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;
}
```

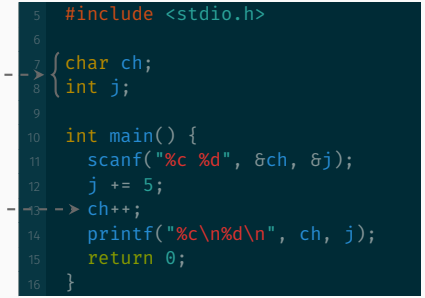
C programs ALWAYS start execution with the `main` function

returning from `main` ends the program

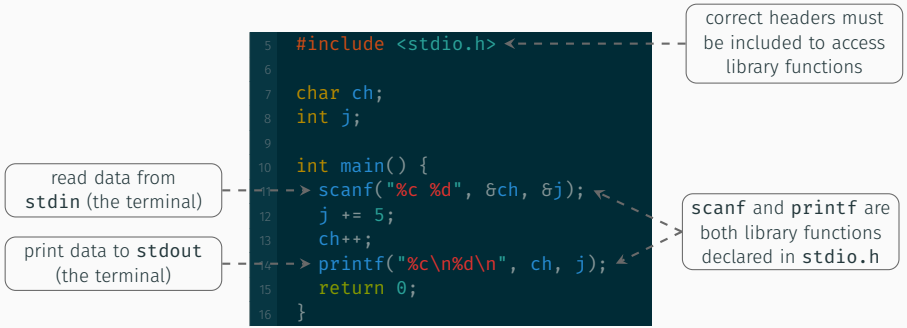
program breakdown

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;
}
```

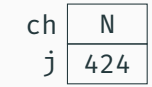
`&` is the address of operator — `scanf` expects the address of the variables where the data will be stored

- address here means the location in memory

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

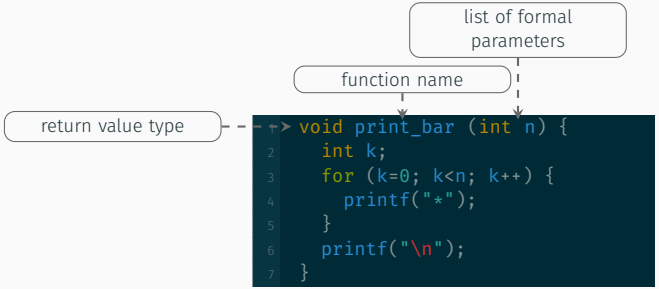
stores information about the active functions of a C program, including:

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

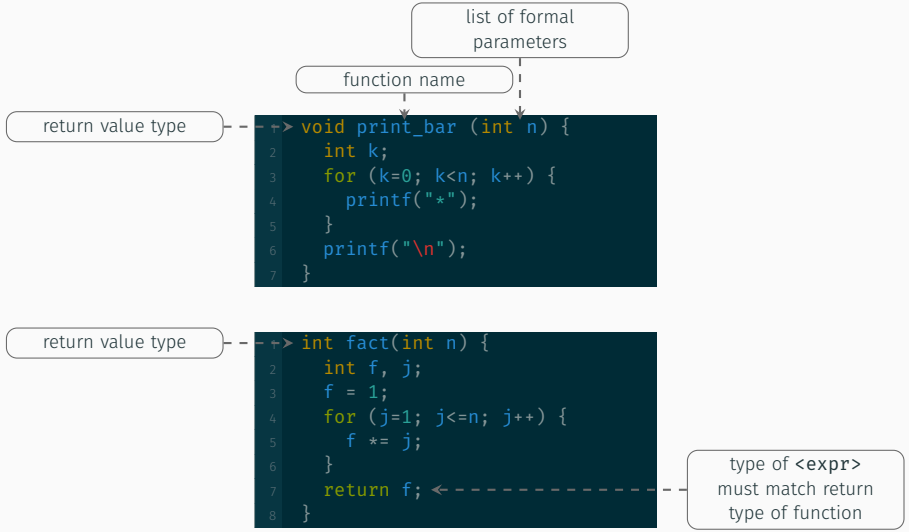
in that order.

