CS 101
Computer Programming and Utilization

Functions

Suyash P. Awate

Functions

- "Function", generalizes notion of "command"
  - Some languages call it a "procedure"
- If we had a gcd() function, we could write:
- How would we create this function ?

```
main_program{
    int a=36,b=24,c=99,d=47;
    cout << gcd(a,b) << endl;
    cout << gcd(c,d) << endl;
}
```

```
int gcd                 // return-type function-name
  (int m, int n)        // parameter list: (parameter-type parameter-name ...)
{                       // beginning of function body
```

  - Function syntax
    - Function name
    - Inputs to the function: "parameters" or "arguments"
      - One or more. They have a name. They have a data type.
    - Output of the function: "return type"
    - Data processing algorithm embodied by the function: "body"
      - Parameter variables will be used within function's body
      - Body should return variable of type "return type"

Functions

- "Function", generalizes notion of "command"
  - Some languages call it a "procedure"
- If we had a gcd() function, we could write:
- How would we create this function ?

```
main_program{
   int a=36,b=24,c=99,d=47;
   cout << gcd(a,b) << endl;
   cout << gcd(c,d) << endl;
}
```

```
int gcd                 // return-type function-name
  (int m, int n)        // parameter list: (parameter-type parameter-name ...)
{                       // beginning of function body
  while(m % n != 0){
    int Remainder = m % n;
    m = n;
    n = Remainder;
  }
  return n;
}                       // end of function body
```

```
type-of-return-value function-name (parameter1-type parameter1-name,
  parameter2-type parameter2-name, ...){
  body
}
```

Functions

- What happens when control comes to "gcd(a,b)" ?
  - Get values of variables a,b
    - If arguments are expressions, evaluate them to get their values
  - Execution of main program is suspended; to be resumed later
  - Function will need to have its own area of memory
    - e.g., the first variables created in this memory are the function's parameters
    - This memory is called the activation frame, or call stack, of function call
  - Copy argument/parameter values from main program's memory region to function's memory region ("call by value", "pass by value")
  - Execute function's body
    - Commands in body can refer to only those variables that are within function's call stack
    - e.g., variable "Remainder" is "local" to function call
  - When a return statement is encountered, return-expression evaluated and value copied into activation frame of main program
  - Activation call of function is destroyed
  - Control returns to main program; execution starts from next statement

Functions

- Function execution: “call by value”
  - Only values of arguments passed from calling program’s frame to called function’s frame
  - Variables within main program aren’t accessible (out of scope) to code within called function;
    and vice versa

```
main_program{
  int a=36,b=24,c=99,d=47;
  cout << gcd(a,b) << endl;
  cout << gcd(c,d) << endl;
}
```

```
int gcd                 // retu
  (int m, int n)        // para
{                       // begi
  while(m % n != 0){
    int Remainder = m % n;
    m = n;
    n = Remainder;
  }
  return n;
}                       // end o
```

Activation frame of main_program	Activation frame of gcd(a,b)
a : 36 b : 24 c : 99 d : 47	m : 36 n : 24

(a) After copying arguments.

Activation frame of main_program	Activation frame of gcd(a,b)
a : 36 b : 24 c : 99 d : 47	m : 24 n : 12 Remainder : 12

(a) At the end of the first iteration of the loop in gcd.

Functions

- Nested function calls

```
main_program{
   cout << lcm(36,24) << endl;
}   int lcm(int m, int n){
      return m*n/gcd(m,n);
    }
        int gcd              // retu
          (int m, int n)     // para
        {                    // begi
          while(m % n != 0){
            int Remainder = m % n;
            m = n;
            n = Remainder;
          }
          return n;
        }                    // end o
```

nest ***verb*** **(FIT INSIDE)**

[I or T]

to fit one object inside another, or to fit inside in this way:

- *nested coffee tables*
- *dolls that nest inside one another*

Functions

- Specification: GCD example
  - "GCD(m,n) returns the greatest common divisor of positive integers m and n"
  - Responsibility of calling program/function
    - Must supply positive integers (pre-condition of function)
      - Can code-up a check on arguments being passed, just before calling
  - Responsibility of called program/function
    - Must return a positive integer (post-condition of function)
      - Can code-up a check on return expression
  - On the safe side:
    - Called function can also check for validity of arguments, before starting processing
    - Calling program/function can also check for validity of return value, before continuing

Functions

- Specification: GCD example
  - Write down specification, within comments, when defining function
  - Specification is more about "what" function does, and less about "how"

```
int gcd(int L, int S)
// Function for computing the greatest common divisor of integers L, S.
// PRE-CONDITION: L, S > 0
{
...
}
```

  - How the function computes the GCD is also important; comment in body
  - Why the code works is also important; also comment in body

```
// Note the theorem: If n divides m, then GCD(m,n) = n.
//                   If n does not divide m, then GCD(m,n) = GCD(n, m mod n)
```

Functions

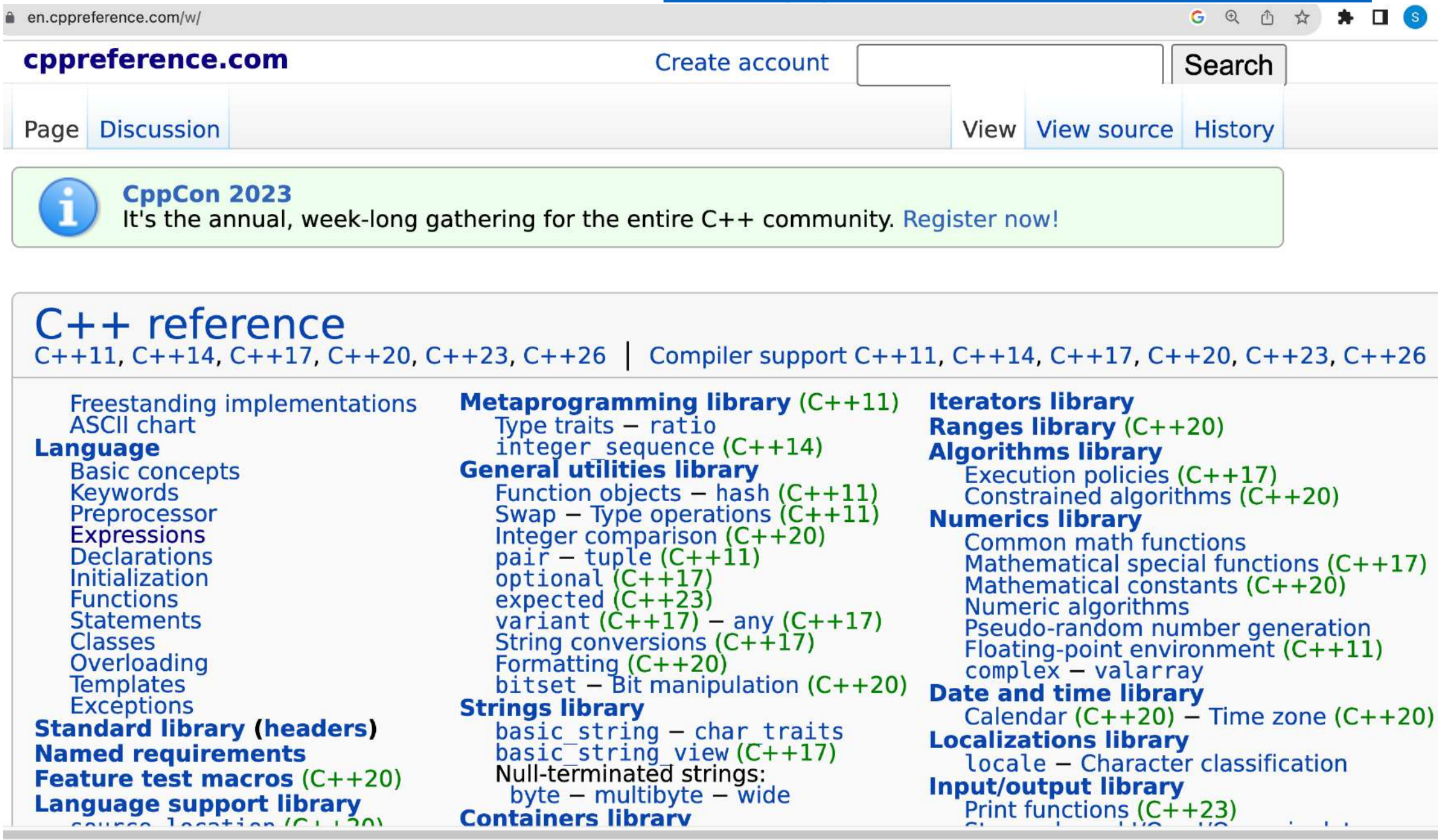
- Functions needn't return a value: return type of "void"
  - e.g., forward() function doesn't return a value
  - e.g., a function to draw a n-sided regular polygon, returning nothing

```
void polygon(int nsides, double sidelength)
// draws polygon with specified sides and specified sidelength.
// PRE-CONDITION: The pen must be down, and the turtle must be
// positioned at a vertex of the polygon, pointing in the clockwise
// direction along an edge.
// POST-CONDITION: At the end the turtle is in the same position and
// orientation as at the start.  The pen is down.
{
  for(int i=0; i<nsides; i++){
    forward(sidelength);
    right(360.0/nsides);
  }
  return;
}
```

Functions

- Main program is a function !
  - SimpleCPP camouflaged it
  - Actually when you write “main_program”,
    SimpleCPP internally replaces it by “int main ()”,
    and then passes that to C++ compiler
  - Operating system (OS) expects a function called “main” in compiled program
  - When you run program (a.out),
    OS first transfers control to main() function
  - When program ends, main() returns an integer code to OS as feedback
    - Code typically conveys if program ran properly or not (e.g., what type of error, etc.)
  - C++ allows main() without a return statement
    - 0 returned by default

C++ Resources on WWW: en.cppreference.com/w/



Functions

- Some challenges (for now)
  - Function cannot return more than 1 value
  - Suppose we write a function to swap values of 2 variables as follows:

```
void swap(int a, int b){  // will it work?
   int temp;
   temp = a;
   a = b;
   b = temp;
}
```

  - Suppose we intend to write a function read_marks_into() to update nextmark AND return bool indicating stopping criterion

```
int main(){
   double nextmark, sum=0;
   int count=0;

   while(read_marks_into(nextmark)){  // will this work?
      sum = sum + nextmark;
      count = count + 1;
   }

   cout << ‘‘The average is: ‘‘ << sum/count << endl;
}
```

Functions

- Function execution: "call by reference", "pass by reference"
  - When you want changes to a function parameter to be reflected in the corresponding variable in the calling program/function
  - Use ampersand symbol (&) prefixed to variable name in function definition

```
void Cartesian_To_Polar(double x, double y, double &r, double &theta){
  r = sqrt(x*x + y*y);
  theta = atan2(y,x);
}
```

```
int main(){
  double x1=1.0, y1=1.0, r1, theta1;
  Cartesian_To_Polar(x1,y1,r1,theta1);
  cout << r1 << ' ' << theta1 << endl;
}
```

    - What happens to values of variables x1, y1, r1, theta1 in main program before, during, after function call ?
  - Within function's activation frame, for variables passed by reference, new memory-storage area isn't allocated, but their memory region in calling program/function is used
    - No need for copying such variable's values from calling to called function

Functions

- Function execution: “call by reference”
  - Example for swapping numbers

```
void swap2(int &a, int &b){
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

```
int main{
  int x=5, y=6;
  swap2(x,y);
  cout << x << " " << y << endl;
}
```

  - Example for reading marks

```
bool read_marks_into(int &var){
  cin >> var;
  return var != 200;
}
```

```
int main(){
  double nextmark, sum=0;
  int count=0;

  while(read_marks_into(nextmark)){  // will this work?
    sum = sum + nextmark;
    count = count + 1;
  }

  cout << ‘‘The average is: ‘‘ << sum/count << endl;
}
```

Functions

- "Reference variables"
  - Variables passed by reference are, essentially, renamed
  - "Call by reference" also called as "call by name"
  - We can define aliases even outside context of function parameters
  - Example
    - "r" becomes a new name for "x"
    - They share memory and values
    - Changes to any one imply changes to other
    - "int &" means that variable name is going to be an alias of another variable, and that association/renaming must be defined during the declaration of the alias itself

```
int x = 10;
int &r = x;
cout << r << endl;
r = 20;
cout << x << endl;
```

  - What is the output of the code ?
    - 10
      20

Functions

- Call/pass by reference/name
- Call/pass by value

pass by reference

cup =

fillCup( )

pass by value

cup =

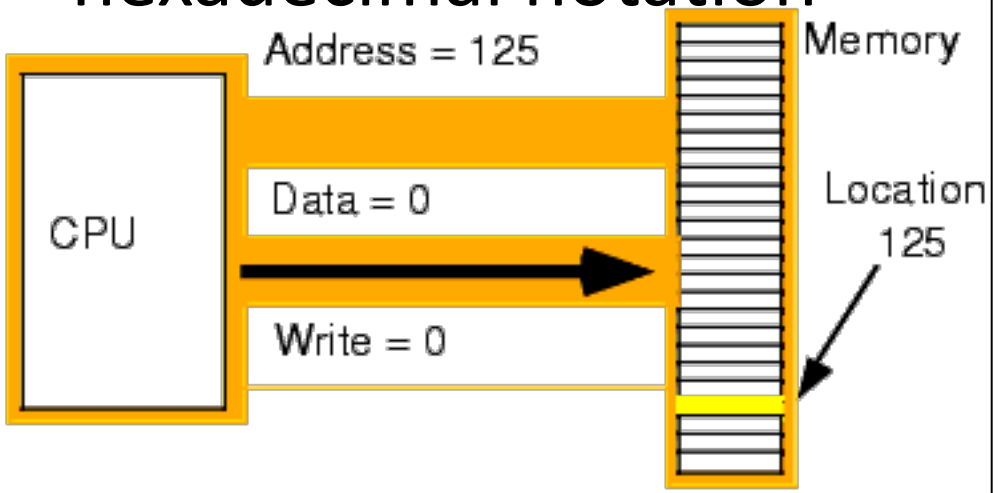
fillCup( )

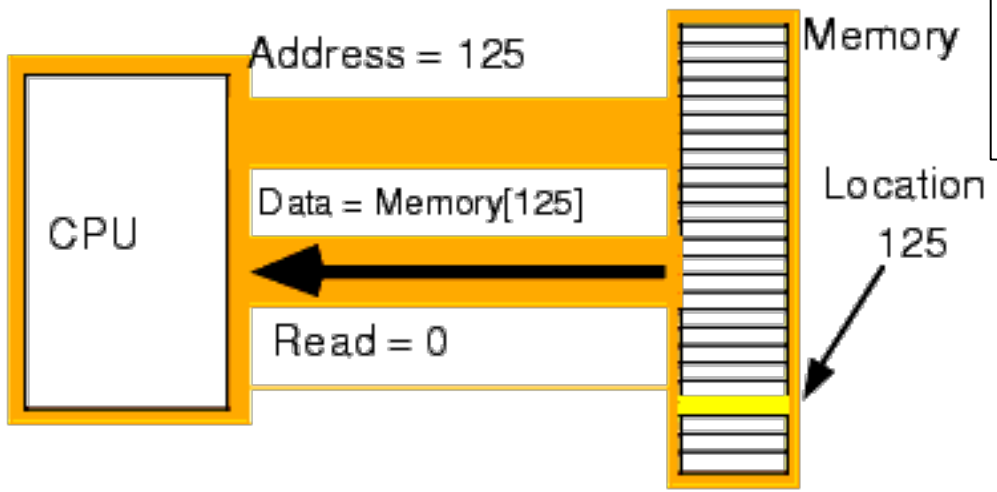
Functions

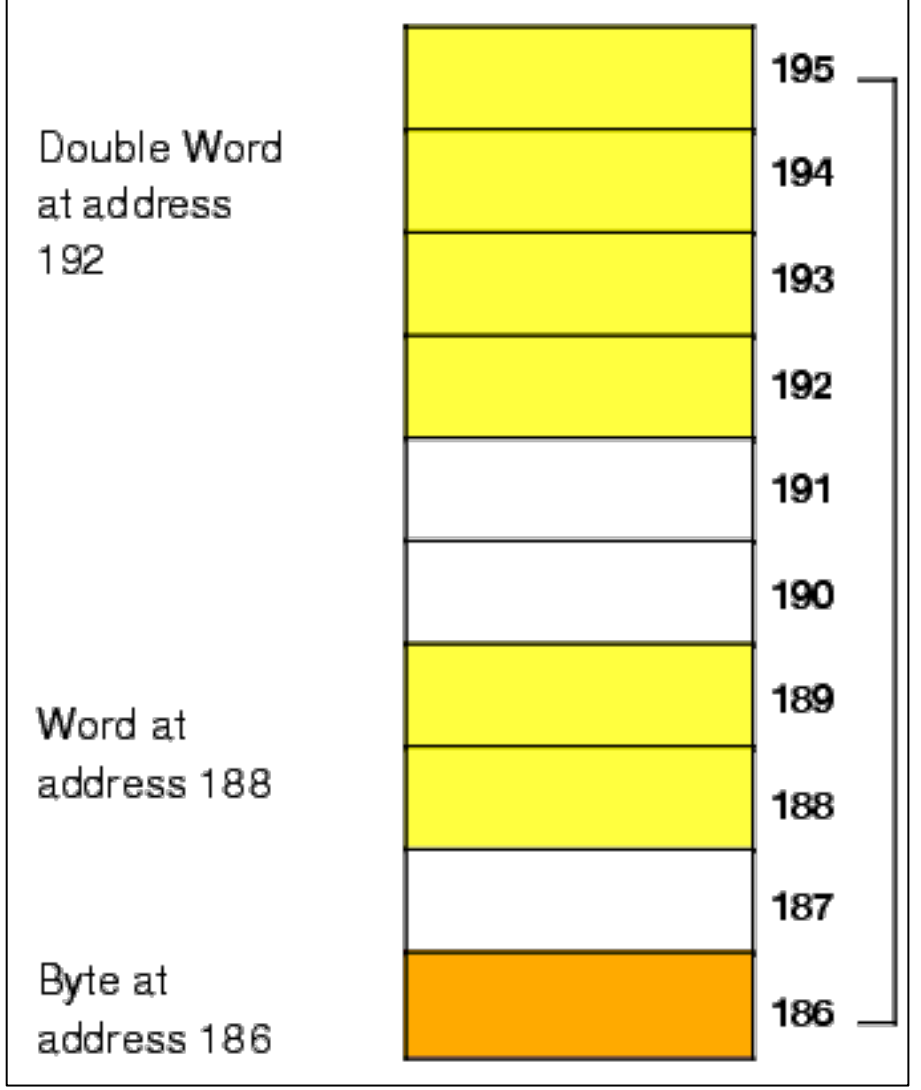
- Memory is organized into bytes (smallest unit in typical hardware)
  - Each byte has a location/address
  - Each byte may also store some data; its associated value
- When memory is allocated to a variable, it is as contiguous bytes
  - e.g., bool and char variables get 1 byte, float variable gets 4 consecutive bytes
- "Address of a variable" = address of memory location of its first byte
- C++ provides an operator to get address of a variable: ampersand (&)
  - Don't confuse this operator & from the & used to create a reference variable (e.g., "int &")
  - Unary operator; signifies "address of"
- Example:
  - "int i; cout << &i;" prints the address associated with variable i

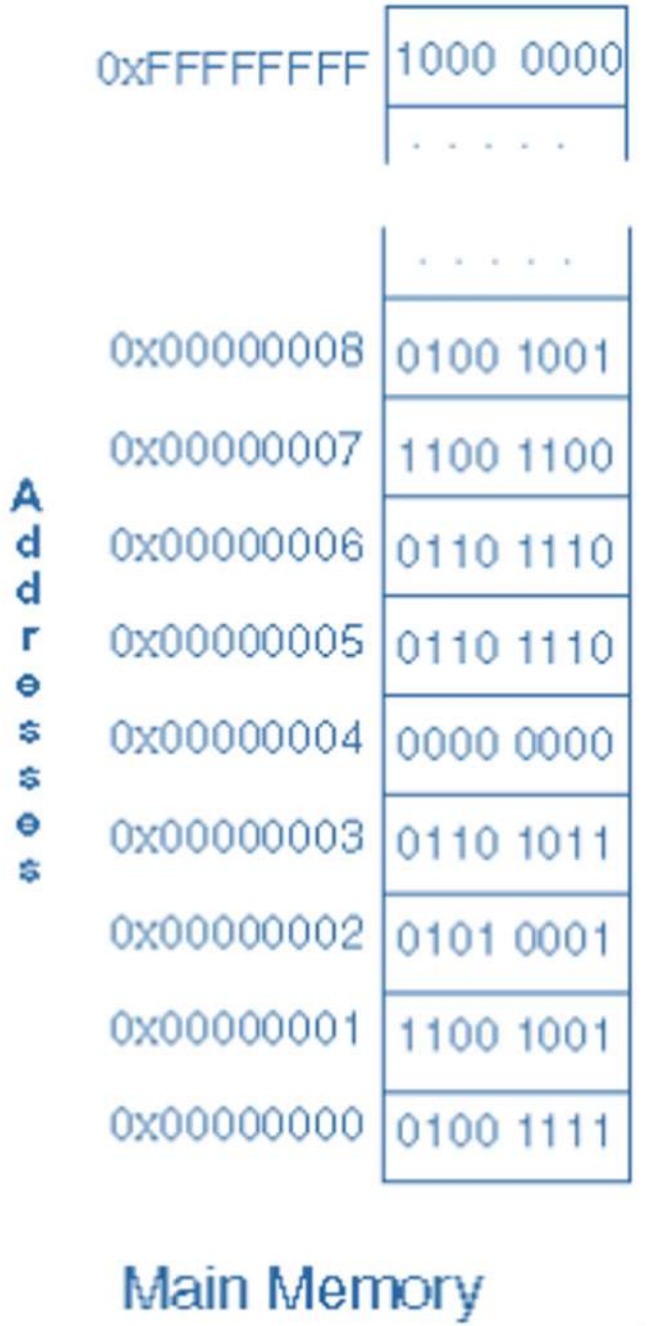
Functions

- Memory addresses often given in hexadecimal notation



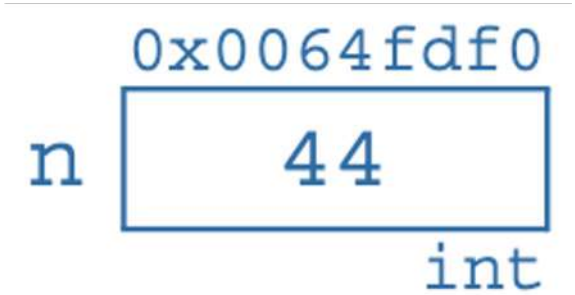






Functions

- Visualizing the representation of a variable, e.g., "int n = 44;"
  - Variable name
  - Variable type
  - Variable address
  - Variable value




