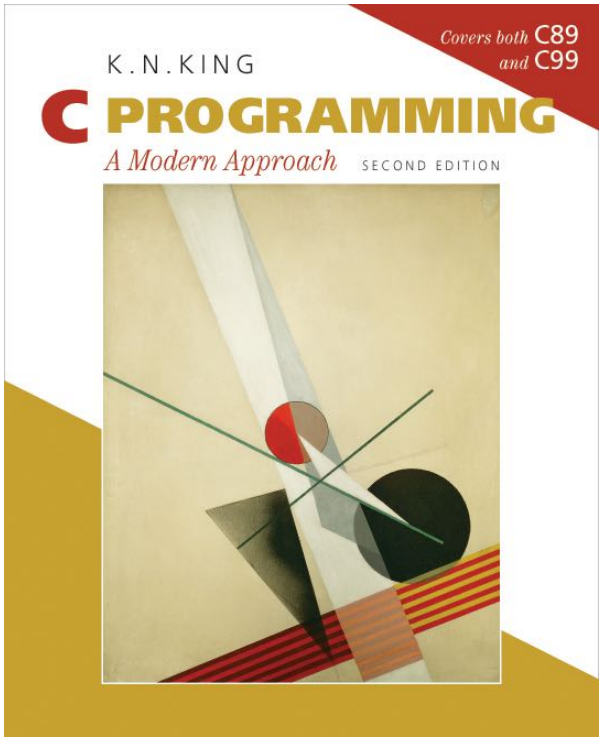
# Fundamentals of C programming

**CS 210 - Fall 2023**
**Vasiliki Kalavri**
**vkalavri@bu.edu**

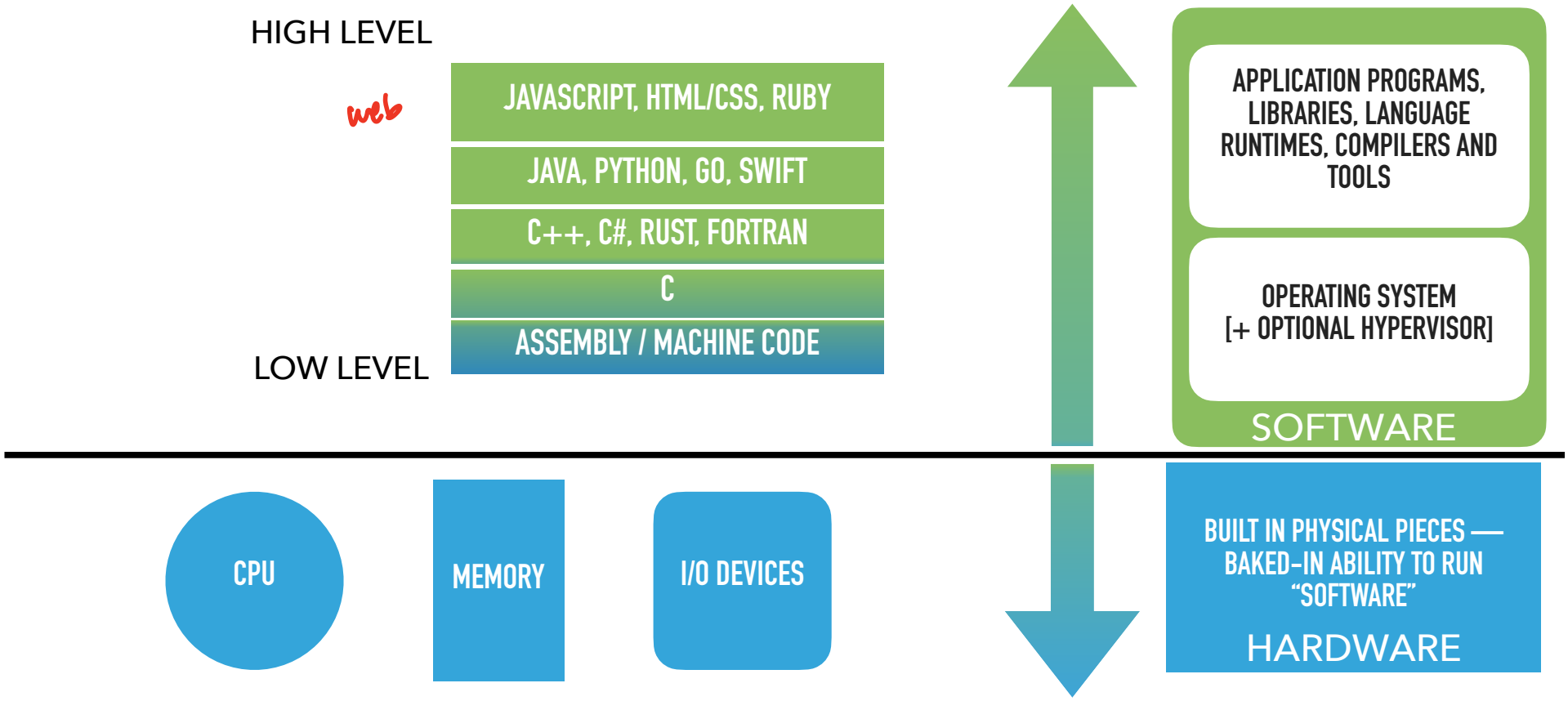
***Some slides adapted from***

# Switching gears…

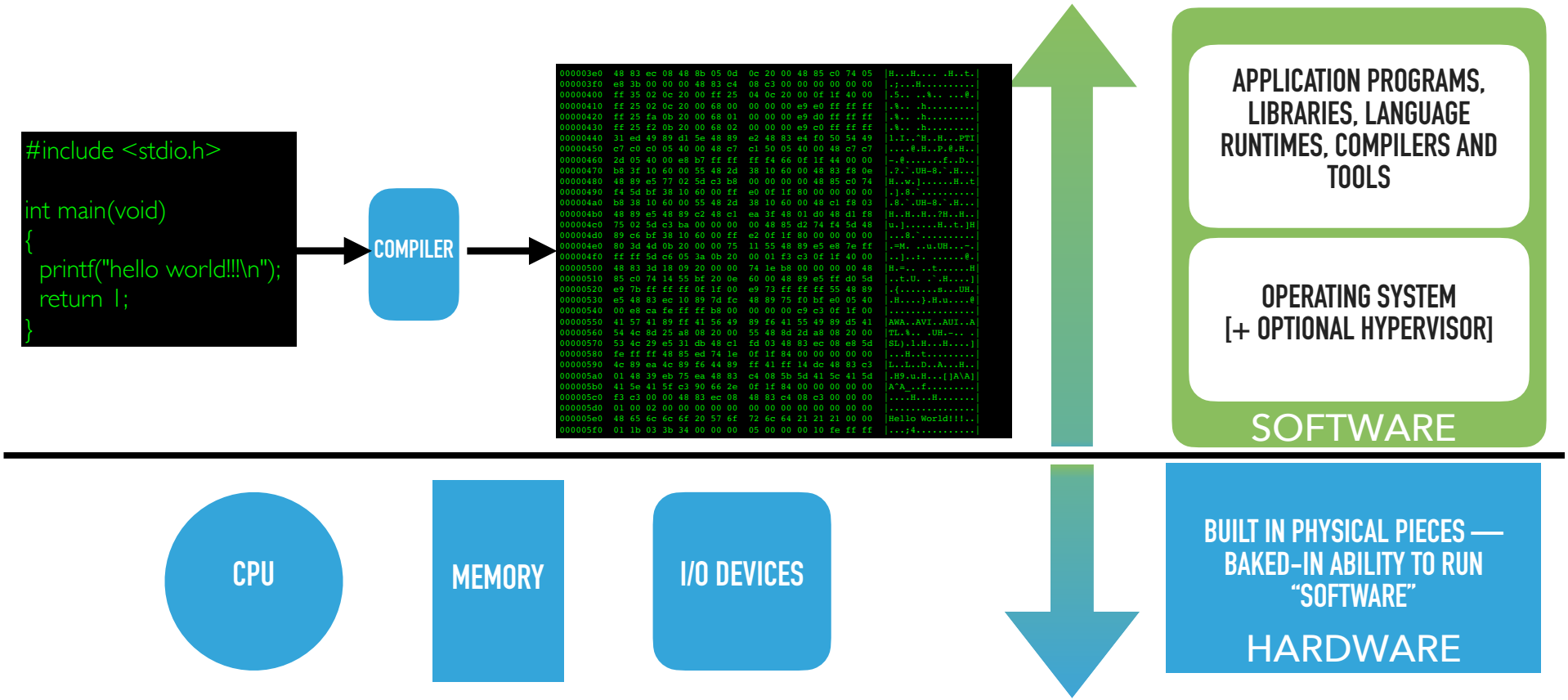
- See Syllabus for assigned readings
- Online lecture notes still useful (https://cs-210-fall-2023.github.io/UndertheCovers/lecturenotes/C/L17.html) but not necessarily in sync with lectures
- Slides will be shared on Piazza



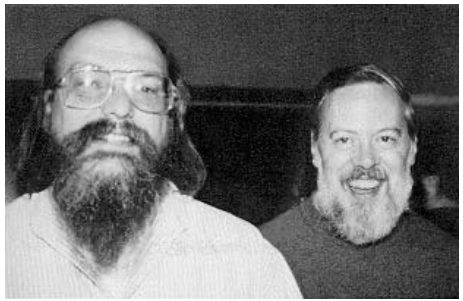
# A programmer's view



# Programs are translated by other programs into different forms



# Origins of C



- C is a by-product of UNIX, developed at Bell Laboratories by Ken Thompson, Dennis Ritchie, and others.
- Thompson designed a small language named B.
- B was based on BCPL, a systems programming language developed in the mid-1960s.

# Origins of C

- By 1971, Ritchie began to develop an extended version of B.
- He called his language NB (“New B”) at first.
- As the language began to diverge more from B, he changed its name to C.
- The language was stable enough by 1973 that UNIX could be rewritten in C.

