C programming language

Jeremy Iverson

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

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

hello, world

```c
#include <stdio.h>

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

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

- 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*

conditions

- under what conditions will each of the following be execute?

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

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

add evens

- create program called `add_even.c` that adds all the even numbers between 1 and 100 and prints the sum

cli

```c
#include <stdio.h>

int main(int argc, char * argv[]) {
  printf("(%d) %s:%s\n", argc, argv[0],
    argv[1]);
  return 0;
}
```

- modify `add_even.c` to get maximum value from the command-line instead of hard-coded as 100

printf / scanf

- `printf()` interprets variables and prints character representations to standard out (usually the terminal)
- `scanf()` scans characters from standard in (usually the terminal) and interprets them for storage in variables

```c
#include <stdio.h>

int main() {
  int i;
  scanf("%d", &i);
  return 0;
}
```

- scanf requires you to pass the address of the variable, so that its value can be changed

echo

- modify `helloworld.c` to ask user for an input and then print it back
- change its name to `echo.c`

```
$ ./echo
Enter a string to echo: hello, world
hello,
```

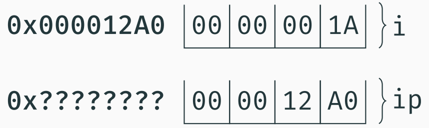
- why did it print `hello,` instead of `hello, world`?

pointers

- a pointer is a variable whose value is a memory address

```c
int    i   = 0x1A;
int  * ip  = &i;
```

- `&i` evaluates to the address where the variable `i` is stored in memory
- `i` is an `int`, so `ip` is a *pointer* to an `int`



pointers cont.

```c
printf("0x%X\n", i);    /* 0x1A */
printf("0x%#X\n", &i);  /* 0x12A0 */
printf("0x%#X\n", ip);  /* 0x12A0 */
printf("0x%#X\n", &ip); /* 0x???????? */
```

- so how can we use the pointer, `ip`, to access the value of `i`?

pointer dereference

- `*ptr` will
  1. treat the value of `ptr` as a memory address
  2. get the bytes of data located at that memory address
  3. interpret those bytes according to the type of pointer that `ptr` is

```
printf("0x%X\n", *ip);   /* 0x1A */
```

- the C compiler is ”smart enough” to ”know” that `+ X` really means add `X * sizeof(*ip)` to `ip`

pointer dereference

- `*ptr` will
  1. treat the value of `ptr` as a memory address
  2. get the bytes of data located at that memory address
  3. interpret those bytes according to the type of pointer that `ptr` is

```
printf("0x%X\n", *ip);   /* 0x1A */
```

- `ip[X] = *(ip + X)`

```
printf("0x%X\n", ip[0]); /* 0x1A */
```

- the C compiler is "smart enough" to "know" that `+ X` really means add `X * sizeof(*ip)` to `ip`

pointers cont.

```c
printf("0x%X\n", i);        /* 0x1A */
printf("0x%X\n", *ip);      /* 0x1A */
printf("0x%X\n", ip[0]);    /* 0x1A */
printf("0x%X\n", *(ip+0)); /* 0x1A */
printf("0x%#X\n", &i);      /* 0x12A0 */
printf("0x%#X\n", ip);      /* 0x12A0 */
printf("0x%#X\n", &ip);     /* 0x???????? */
```

pointers cont.

```c
char * cp = "hello, world";
```

- `cp` is a *pointer* to a `char`

```
0x00004C80 | h | e | l | l | o | , |   | w | o | r | l | d |\0 |

0x???????? |00|00|4C|80|
```

```c
printf("%c\n", *cp);       /* h */
printf("%c\n", cp[0]);     /* h */
printf("%c\n", cp[4]);     /* o */
printf("%c\n", *(cp+4));   /* o */
printf("%s\n", cp);        /* hello, world */
printf("%s\n", cp+7);      /* world */
printf("0x%#X\n", cp);     /* 0x4C80 */
printf("0x%#X\n", &cp);    /* 0x???????? */
```

- why not say `cp` is a *pointer* to a `char` array?

practice

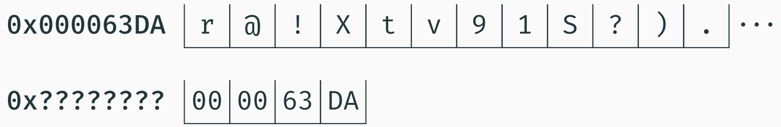
```c
#include <stdio.h>

void swap(int n1, int n2) {
  int tmp = n1;
  n1 = n2;
  n2 = tmp;
}

int main() {
  int v1 = 11, v2 = 77;
  printf("BEFORE  v1=%d, v2=%d\n", v1, v2);
  swap(v1, v2);
  printf("AFTER  v1=%d, v2=%d\n", v1, v2);
  return 0;
}
```

- what's wrong with this program?
- fix the program so that it correctly swaps the two variables' values

heap memory

- designate a block of memory to store value(s) of a particular data type

```c
int * ip = malloc(100*sizeof(int));
```



- allocates enough consecutive memory for 100 `int` values

heap memory

- designate a block of memory to store value(s) of a particular data type

```c
int * ip = malloc(100*sizeof(int));
```

0x000063DA	r	@	!	X	t	v	9	1	S	?	)	.	···

0x????????	00	00	63	DA

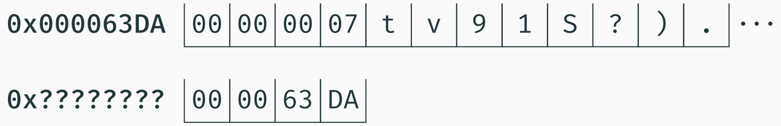
- release a block of memory back to system to be used elsewhere

```c
free(ip);
```

- allocates enough consecutive memory for 100 `int` values

heap memory cont.

```c
ip[0] = 0x7; /* *ip = 0x7; */
```



heap memory cont.

```c
ip[0] = 0x7; /* *ip = 0x7; */
```

```
0x000063DA |00|00|00|07|t |v |9 |1 |S |? |) |. |···
0x???????? |00|00|63|DA|
```

```c
ip[1] = 0xA; /* *(ip + 1) = 0xA; */
```

```
0x000063DA |00|00|00|07|00|00|00|0A|S |? |) |. |···
```

file i/o

```c
#include <stdio.h>

int main() {
  int m, n;
  FILE * fp;

  if (!(fp = fopen("example.txt", "r")))
    return EXIT_FAILURE;
  if (2 != fscanf(fp, "%d %d", &m, &n))
    return EXIT_FAILURE;
  if (!fclose(fp))
    return EXIT_FAILURE;

  return 0;
}
```

final thoughts

- matrices