```
int main()
{ int n=44;
  cout << "n = " << n << endl;     // prints the value of n
  cout << "&n = " << &n << endl;   // prints the address of n
}
```

```
n = 44
&n = 0x0064fdf0
```

Functions

- Using references

This declares rn as a reference to n:

```
int main()
{  int n=44;
   int& rn=n;  // r is a synonym for n
   cout << "n = " << n << ", rn = " << rn << endl;
   --n;
   cout << "n = " << n << ", rn = " << rn << endl;
   rn *= 2;
   cout << "n = " << n << ", rn = " << rn << endl;
}
```

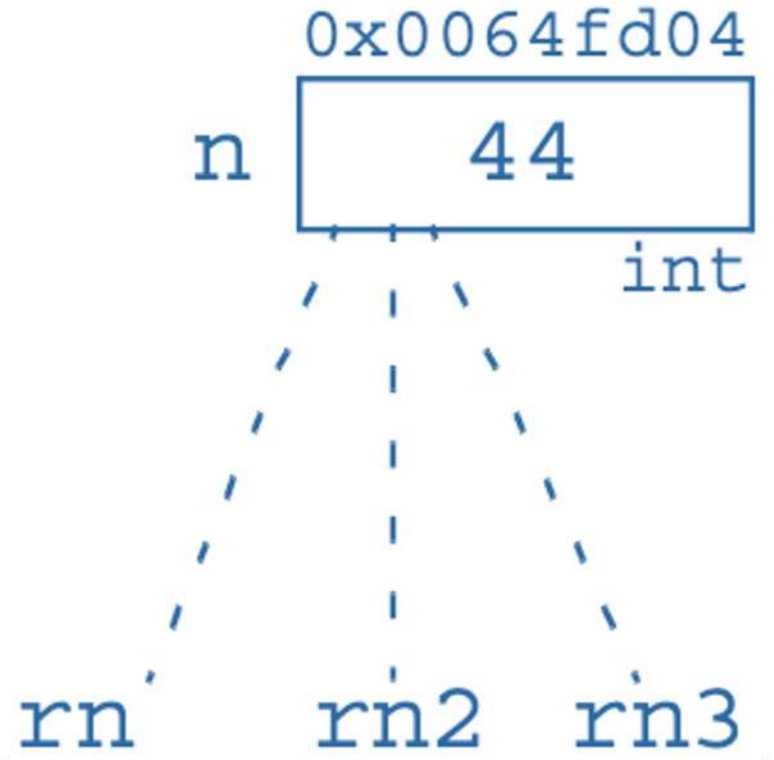
```
n = 44, rn = 44
n = 43, rn = 43
n = 86, rn = 86
```

Like constants, references must be initialized when they are declared. But unlike a constant, a reference must be initialized to a variable, not a literal:

```
int& rn=44;  // ERROR: 44 is not a variable!
```

Functions

- References aren't separate variables
  - "A reference to a a reference is the same as a reference to the object to which it refers" ☺
    [From Schaum's book on Programming in C++]




```
int main()
{  int n=44;
   int& rn=n;   //    is a synonym for n
   cout << "   &n = " << &n << ",   &rn = " << &rn << endl;
   int& rn2=n;    //    is another synonym for n
   int& rn3=rn;   //    is another synonym for n
   cout << "&rn2 = " << &rn2 << ", &rn3 = " << &rn3 << endl;
}
```

```
   &n = 0x0064fde4,   &rn = 0x0064fde4
&rn2 = 0x0064fde4, &rn3 = 0x0064fde4
```

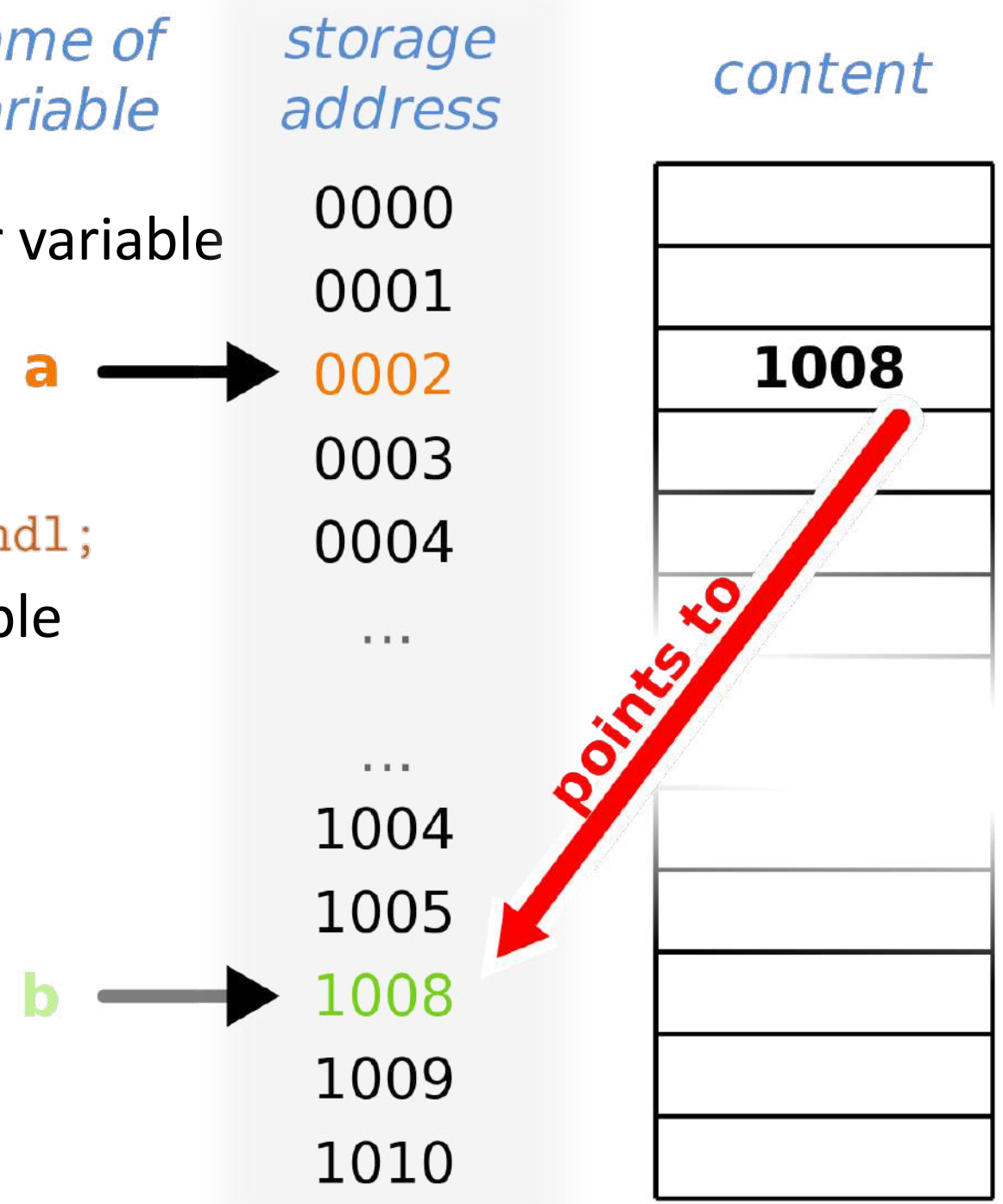
Functions

- "Pointers"
  - A variable that stores address of another variable
  - Example

```
int p=15;
int *r;
r = &p;
cout << &p << " " << r << endl;
```

  - Data type "T *" indicates a pointer variable (of derived type T *) that will store an address of variable of type T

Address	Content	Remarks
104 105 106 107	15	Allocated to p
108 109 110 111	104	Allocated to r



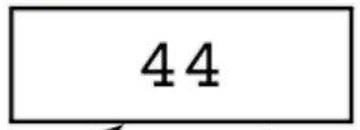
Functions

- Pointer dogs
  - Used in hunting
  - Point towards game

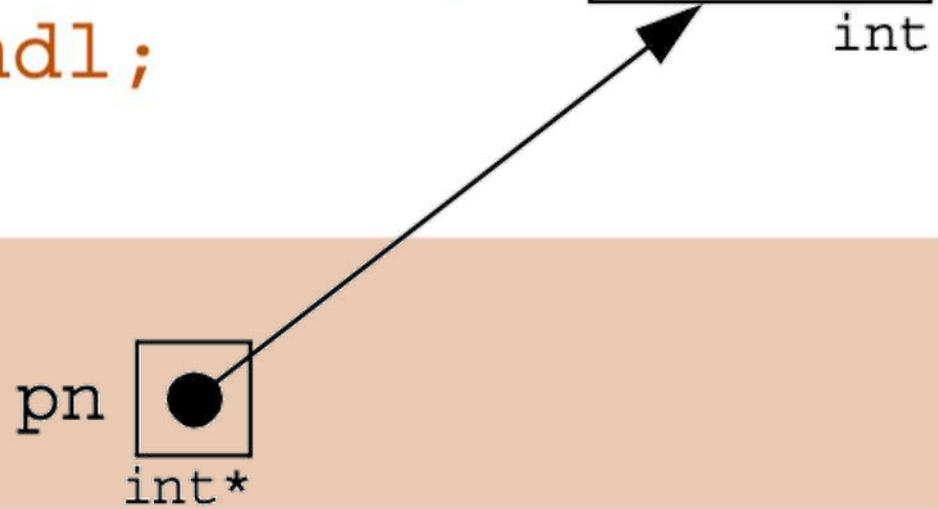
Functions

- Visualizing the representation of a pointer variable


```
int main()
{  int n=44;
   cout << "n = " << n << ", &n = " << &n << endl;
   int* pn=&n;   // pn holds the address of n
   cout << "          pn = " << pn << endl;
   cout << "&pn = " << &pn << endl;
}
```




```
n = 44,  &n = 0x0064fddc
         pn = 0x0064fddc
&pn = 0x0064fde0
```



Functions

- Pointers
  - Equivalent definitions

```
int* p;     // indicates that p has type int* (pointer to int)
int * p;    // style sometimes used for clarity
int *p;     // old C style
```

  - Be careful: “int * p, q;” declares p as pointer and q as integer
- “Dereferencing” pointers
  - “reference” applied to variable gives “address” of variable, “dereference” applied to address gives back (value of) variable
  - Inverse of unary & operator is unary * operator
  - Just as operator & refers to “address of” variable, operator * refers to “variable at” address
  - Don’t confuse dereferencing * operator with “int *” declaration or multiplication operator *

Functions

- Dereferencing a pointer

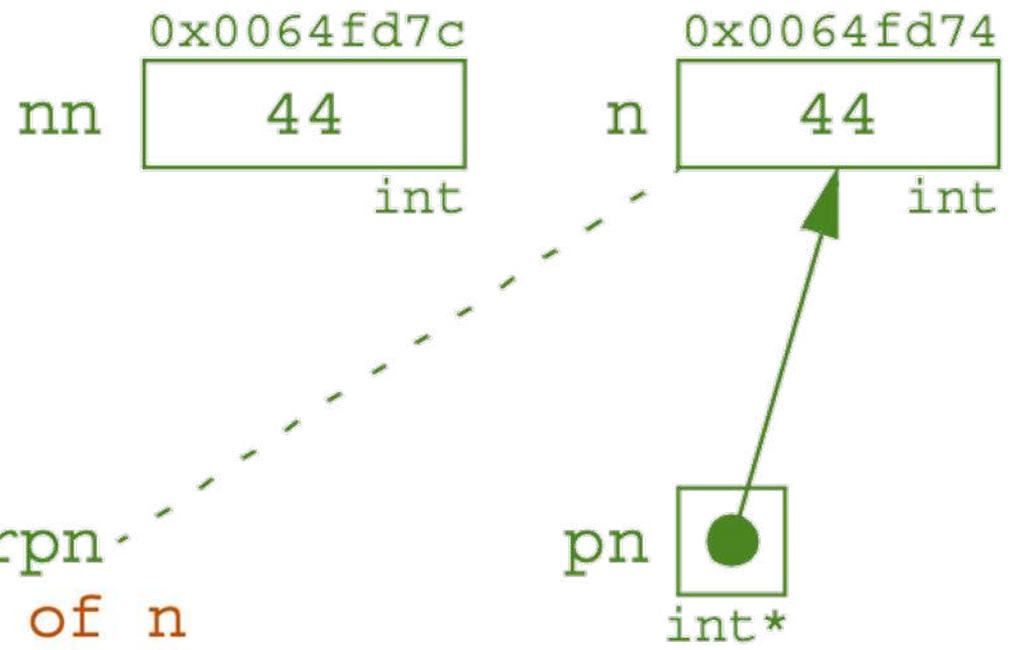
```
int main()
{  int n=44;
   cout << "n = " << n << ", &n = " << &n << endl;
   int* pn=&n;   // pn holds the address of n
   cout << "          pn = " << pn << endl;
   cout << "&pn = " << &pn << endl;
   cout << "*pn = " << *pn << endl;
}
```

```
n = 44, &n = 0x0064fdcc
        pn = 0x0064fdcc
&pn = 0x0064fdd0
*pn = 44
```

This shows that `*pn` is an alias for `n`: they both have the value 44

Functions

- Referencing is Opposite of Dereferencing




```
int main()
{ int n=44;
  cout << "     n = " << n << endl;
  cout << "    &n = " << &n << endl;
  int* pn=&n;  // pn holds the address of n
  cout << "    pn = " << pn << endl;
  cout << "   &pn = " << &pn << endl;
  cout << "   *pn = " << *pn << endl;
  int nn=*pn;  // nn  is a duplicate of n
  cout << "    nn = " << nn << endl;
  cout << "   &nn = " << &nn << endl;
  int& rpn=*pn;  // rpn is a reference for n
  cout << "   rpn = " << rpn << endl;
  cout << " &rpn = " << &rpn << endl;
}
```

```
     n = 44
    &n = 0x0064fd74
    pn = 0x0064fd74
   &pn = 0x0064fd78
   *pn = 44
    nn = 44
   &nn = 0x0064fd7c
   rpn = 44
  &rpn = 0x0064fd74
```

Functions

- "Dereferencing" pointers
  - Example: old
  - Example: new

```
void Cartesian_To_Polar(double x, double y, double &r, double &theta){
  r = sqrt(x*x + y*y);
  theta = atan2(y,x);
}
```

```
int main(){
  double x1=1.0, y1=1.0, r1, theta1;
  Cartesian_To_Polar(x1,y1,r1,theta1);
  cout << r1 << ' ' << theta1 << endl;
}
```

```
void CartesianToPolar(double x, double y, double* pr, double* ptheta){
  *pr = sqrt(x*x + y*y);
  *ptheta = atan2(y,x);
}
```