# C is a Systems Programming Language

Operating Systems
Database Management Systems
Interpreters
3D engines
Web servers
Audio processing
Scientific software




# Properties of C

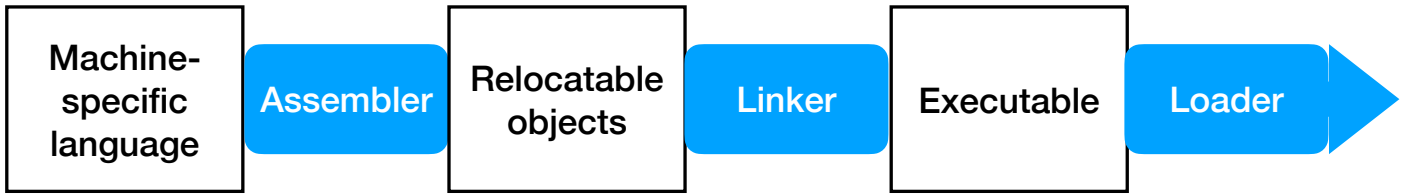
- Low-level ✓
- Small
- Permissive

- Efficiency
- Portability
- Power
- Flexibility
- Standard library
- Integration with UNIX

- Error-prone

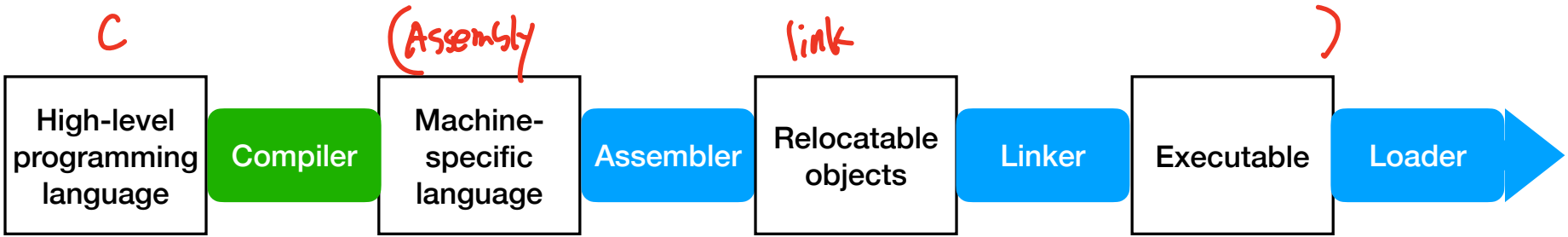
- Learn how to avoid pitfalls.
- Use a debugger! ✓
- Take advantage of existing code libraries.
- Avoid “tricks” and overly complex code.
- Stick to the standard.

# Adding a link to our toolchain




2) compile
3) Assemble
4) link
5) load

# Adding a link to our toolchain




- The **compiler** translates program *source* files, *a description of operations and data types conforming to a machine-independent language*, into machine-specific assembly language.

Description to a machine-independent language

# Two components of the C programming language

- **Core language**: the language syntax and built-operations that you can use to write your programs.
  - Source files that contain descriptions of
    - Function definitions and declarations
    - Data: variable instances, data type definitions
- The compiler translates the source code written in C into binary fragments and the linker can combine them into executables without the source.

1) core language
   function
   Data

2)

# Two components of the C programming language

Core language

- **Core language**: the language syntax and built-operations that you can use to write your programs.
  - Source files that contain descriptions of
    - Function definitions and declarations
    - Data: variable instances, data type definitions
- The compiler translates the source code written in C into binary fragments and the linker can combine them into executables without the source.

Standard library

- **Standard library (libc)**: A base library of common functions.
  - Relocatable object files where the linker can find symbols that other object files reference.
    - Header files with function declarations that you include in your source code
- Standard C library is provided with all compilers and provides C functions for system calls, APIs for dynamic memory and IO, other useful functions.

# Our first C program: csumit.c

```c
long long XARRAY[1024];

long long sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

# Our first C program: csumit.c

1) define variable

```
long long XARRAY[1024];

