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

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
```

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
```

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**

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

conditions

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

conditions

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

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

switch

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

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

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

loops

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

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

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

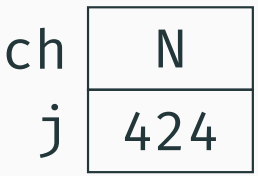
memory model — part i

global variables

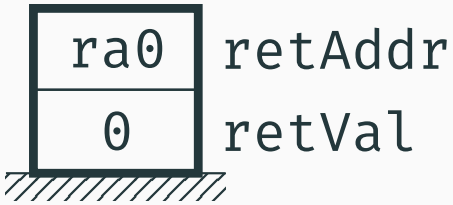
declared outside of any function and remain in place throughout the execution of the entire program. they are stored at a fixed location in memory.

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.

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.

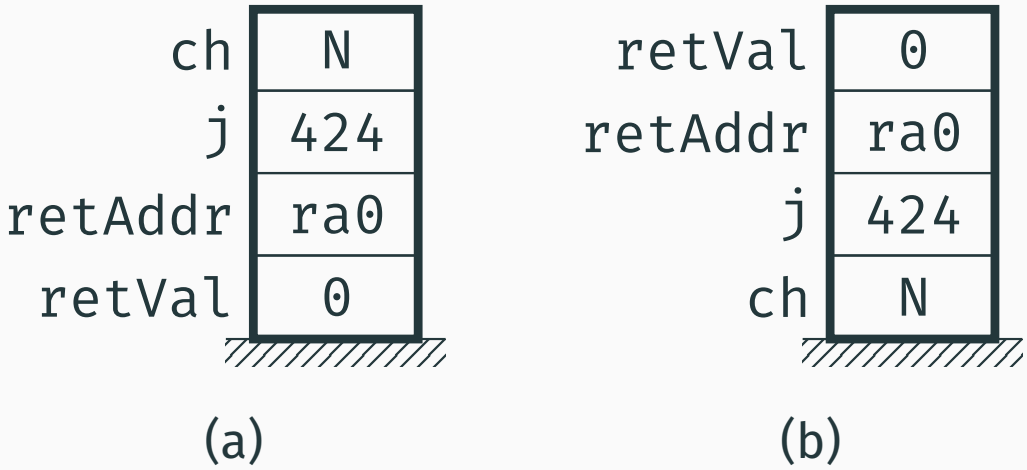
run-time stack a.k.a. “the stack”

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.



functions

list of formal parameters

function name

return value type

```
1  void print_bar (int n) {
2    int k;
3    for (k=0; k<n; k++) {
4      printf("*");
5    }
6    printf("\n");
7  }
```

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");
}
```

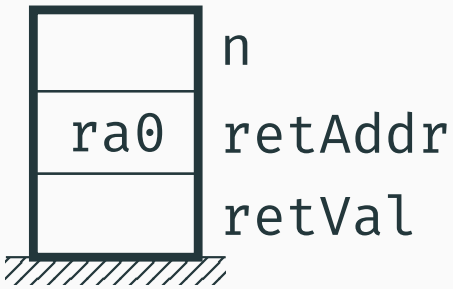
return value type

```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

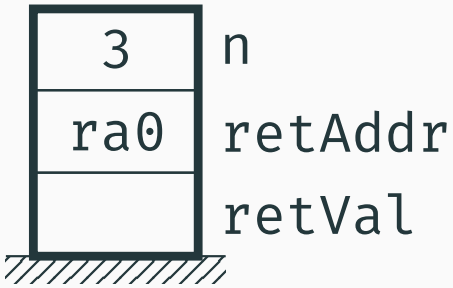
functions — call-by-value

```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;
}
```



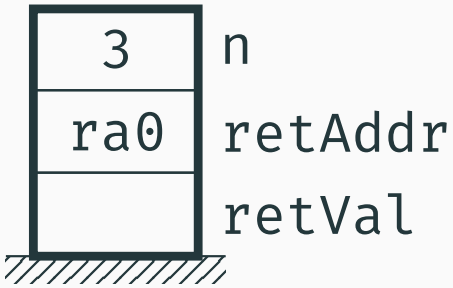
functions — call-by-value

```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;
}
```