```
int main{
  double r,theta;
  CartesianToPolar(1.0, 1.0, &r, &theta);
  cout << r << ‘‘ ‘‘ << theta << endl;
}
```

- No parameters passed by reference; all parameters copied, i.e., value 1 into x, value 1 into y, address of r into pr, address of theta into ptheta
  - This style of calling a function called "pass by address"

Functions

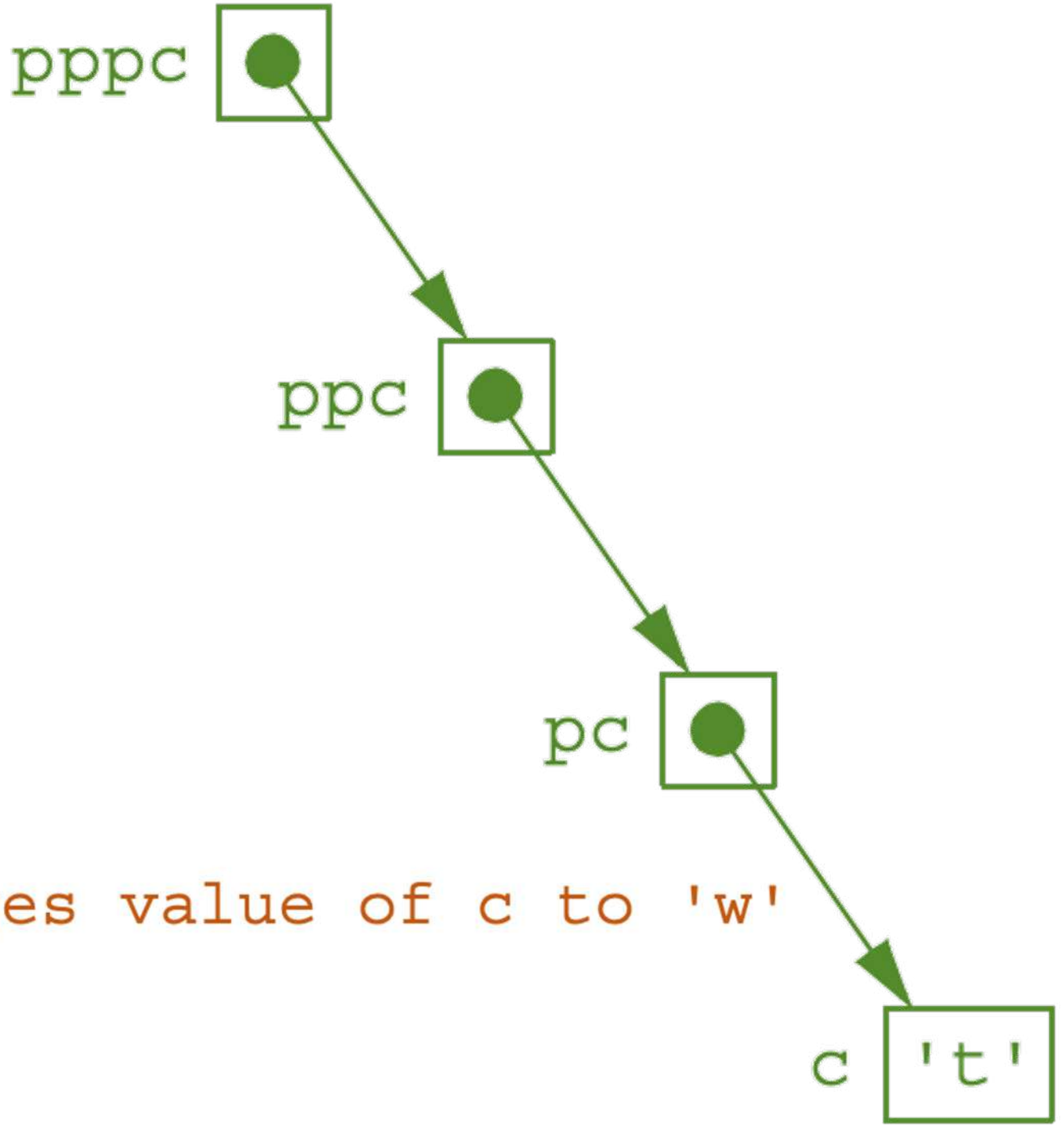
- "Dereferencing" pointers
  - Example

```
void swap(int* pa, int* pb){
   int temp;
   temp = *pa;
   *pa = *pb;
   *pb = temp;
}

int main{
   int x=5, y=6;
   swap(&x,&y);
   cout << x << ‘‘ ‘‘<< y << endl;
}
```

Functions

- Pointers to pointers
  - [From Schaum's book on Programming in C++]




```
char c = 't';
char* pc = &c;
char** ppc = &pc;
char*** pppc = &ppc;
***pppc = 'w';  // changes value of c to 'w'
```

Functions

- Constant pointer. Pointer to a constant.
  - [From Schaum's book on Programming in C++]

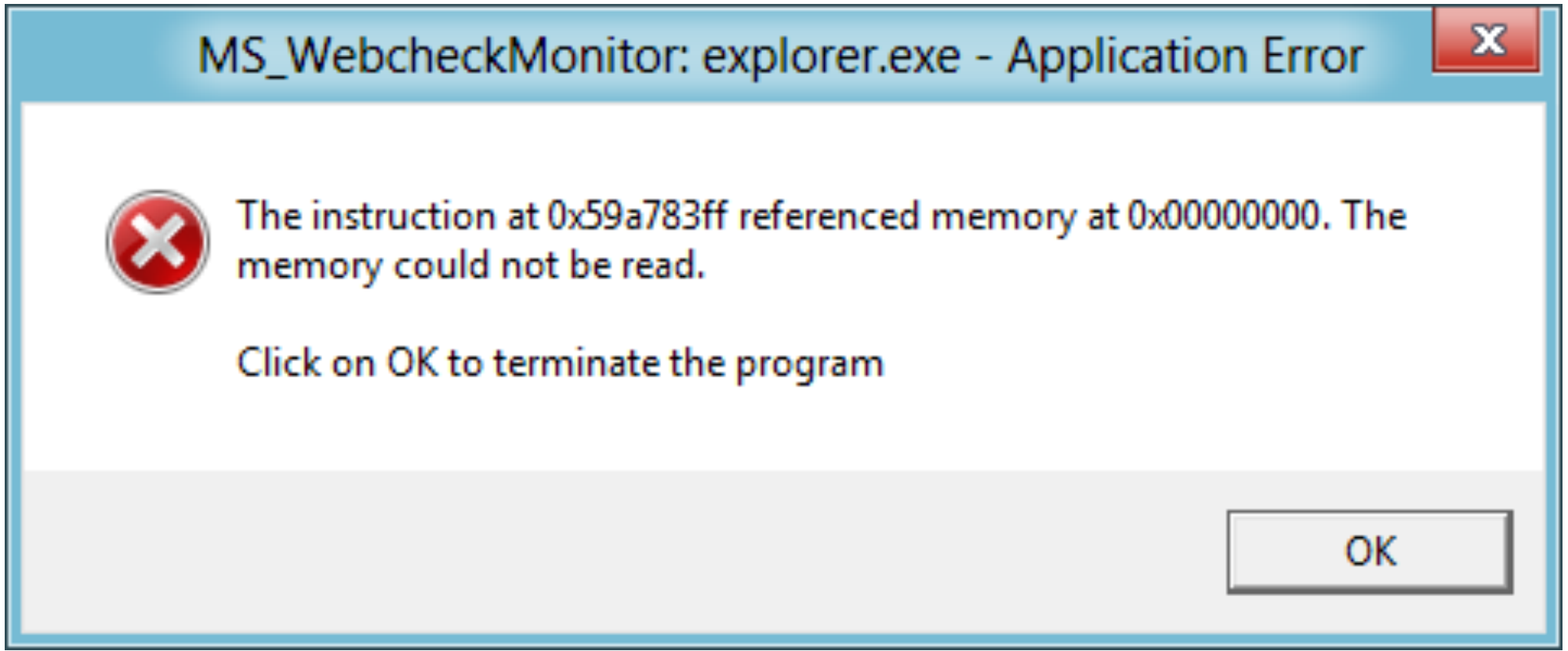
```
int n = 44;                          // an int
int* p = &n;                         // a pointer to an int
++(*p);                              // ok: increments int *p
++p;                                 // ok: increments pointer p
int* const cp = &n;                  // a const pointer to an int
++(*cp);                             // ok: increments int *cp
++cp;                                // illegal: pointer cp is const
const int k = 88;                    // a const int
const int * pc = &k;                 // a pointer to a const int
++(*pc);                             // illegal: int *pc is const
++pc;                                // ok: increments pointer pc
const int* const cpc = &k;           // a const pointer to a const int
++(*cpc);                            // illegal: int *cpc is const
++cpc;                               // illegal: pointer cpc is const
```

Functions

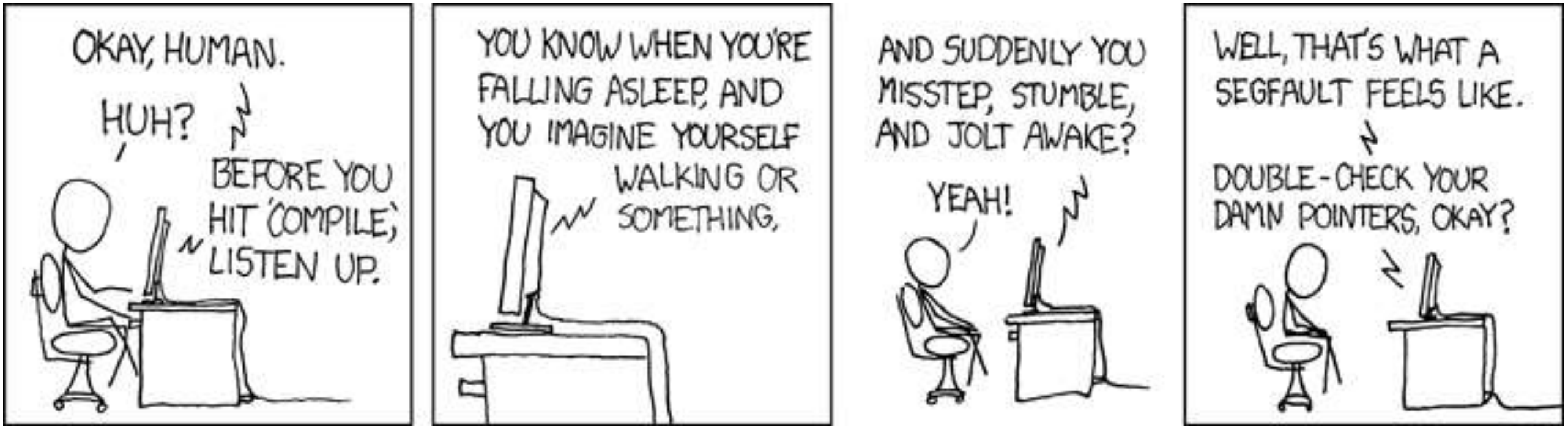
- Call-by-reference (passing aliases) versus call-by-address (passing pointers)
  - Call-by-reference implicitly (behind the scenes):
    - Passes addresses of variables to function
    - Performs dereferencing inside the function
  - Passing addresses and dereferencing pointers requires utmost care lest the addresses aren't accessible by program (e.g., address of zero= NULL) leading to a program error
    - Such an error is called a "segmentation fault"
    - en.wikipedia.org/wiki/Segmentation_fault
    - "A segmentation fault (often shortened to segfault) or access violation is a fault, or failure condition, raised by hardware with memory protection, notifying an operating system
      the software has attempted to access a restricted area of memory (a memory access violation)"

Functions

- Segfault
  - en.wikipedia.org/wiki/Segmentation_fault
  - “Common class of error in programs written in languages like C that provide low-level memory access and few to no safety checks”
  - A null pointer dereference on Windows 8



Functions


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE', LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF WALKING OR SOMETHING.
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Practice Examples for Lab: Set 8

- 1

The k-norm of a vector (x, y, z) is defined as $\sqrt[k]{x^k + y^k + z^k}$. Note that the 2-norm is in fact the Euclidean length. Indeed, the most commonly used norm happens to be the 2 norm. Write a function to calculate the norm such that it can take k as well as the vector components as arguments. You should also allow the call to omit k, in which case the 2 norm should be returned.

- 2

Write a function to find the cube root of a number using Newton's method. Accept the number of iterations as an argument.

- 3

Modify the function `polygon` so that it returns the perimeter of the polygon drawn (in addition to drawing the polygon).

- 4

Write the function `read_marks_into` and the main program for mark averaging using pointers.

Practice Examples for Lab: Set 8

- 5

A key rule in assignment statements is that the type of the value being assigned must match the type of the variable to which the assignment is made. Consider the following code:

```
int *x,*y, z=3      ;


y = &x;
z = y;
y = *x;
```

Each of the assignments is incorrect. Can you guess why? If not, write the code in a program, compile it, and the compiler will tell you!

Functions

- Functions can return a reference
  - Return type will be “Type &”

```
int &f(int &x, int &y){
  if(x > y) return x;
  else return y;
}
main_program{
  int p=5, q=4;
  f(p,q) = 2;                    // function call appears on the left!
  cout << p <<' '<< q << endl;
  cout << f(p,q) << endl;
}
```

  - Doesn’t make the code more readable
  - Is dangerous if returning a reference to a local variable

```
double &h(){
  double x = 5;
  return x;
}
```

Functions

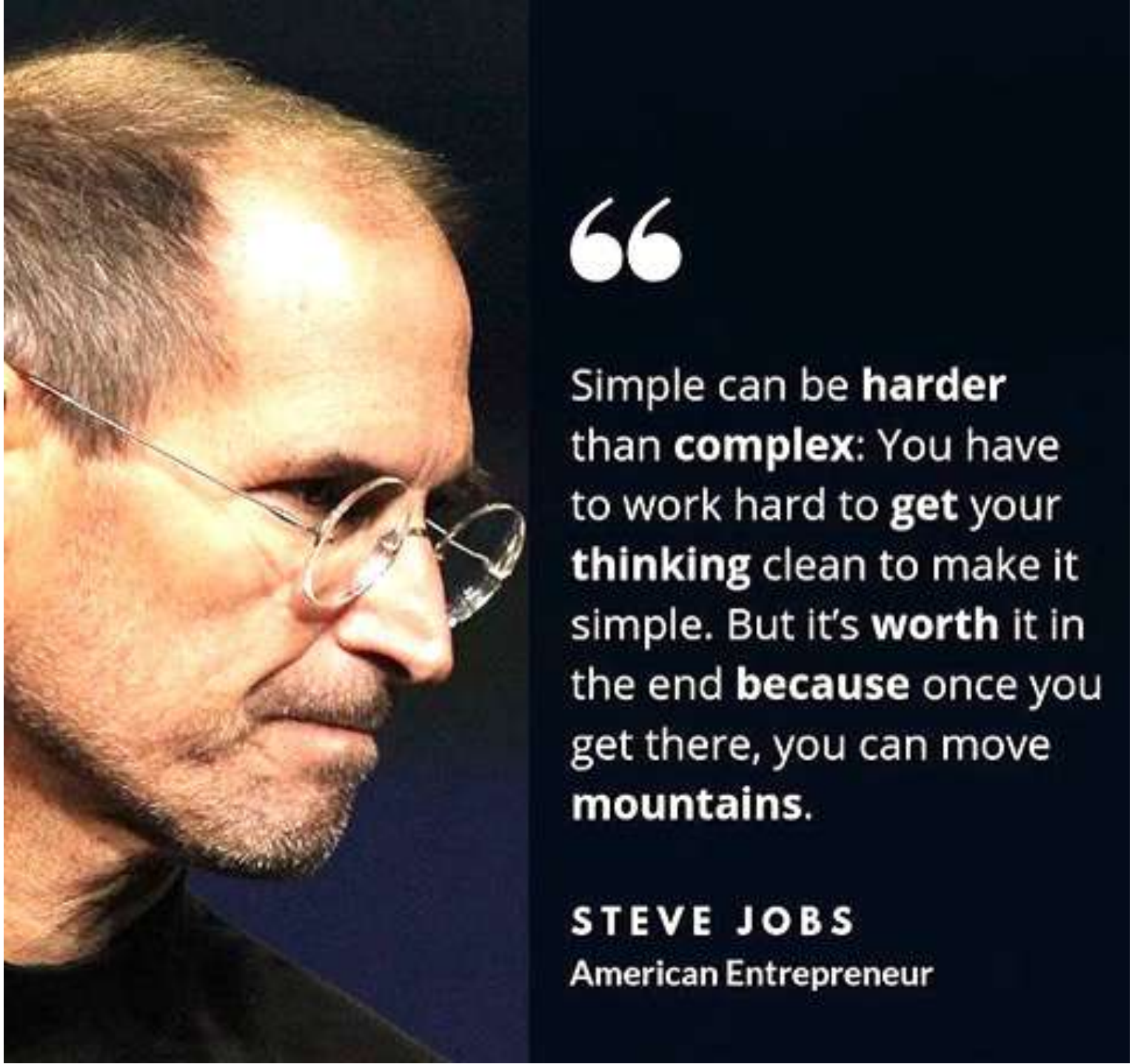
- Functions can return a pointer
- Stay away from such theatrics
- Doesn't usually help readability
- Can often be dangerous

