

Lecture 4

Functional Abstraction and Pointers

TIC1001 Introduction to Computing and Programming I

Archipelago





If you are given one hour
to chop down a tree

You should spend 45 mins
sharpening your axe

DO WE HAVE TO CODE
2 HOURS
PER DAY?



More Thinking
Less Coding
Less is more

Sleep is good for you

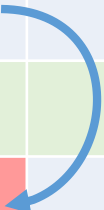


Submission is Final
Just remember to click

Finalize Submission

Calendar

Week	Mon	Tue	Wed	Thu	Fri	Sat	Sun
1				Lecture (P)		Technical Support	
2				Lecture (P)		Lab	
3				Lecture (C)		Tutorial 1 + Lab	
4				Lecture (P)		Tutorial 2 + Lab	
5				Lecture (C)		Tutorial3 + Lab	
6				Lecture (P)		Tutorial4 + Lab	
R							
7				Midterm Test 7—8pm		Practical Exam 1 10:30—11:30am	



Online Assessments

Will be conducted online

- Use own PC/Laptop
- Stable Internet
- Handphone/Ipad (2nd device) for proctoring

Only valid excuse, e.g. MC or LOA, will be accepted

- Makeup will be arranged

Online Assessments

Midterm Test

- Exemplify (Supported by CIT)
- CIT will arrange briefing

Practical Exam

- ExamiNIX
- Our customized Linux OS
 - VSC + Coursemology + Cake
 - Boot from USB thumbdrive (install on your own 8GB thumbdrive)

Zoom proctoring

- Using Zoom app on phone or iPad

Lecture 4

Functional Abstraction and Pointers

TIC1001 Introduction to Computing and Programming I

Variables

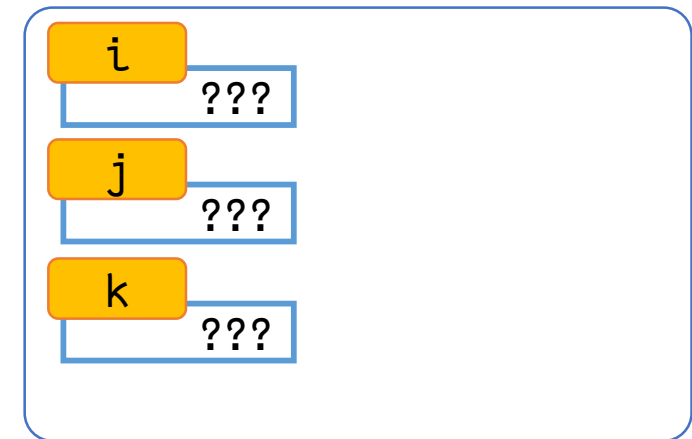
Symbols/Tokens

- that alias a memory location
- strict format: `[_a-zA-Z][_a-zA-Z0-9]*`

```
int i, j, k;
```

```
i = 10
```

main



Variables

Symbols/Tokens

- that alias a memory location
- strict format: `[_a-zA-Z][_a-zA-Z0-9]*`

```
int i, j, k;
```

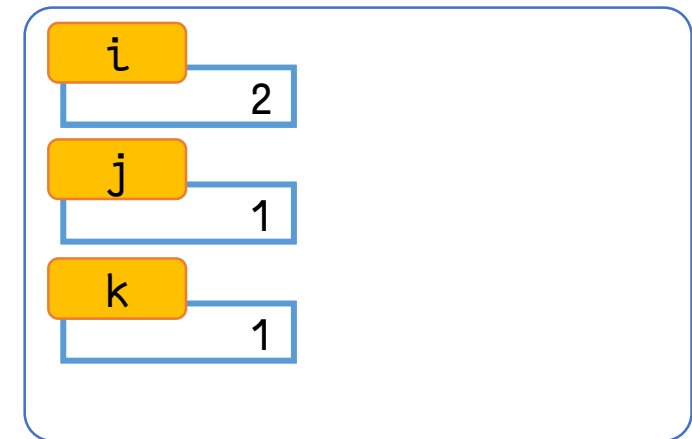
```
i = 1
```

```
j = i
```

```
i = i + 1
```

```
k = j
```