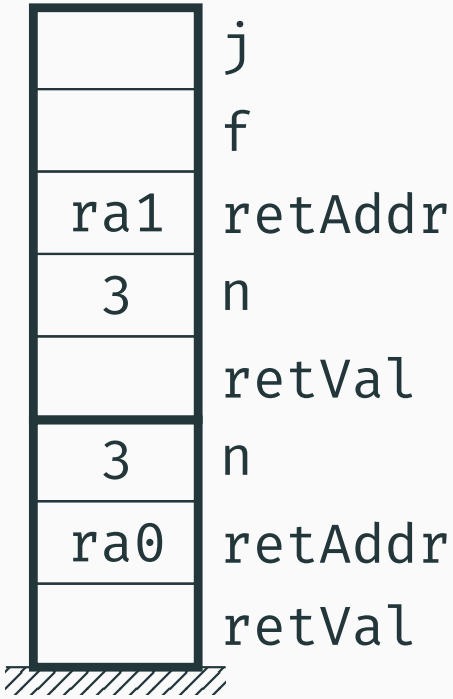
functions — call-by-value

```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



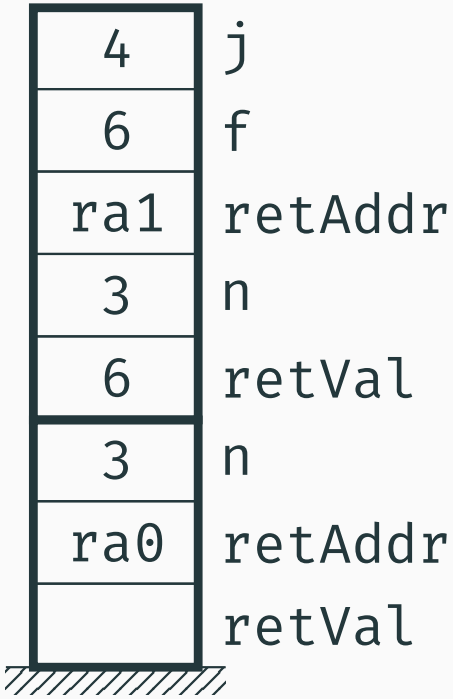
functions — call-by-value

```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



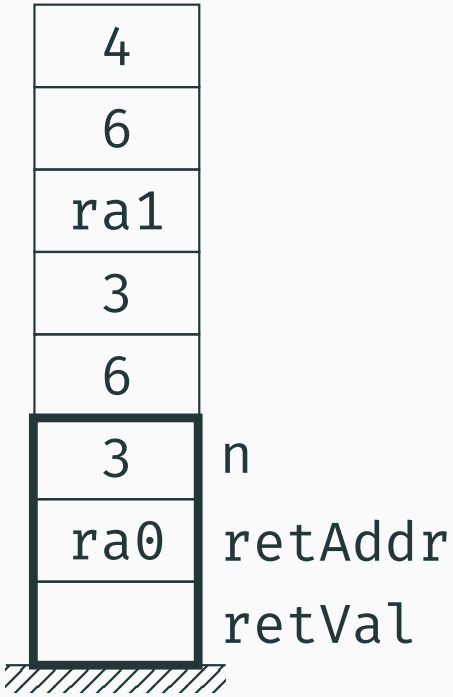
functions — call-by-value

```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



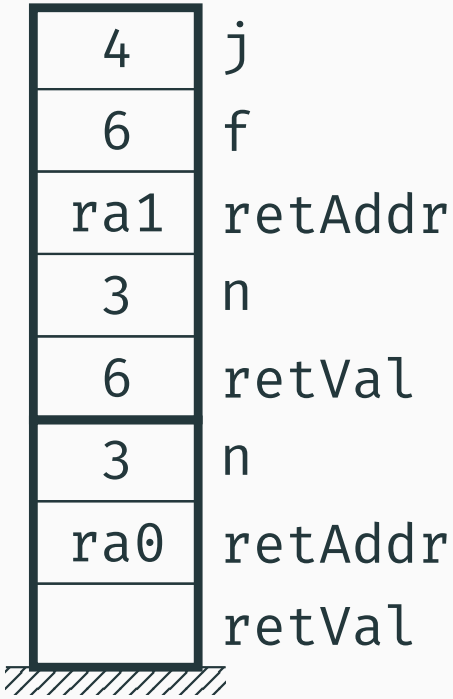
functions — call-by-value