long long sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

Variable declaration

# Our first C program: csumit.c

type

```
long long XARRAY[1024];
```

Variable declaration

```
long long sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

# Our first C program: csumit.c

type

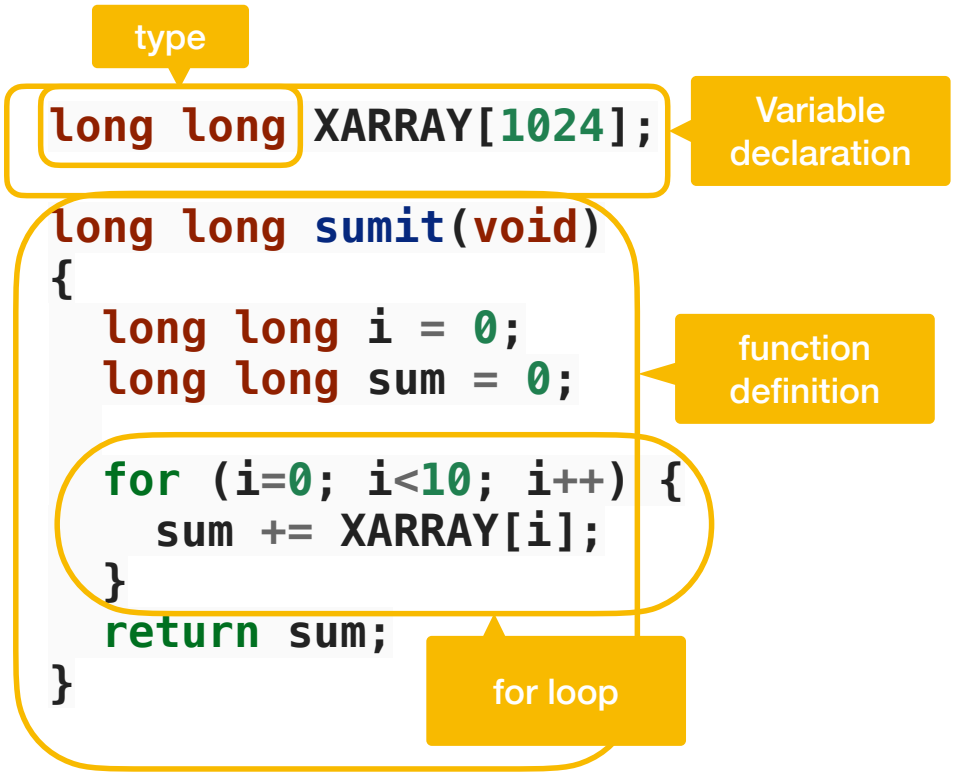
```
long long XARRAY[1024];
```

Variable declaration

```
long long sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

function definition

# Our first C program: csumit.c



# Our first C program: csumit.c

type

```
long long XARRAY[1024];
```

Variable declaration

```
long long sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

function definition

for loop

**Running the compiler: csumit.c —> csumit.s**

```
gcc -fno-inline -fno-stack-protector
-fno-pic  -static -Werror -fcf-
protection=none -fno-asynchronous-
unwind-tables -Os -S -masm=intel
csumit.c -o csumit.s
```

# csumit.c —> csumit.s

```
long long XARRAY[1024];

long long sumit(void)
{
  long long i = 0;
  long long sum = 0;

  for (i=0; i<10; i++) {
    sum += XARRAY[i];
  }
  return sum;
}
```

```
…
sumit:
        xor  r8d, r8d
        xor  eax, eax
.L2:
        add  r8, QWORD PTR XARRAY[0+rax*8]
        inc  rax
        cmp  rax, 10
        jne  .L2
        mov  rax, r8
        ret
…
```

i : rax

We can now run the assembler and linker:
% as -g csumit.s -o csumit.o
% ld -g usecsumit.o csumit.o -o usecsumit

# Call csumit: usesumit.s

```
    .intel_syntax noprefix
    .global _start
_start:
     call sumit
     mov rdi, 0
     mov rax, 60
     syscall       # exit(0)
```

We can now run the assembler and linker:

```
% as -g  usecsumit.s -o usecsumit.o
% as -g  csumit.s -o csumit.o
% ld -g  usecsumit.o csumit.o -o usecsumit
```

# The gcc compiler

- A very sophisticated program with many options that control the assembly code it creates
  - See https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- The term *compiler* is used in multiple ways:
  - The component of the toolchain that translates C into assembly.
  - The master command of the toolchain that knows how to invoke each component:
    - The default behavior is to try and run all steps and create an executable
      - eg. **gcc my1.c my2.c myasm.S neuralnet.o -o myexe**
    - if any errors occur, stop and report them
    - it creates all intermediary files as temporaries and removes them once done
      - eg. it creates `.s` and `.o` files as it needs too and deletes them when it is done
  - See UndertheCovers/lecturenotes/C/L17.html for details

# Gdb is still your best friend!

You can do everything we've been doing so far:

- examine memory ✔
- list assembly source ✔
- disassemble
- set breakpoints

- But now, we can work with C source level using `-g` to produce debug info. We can:
  - list C source that corresponds to the opcodes
  - set breakpoints via C source lines, e.g. `break 8`
  - examine C variables with the debugger knowing the correct types
    - `p I`
    - `p sum`
    - `p XARRAY`
    - `p XARRAY[0]`

# “hello world” in C