main



Constants

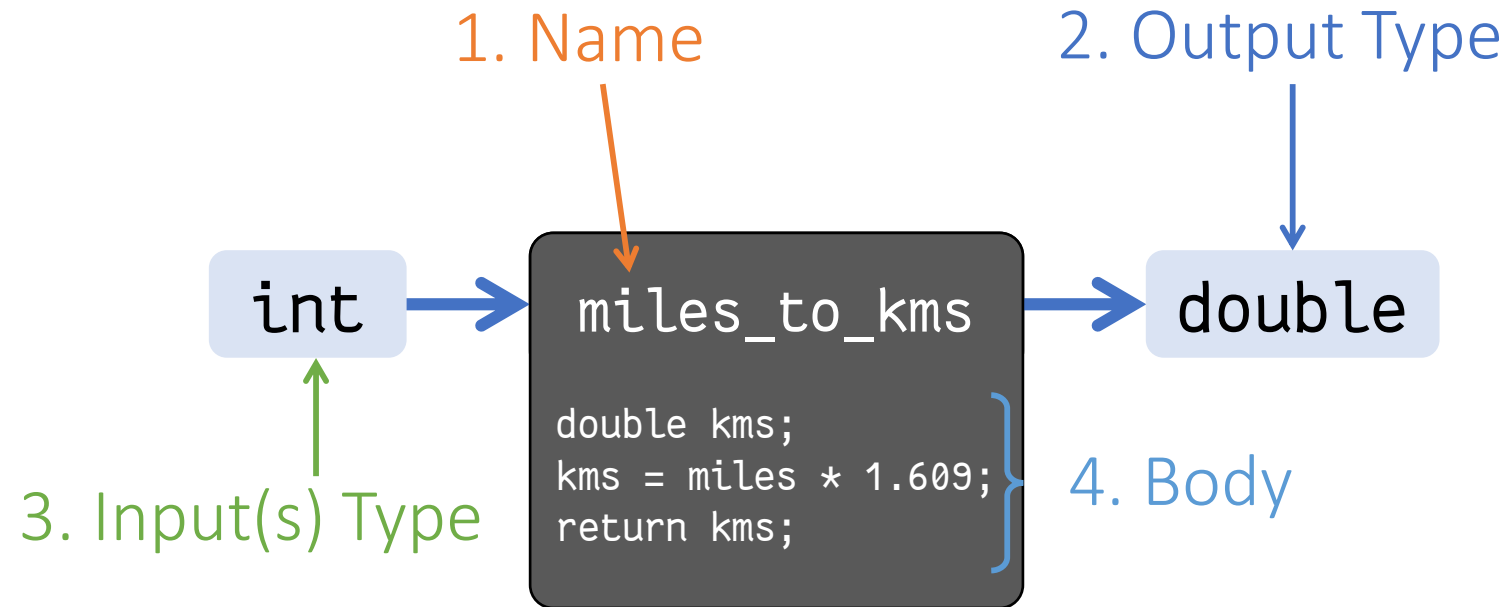
`const` keyword in declaration

```
int const TXT_WIDTH = 80;
```