```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



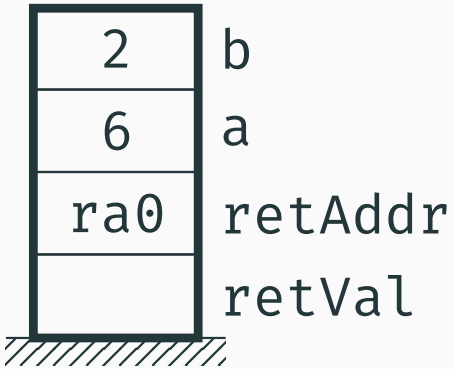
functions — call-by-value

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



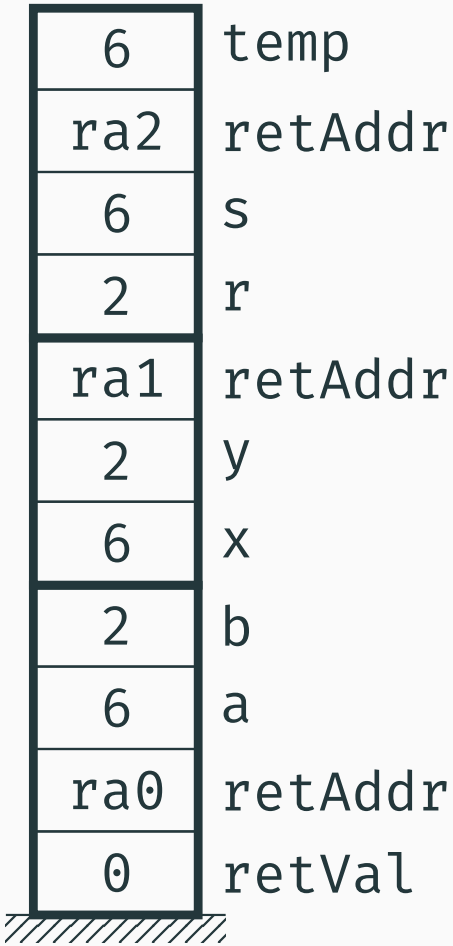
functions — call-by-“reference”

```c
5  void swap(int r, int s) {
6    int temp;
7    temp = r;
8    r = s;
9    s = temp;
10 }
11
12 void order(int x, int y) {
13   if (x > y) {
14     swap(x, y);
15   } // ra2
16 }
17
18 int main() {
19   int a, b;
20   scanf("%d %d", &a, &b);
21   order(a, b);
22   printf("d %d\n", a, b); // ra1
23   return 0;
24 }
```



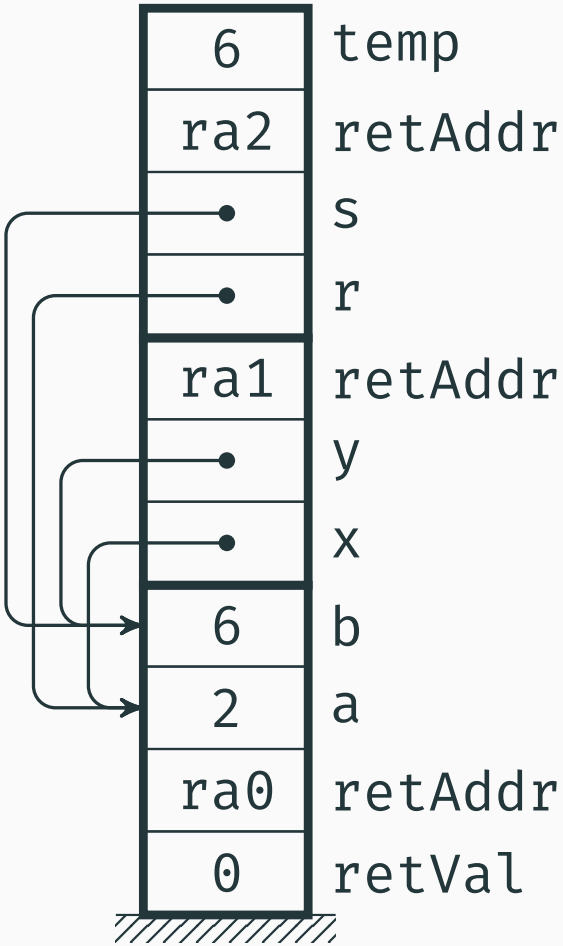
functions — call-by-“reference”

```c
5  void swap(int r, int s) {
6    int temp;
7    temp = r;
8    r = s;
9    s = temp;
10 }
11
12 void order(int x, int y) {
13   if (x > y) {
14     swap(x, y);
15   } // ra2
16 }
17
18 int main() {
19   int a, b;
20   scanf("%d %d", &a, &b);
21   order(a, b);
22   printf("d %d\n", a, b); // ra1
23   return 0;
24 }
```



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;
}
```



pointers

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

```
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???????? */
```

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 */
```

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 */
```

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???????? */
```

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