- 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

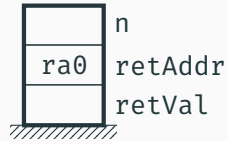

list of formal parameters
function name
return value type
1 void print_bar (int n) {
2 int k;
3 for (k=0; k<n; k++) {
4 printf("*");
5 }
6 printf("\n");
7 }

functions


list of formal parameters
function name
return value type
1 void print_bar (int n) {
2 int k;
3 for (k=0; k<n; k++) {
4 printf("*");
5 }
6 printf("\n");
7 }
return value type
1 int fact(int n) {
2 int f, j;
3 f = 1;
4 for (j=1; j<=n; j++) {
5 f *= j;
6 }
7 return f;
8 }
type of <expr> must match return type of function

functions — call-by-value

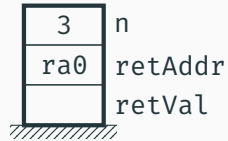
```c
int fact(int n) {
  int f, j;
  f = 1;
  for (j=1; j<=n; j++) {
    f *= j;
  }
  return f;
}

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

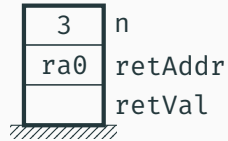
```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



- assume the user inputs the number 3

functions — call-by-value

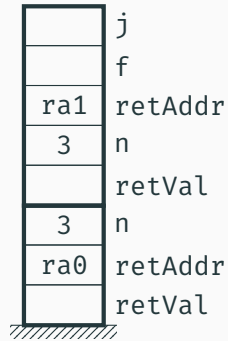
```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



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

functions — call-by-value

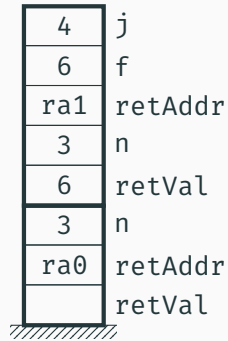
```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



- `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

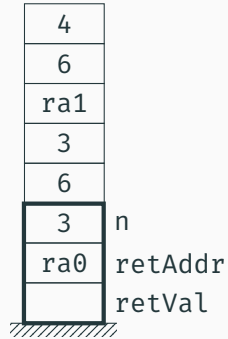
```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



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

functions — call-by-value

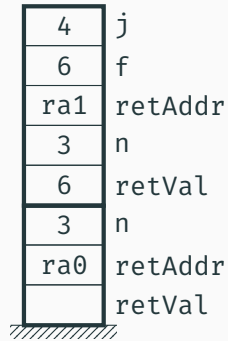
```
5  int fact(int n) {
6    int f, j;
7    f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n)); // ra1
18   return 0;
19 }
```



- 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

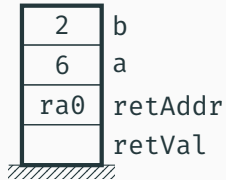
```
5  int fact(int n) {
6    int f, j;
7    // f = 1;
8    for (j=1; j<=n; j++) {
9      f *= j;
10   }
11   return f;
12 }
13
14 int main() {
15   int n;
16   scanf("%d", &n);
17   printf("%d\n", fact(n));
18   scanf("%d", &n);
19   printf("%d\n", fact(n)); // ra1
20   return 0;
21 }
```



- 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”

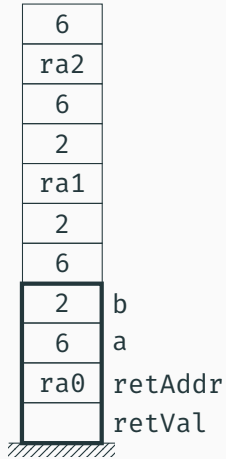
```
5  void swap(int r, int s) {
6    int temp;
7    temp = r;
8    r = s;
9    s = temp;
10 }
11
12 void order(int x, int y) {
13   if (x > y) {
14     swap(x, y);
15   } // ra2
16 }
17
18 int main() {
19   int a, b;
20   scanf("%d %d", &a, &b);
21   order(a, b);
22   printf("d %d\n", a, b); // ra1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



- 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 after the call to `order`, but before the call to `printf`?

functions — call-by-“reference”