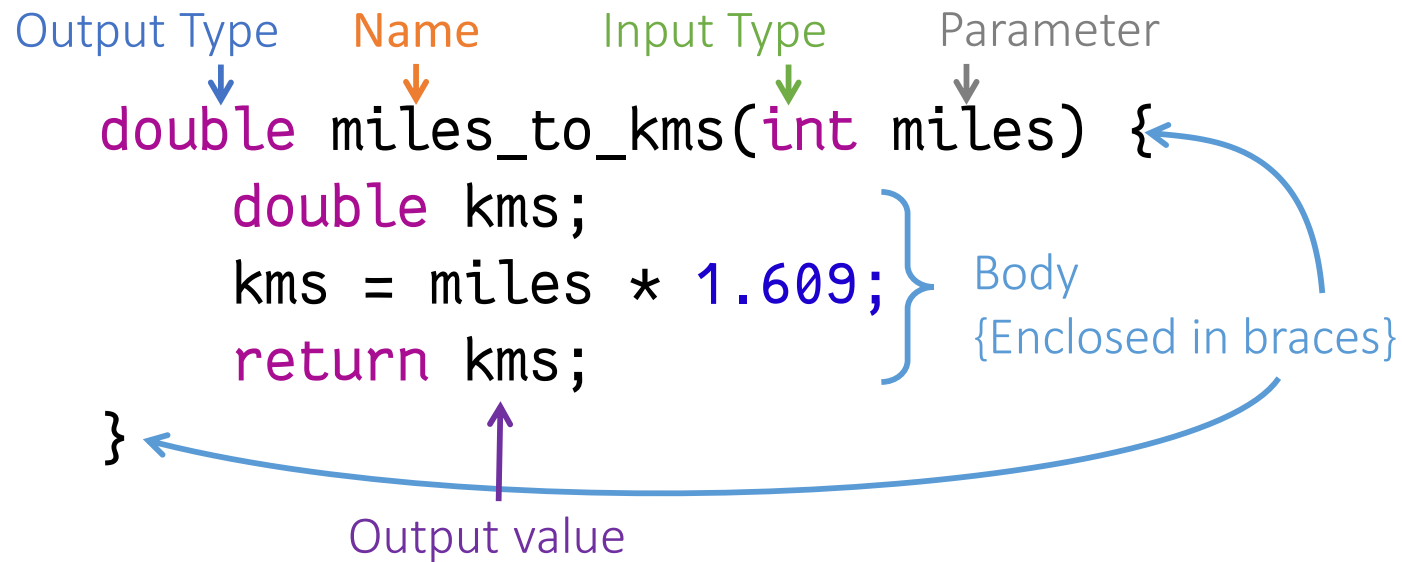
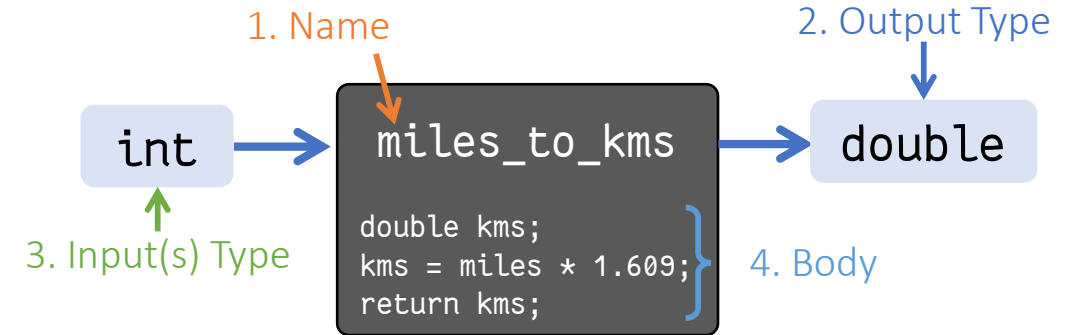
```
const int MAX_ROWS = 25;
```

- `const` can be either before or after type
- Once defined, value cannot change

Components of a Function



In C/C++ Syntax



Consider this function

```
int square(int x) {  
    return x * x;  
}
```

```
square(2);  
square(2 + 5);  
square(square(3));
```

More Functions

```
int sum_of_squares(int x, int y) {  
    return square(x) + square(y);  
}
```

```
sum_of_square(3, 4);
```

```
double hypotenuse(int a, int b) {  
    return sqrt(sum_of_squares(a, b));  
}
```

```
hypotenuse(5, 12);
```

All together now

```
int square(int x) {  
    return x * x;  
}
```

```
int sum_of_squares(int x, int y) {  
    return square(x) + square(y);  
}
```

```
double hypotenuse(int a, int b) {  
    return sqrt(sum_of_squares(a, b));  
}
```

Notice something?

Functional Abstraction

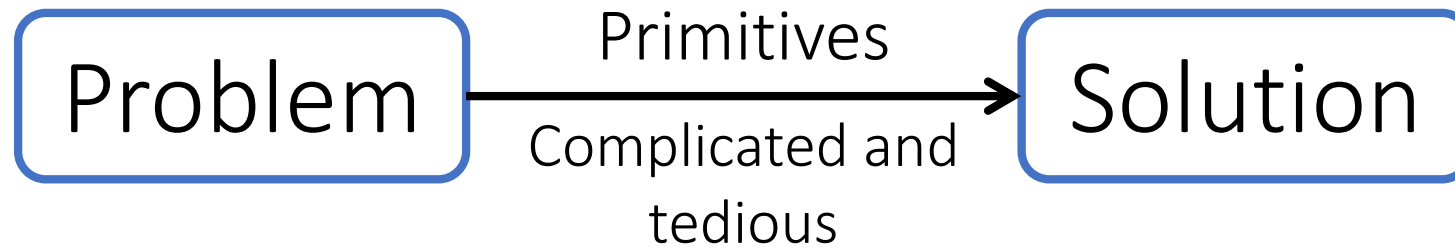
WHY?

