C programming language

College of Saint Benedict & Saint John's University

origins

Dennis Ritchie in 2011 / CC BY 2.0

Brian Kernighan in 2012 / CC BY 2.0

- Dennis Ritchie and Brian Kernighan creators of C circa 1972
- TODO: more thorough history

hello, world

```c
/* file: helloworld.c */

#include <stdio.h>

int main() {
  printf("hello, world\n");
  return 0;
}
```

```
$ gcc -o helloworld helloworld.c
$ ./helloworld
hello, world
```

- Remind students that not everyone has same background in C — those with experience can help those without

- The tradition of using the phrase ”Hello, world!” as a test message was influenced by an example program in the seminal book *The C Programming Language*
- Every statement in C exists in a function, starting with `main`
- Refresh students memory on the *traditional* compilation process

global variables

```c
// file: figure2-4.c
// Stan Warford
// A nonsense program to illustrate global variables

#include <stdio.h>

char ch;
int j;

int main() {
  scanf("%c %d", &ch, &j);
  j += 5;
  ch++;
  printf("%c\n%d\n", ch, j);
  return 0;
}
```

```
$ gcc -o figure2-4 figure2-4.c
$ ./figure2-4
M 419
N
424
```

- Every C variable has three attributes:

  name an identifier determined arbitrarily by the programmer

  type specifies the kind of values it can have

  value

- In C, variable declaration only reserves storage for the value; nothing can be assumed about the initial value

- What would you expect for input '`Z -3`'?
- What would you expect for input '`9 a`'?
- What would you expect for input '`~ 2147483643`'?

program breakdown

```c
#include <stdio.h>

char ch;
int j;

int main() {
  scanf("%c %d", &ch, &j);
  j += 5;
  ch++;
  printf("%c\n%d\n", ch, j);
  return 0;
}
```

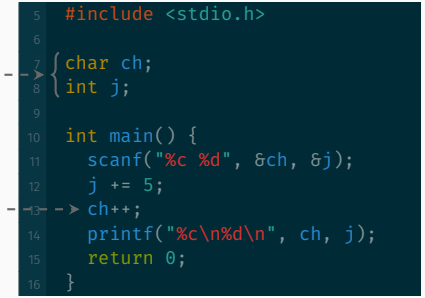
C programs ALWAYS start execution with the `main` function

returning from `main` ends the program

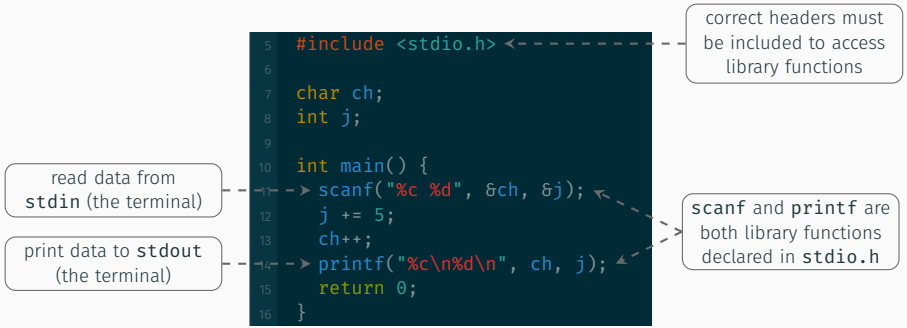
program breakdown

global variables are declared here — outside of any function

characters in C are treated internally like signed integers



program breakdown



- C has no “built-in” functions; however, it does have a standard library that includes many useful utility functions.

program breakdown

```c
#include <stdio.h>

char ch;
int j;

int main() {
  scanf("%c %d", &ch, &j);
  j += 5;
  ch++;
  printf("%c\n%d\n", ch, j);
  return 0;
}
```

`&` is the address of operator — `scanf` expects the address of the variables where the data will be stored

- address here means the location in memory

conditions

```
if (<cond>) {
  /* ... */
}
else (<cond>) {
  /* ... */
}
else {
  /* ... */
}
```

- C only expects that the expression yield an integer value
- 0 is false, non-zero is true

conditions

```
if (x) {
  /* ??? */
}
if (x-y) {
  /* ??? */
}
if (x=y) {
  /* ??? */
}
```

- under what conditions will each of the above be executed?

- x != 0
- x != y
- y != 0

switch

```
switch (<expr>) {
  case <const>:
  /* ... */

  case <const>: /* fall-through */
  /* ... */
  break;

  default:
  /* ... */
  break;
}
```

loops

```
for (<init>; <cond>; <incr>) {
  /* ... */
}

while (<cond>)
  /* ... */
}

do {
  /* ... */
} while (<cond>);
```

- `<init>` is for initializing variables once, before loop begins execution — normally the loop control variable
- Same rules apply for `<cond>`.
- `<incr>` is for assigning variables a value, happens once after the body of the loop has executed each iteration — again, normally the loop control variable

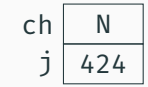
memory model — part i

global variables

declared outside of any function and remain in place throughout the execution of the entire program. they are stored at a fixed location in memory.

local variables

declared within a function and come into existence when the function is called and cease to exist when the function terminates. they are stored on the run-time stack.



(a) Fixed location.

(b) Run-time stack.

- I will be using graphical notation consistent with that of the book.
- In this case, (a) and (b) represent the state of relevant memory for the previous program just before it terminates, i.e., in the process of executing line 15.
- How would the previous program behave had it declared `ch` and `j` as local variables instead of global variables?
- What would the memory model look like given the above?

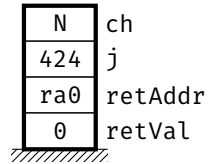


Figure 2: Run-time stack.

run-time stack a.k.a. “the stack”

run-time stack

stores information about the active functions of a C program, including:

- return value,
- parameters,
- return address, and
- local variables

in that order.

- each program has one stack

run-time stack a.k.a. “the stack”

run-time stack

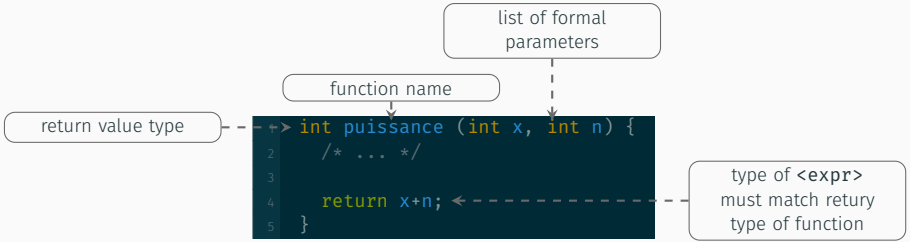
stores information about the active functions of a C program, including:

- return value,
- parameters,
- return address, and
- local variables

in that order.

- each program has one stack
- which visualization is correct?

functions


list of formal parameters
function name
return value type
1 int puissance (int x, int n) {
2 /* ... */
3
4 return x+n;
5 }
type of <expr> must match retury type of function

comparison

Java	C
object-oriented	procedural
interpreted	compiled
`String`	`char` array
condition (`boolean`)	condition (`int`)
garbage-collected	no memory management
references	pointers
exceptions	error codes

- in Java, everything is a method that is called on an object
- in C, everything is a function

- in Java, source code is compiled to byte code, which is then interpreted by Java VM
- in C, source code is compiled into binary machine code

- in Java, String is a class
- in C, a string is just an array of `char` values which ends with the `char '\0'`

- in Java, the Java VM takes care of deallocating memory used
- in C, any memory you allocate, you must also deallocate