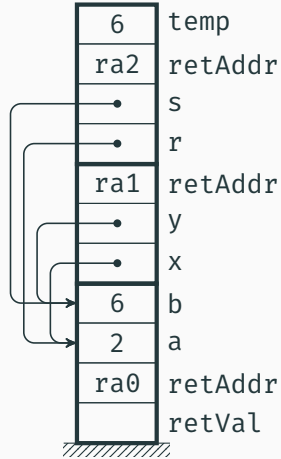
```
5  void swap(int r, int s) {
6    int temp;
7    temp = r;
8    r = s;
9    s = temp;
10 }
11
12 void order(int x, int y) {
13   if (x > y) {
14     swap(x, y);
15   } // ra2
16 }
17
18 int main() {
19   int a, b;
20   scanf("%d %d", &a, &b);
21   order(a, b);
22   printf("d %d\n", a, b); // ra1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



- 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 after the call to `order`, but before the call to `printf`?

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;
}
```



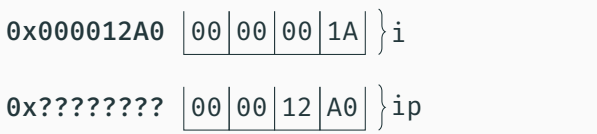
- this is what the stack will look like just before returning from the `swap` function
- now, instead of passing the values **a** and **b**, we will pass their addresses, so that we can update their values, i.e., pass them by “reference”

pointers

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

```
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

```
1 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`


```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?

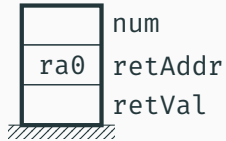
recursion

```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra1
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```

recursion

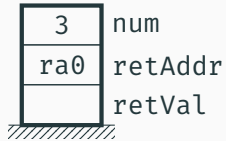
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```

```
$ ./figure2-22
```



recursion

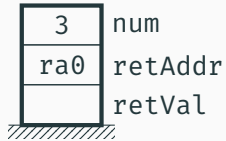
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
```

recursion

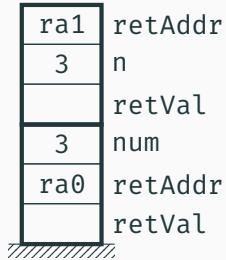
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
```

recursion

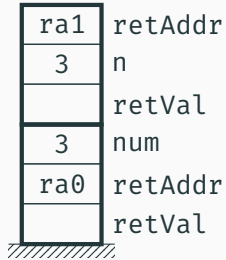
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
```

recursion

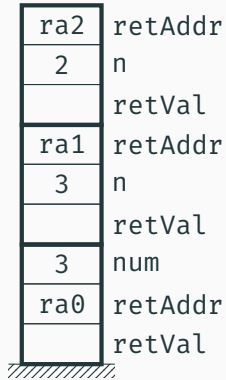
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
```

recursion

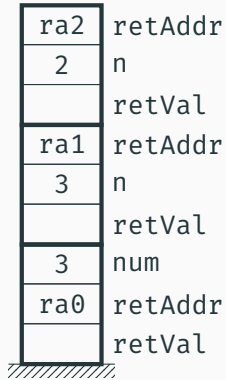
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
```

recursion

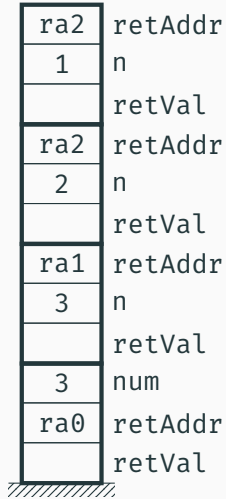
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
```

recursion

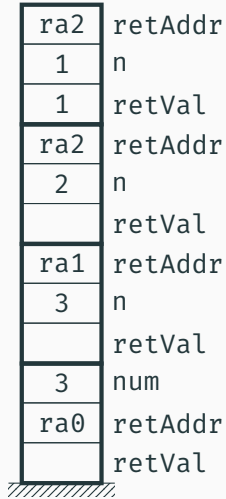
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
```

recursion