Help us to manage complexity

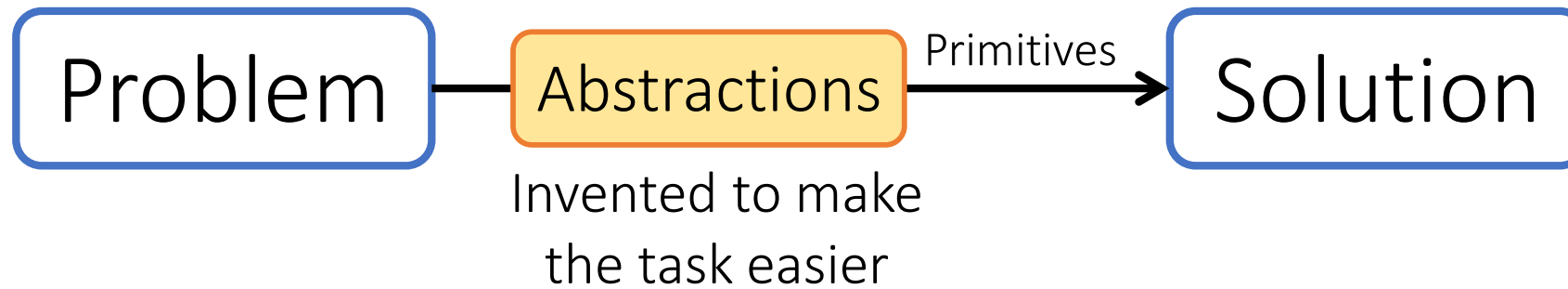
WHY?