```
double *h(){
   double x = 5;
   return &x;
}
int main(){
   *h() = 7;
}
```

```
int *f(int *x, int *y){
   if(*x > *y) return x;
   else return y;
}
int main(){
   int p=5, q=4;
   *f(&p,&q) = 2;
   cout << p <<' '<< q << endl;
   cout << *f(&p,&q) << endl;
}
```

KISS principle

- KISS = acronym for "Keep it simple, stupid!”
- en.wikipedia.org/wiki/KISS_principle
- Design principle noted by U.S. Navy in 1960

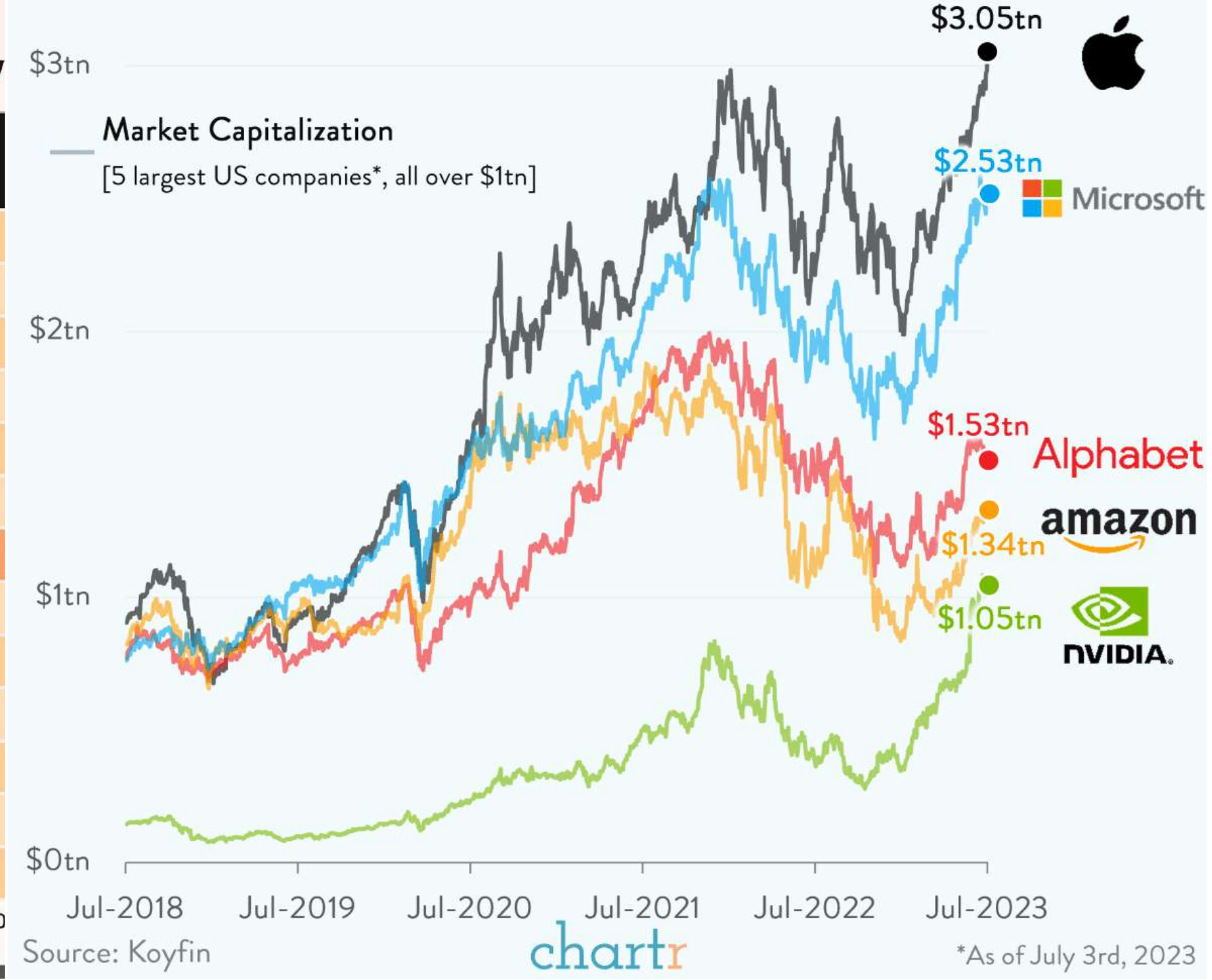


Market & Economy

Country	Current Mcap	FY21 GDP
World	**1,16,073**	84,680
United States	**50,536**	20,894
China	**11,812**	14,723
Japan	**6,445**	5,058
Hong Kong	**6,030**	347
UK	**3,519**	2,764
India	**3,524**	**2,660**
France	**3,334**	2,630
Canada	**3,166**	1,644
Saudi Arabia	**2,819**	700
Germany	**2,663**	3,846
Switzerland	**2,247**	752
South Korea	**1,988**	1,638

(Figures in $ bn)

So



Functions

- Default values of parameters
  - Can assign default values to any suffix (last K elements) of parameter list
    - Compiler needs to know which arguments correspond to which parameters
  - In that case, when calling a function, some arguments (last K in the list of arguments) may be omitted
  - Example
    - Valid calls:
      polygon(5, 50)
      polygon(5)
      polygon()
  - When designing such a function, keep parameters with default values as the last few in the parameter list

```
void polygon(int nsides=4, double sidelength=100)
{
  for(int i=0; i<nsides; i++){
    forward(sidelength);
    right(360.0/nsides);
  }
  return;
}
```

Functions

- Function "overloading"
  - Multiple functions with same name but with different parameter-type lists
  - This set of functions perform similar functionality (called by same name) but with different type of parameters, or different number of parameters, etc.

```
int gcd(int p, int q, int r){
  return gcd(gcd(p,q),r);
}

int gcd(int p, int q, int r, int s){
  return gcd(gcd(p,q),gcd(r,s));
}
```

```
int Abs(int x){
  if (x>0) return x;
  else return -x;
}

double Abs(double x){
  if (x>0) return x;
  else return -x;
}
```

  - How does compiler know which function to call ?
    - By number of parameters
    - By type of argument passed

Functions

- "Templates"
  - During function overloading, if functions have identical bodies, then why not define them just once via a generalizable scheme/"template"
  - Example

```
template<typename T>
T Abs(T x){
  if (x>0) return x;
  else return -x;
}
```

```
int main(){
  int x=3;
  float y=-4.6;

  cout << Abs(x) << endl;
  cout << Abs(y) << endl;
}
```

    - T = template variable
      - Can be any other name, other than "T"
    - Function for handling a specific type T is instantiated when compiler encounters a function call with that type T

Recursive Functions

- Recursion
  - Defining a concept/process relying on a simpler version of that same concept/process
  - Example
    - Factorial defined recursively, using a recurrence relation:
      for positive integer n, we define n! = n * (n-1)!
      - Computing (n-1)! "simpler" than n!
      - If recursion goes on infinitely long, it isn't useful in practice: e.g., GNU = "GNU's Not Unix"
    - Factorial base case: we define 0! = 1 (or 1! = 1)
      - Simplest case
  - A class of objects or methods exhibits recursive behavior when it can be defined by two properties:
    - Simple base case(s): terminal cases that don't use recursion to produce an answer
    - Recursive step — a set of rules that reduces all successive cases toward the base case

Recursive Functions

$$\gcd(x, y) = \begin{cases} x & \text{if } y = 0 \\ \gcd(y, \text{remainder}(x, y)) & \text{if } y > 0 \end{cases}$$

- GCD of natural numbers
  - Recurrence relation

```
function gcd is:
input: integer x, integer y such that x > 0 and y >= 0


    1. if y is 0, return x
    2. otherwise, return [ gcd( y, (remainder of x/y) ) ]


end gcd
```

Computing the recurrence relation for x = 111 and y = 259:

```
gcd(111, 259)   = gcd(259, 111 % 259)
                = gcd(259, 111)
                = gcd(111, 259 % 111)
                = gcd(111, 37)
                = gcd(37, 111 % 37)
                = gcd(37, 0)
                = 37
```

Recursive Functions

- GCD definition based on recurrence relation

Theorem 2 (Euclid) *Let m, n be positive integers. If $m\%n = 0$ then $GCD(m, n) = n$. If $m\%n \neq 0$ then $GCD(m, n) = GCD(n, m\%n)$.*

  - Motivates calling GCD(.) within body of GCD(.) itself

```
int gcd(int m, int n)
// finds GCD(m,n) for positive integers m,n
{
  if(m % n == 0) return n;
  else return gcd(n,m % n);
}
```

  - How would the following code execute ?
    - int main() { cout << gcd (36,24) << endl; }

Recursive Functions

- GCD
  - Number of times recursive calls were made = “depth of recursion”

Function call	main	gcd(36,24)	gcd(24,12)
Activation Frame contents		m : 36 n : 24	m : 24 n : 12
State of call	Suspended	Suspended	Executing
Code with ▷ showing next statement to be executed	cout << ▷ gcd(36,24) << endl;	if(m % n == 0) return n; else return ▷ gcd(n, m % n);	▷ if(m % n == 0) return n; else return gcd(n, m % n);

Recursive Functions

- Factorial of a non-negative int

$$\text{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \text{fact}(n-1) & \text{if } n > 0 \end{cases}$$

  - Defining as recurrence relation:
    $f_n = n\, f_{n-1}$; $f_0 = 1$

Computing the recurrence relation for n = 4:

$$\begin{aligned} b_4 &= 4 \times b_3 \\ &= 4 \times (3 \times b_2) \\ &= 4 \times (3 \times (2 \times b_1)) \\ &= 4 \times (3 \times (2 \times (1 \times b_0))) \\ &= 4 \times (3 \times (2 \times (1 \times 1))) \\ &= 4 \times (3 \times (2 \times 1)) \\ &= 4 \times (3 \times 2) \\ &= 4 \times 6 \\ &= 24 \end{aligned}$$

Pseudocode (recursive):

```
function factorial is:

input: integer n such that n >= 0

output: [n × (n-1) × (n-2) × ... × 1]


    1. if n is 0, return 1
    2. otherwise, return [ n × factorial(n-1) ]


end factorial
```

Recursive Functi

- Factorial computation can be converted into a loop
  - True for any recursive function
- Recursive versus looping
  - Trade-off between readability and cost of space/ computation

Pseudocode (iterative):

```
function factorial is:

input: integer n such that n >= 0

output: [n × (n-1) × (n-2) × ... × 1]


    1. create new variable called running_total with a value of 1


    2. begin loop
           1. if n is 0, exit loop
           2. set running_total to (running_total × n)
           3. decrement n
           4. repeat loop


    3. return running_total


end factorial
```

Recursive Functions

- Tail-recursive functions
  - Functions in which all recursive calls are "tail" calls (last statement in function)
  - They don't build up any deferred operations; program is close-to iterative
  - IF a compiler treats tail-recursive calls as jumps rather than function calls, then a tail-recursive function will execute using constant space

Tail recursion:

```
//INPUT: Integers x, y such that x >= y
and y >= 0
int gcd(int x, int y)
{
  if (y == 0)
     return x;
  else
     return gcd(y, x % y);
}
```

Augmenting recursion:

```
//INPUT: n is an Integer such
that n >= 0
int fact(int n)
{
    if (n == 0)
       return 1;
    else
       return n * fact(n - 1);
}
```

Recursive Functions

- Building towers with bricks
  - You have 2 kinds of bricks: those of height 1 unit and those of height 2 units
  - In how many ways (say, V_n) can we build a tower of height n ?
    - Example: n = 4 gives 5 ways: 1111 / 112 / 121 / 211 / 22

$$V_n = \text{Number of ways of building a tower of height } n = \text{Number of ways of building a tower of height } n \text{ with bottom-most brick of height 1} + \text{Number of ways of building a tower of height } n \text{ with bottom-most brick of height 2.}$$

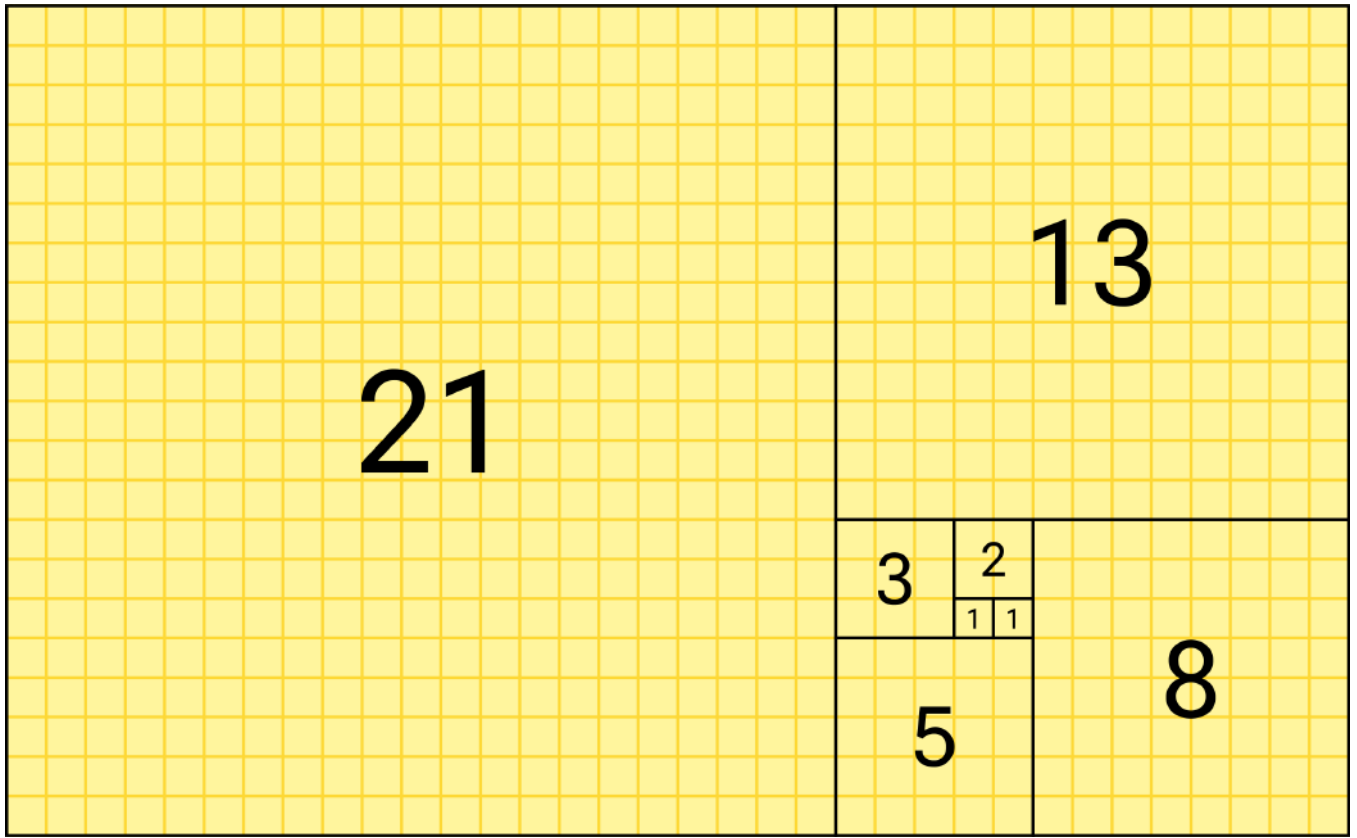
$$\text{Number of ways of building a tower of height } n \text{ with bottom-most brick of size 1} = \text{Number of ways of building a tower of height } n-1 = V_{n-1}$$

$$\text{Number of ways of building a tower of height } n \text{ with bottom-most brick of height 2} = \text{Number of ways of building a tower of height } n-2 = V_{n-2}$$

  - Recurrence relation $V_n = V_{n-1} + V_{n-2}$
    - Proposed by Virahaanka (विरहाङ्क)

Recursive Functions

- Virahaanka (विरहाङ्क) en.wikipedia.org/wiki/Virahanka
  - Indian prosodist and mathematician (around 600 AD)
    - "The study of the metrical structure of verse"
  - Work builds on छंद सूत्र of Pingala (400-300 BC) en.wikipedia.org/wiki/Pingala
    - Presents first known description of a binary numeral system in connection with systematic enumeration of metres with fixed patterns of short (लघु) & long (गुरु) syllables
  - Fibonacci (Italian mathematician; 1200s) popularized Indo-Arabic numeral system in Western world primarily through his composition *Liber Abaci* (Book of Calculation)



Recursive Functions

- Virahaanka
  - www.sciencedirect.com/science/article/pii/0315086085900217
  - Rhythmic structure of a verse
    - How many short (लघु) & long (गुरु) syllabus can be fit in N beats
    - en.wikipedia.org/wiki/Sanskrit_prosody

ELSEVIER

Historia Mathematica

Volume 12, Issue 3, August 1985, Pages 229-244

The so-called fibonacci numbers in ancient and medieval India

Parmanand Singh

Department of Mathematics, Raj Narain College, Hajipur-844101, District-Vaishali Bihar, India

Available online 2 September 2004.

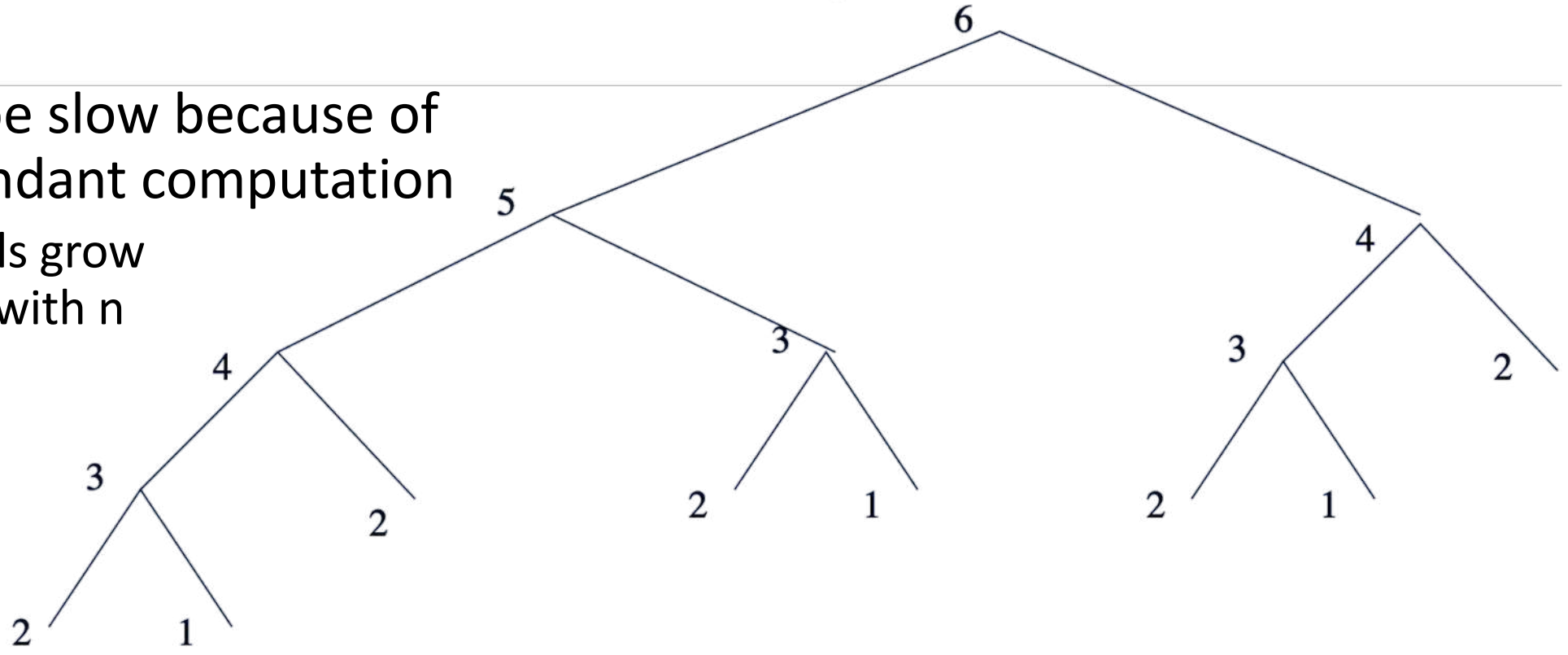
The basic units in Sanskrit prosody are a letter having a single *mātrā* (mora or a syllabic instant) called *laghu* (light) and that having two morae called *guru* (heavy). The former is denoted by | and the latter by *S*, and their role in metric is the same as that of 1 and 2 in combinatorics.

Recursive Functions

- Building Virahaanka towers

```
int Virahanka(int n){
  if(n == 1) return 1; // V_1
  if(n == 2) return 2; // V_2
  return Virahanka(n-1) + Virahanka(n-2); // V_{n-1} + V_{n-2}
}
```

  - Execution will be slow because of too much redundant computation
    - Number of calls grow exponentially with n



Recursive Functions

- Building Virahaanka towers
  - How slow is this function ?
  - For large n,
    when n increases by 1,
    time taken increases
    almost ~1.6 times

```
int Virahanka(int n){
  if(n == 1) return 1; // V_1
  if(n == 2) return 2; // V_2
  return Virahanka(n-1) + Virahanka(n-2);
}
```

```
Recursive computation and runtimes:


f(35) = 14930352. That took 42ms to complete.
f(36) = 24157817. That took 66ms to complete.
f(37) = 39088169. That took 79ms to complete.
f(38) = 63245986. That took 130ms to complete.
f(39) = 102334155. That took 207ms to complete.
f(40) = 165580141. That took 309ms to complete.
f(41) = 267914296. That took 503ms to complete.
f(42) = 433494437. That took 808ms to complete.
f(43) = 701408733. That took 1468ms to complete.
f(44) = 1134903170. That took 2136ms to complete.
f(45) = 1836311903. That took 3477ms to complete.
```

willrosenbaum.com/teaching/2021s-cosc-112/notes/recursive-fibonacci/

Recursive Functions

- Building Virahaanka towers
  - Counting number of calls
  - Writing another recursive function