```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```

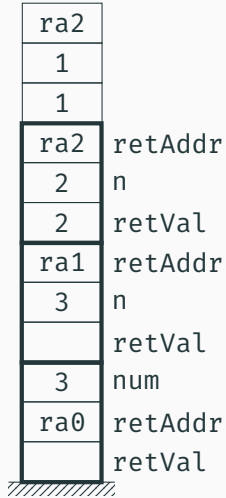



```
$ ./figure2-22
Enter a small integer: 3
```

recursion

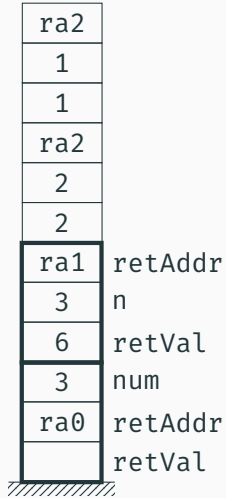
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```

```
$ ./figure2-22
Enter a small integer: 3
```



recursion

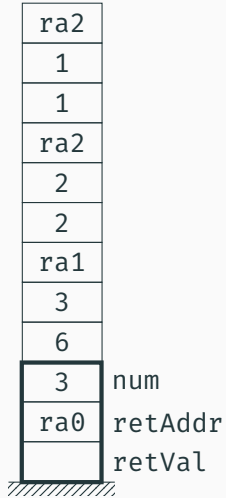
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
```

recursion

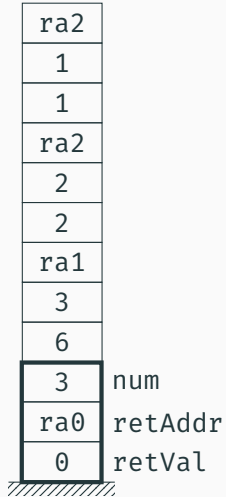
```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
Its factorial is: 6
```

recursion

```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```




```
$ ./figure2-22
Enter a small integer: 3
Its factorial is: 6
```

recursion

```c
#include <stdio.h>

int fact(int n) {
  if (n <= 1) {
    return 1;
  }
  else {
    return n * fact(n - 1); // ra2
  }
}

int main() {
  int num;
  printf("Enter a small integer: ");
  scanf("%d", &num);
  printf("Its factorial is: ");
  printf("%d\n", fact(num)); // ra1
  return 0;
}
```

```
$ ./figure2-22
...
$
```

heap memory

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

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

0x000063DA	A7	38	DC	91	0F	F3	21	1E	76	4B	AA	01	···
0x????????	00	00	63	DA									

heap memory

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

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

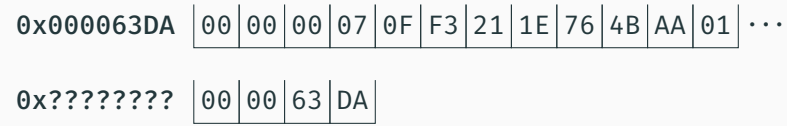
0x000063DA	A7	38	DC	91	0F	F3	21	1E	76	4B	AA	01	···
0x????????	00	00	63	DA									

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

```
1 free(ip);
```

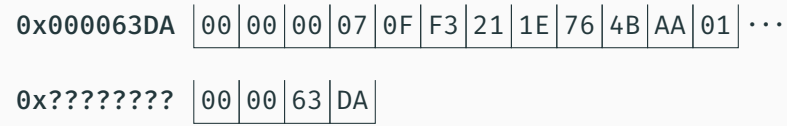
heap memory cont.

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



heap memory cont.

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




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

structures

```c
#include <stdio.h>

int main() {
  struct {
    char first;
    char last;
    int age;
    char gender;
  } bill;

  scanf("%c%c%d%c", &bill.first, &bill.last, &bill.age,
    &bill.gender);
  printf("Initials: %c%c\n", bill.first, bill.last);
  printf("Age: %d\n", bill.age);
  printf("Gender: ");
  if (bill.gender == 'f') {
    printf("fe");
  }
  printf("male\n");
  return 0;
}
```

structures cont.

```c
#include <stdio.h>
#include <stdlib.h>

struct { char *name; } bill;

char * find(char *str, char c) {
  for (; *str != c; str++);
  return str;
}

int main() {
  char *first, *last;

  first = malloc(100);
  scanf("%s", first);
  last = find(first, '-') + 1;
  last[-1] = '\0';
  printf("Initials: %c%c\n", first[0], *last);
  free(first);
  bill.name = malloc(100);
  printf("Full name: %s\n", bill.name);
  free(bill.name);

  return 0;
}
```

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