Allows us to focus on high level problem solving

Abstractions



Abstractions

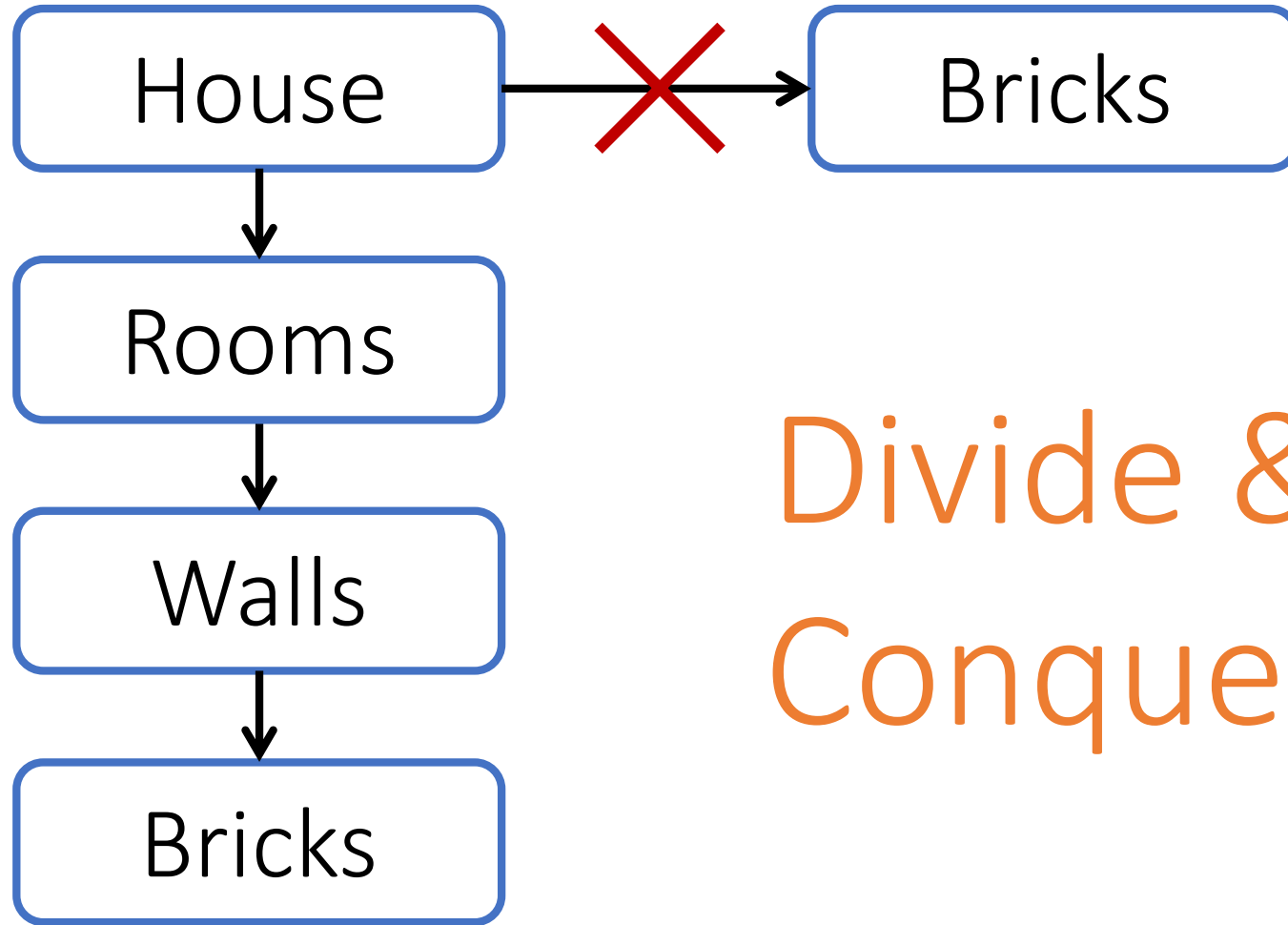


What makes a good abstraction?

Good Abstraction

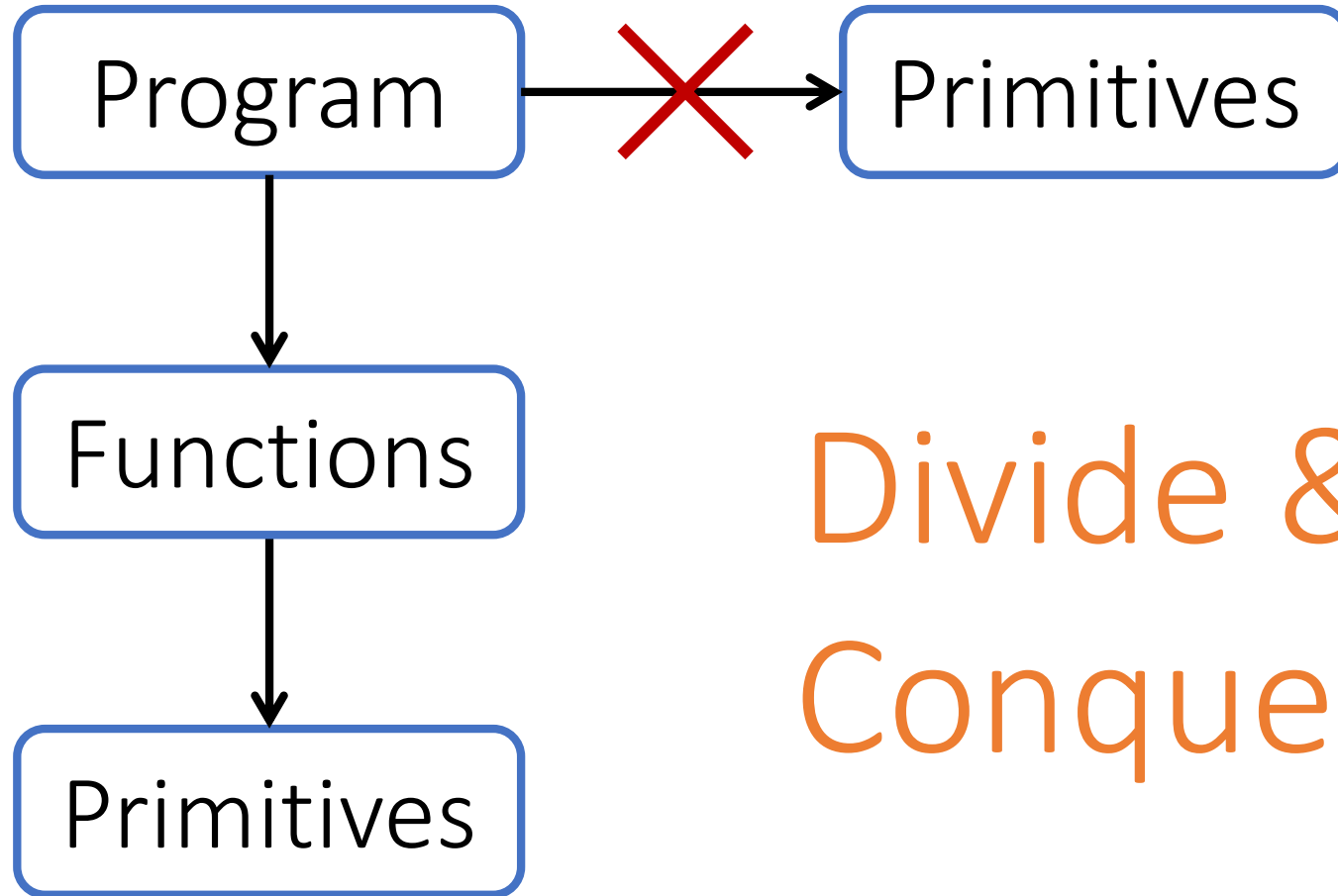
1. Makes it more natural to think about tasks and subtasks