```
int numberOfCalls (int n) {
    if (n == 1 || n == 2) {
        return 0;
    }

    return 2 + numberOfCalls(n - 1) + numberOfCalls(n - 2);
}
```

  - For large n, when n increases by 1, number of calls increase ~1.6 times

```
int Virahanka(int n){
  if(n == 1) return 1; // V_1
  if(n == 2) return 2; // V_2
  return Virahanka(n-1) + Virahanka(n-2);
}
```

```
fibonacci(1) requires 0 recursive function calls
fibonacci(2) requires 0 recursive function calls
fibonacci(3) requires 2 recursive function calls
fibonacci(4) requires 4 recursive function calls
fibonacci(5) requires 8 recursive function calls
fibonacci(6) requires 14 recursive function call
fibonacci(7) requires 24 recursive function call
fibonacci(8) requires 40 recursive function call
fibonacci(9) requires 66 recursive function call
fibonacci(10) requires 108 recursive function ca
fibonacci(11) requires 176 recursive function ca
fibonacci(12) requires 286 recursive function ca
fibonacci(13) requires 464 recursive function ca
fibonacci(14) requires 752 recursive function ca
fibonacci(15) requires 1218 recursive function c
fibonacci(16) requires 1972 recursive function c
fibonacci(17) requires 3192 recursive function c
fibonacci(18) requires 5166 recursive function c
fibonacci(19) requires 8360 recursive function c
fibonacci(20) requires 13528 recursive function
fibonacci(21) requires 21890 recursive function
fibonacci(22) requires 35420 recursive function
fibonacci(23) requires 57312 recursive function
fibonacci(24) requires 92734 recursive function
fibonacci(25) requires 150048 recursive function
fibonacci(26) requires 242784 recursive function
```

willrosenbaum.com/teaching/2021s-cosc-112/notes/recursive-fibonacci/

Recursive Functions

- Virahaanka without recursion

```
int VirahankaByLooping(int n){   // Program to compute Virahanka number V_n
  int v1=1,v2=2;                 // v1 = V_1,  v2 = V_2
  if(n == 1) return v1;
  else if(n == 2) return v2;
  else {
    int secondlast=v1, last=v2, current;
    repeat(n-2){
      current = secondlast + last;

      secondlast = last;  // prepare for next iteration
      last = current;
    }
    return current;
  }
}
```

Practice Examples for Lab: Set 9

- 1

The factorial of a number n is denoted as $n!$, and can be defined using the recurrence $n! = (n-1)!$ for $n > 0$ and $0! = 1$. Write a recursive function to compute $n!$.

- 2

The binomial coefficient $\binom{n}{r}$ can be defined recursively as $\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1}$, for $n, r > 0$ and $\binom{n}{0} = \binom{n}{n} = 1$ for all $n \geq 0$. Write a function to compute $\binom{n}{r}$.

- 3

Consider the recurrence $W_n = W_{n-1} + W_{n-2} + W_{n-3}$, with $W_0 = W_1 = W_2 = 1$. Write a recursive program for printing W_n. Also write a loop based program.

Practice Examples for Lab: Set 9

- 4

Consider an equation $ax + by = c$, where a, b, c are integers, and the unknowns x, y are required to be integers. Such equations are called Diaphontine equations. If $GCD(a, b)$ does not divide c, then the equation does not have any solution. However, the equation will have infinitely many solutions if $GCD(a, b)$ does divide c. Write a program which takes a, b, c as input and prints a solution if $GCD(a, b)$ divides c.

Hint 1: Suppose $a = 1$. Show that in this case an integer solution is easily obtained.

Hint 2: Suppose the equation is $17x + 10y = 4$. Suppose you substitute $y = z - x$. Then you get the new equation $7x + 10z = 4$. Observe that the new equation has smaller coefficients, and given a solution to the new equation you can get a solution to the old one.

Program Organization and Functions

- Organizing functions into files
  - Can be easier to manage when functions involve lots of lines of code
  - Rule 1:
    If a function f() is being called by code in file T,
    then function f() must be "declared" inside T before statement calling f()
    - Function definition is also a declaration
    - Function declaration states function name, types of arguments, return type
      - Doesn't need a body, e.g., "int lcm (int m, int n); int gcd (int m, int n);" or
        "int lcm (int, int); int gcd (int, int);"
    - This is true even if there is a single file containing main() and f()
      - e.g., If main() is calling function f(),
        but f() is defined after main(),
        then there must be a "declaration statement" of f() before main()
  - Rule 2:
    Every function being called must be defined exactly once in some file

Program Organization and Functions

- Splitting program code into multiple files
  - Files: gcd.cpp lcm.cpp abc.cpp
  - Compile command: "c++ gcd.cpp lcm.cpp abc.cpp"

gcd.cpp

```
int gcd(int m, int n){
  int mdash,ndash;

  while(m % n != 0){
    mdash = n;
    ndash = m % n;
    m = mdash;
    n = ndash;
  }
  return n;
}
```

lcm.cpp

```
int gcd(int, int);

int lcm(int m, int n){
  return m*n/gcd(m,n);
}
```

abc.cpp (main ?? s++ ??)

```
#include <simplecpp>
int lcm(int m, int n);    //

int main(){
  cout << lcm(36,24) << endl;
}
```

Program Organization and Functions

- Splitting program code into multiple files
  - Files: gcd.cpp lcm.cpp main.cpp
  - Another way to compile using notion of "object modules"
  - Motivation: Different programmers may be working with different files. Each can compile their work separately; useful to find compilation errors. Because each file contains only a part of program, executable cannot be produced.
  - Use s++ with "-c" flag: "s++ -c filename" produces filename.o
  - To get executable: run "s++ main.o gcd.o lcm.o"
    - This involves "linking" object modules together
    - e.g., ensure that for all function declarations, a function definition exists in some .o file
  - Can call s++ passing a mix of cpp files and o files
    - e.g., "s++ main.cpp gcd.o lcm.o"

Program Organization and Functions

- Header files
  - Programmer working on gcd.cpp can share declaration of function gcd() with other programmers
    by putting the declaration line in a different file (so-called header file) gcd.h
  - Other programmers (e.g., one working on lcm.cpp) wanting to declare gcd() can use, within their file: #include “gcd.h”
    instead of writing their own declaration →
    - Angle brackets <…> used when
      .h file is in system directory
    - Double quotes “…” used when
      .h file is in same directory
      as file that has the #include command

```
int gcd(int, int);


int lcm(int m, int n){
  return m*n/gcd(m,n);
}
```

Program Organization and Functions

- Avoiding multiple #includes of same .h file
  - C/C++ allows duplicate declarations of the same function (or data structure)
    - Note: C/C++ standards change over time. In early days, this may have been disallowed
  - Duplicate definitions of a function (or data structure) gives a compilation error
  - e.g., if main.cpp includes b.h and c.h,
    and both b.h and c.h include d.h , then
    definitions within d.h (if any) get included twice, leading to compilation error
  - We avoid this situation by modifying each header file (including d.h) as:
    #ifndef D_H
    #define D_H
    … all declarations …
    … all definitions …
    #endif
  - C++ has a so-called “one **definition** rule”

Program Organization and Functions

- Avoiding multiple #includes of same .h file

geometry.h:

```
1 | #include "square.h"
```

main.cpp:

```
1 | #include "square.h"
2 | #include "geometry.h"
3 |
4 | int main()
5 | {
6 |     return 0;
7 | }
```

square.h:

```
1 | int getSquareSides()
2 | {
3 |     return 4;
4 | }
```

square.h:

```
1 | #ifndef SQUARE_H
2 | #define SQUARE_H
3 |
4 | int getSquareSides()
5 | {
6 |     return 4;
7 | }
8 |
9 | #endif
```

Program Organization and Functions

- Packaging software – how to share without disclosing “source”
  - Creator
    - Create F.cpp (your source; function definitions)
    - Create F.h (functions declarations that you want to share; may be also some definitions)
      - This is “interface”
    - Compile F.cpp using –c option to produce F.o
    - Share F.o (functionality) and F.h (interface)
  - User
    - Places F.h in their directory and #includes it in their cpp/h files
    - When compiling their cpp files, they provide F.o file also to compiler

Program Organization and Functions

- Forward declarations for functions
  - A function must be declared before usage (definition subsumes declaration)
  - Benefits
    - Allows flexible layout of functions within program files
      - Otherwise called functions must be placed before calling functions
    - Allows mutually-recursive functions
      - functionA calls functionB, and functionB calls functionA
      - Although this isn't popular usage
    - Needed for "separate compilation" when several programmers work on a project
      - Programmer A's cpp file needs to call function present programmer B's cpp
      - Programmer A can compile cpp

```
declarations of functions in any order.
definition of main
definitions of other functions in any order.
```

```
int first(int x) {
    if (x == 0)
        return 1;
    else
        return second(x-1); // forward reference to second
}

int second(int x) {
    if (x == 0)
        return 0;
    else
        return first(x-1); // backward reference to first
}
```

Program Organization and Functions

- Function templates and header files
  - When compiling a use/call of a template function F(.),
    C++ compiler
    "looks into" template function F(.)'s argument types
    and also looks into template function F's body
  - When a source file includes F.h (where function is declared) and
    calls a template function,
    compiler asks for access to F's body also (e.g., to generate optimized code)
  - So, we need to define template function F(.) in its header file F.h,
    instead of a cpp file F.cpp

Program Organization and Functions

- Namespaces
  - What if you borrow and use code (.h .o files) from 2 different programmers, where each .h file has a function of same name and signature ?
    - Assume source code isn't available
    - e.g., a function lcm(.)
  - These 2 programmers should have used the notion of a namespace
  - Namespace is like a catalog (think of it as a family name or surname)
  - Placing a name N (e.g., of function, variable, etc.) within a namespace C, the defined name is actually C::N
  - A programmer creates functions (shared publicly) within unique namespace
    - e.g., spaceA and spaceB
  - Then you can access the 2 different functions as spaceA::lcm(.) and spaceB::lcm(.)

Program Organization and Functions

- Namespaces
  - Defining a namespace and entities within it

```
namespace name-of-namespace{
  declarations of definitions of names
}
```

```
namespace mySpace{
  int gcd(int,int);
  int lcm(int m,int n){return m*n/gcd(m,n);}
}
```

  - Inside namespace block, we can define functions as lcm and gcd
  - But outside namespace block, define using full name

```
int mySpace::gcd(int m, int n){
  if(n == 0) return m;
  else return gcd(n, m % n);
}
```

  - Use them elsewhere by using full name

```
int main(){
  cout << mySpace::lcm(36,24) << endl;
}
```

Program Organization and Functions

- Namespaces
  - If you find writing full name of function at multiple occasions too much work, then you can define a short form through the "*using*" declaration "*using* namespaceName::functionName"
    - e.g., declaring "*using* mySpace::lcm" before main() tells compiler that calls to lcm() within main() are to mySpace::lcm()
  - Alternatively, just declare "*using namespace* namespaceName", which will allow usage of all names within namespace ns without needing to use their fullname
    - e.g., "*using namespace* mySpace" and then calling lcm() from main() will effectively call mySpace::lcm()

Program Organization and Functions

- Namespaces
  - What happens when you declare functions outside of any namespace ?
    - They enter a global namespace
  - When you use a name (e.g., function name) without a namespace qualifier, that name should be present either in global namespace or some namespace for which a *using* directive has been given
  - If same name (same signature) present in 2 namespaces → compilation error
    - e.g., when mySpace already had a lcm function and your code is as follows:

```
using namespace mySpace;

int lcm(int m,int n){return m*n;}  // not really computing the lcm!

int main(){
  cout << lcm(36,24) << endl;
}
```

    - If you want to call lcm in global namespace, you should use ::lcm(...) otherwise we should use mySpace::lcm(...)

Program Organization and Functions

- Global variables = a variable defined outside all functions, even main()
  - Example
  - Makes code less readable
  - Risk any function modifying global variable, leading to unintended effects

```
int i=5;                        // global variable definition

void f(){ i = i * i; }          // refers to global variable

int main(){
  cout << i << endl;            // refers to global variable
  f();
  cout << i << endl;            // refers to global variable
}
```

  - To use global variable V in fileA, but where V is defined in fileB
    - fileA needs to have a declaration: e.g., "extern int V;"
  - Global variables can be put into namespaces
  - Global variable has global scope; visible throughout program, unless shadowed

Program Organization and

- Global variables

Global-data coupling. Two routines are global-data–coupled if they make use of the same global data. This is also called "common coupling" or "global coupling." If use of the data is read-only, the practice is tolerable. Generally, however, global-data coupling is undesirable because the connection between routines is neither intimate nor visible. The connection is so easy to miss that you could refer to it as information hiding's evil cousin—"information losing." (McConnell 1993, p. 90; book on accepted practices)

Program Organization and Functions

- Namespace “std”
  - Popular functions: cin, cout, endl
  - Why didn’t we have to use std::cin ?
    - #include <simplecpp>
    - File simplecpp contains the directive “using namespace std;”
- Namespace “simplecpp”
  - Contains initCanvas, forward, left, Circle, Line, …
  - File simplecpp contains the directive “using namespace simplecpp;”

Program Organization and Functions

- C++ without simplecpp
- File: sidelength.cpp
- Command to compile:
  g++ sidelength.cpp
- "iostream" is a header file containing declarations of functions for input-output
  - e.g., cin, cout [very early versions of C++ used iostream.h]

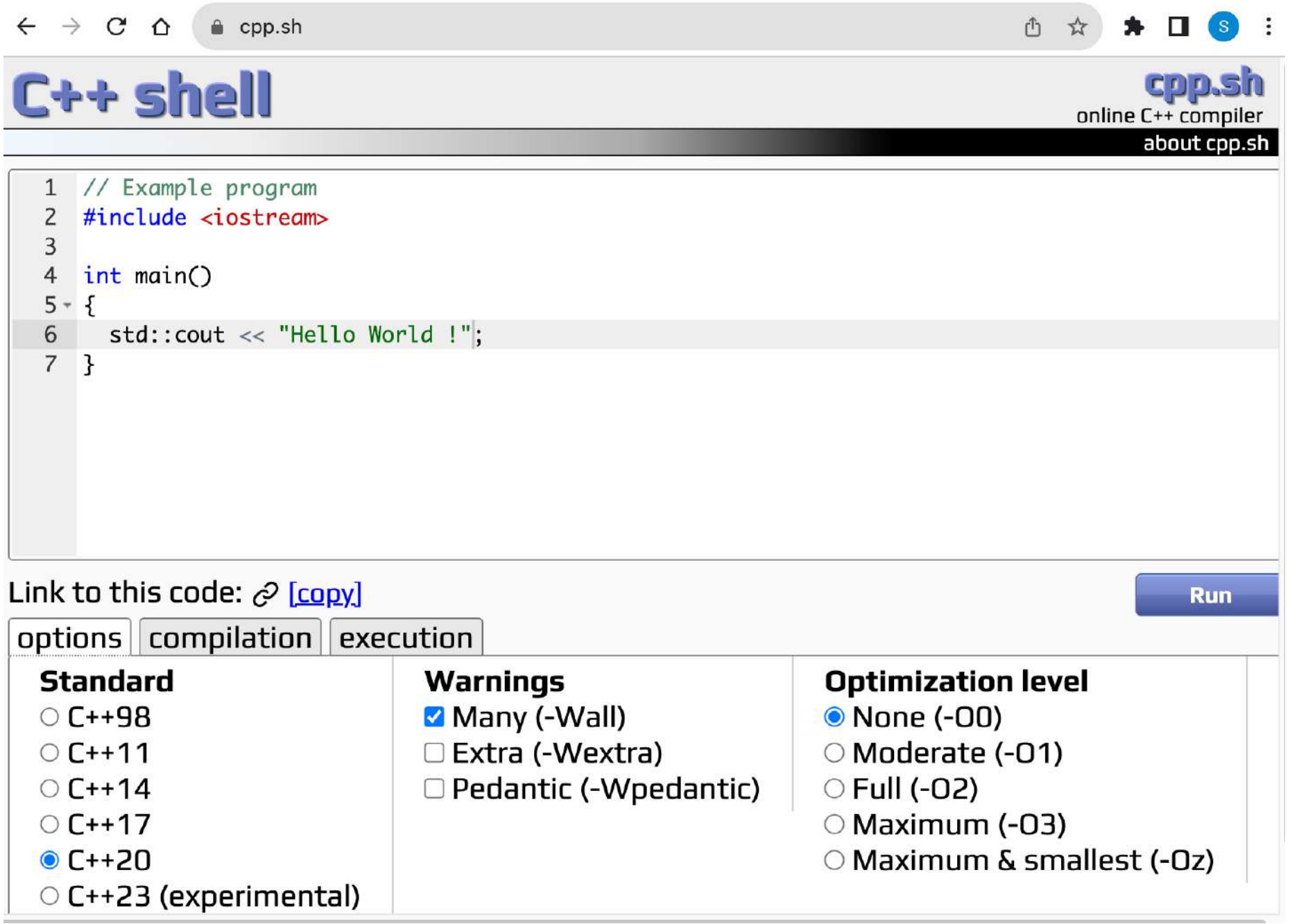
```
#include <iostream>
#include <cmath>
using namespace std;

int main(){
   cout << "Give area of square: ";
   double area;
   cin >> area;
   cout << "Sidelength is: " << sqrt(area) << endl;
}
```

- Similarly, "cmath" contains declarations of functions like sqrt()
- So far, <simplecpp> header file was including these header files
- "int main()"

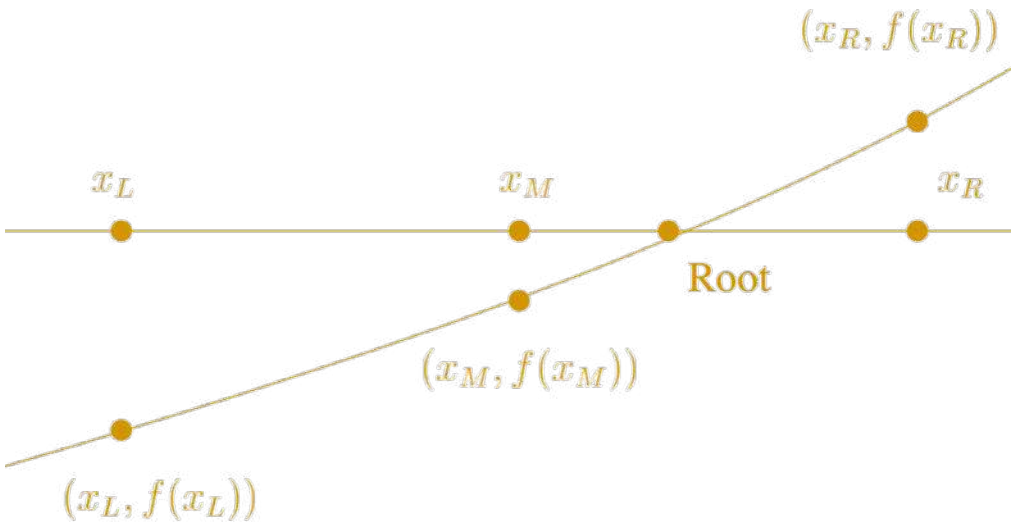
C++ Resources on WWW

- [cpp.sh/](cpp.sh/)



Program Organization and Functions

- Our code for bisection was specific to a mathematical function f(x)
- Can we generalize this code so that it can work for any function ?