```c
/* Name: hello.c
Purpose: Prints “hello, world. */

#include <stdio.h>

int main(void)
{
  printf(“hello, world.\n”);
  return 0;
}
```

```
% gcc -o hello -g hello.c
% ./hello
```

# “hello world” in C

Standard IO

```c
/* Name: hello.c
Purpose: Prints “hello, world. */

#include <stdio.h>

int main(void)
{
  printf(“hello, world.\n”);
  return 0;
}
```

Directive: in this case, a header containing information about C’s standard IO library.

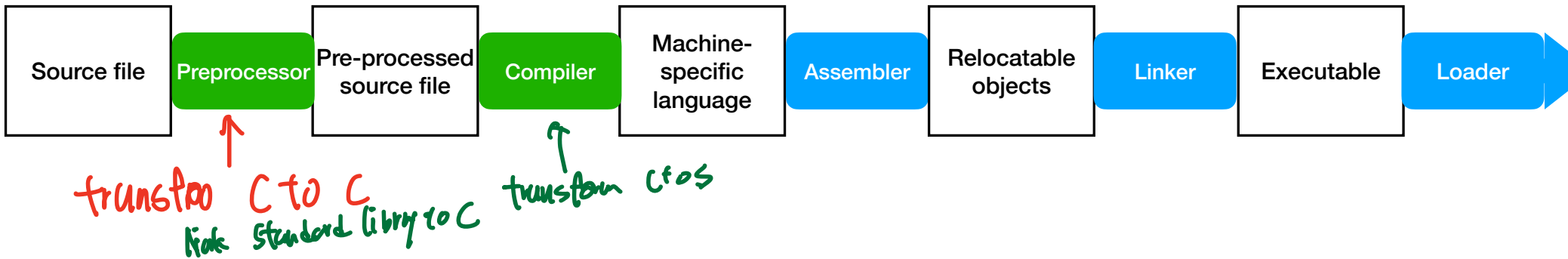
```
% gcc -o hello -g hello.c
% ./hello
```

보통의 -o hello -g hello.c

name of executable

source file

# Another link: the preprocessor




- The **preprocessor** runs prior to the compiler (or assembler) and expands directives that start with '#':
  - Include header files
  - Expand macros
  - Enable conditional compilation

# Program: Converting from Fahrenheit to Celsius

- The `celsius.c` program prompts the user to enter a Fahrenheit temperature; it then prints the equivalent Celsius temperature.
- Sample program output:

```
Enter Fahrenheit temperature: 212
Celsius equivalent: 100.0
```

- The program will allow temperatures that aren't integers.

```c
/* Converts a Fahrenheit temperature to Celsius */

#include <stdio.h>

#define FREEZING_PT 32.0f
#define SCALE_FACTOR (5.0f / 9.0f)

int main(void)
{
  float fahrenheit, celsius;

  printf("Enter Fahrenheit temperature: ");
  scanf("%f", &fahrenheit);

  celsius = (fahrenheit - FREEZING_PT) * SCALE_FACTOR;

  printf("Celsius equivalent: %.1f\n", celsius);

  return 0;
}
```

f: float

# Attendance time

# Formatted Input/Output

# The `printf` Function

- The `printf` function must be supplied with a ***format string***, followed by any values that are to be inserted into the string during printing:

  `printf(`*string*`, `*expr*`1, `*expr*`2, ...);`

- The format string may contain both ordinary characters and ***conversion specifications***, which begin with the `%` character.
  - %d is used for int values
  - %f is used for float values

# The `printf` Function

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

- Ordinary characters in a format string are printed as they appear in the string; conversion specifications are replaced.
- Output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

# The `printf` Function

```
printf("%d %d\n", i); // Too many specifications
printf("%d\n", i, j); // Too few specifications
printf("%f %d\n", i, x);// Inappropriate specification
```

- Compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items.
- Or that a conversion specification is appropriate:

```c
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

- Output:

```
|40|000 40|40   |  040|
|   839.210| 8.392e+02|839.21    |
```

```c
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

minimum field width

- Output:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21    |
```

```c
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

precision

minimum field width

- Output:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21     |
```

# The `scanf` Function

- `scanf` reads input according to a particular format.
- A `scanf` format string may contain both ordinary characters and conversion specifications.
- The conversions allowed with `scanf` are essentially the same as those used with `printf`.

# How `scanf` works

- `scanf` tries to match groups of input characters with conversion specifications in the format string.

- For each conversion specification, `scanf` tries to locate an item of the appropriate type in the input data, **skipping blank space** if necessary.

- `scanf` then reads the item, stopping when it reaches a character that can't belong to the item.
  - If the item was read successfully, `scanf` continues processing the rest of the format string.
  - If not, `scanf` returns immediately.

# How `scanf` Works

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Sample input:

  `1-20.3-4.0¤`

# How `scanf` Works

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

- Sample input:

  `1-20.3-4.0¤`

앞에서 부터 들어 와서 그것부터 가져감

- Here's how `scanf` would process the new input:
  - `%d`. Stores 1 into `i` and puts the `-` character back.
  - `%d`. Stores –20 into `j` and puts the `.` character back.
  - `%f`. Stores 0.3 into `x` and puts the `-` character back.
  - `%f`. Stores –4.0 into `y` and puts the new-line character back.

# Confusing `printf` with `scanf`

```c
printf("%d %d\n", &i, &j);   /*** WRONG ***/
```

```c
scanf("%d, %d", &i, &j);
```


- Sample input:

```
42 -99
```

# Confusing `printf` with `scanf`

- Putting a new-line character at the end of a `scanf` format string is usually a bad idea.
- To `scanf`, a new-line character in a format string is equivalent to a space; both cause `scanf` to advance to the next non-white-space character.
- If the format string is `"%d\n"`, `scanf` will skip white space, read an integer, then skip to the next non-white-space character.
- A format string like this can cause an interactive program to “hang.”

# Program: Adding Fractions

- The `addfrac.c` program prompts the user to enter two fractions and then displays their sum.

