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

```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;
}
```

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;
}
```

correct headers must be included to access library functions

read data from `stdin` (the terminal)

print data to `stdout` (the terminal)

`scanf` and `printf` are both library functions declared in `stdio.h`

- 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

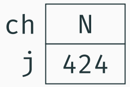
```
for (<init>; <cond>; <incr>) {
  /* ... */
}

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

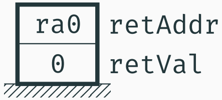
memory model — part i

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.

run-time stack a.k.a. "the stack"

run-time stack
: stores information about the active functions of a C program, including:

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

in that order.

- I will not distinguish between functions and procedures
  - functions that have no return value, i.e., return type `void`, just omit the first bullet point
- 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,
  - actual parameters,
  - return address, and
  - local variables

  in that order.

(a) (b)

- I will not distinguish between functions and procedures
  - functions that have no return value, i.e., return type `void`, just omit the first bullet point
- each program has one stack
- which visualization is correct, had `ch` and `j` been declared as local variables instead of global?

functions

list of formal parameters

function name

return value type

```c
void print_bar (int n) {
  int k;
  for (k=0; k<n; k++) {
    printf("*");
  }
  printf("\n");
}
```

functions

list of formal parameters

function name

return value type

```c
void print_bar (int n) {
  int k;
  for (k=0; k<n; k++) {
    printf("*");
  }
  printf("\n");
}
```

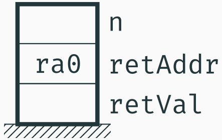
return value type

```c
int fact(int n) {
  int f, j;
  f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}
```

type of `<expr>` must match return type of function

functions — call-by-value

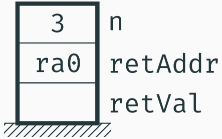
```c
int fact(int n) {
  int f, j;
  f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", fact(n)); // ra1
  return 0;
}
```



- you will have to reconcile this example a little bit with the book, since the solution is written slightly different
- assume nothing about values that are blank — they are not 0

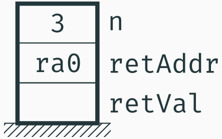
functions — call-by-value

```c
int fact(int n) {
  int f, j;
  f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", fact(n)); // ra1
  return 0;
}
```



- assume the user inputs the number 3

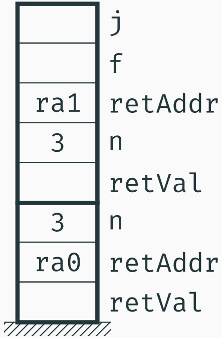
```c
int fact(int n) {
  int f, j;
  f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", fact(n)); // ra1
  return 0;
}
```



- this statement requires that we evaluate the expression `fact(n)`

functions — call-by-value

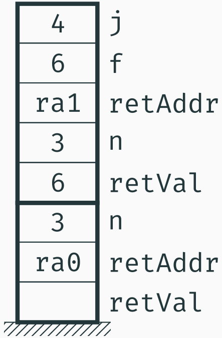
```c
int fact(int n) {
  int f, j;
  f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", fact(n)); // ra1
  return 0;
}
```



- `fact(n)` is a function call, so a new stack frame is pushed on to the stack in the sequence described before:
  - return value
  - actual parameters
  - return address
  - local variables

functions — call-by-value

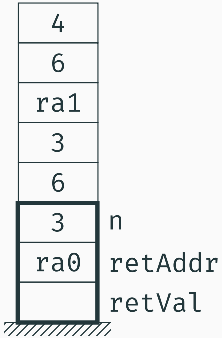
```c
int fact(int n) {
  int f, j;
  f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", fact(n)); // ra1
  return 0;
}
```



- the state of the run-time stack after `fact` function has completed, but before returning

functions — call-by-value

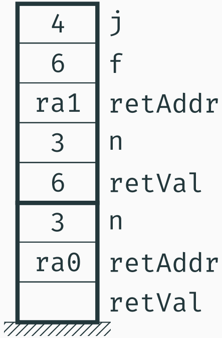
```c
int fact(int n) {
  int f, j;
  f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", fact(n)); // ra1
  return 0;
}
```



- after `fact` returns, its stack frame is deallocated, however, the values computed are still in memory — since return value is always first thing pushed on to stack, the main function knows exactly where to find the value returned by `fact(n)`

functions — call-by-value

```
int fact(int n) {
  int f, j;
  // f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", fact(n));
  scanf("%d", &n);
  printf("%d\n", fact(n)); // ra1
  return 0;
}
```



- assume the the first call to `fact` got lucky and the value 1 happended to be in the memory cell designated to the variable `f`, so that the calculation worked out correctly
- this would be the state of the stack at the beginning of execution of the second call to `fact`

functions — call-by-“reference”

```c
void swap(int r, int s) {
  int temp;
  temp = r;
  r = s;
  s = temp;
}

void order(int x, int y) {
  if (x > y) {
    swap(x, y);
  } // ra2
}

int main() {
  int a, b;
  scanf("%d %d", &a, &b);
  order(a, b);
  printf("d %d\n", a, b); // ra1
  return 0;
}
```

2	b
6	a
ra0	retAddr
	retVal

- in C, parameters are ALWAYS call-by-value, if you want the behavior of call-by-reference, then you pass the address of the variable instead of its value
- but this is still call-by-value, its just that now, the value happens to represent an address of a value
- you will have to reconcile this example a little bit with the book, since the solution is written slightly different
- assume the user inputs the numbers 6 & 2
- what will the stack look like at the end of execution, i.e., when returning from `main` function?

functions — call-by-“reference”

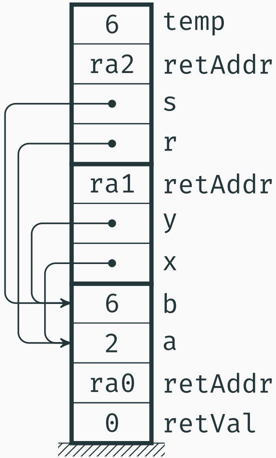
```c
void swap(int r, int s) {
  int temp;
  temp = r;
  r = s;
  s = temp;
}

void order(int x, int y) {
  if (x > y) {
    swap(x, y);
  } // ra2
}

int main() {
  int a, b;
  scanf("%d %d", &a, &b);
  order(a, b);
  printf("d %d\n", a, b); // ra1
  return 0;
}
```

6	temp
ra2	retAddr
6	s
2	r
ra1	retAddr
2	y
6	x
2	b
6	a
ra0	retAddr
0	retVal

- in C, parameters are ALWAYS call-by-value, if you want the behavior of call-by-reference, then you pass the address of the variable instead of its value
- but this is still call-by-value, its just that now, the value happens to represent an address of a value
- you will have to reconcile this example a little bit with the book, since the solution is written slightly different
- assume the user inputs the numbers 6 & 2

functions — call-by-“reference”

```c
void swap(int *r, int *s) {
  int temp;
  temp = *r;
  *r = s;
  *s = temp;
}

void order(int *x, int *y) {
  if (*x > *y) {
    swap(x, y);
  } // ra2
}

int main() {
  int a, b;
  scanf("%d %d", &a, &b);
  order(&a, &b);
  printf("d %d\n", a, b); // ra1
  return 0;
}
```



- now, instead of passing the values **a** and **b**, we will pass their addresses, so that we can update their values, i.e., pass them by “reference”

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`

0x000012A0	00	00	00	1A	i
0x????????	00	00	12	A0	ip

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.

```
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?

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