```
main_program{                    // find root of f(x) = x*x - 2
  float xL=0,xR=2; // invariant: f(xL),f(xR) have diffe
  float xM,epsilon;
  cin >> epsilon;
  bool xL_is_positive, xM_is_positive;
  xL_is_positive = (xL*xL - 2) > 0;
  // Invariant: xL_is_positive gives the sign of f(x_L)

  while(xR-xL >= epsilon){
    xM = (xL+xR)/2;
    xM_is_positive = (xM*xM -2) > 0;
    if(xL_is_positive == xM_is_positive)
      xL = xM;  // does not upset any invariant!
    else
      xR = xM;  // does not upset any invariant!
  }
  cout << xL << endl;
}
```

Program Organization and Functions

- Passing functions as arguments
  - C++ concept of "function pointer"
  - May think of & as "address-of" start of function code (intuitively)

```
double f(double x){
  return x*x -2;
}
```

```
double g(double x){
  return sin(x) - 0.3;
}
```

```
int main(){
  double y = bisection(1,2,0.0001,&f);
  cout << "Sqrt(2): " << y << " check square: " << y*y << endl;

  double z = bisection(0,PI/2,0.0001,&g);
    cout << "Sin inverse of 0.3: " << z << " check sin: " << sin(z) << endl;
}
```

Program Organization and Functions

- Function pointer = pf
- In function body, * = dereferencing operator, so *pf denotes function
  - (*pf)(xL) makes call f(xL)
- Argument pf is a pointer to function taking double and returning double

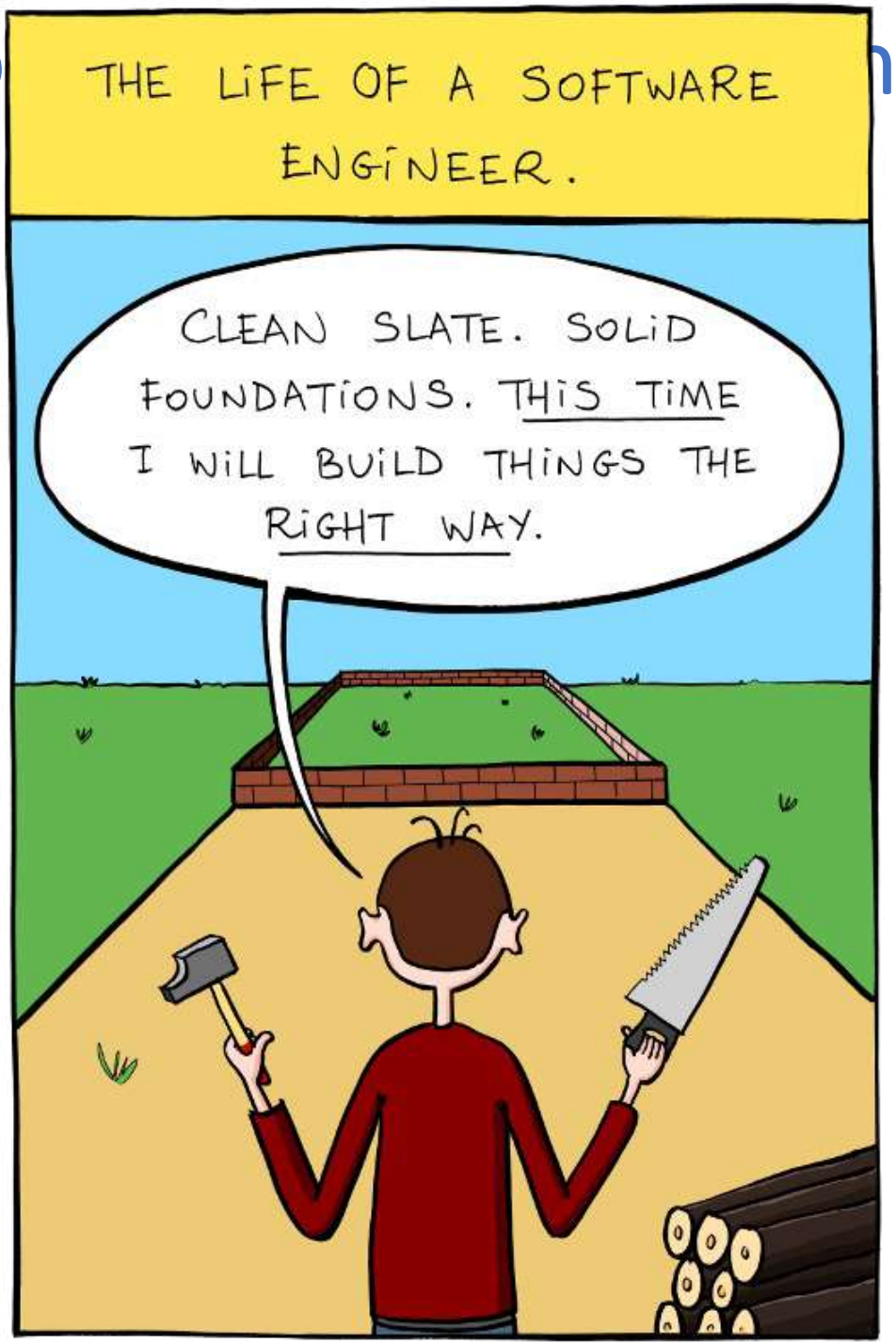
```
double bisection(double xL, double xR, double epsilon, double (*pf)(double x))
// precondition: f(xL),f(xR) have different signs. ( >0 and <=0).
{
  bool xL_is_positive = (*pf)(xL) > 0;
  // Invariant: x_L_is_positive gives the sign of f(x_L).
  // Invariant: f(xL),f(xR) have different signs.