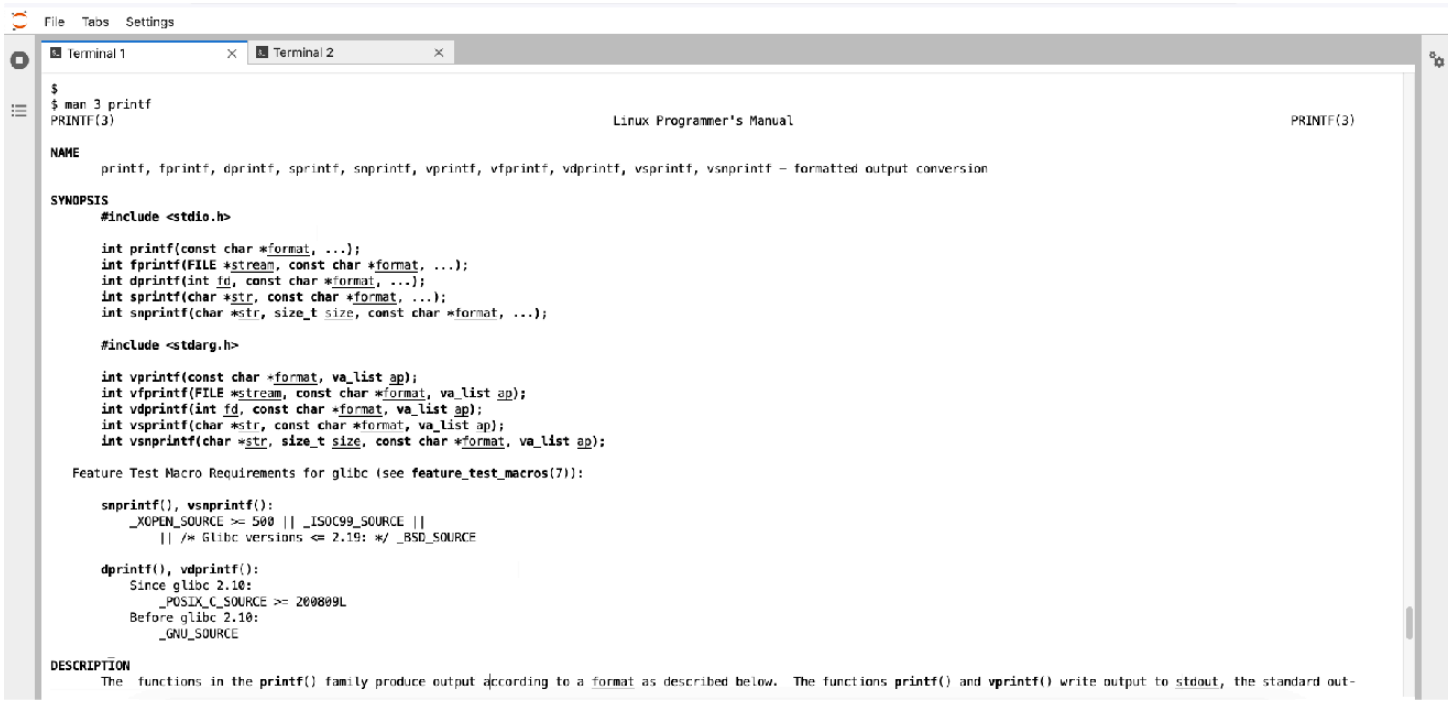
- Sample program output:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

```c
/* Adds two fractions */

#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Enter first fraction: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Enter second fraction: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 *denom1;
  result_denom = denom1 * denom2;
  printf("The sum is %d/%d\n", result_num, result_denom);

  return 0;
}
```

# See man pages for more information

- `man 3 printf`
- `man 3 scanf`



# Assignments and side effects

# Assignments are operators

```c
int i;
float f;

i = 72.99f;   /* i is now 72 */
f = 136;      /* f is now 136.0 */
```

- The value of an assignment $v = e$ is the value of $v$ after the assignment.
  - The value of `i = 72.99f` is 72 (not 72.99).

# Side Effects

- An operator that modifies one of its operands is said to have a ***side effect.***
- The simple assignment operator has a side effect: it modifies its left operand.
- Evaluating the expression `i = 0` produces the result 0 and—as a side effect—assigns 0 to `i`.

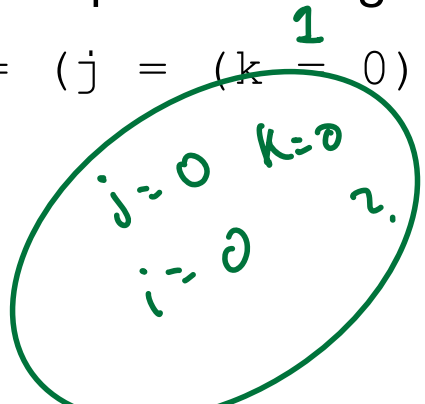
# Side Effects

- Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

- The = operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```



# Side Effects

- Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;

f = i = 33.3f;
```

- `i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3).

# Side Effects

- An assignment of the form $v = e$ is allowed wherever a value of type $v$ would be permitted:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k);
  /* prints "1 1 2" */
```

- "Embedded assignments" can make programs hard to read.

- They can also be a source of subtle bugs.

# Selection statements

# Statements

- Most of C's remaining statements fall into three categories:
  - ***Selection statements:*** `if` and `switch`
  - ***Iteration statements:*** `while`, `do`, and `for`
  - ***Jump statements:*** `break`, `continue`, and `goto`. (`return` also belongs in this category.)

