absurd, but it does compile!

```
for (i = 1 ; i < 2 ; i = i + 1){
      i = i - 1;
}
```

Note: You can press ctrl-c to stop it.

The for loop

- The following program is even more absurd, but it does compile!

```
for ( ; 1 ; );
```

Note: You can press ctrl-c to stop it.

- The initialization instruction is empty
- The condition test is always successful
- The incrementation instruction is empty
- Only the ctrl-c from the shell can stop it.

The while loop example

Computation of the first power of 2 greater than 10000

```
int power,n;

power = 1;

while (power < 10000){
     power = power * 2;
     n = n + 1;
}

printf("Two to the power %d = %d is the first
power of two greater than 10000\n", n, power);
```

The number of loops is not known at the beginning.

Note: there is a bug in the code, can you find it?

while loop: general form

```
while (boolean condition){ body of the while };
```

- The condition is evaluated. If it is true (i.e., non zero) then,
  - The body of the while is executed
  - One goes back to the condition evaluation
  - if the condition is false, the while loop stops, and one carries on.
- It is possible that the body of the `while` is never executed when the condition is immediately false (equal to zero).
- Nothing guarantees that the `while` loop will stop -- as such, it a bit more risky than a for loop.

Exercise

- It is easy to encode a `for` loop using a `while` loop. How can you do this?

```
5       int power, n;
6       power = 1;
7       n = 0;
8       while (power<10000){
9           power = power*2;
10          n = n+1;
11      }
12      printf("2^%d = %d is the first power of two \
13      greater than 10000\n", n, power);
14      return 0;
```

Can you use a for loop for that?

- It is even easier to implement an infinite `while` loop. How can you do it?

The do-while loop

```
int n;

do {
    printf("Enter a positive number : ");
    scanf("%d",&n);
    if (n < 0) {printf("Sorry, I have said positive...\n")
} while (n < 0);
```

do { body of while } while (boolean condition);

- The body of the while is executed at least once. Then, the condition is evaluated.
  - If it is true then one goes back to execute the body
  - Otherwise, one carries on and goes to the next instruction.

The switch structure

```c
int main(){
    int n, m; char choice;
    printf("Enter two numbers : ");
    scanf("%d %d", &n, &m);
    printf("What do you want to do with them ?\n");
    printf("Add them (+) ?\n");
    printf("Multiply them (*) ?\n");
    scanf(" %c", &choice);

    switch (choice) {
        case '+': printf("Their sum is equal to : %d\n", n+m);
                  break;
        case '*': printf("Their product is equal to : %d\n", n*m);
                  break;
        default : printf("Unknown operation\n");
                  break;
    }
    return 0;
}
```

The switch structure

```
switch (expression) {
    case constant1 : sequence of instructions;
                     break;
    case constant2 : sequence of instructions;
                     break;
    ...
    default : sequence of instructions by default;
              break;
```

1. the expression is evaluated (i.e., computed).
2. The first sequence of the instructions whose constant is equal to the value just computed is executed.
3. the break instruction at the end of the sequence causes an immediate exit from the switch.
4. The sequence of instructions at default is executed when the computed value is different from all the constants.

The switch structure

- The **break** instruction is **not** necessary **at the end** of the instruction. ⇒ In that case, the next instruction of the switch is performed.
- Similarly, it is **not** necessary to end the switch with a default case.

```
int n=2;

  switch (n) {
  case 0: printf("is switched ON at 0\n");
  case 1: printf("is switched ON at 1\n");
  case 2: printf("is switched ON at 2\n");
  case 3: printf("is switched ON at 3\n");
  case 4: printf("is switched ON at 4\n");
  }
printf("end of the switch");
```

```
is switched ON at 2
is switched ON at 3
is switched ON at 4
end of the switch
```

Try it:

```
#include <stdio.h>

int main(){  /* count digits, white spaces, others */

  int c, i, nwhite, nother, ndigit[10];

  nwhite = nother = 0;
  for (i=0; i < 10; i=i+1)
    {ndigit[i] = 0;}  /* initialization of the array of numbers */
  while ((c=getchar()) != EOF)
    {switch (c) {
      case '0': case '1': case '2': case '3': case '4':
      case '5': case '6': case '7': case '8': case '9':
        ndigit[c-'0'] = ndigit[c-'0']+1;
        break;
      case ' ': case '\n': case '\t':
        nwhite=nwhite+1;
        break;
      default:
        nother=nother+1;
        break;
      }
    }
  printf("digits =");
  for (i = 0; i<10; i=i+1)
    {printf(" %d", ndigit[i]);}
  printf(", white space = %d, other = %d\n", nwhite, nother);
  return 0;
}
```

Note:

- when you finish your input, press the Enter, and then, press **ctrl-d** and press Enter.
- ctrl-d is the EOF.

The break instruction

More generally, the break instruction causes immediate exit from:

- The body of a switch
- The body of a for loop
- The body of a while loop
- The body of a do-while loop

```
for ( ... ){
      ...
      break;  // exit from the body
      ...
}
//  the execution carries on here
```

It is not possible to exit with one single break instruction from several nested bodies: one needs a break instruction for each level of nesting.

The continue instruction

The `continue` instruction enables one

- in the body of a `for` loop: to jump directly to the increment instruction
- in the body of a `while` loop or of a `do-while` loop: to jump directly to the evaluation of the condition

In other words, the `continue` instruction jumps immediately to the end of the body of the loop.

Very often, the `continue` instruction can be simulated by an `if`.

Exercise

Write a program that will calculate the result for the first **N**-th terms of the following series. [In that series sum, dot sign (.) means multiplication]

$1^2 * 2 + 2^2 * 3 + 3^2 * 4 + 4^2 * 5 + ...$

Input	Output
2	14
3	50
4	130
7	924

Exercise

Write a program that will print Fibonacci series up to N-th terms.

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89,…….

Input	Output
1	1
2	1 1
4	1 1 2 3
7	1 1 2 3 5 8 13

Acknowledgements

The slides of the lectures of C programming are largely based on very nice and cleverly crafted lecture materials from Prof. Paul Mellies.

Paul is a visiting professor at NYU Shanghai. He teaches functional programming and computer architecture in spring semesters.