  while(xR-xL >= epsilon){
    double xM = (xL+xR)/2;
    bool xM_is_positive = (*pf)(xM) > 0;
    if(xL_is_positive == xM_is_positive)
      xL = xM;  // maintains both invariants
    else
      xR = xM;  // maintains both invariants
  }
  return xL;
}
```

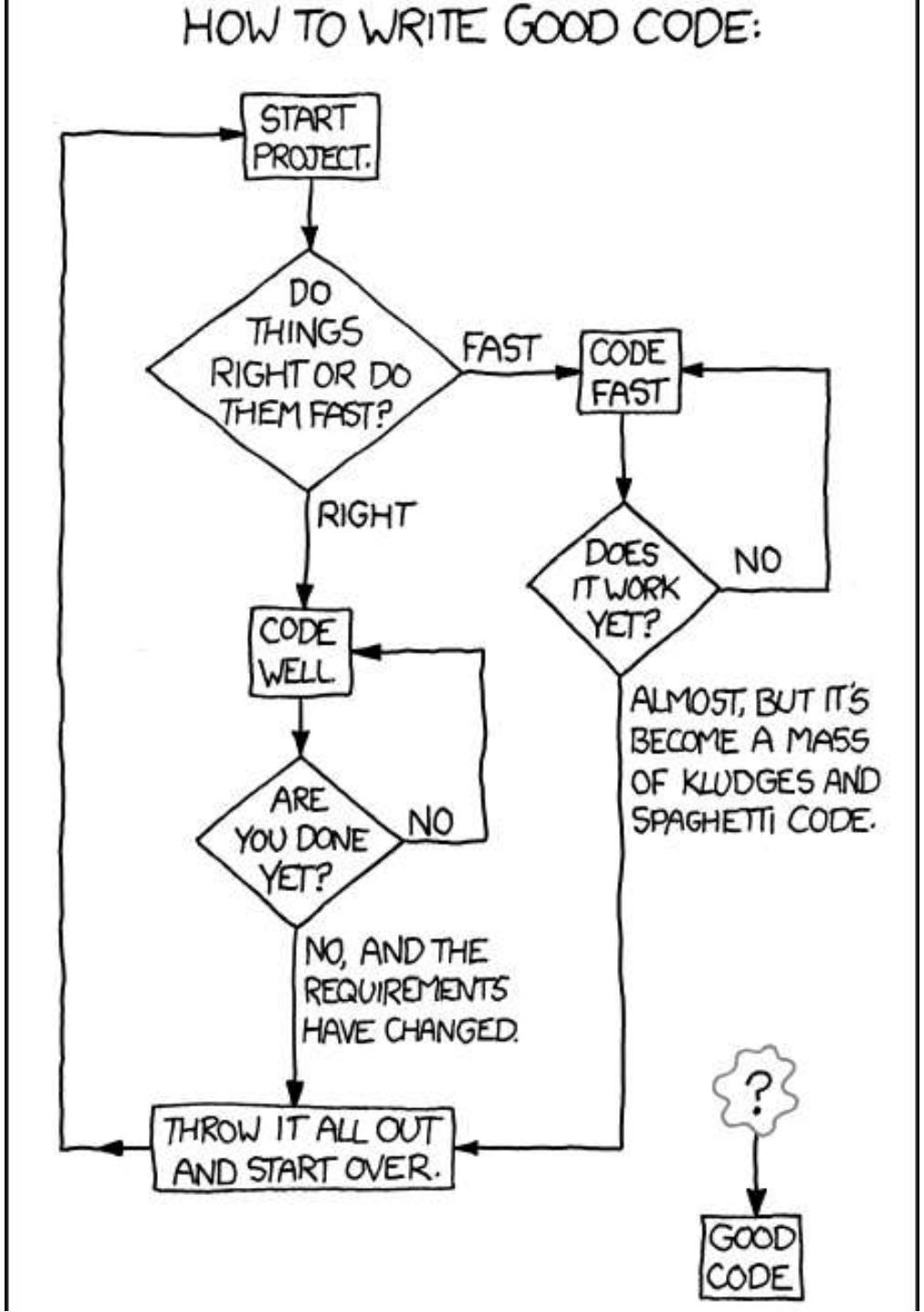
Program Organization and Functions

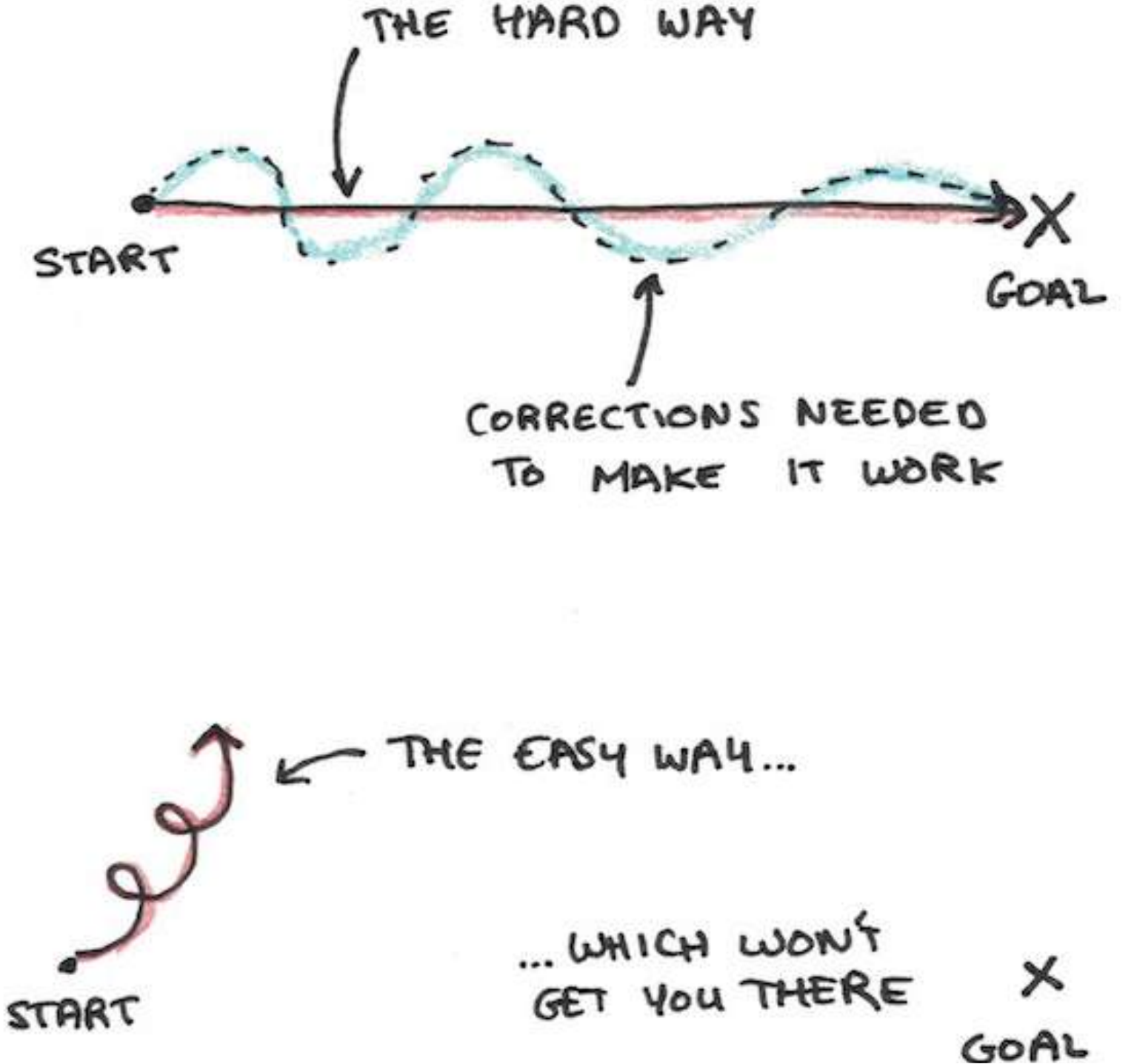
- Functions are key elements in programming,
  which allow abstraction of elemental concepts in modules
- Modules
  - Help break large piece of logic into smaller logical pieces,
    each of which is easier to understand
  - Can be reused by programmer, and across programmers
  - Make it easier to test code (per module), maintain code (per module)
- "Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules,
  such that each contains everything necessary to execute only one aspect of the desired functionality."
  - en.wikipedia.org/wiki/Modular_programming

Pro nd Funct


THE LIFE OF A SOFTWARE ENGINEER.
CLEAN SLATE. SOLID FOUNDATIONS. THIS TIME I WILL BUILD THINGS THE RIGHT WAY.


MUCH LATER...
OH MY. I'VE DONE IT AGAIN, HAVEN'T I ?
MANU


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL.
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
?
THROW IT ALL OUT AND START OVER.
GOOD CODE

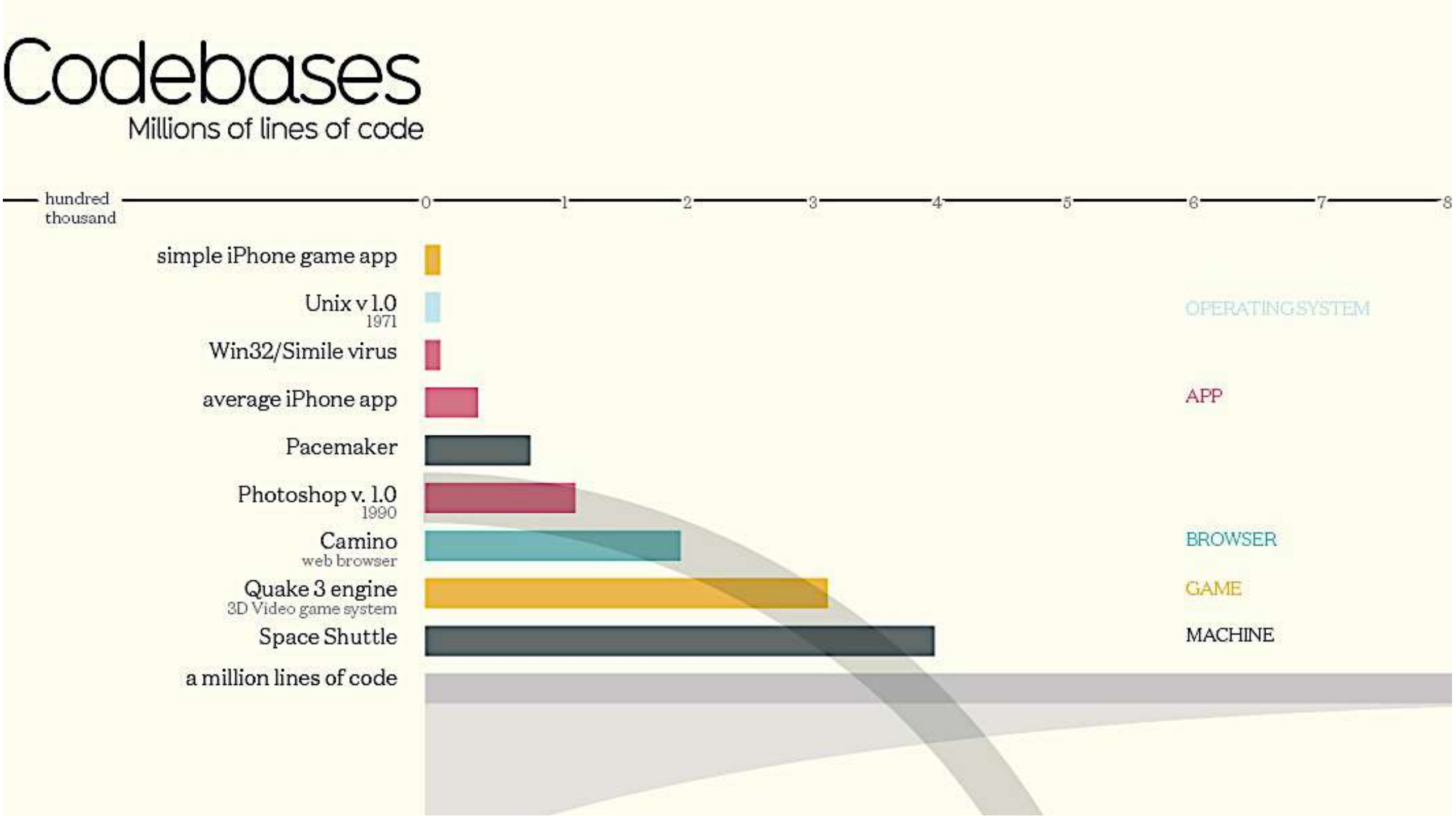

THE HARD WAY
START
GOAL
CORRECTIONS NEEDED TO MAKE IT WORK
THE EASY WAY...
START
...WHICH WON'T GET YOU THERE
GOAL

Program

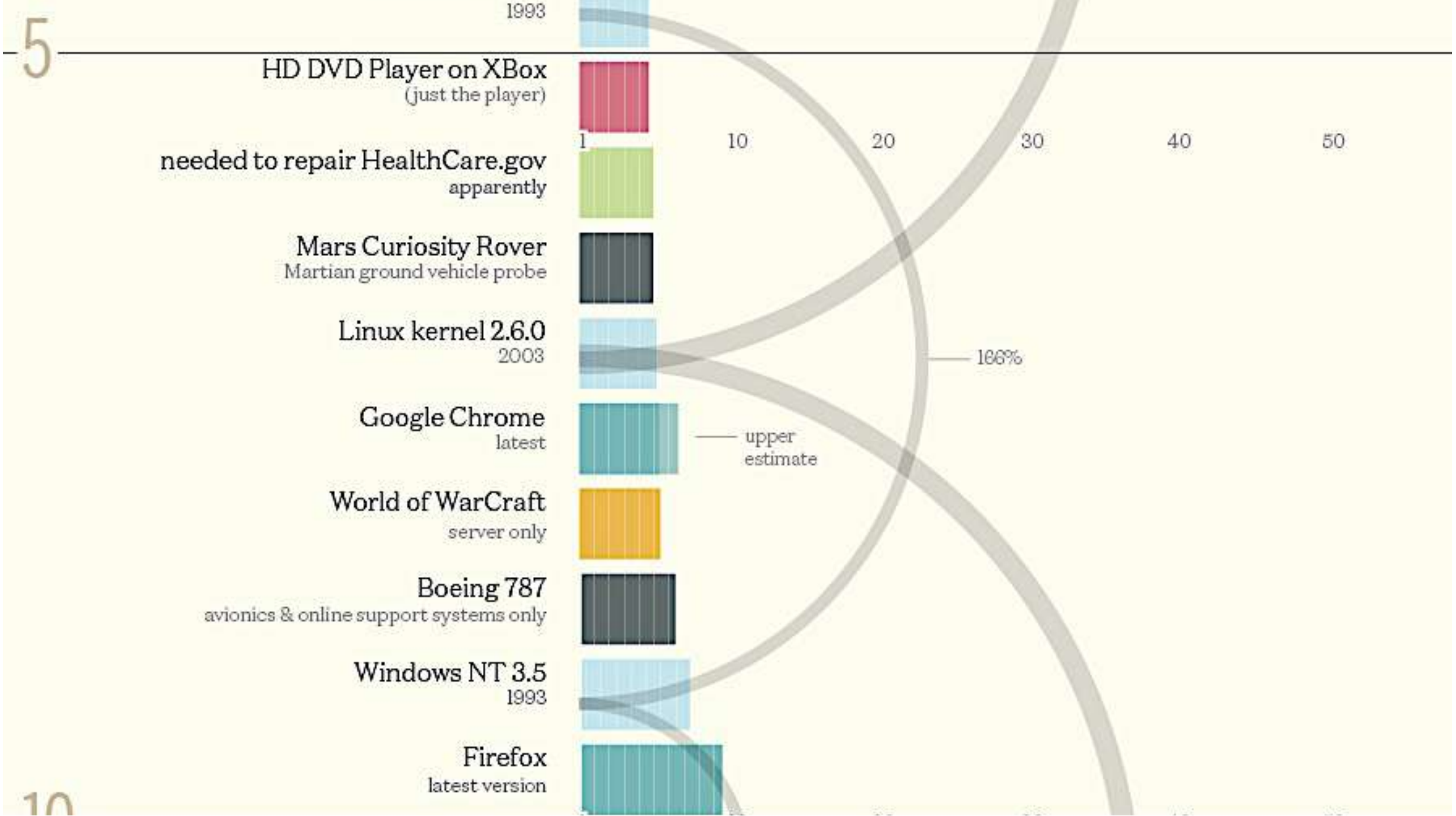

When you are the only developer who writes code for a feature with no code reviewers and yet you make it modular, write it with adequate comments and follow strict naming conventions...
fb.me/yuva.krishna.memes
I did it for me.

Codebases

Millions of lines of code




1
million
1
10
20
30
40
50
60
70
80
90
a million lines of code
18,000 pages of printed text
War And Peace x 14, or
Ulysses x 25, or
The Catcher in The Rye x 63
CryEngine 2
3D video game system
Bacteria
Syphillis (Treponema pallidum)
Age of Empires online
CESM Climate Model
National Center for Atmospheric Research
F-22 Raptor fighter jet
Linux Kernel 2.2.0
core code
Jurassic Park codebase
source: Dennis Nedry
Hubble Space Telescope
Unreal engine 3
3D video game system
Windows 3.1
1992
Large Hadron Collider
(root software)
US military drone
(control software only)
Photoshop C.S. 6
image editing software
Windows NT 3.1
ORGANISM
3750%
288%


1993
5
HD DVD Player on XBox
(just the player)
1
10
20
30
40
50
needed to repair HealthCare.gov
apparently
Mars Curiosity Rover
Martian ground vehicle probe
Linux kernel 2.6.0
2003
166%
Google Chrome
latest
upper
estimate
World of WarCraft
server only
Boeing 787
avionics & online support systems only
Windows NT 3.5
1993
Firefox
latest version
10

latest version

10

1 10 20 30 40

Chevy Volt
electric car

Intuit Quickbooks
accounting software

Windows NT 4.0
1996

Android
mobile device operating system

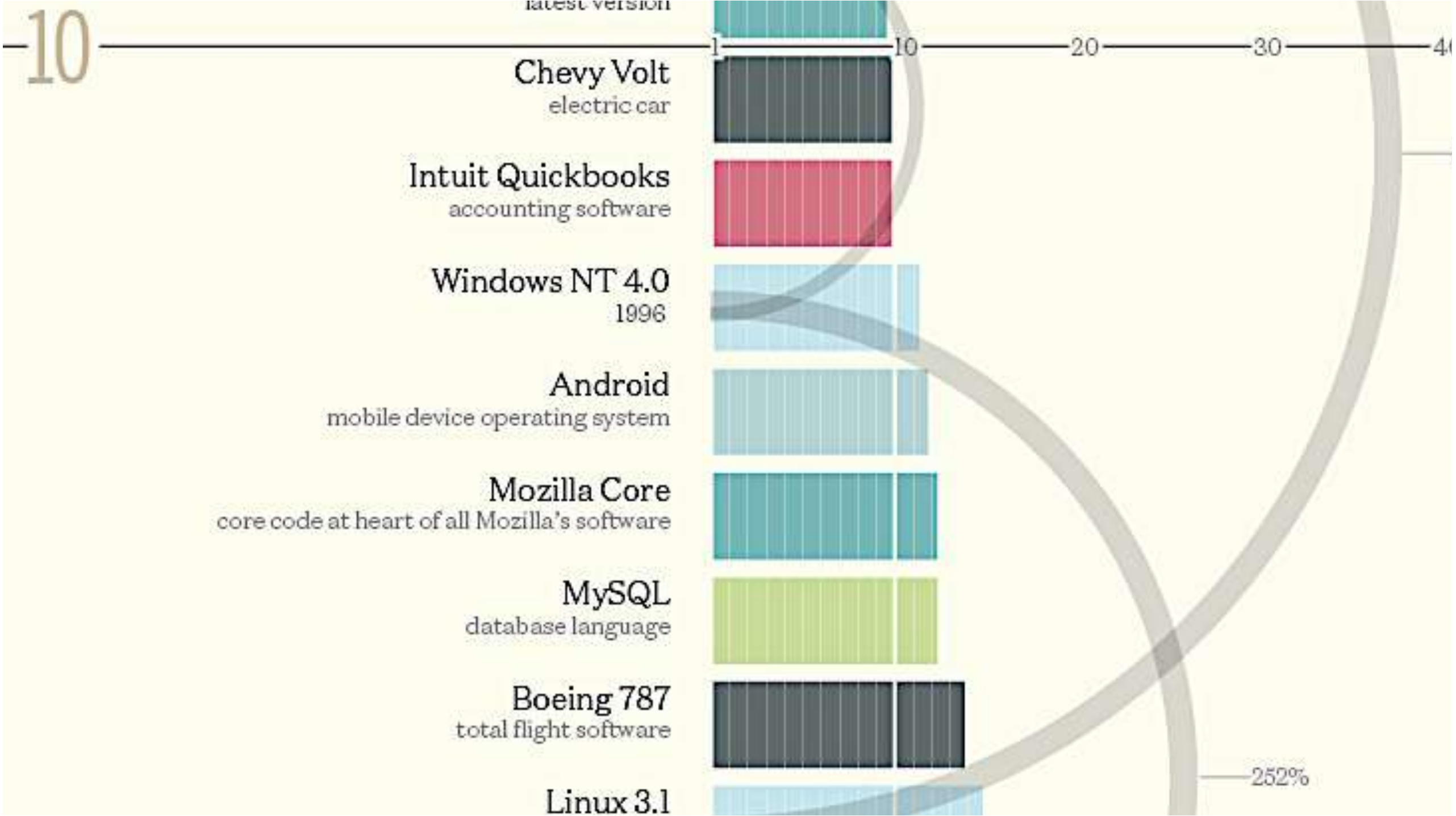
Mozilla Core
core code at heart of all Mozilla's software

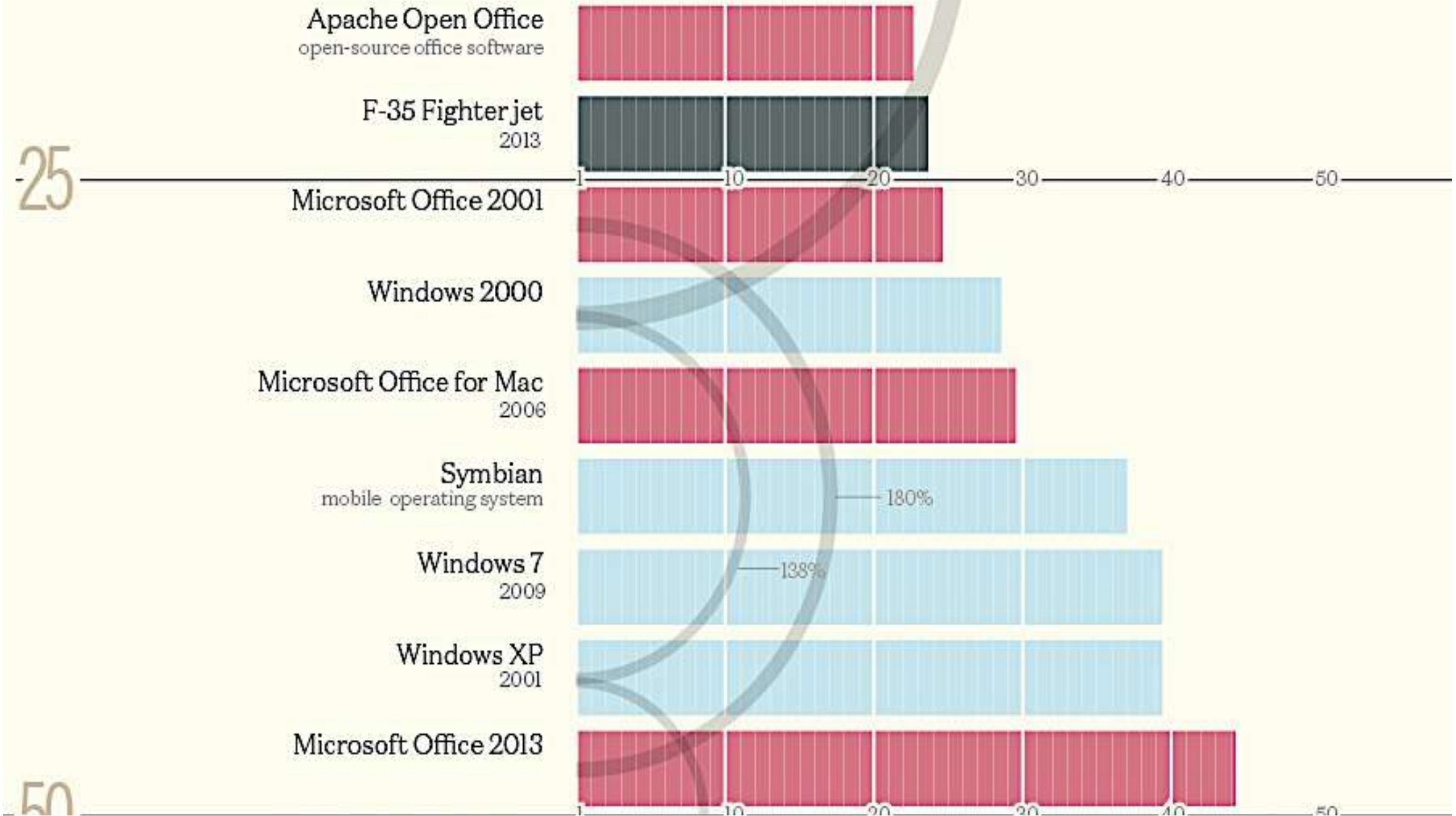
MySQL
database language

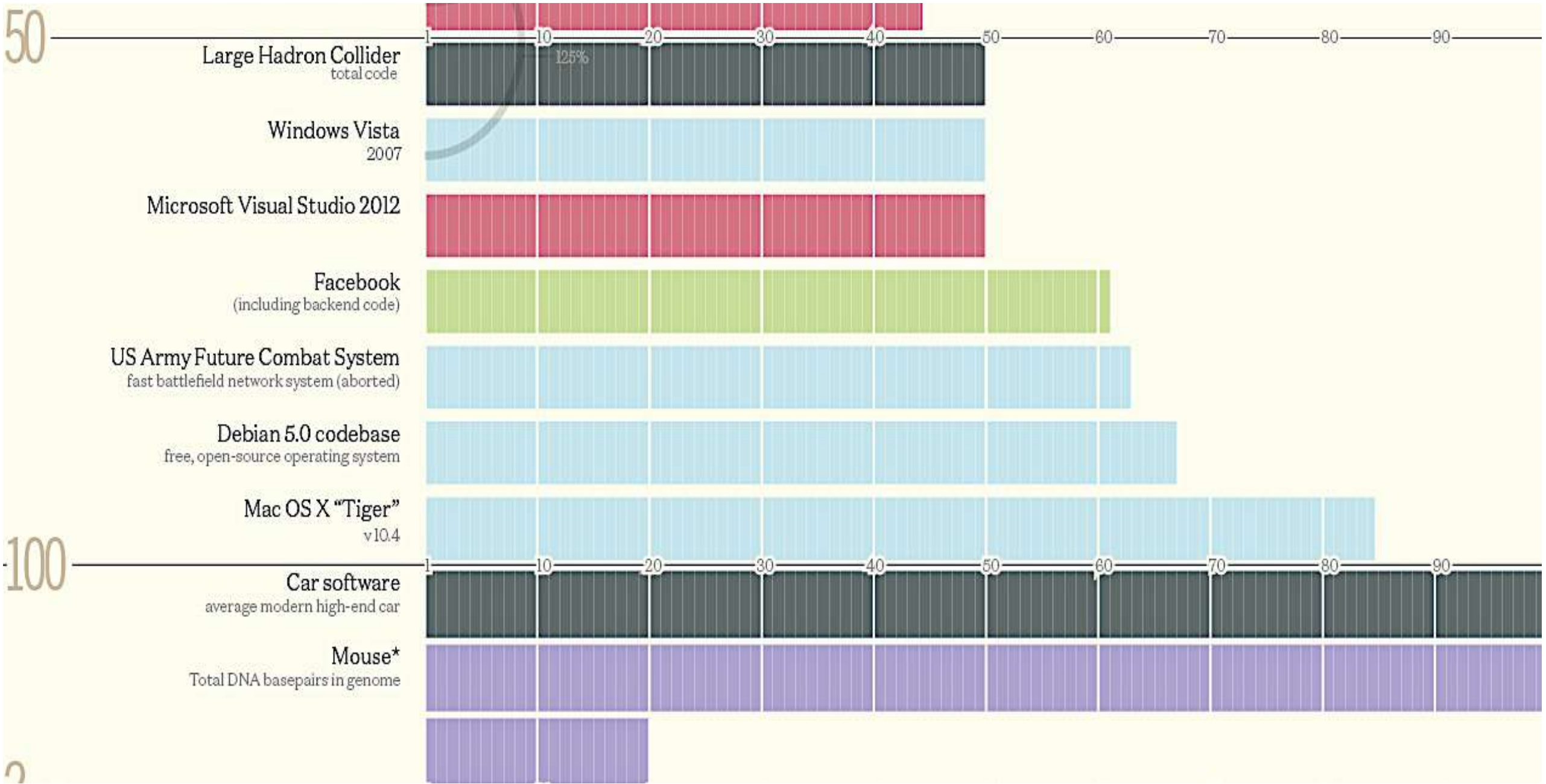
Boeing 787
total flight software

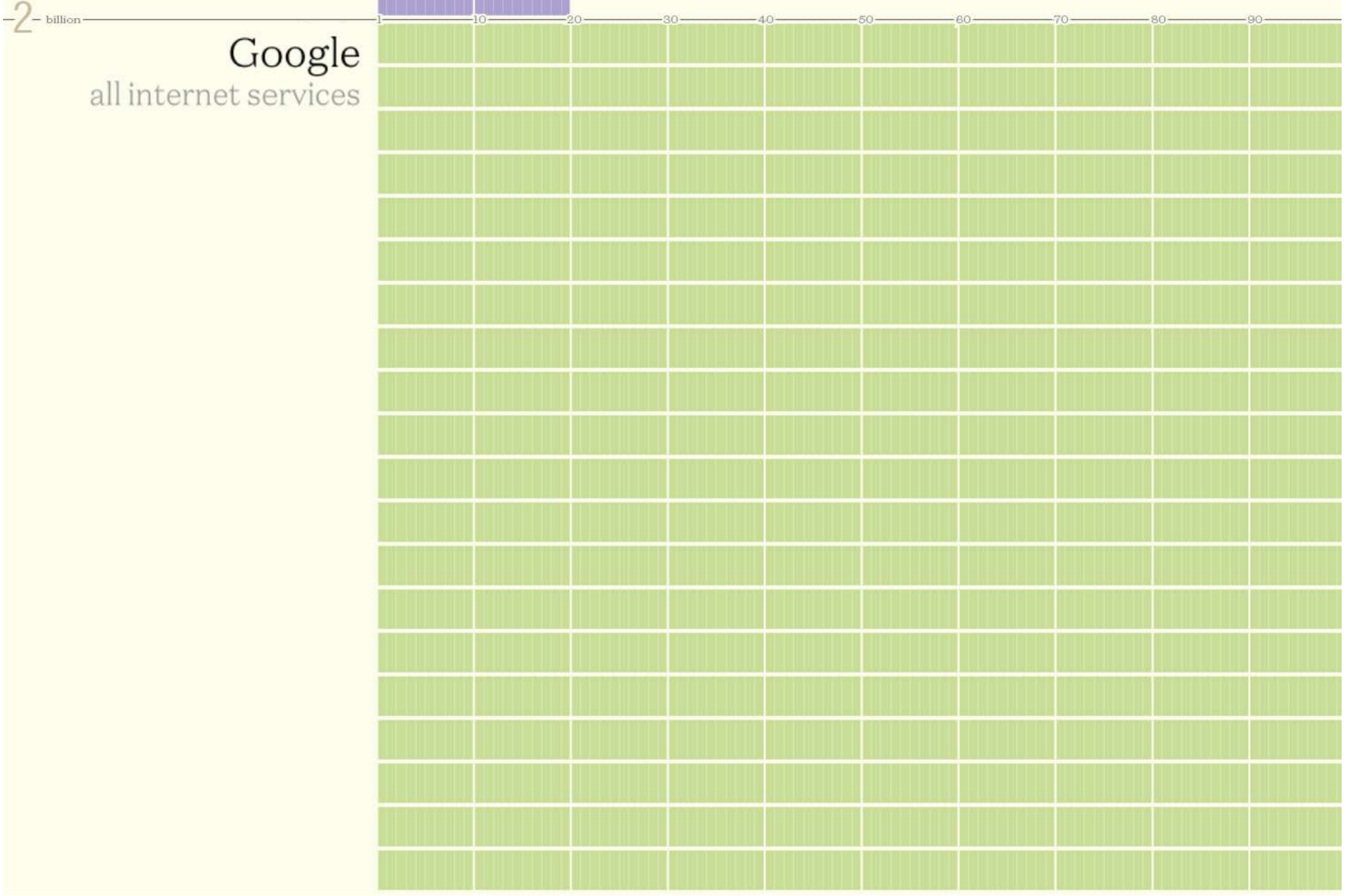
252%

Linux 3.1




Apache Open Office
open-source office software
F-35 Fighter jet
2013
25
1
10
20
30
40
50
Microsoft Office 2001
Windows 2000
Microsoft Office for Mac
2006
Symbian
mobile operating system
180%
Windows 7
2009
138%
Windows XP
2001
Microsoft Office 2013
50
1
10
20
30
40
50


50
Large Hadron Collider
total code
1
10
20
30
40
50
60
70
80
90
125%
Windows Vista
2007
Microsoft Visual Studio 2012
Facebook
(including backend code)
US Army Future Combat System
fast battlefield network system (aborted)
Debian 5.0 codebase
free, open-source operating system
Mac OS X "Tiger"
v 10.4
100
Car software
average modern high-end car
Mouse*
Total DNA basepairs in genome
2


2 billion
1
10
20
30
40
50
60
70
80
90
Google
all internet services

*Human Genome = 3,300 billion “lines” of code

concept & design: David McCandless
informationisbeautiful.net
research: Pearl Doughty-White, Miriam Quick

sources NASA, Quora, Ohloh, Wired & press reports
note some guess work, rumours & estimates
data bit.ly/KIB_linescode

Debugging

- Debugging = process of finding and correcting bugs in software
- Why is the program not producing correct output ?
  - Incorrect input
  - Incorrect logic. If so, in which function ? In which line ?
  - Is the output incorrect only for certain inputs ?
    - If so, can you find out which inputs ?
- Maintain discipline in coding
  - First clarify specification
  - Then plan entire algorithm in mind and on paper. Plan functions/modules.
  - Then write code
    - While you write each line, put comments. Think about test cases.
  - Before running code on computer, run it in your mind

Debugging

- Conditional compilation
  - Six directives are available to control conditional compilation
  - They delimit blocks of program text that are compiled only if a specified condition is true
  - #ifdef identifier
    #ifndef identifier
    #if expression
    #elif expression
    #else
    #endif
  - Use with #define, e.g., #define ENABLE_MY_DEBUG_MODE
  - Use cout statements liberally during debugging, e.g.,
    - #ifdef ENABLE_MY_DEBUG_MODE
      cout ….;
      #endif

Debugging

- "assert (condition);" statement
  - An assertion is a condition that should always evaluate to true at the point in execution where it is placed
  - Programmers use assertions to help specify programs and to reason about program correctness