# The `if` Statement

- In its simplest form, the `if` statement has the form

  `if ( `*expression*` ) `*statement*

- When an `if` statement is executed, *expression* is evaluated; if its value is nonzero, *statement* is executed.

- Example:

```
if (line_num == MAX_LINES)
  line_num = 0;
```

# The `if` Statement

- Confusing `==` (equality) with `=` (assignment) is perhaps the most common C programming error.
- The statement

  ```
  if (i == 0) …
  ```

  tests whether `i` is equal to 0.
- The statement

  ```
  if (i = 0) …
  ```

  assigns 0 to `i`, then tests whether the result is nonzero.

# The `else` Clause

- An `if` statement may have an `else` clause:

  `if (` *expression* `)` *statement* `else` *statement*

- The statement that follows the word `else` is executed if the expression has the value 0.

- Example:

```
if (i > j)
  max = i;
else
  max = j;
```

# Program: Calculating a Broker's Commission

- The `broker.c` program asks the user to enter the amount of the trade, then displays the amount of the commission:

```
Enter value of trade: 30000
Commission: $166.00
```

- The heart of the program is a cascaded `if` statement that determines which range the trade falls into.

```c
/* Calculates a broker's commission */
#include <stdio.h>

int main(void)
{
  float commission, value;

  printf("Enter value of trade: ");
  scanf("%f", &value);

  if (value < 2500.00f)
    commission = 30.00f + .017f * value;
  else if (value < 6250.00f)
    commission = 56.00f + .0066f * value;
  else if (value < 20000.00f)
    commission = 76.00f + .0034f * value;
  else if (value < 50000.00f)
    commission = 100.00f + .0022f * value;
  else if (value < 500000.00f)
    commission = 155.00f + .0011f * value;
  else
    commission = 255.00f + .0009f * value;
  if (commission < 39.00f)
    commission = 39.00f;

  printf("Commission: $%.2f\n", commission);

  return 0;
}
```

# The `switch` Statement

- A cascaded `if` statement can be used to compare an expression against a series of values:

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf("Illegal grade");
```

# The `switch` Statement

- The `switch` statement is an alternative:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

# The `switch` Statement

```
switch ( expression ) {
  case constant-expression : statements
  …
  case constant-expression : statements
  default : statements
}
```

# The `switch` Statement

- Duplicate case labels aren't allowed.
- The order of the cases doesn't matter, and the `default` case doesn't need to come last.
- Several case labels may precede a group of statements:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:  printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

# The Role of the `break` Statement

```
switch (grade) {
   case 4:  printf("Excellent");
   case 3:  printf("Good");
   case 2:  printf("Average");
   case 1:  printf("Poor");
   case 0:  printf("Failing");
   default: printf("Illegal grade");
 }
```

**What's printed if the value of grade is 3?**

- Without `break` (or some other jump statement) at the end of a case, control will flow into the next case.

# Program: Printing a Date in Legal Form

- Contracts and other legal documents are often dated in the following way:

```
Dated this __________ day of __________ , 20__ .
```

- The `date.c` program will display a date in this form after the user enters the date in month/day/year form:

```
Enter date (mm/dd/yy): 7/19/14
Dated this 19th day of July, 2014.
```

- The program uses `switch` statements to add “th” (or “st” or “nd” or “rd”) to the day, and to print the month as a word instead of a number.

```c
/* Prints a date in legal form */
#include <stdio.h>

int main(void)
{
  int month, day, year;

  printf("Enter date (mm/dd/yy): ");
  scanf("%d /%d /%d", &month, &day, &year);

  printf("Dated this %d", day);
  switch (day) {
    case 1: case 21: case 31:
      printf("st"); break;
    case 2: case 22:
      printf("nd"); break;
    case 3: case 23:
      printf("rd"); break;
    default: printf("th"); break;
  }
  printf(" day of ");
```

```
    switch (month) {
    case 1:  printf("January");   break;
    case 2:  printf("February");  break;
    case 3:  printf("March");     break;
    case 4:  printf("April");     break;
    case 5:  printf("May");       break;
    case 6:  printf("June");      break;
    case 7:  printf("July");      break;
    case 8:  printf("August");    break;
    case 9:  printf("September"); break;
    case 10: printf("October");   break;
    case 11: printf("November");  break;
    case 12: printf("December");  break;
  }

  printf(", 20%.2d.\n", year);
  return 0;
}
```

# Loops

```c
/* Prints a table of squares using a while statement */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  i = 1;
  while (i <= n) {
    printf("%10d%10d\n", i, i * i);
    i++;
  }

  return 0;
}
```

```c
/* Prints a table of squares using a for statement */

#include <stdio.h>

int main(void)
{
  int i, n;

  printf("This program prints a table of squares.\n");
  printf("Enter number of entries in table: ");
  scanf("%d", &n);

  for (i = 1; i <= n; i++)
    printf("%10d%10d\n", i, i * i);

  return 0;
}
```

# The `break` Statement

- The `break` statement can transfer control out of a switch statement, but it can also be used to jump out of a `while`, `do`, or `for` loop.

- A loop that checks whether a number `n` is prime can use a `break` statement to terminate the loop as soon as a divisor is found:

```
for (d = 2; d < n; d++)
  if (n % d == 0)
    break;
```

# The `break` Statement

- The `break` statement is particularly useful for writing loops in which the exit point is in the middle of the body rather than at the beginning or end.
- Loops that read user input, terminating when a particular value is entered, often fall into this category:

```
for (;;) {
  printf("Enter a number (enter 0 to stop): ");
  scanf("%d", &n);
  if (n == 0)
    break;
  printf("%d cubed is %d\n", n, n * n * n);
}
```

# The `break` Statement

- A `break` statement transfers control out of the innermost enclosing `while`, `do`, `for`, or `switch`.
- When these statements are nested, the `break` statement can escape only one level of nesting.
- Example:

```
while (…) {
  switch (…) {
    …
    break;
    …
  }
}
```

- `break` transfers control out of the `switch` statement, but not out of the `while` loop.

# The `continue` Statement

- The `continue` statement is similar to `break`:
  - `break` transfers control just past the end of a loop.
  - `continue` transfers control to a point just before the end of the loop body.

- With `break`, control leaves the loop; with `continue`, control remains inside the loop.

- There's another difference between `break` and `continue`: `break` can be used in `switch` statements and loops (`while`, `do`, and `for`), whereas `continue` is limited to loops.

# The `continue` Statement

- A loop that uses the `continue` statement:

```
n = 0;
sum = 0;
while (n < 10) {
  scanf("%d", &i);
  if (i == 0)
    continue;
  sum += i;
  n++;
  /* continue jumps to here */
}
```

```c
/**
* A basic integer calculator.
* It supports addition, subtraction,
* multiplication, division, and power.
*
* **/

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

void print_help();
void add(int, int);
int mul(int, int);
int division(int i, int j);
int power(int i, int j);
int rec_power(int i, int j);
```

```c
int main() {

  int i, j;
  char op;

  // print instructions
  print_help();

  while (1) {
    printf("Enter operator or Q to exit: ");
    // read operation
    scanf(" %c ", &op);

    if (op != 'Q') {
      // read input numbers
      printf("Enter two integers: ");
      scanf("%d %d", &i, &j);

      if (i < 0 || j < 0) {
        printf("Negative numbers are ignored.\n");
        continue;
      }

      // apply operation
      switch (op) {
        case '+': add(i, j); break;
        case '-': printf("Result: %d\n", i-j); break;
        case '*': mul(i, j); break;
        case '/': {
          int ret = division(i, j);
          if (ret < 0) break;
          printf("Result: %d\n", ret);
          break;
        }
        case '^': printf("Result: %d\n", power(i, j)); break;
      default:  printf("Invalid operator.\n"); break;
      }
    }
    else {
      printf("Sorry to see you go :/\n");
      return 0;
    }
  }
  return 0;
}
```

```c
int main() {

  int i, j;
  char op;

  // print instructions
  print_help();

  while (1) {
    printf("Enter operator or Q to exit: ");
    // read operation
    scanf(" %c ", &op);

    if (op != 'Q') {
      // read input numbers
      printf("Enter two integers: ");
      scanf("%d %d", &i, &j);

      if (i < 0 || j < 0) {
        printf("Negative numbers are ignored.\n");
        continue;
      }

      // apply operation
      switch (op) {
        case '+': add(i, j); break;
        case '-': printf("Result: %d\n", i-j); break;
        case '*': mul(i, j); break;
        case '/': {
          int ret = division(i, j);
          if (ret < 0) break;
          printf("Result: %d\n", ret);
          break;
        }
        case '^': printf("Result: %d\n", power(i, j)); break;
      default:  printf("Invalid operator.\n"); break;
      }
    }
    else {
      printf("Sorry to see you go :/\n");
      return 0;
    }
  }
  return 0;
}
```

```c
/** Prints the calculator usage instructions **/
void print_help() {
  printf("\n***Welcome to the basic calculator program!***\n\n");
  printf("Select one of the following operations or type 'Q' to exit\n");
  printf("+\t: Addition\n");
  printf("-\t: Subtraction\n");
  printf("*\t: Multiplication\n");
  printf("\\\t: Division\n");
  printf("^\t: Power\n");
}

// adds two integers
void add(int i, int j) {
  int res = i + j;
  printf("Result: %d\n", res);
}

// multiplies two numbers
int mul(int i, int j) {
  int k, res = 0;
  for (k=0; k<j; k++) {
    res += i;
  }
  printf("Result: %d\n", res);
  return res;
}
```

```c
// divides two positive numbers i, j
// returns -1 if division with 0 is attempted
int division(int i, int j) {
  if (j==0) {
    printf("ERROR. Division with 0\n");
    return -1;
  }
  else {
    return i/j;
  }
}

// computes i^j using mul()
int power(int i, int j) {
  int res = 1;
  while(j > 0) {
    res = mul(res, i);
    j--;
  }
  return res;
}

// computes i^j recursively
int rec_power(int i, int j) {
  if (j==0) {
    return 1;
  }
  else {
    return i * rec_power(i, j-1);
  }
}
```