```c
#include <assert.h>
int main()
{
    void* pointer = 0;
    assert(pointer);
    return 0;
}
```

Upon compiling the program and running it, a message similar to the following will be output:

```
program: source.c:5: main: Assertion 'pointer' failed.
Aborted (core dumped)
```

  - Disabling assert statements is easily done by stating "#define NDEBUG"
    - Effect of assert statement depends on whether NDEBUG has been #define-d
    - If #define NDEBUG is present, then assert is (re)defined as an expression doing nothing

Debugging

- Checking for errors in data input
  - If you execute “int x; cin >> x”, and if your typed response is character ‘a’, then cin becomes unusable !
    - Need to use cin.clear(), cin.ignore(), etc.
  - cplusplus.com/forum/beginner/2957/
  - copyprogramming.com/howto/how-to-use-cin-fail-in-c-properly

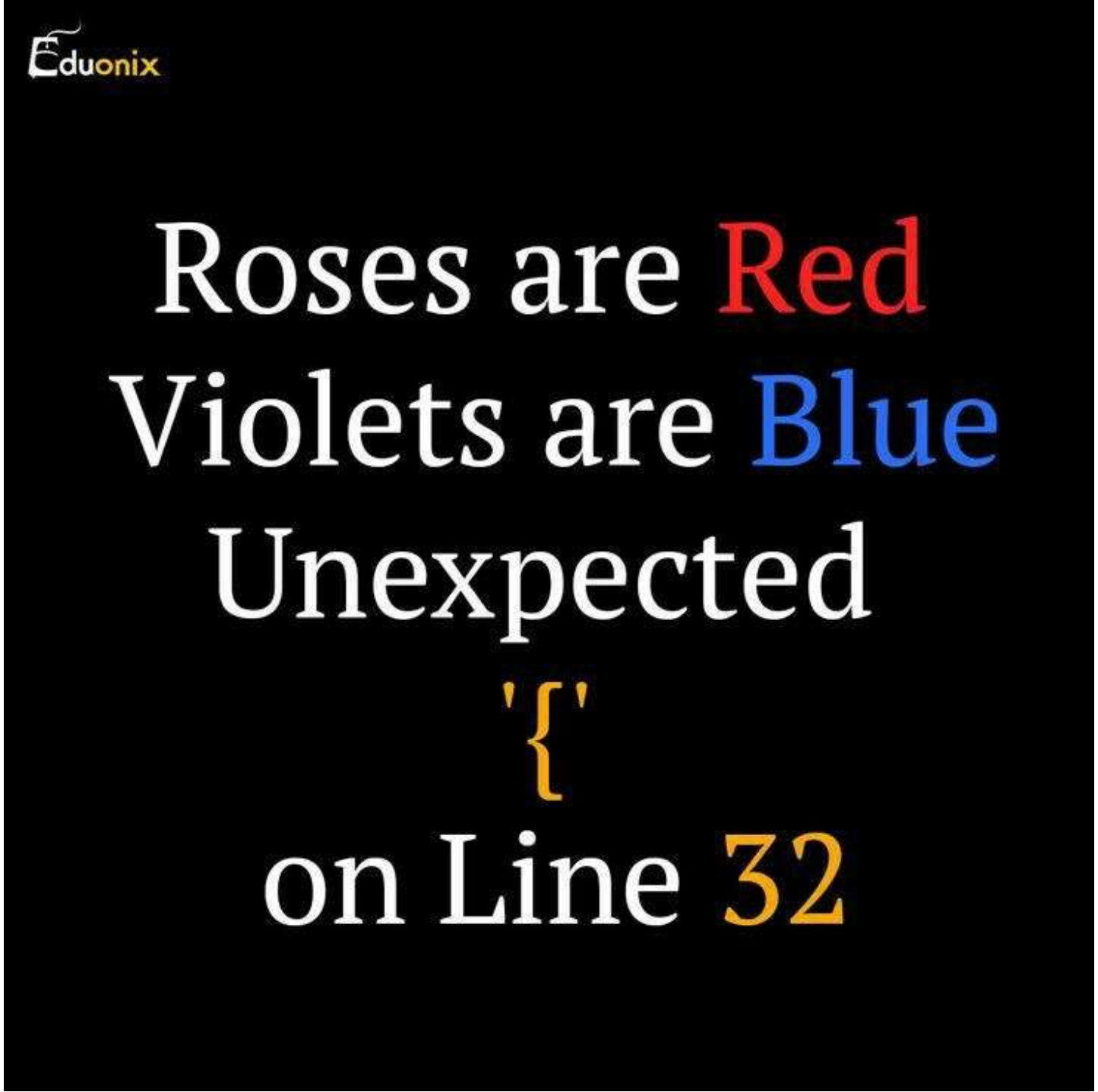
```
1  #include <iostream>
2  int main()
3  {
4      int x = 1;
5      std::cin >> x;
6      if (std::cin.fail()) {
7          //Not an int.
8          std::cout << "cin.fail() is true" << std::endl;
9      }
10     std::cout << x;
11     return 0;
12 }
```

Link to this code: [copy]

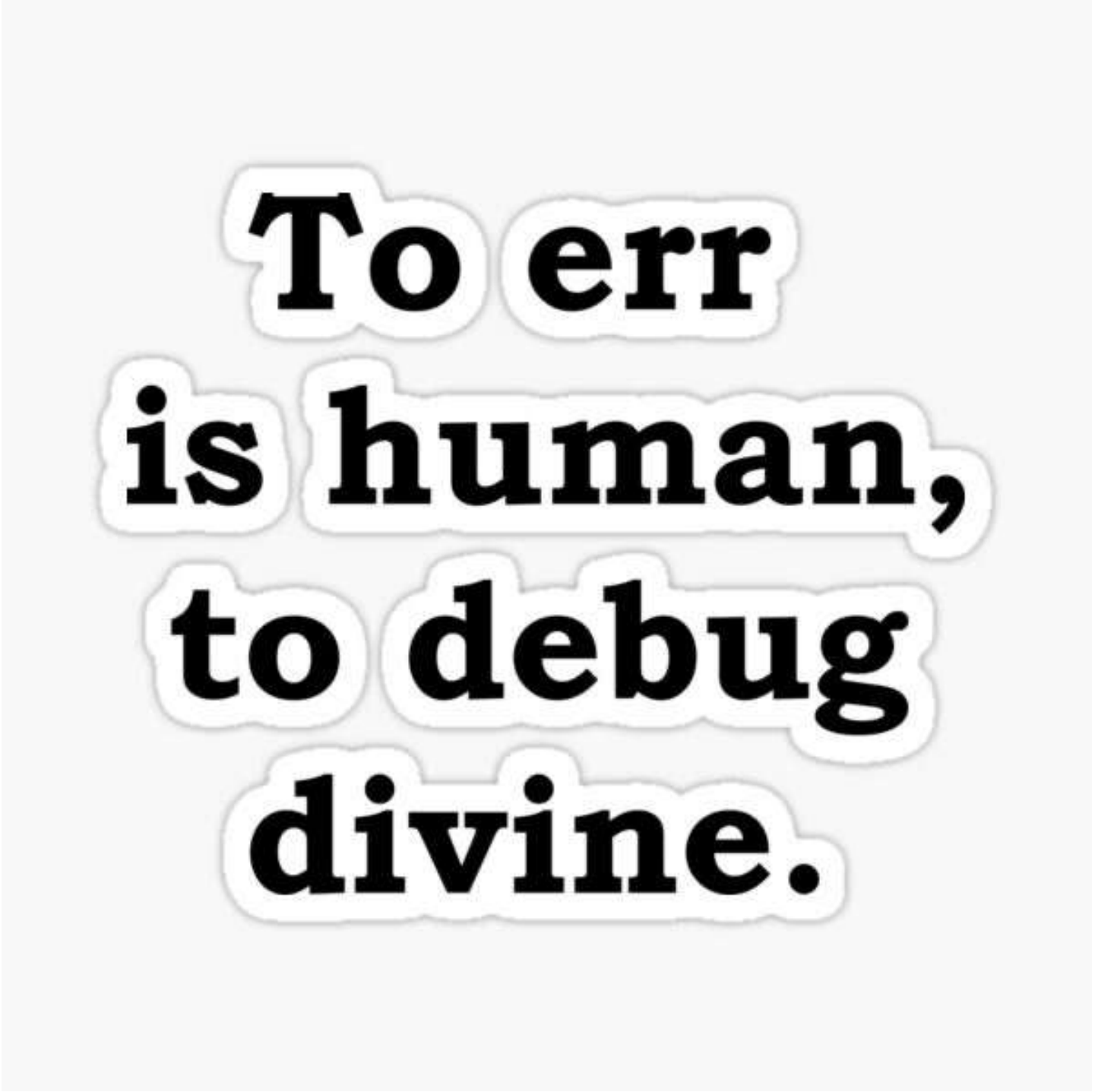
options | compilation | execution

```
q
cin.fail() is true
0
```

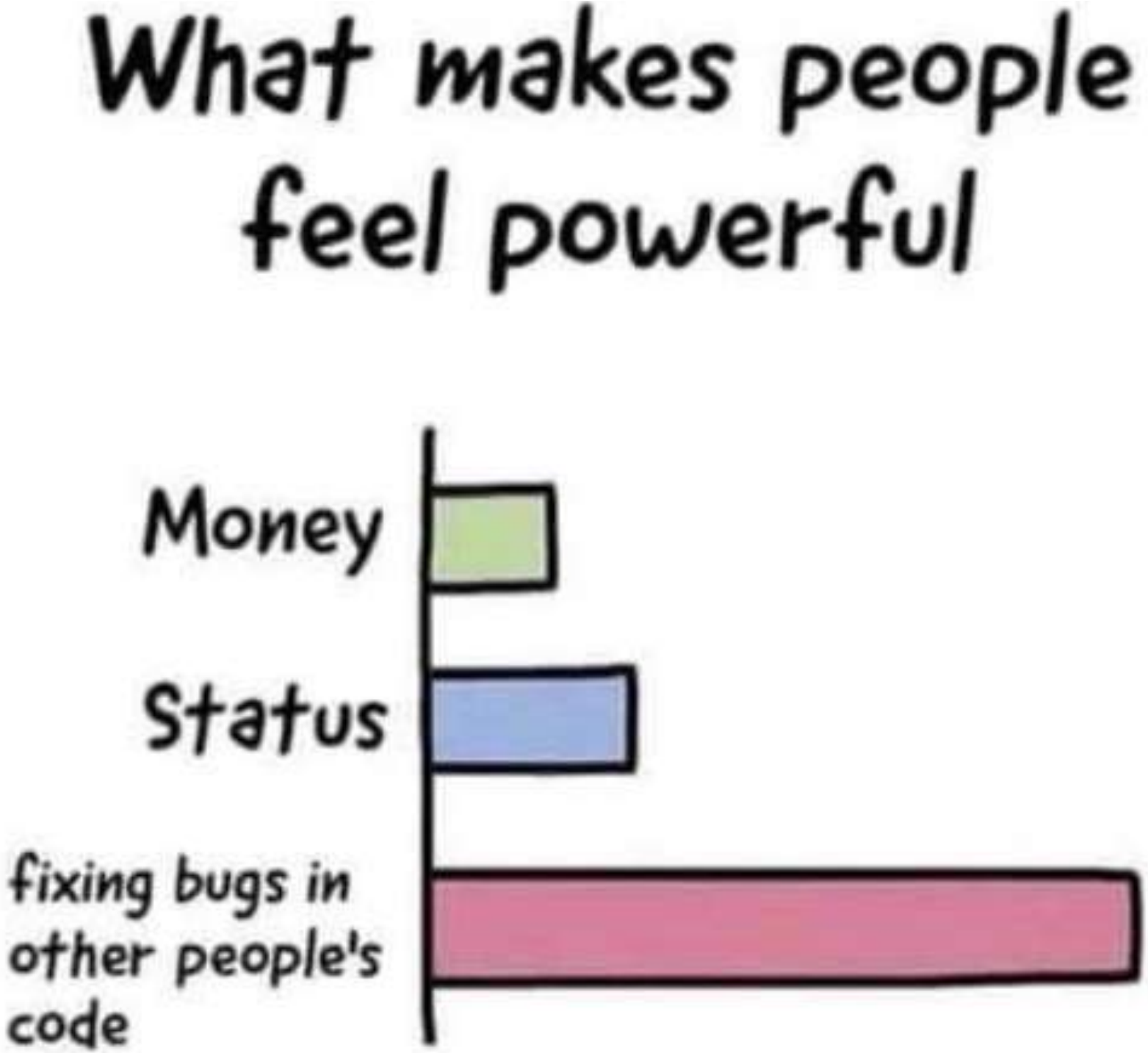
Debugging


Eduonix
Roses are Red
Violets are Blue
Unexpected
'{'
on Line 32

Debugging



Progra


What makes people feel powerful
Money
Status
fixing bugs in other people's code

Random Numbers

- Program needs to generate/simulate random numbers in applications
- Function rand() returns integer in range [0,RAND_MAX]
  - Each call mimics a uniformly random draw from the range
  - Defined in <stdlib.h>, which is #included within <simplecpp>

```
#include <cstdlib> // defines the rand() function and RAND_MAX const
#include <iostream>
using namespace std;

int main()
{ // prints pseudo-random numbers
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
  cout << "RAND_MAX = " << RAND_MAX << endl;
}
```

```
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
RAND_MAX = 2147483647
```


```
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
RAND_MAX = 2147483647
```

Random Numbers

- Pseudorandom numbers
  - Numbers drawn from an actually a deterministic sequence, which is generatable by a computer, such that the sequence appears random
  - First number in sequence called “seed”
  - C++ allows you to set seed via call to function srand(seedValue)
  - For debugging programs involving generating random numbers, often useful to set a seed so that multiple instance of program are guaranteed to be consistent
  - If you want program to run differently each time, you can set seed differently each time via “srand(time())”
    - time() returns the time in seconds since midnight of 1 Jan 1970

Random Numbers

```
#include <cstdlib>  // defines the rand() and srand() functions
#include <iostream>
using namespace std;

int main()
{ // prints pseudo-random numbers:
  unsigned seed;
  cout << "Enter seed: ";
  cin >> seed;
  srand(seed);                    // initializes the seed
  for (int i = 0; i < 8; i++)
    cout << rand() << endl;
}
```

```
Enter seed: 0
12345
1406932606
654583775
1449466924
229283573
1109335178
1051550459
1293799192
```

```
Enter seed: 1
1103527590
377401575
662824084
1147902781
2035015474
368800899
1508029952
486256185
```

```
Enter seed: 12345
1406932606
654583775
1449466924
229283573
1109335178
1051550459
1293799192
794471793
```

Random Numbers

- Simulating random numbers (real numbers) between [u,v]
  - <simplecpp> has a function randuv(u,v) that returns u + (v-u)*rand()/(1.0 + RAND_MAX)
- Simulating random numbers (integers) between [i,j]
  - Call round (randuv (i, j+1))

Practice Examples for Lab: Set 10

- 1) Write a program to simulate 100 tosses of a coin,
  where each coin toss produces a head with probability 0.5 and a tail
  with probability 0.5.
- 2) Write a program to simulate 100 throws of a fair die,
  where each die throw produces a number from the set {1,2,3,4,5,6}
  with equal probability.
- 3) Graphically simulate, using the turtle,
  a 1-dimensional random walk over 100 steps.
  - https://mathworld.wolfram.com/RandomWalk1-Dimensional.html
- 4) Graphically simulate, using the turtle,
  a 2-dimensional random walk on a lattice, over 100 steps.
  - https://mathworld.wolfram.com/RandomWalk2-Dimensional.html