Example



Divide &
Conquer

Programming



Divide &
Conquer

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand

Compare

```
double hypotenuse(int a, int b) {  
    return sqrt((a*a) + (b*b))  
}
```

Versus

```
int square(int x) { return x * x; }  
  
int sum_of_squares(int x, int y) {  
    return square(x) + square(y);  
}  
  
double hypotenuse(int a, int b) {  
    return sqrt(sum_of_squares(a, b));  
}
```

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
 - Function **square** used in **sum_of_squares**.
 - **square** can also be used in calculating area of circle.

Another Example

Function to calculate area of circle given the radius

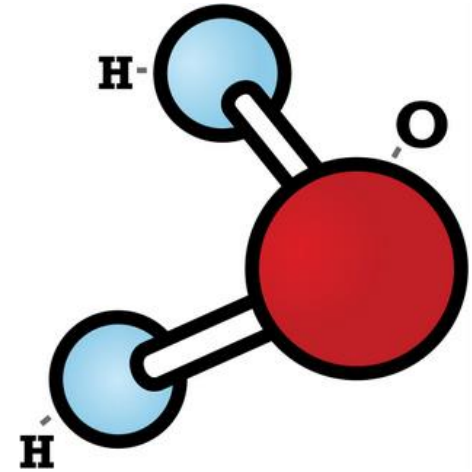
```
double pi = 3.14159
double circle_area_from_radius(int r) {
    return pi * square(r);
}
```

given the diameter:

```
double circle_area_from_diameter(int d) {
    return circle_area_from_radius(d/2);
}
```


Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
5. Hides irrelevant details



Water molecule
represented as 3 balls
Ok for some chemical analyses,
inadequate for others.

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
5. Hides irrelevant details
6. Separates specification from implementation

Recall

Function = Black Box

Separates specification from implementation

Specification: WHAT

Implementation: HOW

Do you know how to drive a car?

Do you know how a car works?

Example

One way to define

```
double square(double x) {  
    return x * x;  
}
```

Another way

```
double square(double x) {  
    return exp(2 * (log(x)));  
}
```

To think about

Why would we want to implement a function
in different ways?

Good Abstraction

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
5. Hides irrelevant details
6. Separates specification from implementation
7. Makes debugging (fixing errors) easier

Where's the bug?


```
double hypotenuse(int a, int b) {  
    return sqrt(sum_of_squares(a, b));  
}
```

```
int sum_of_squares(int x, int y) {  
    return square(x) + square(y);  
}
```

```
int square(int x) {  
    return x + x;  
}
```

Compare with

```
double hypotenuse(int a, int b) {  
    return sqrt((a+a)*(b+b));  
}
```



Error: sum_of_squares
not defined

Scoping



Recall

All statements must belong in a function

- Elements declared in a function, exists only in that function

well not quite...

- Globals
- Preprocessor Directives

Globals

Variables declared outside of functions

- exist for the entire lifetime of the program

Avoid global variables

- they can be modified at any time, at any place

Mostly used as constants

```
const int MAX_AMOUNT = 50000;
```

But still pollutes the namespace

```
int Global = 0;

int main(void) {
    Global += 5;
    printf("%d\n", Global);
}
```

Preprocessor Directives

Lines starting with

– #include, #define, etc.

```
#include <stdio>;
```

```
int main() {  
    printf("Hello World\n");  
    return 0;  
}
```

Error: printf not defined

```
#undef printf
```

```
/* Write formatted output to stdout from the format  
.../* VARARGS1 */
```

```
int  
__printf (const char *format, ...)  
{
```

```
    va_list arg;  
    int done;
```

```
    va_start (arg, format);  
    done = __vfprintf_internal (stdout, format, arg, 0);  
    va_end (arg);
```

```
    return done;  
}
```

```
#undef _IO_printf
```

```
ldbl_strong_alias (__printf, printf);  
ldbl_strong_alias (__printf, _IO_printf);
```

Preprocessor Directives

```
#include <stdio>;
```

```
int main() {  
    printf("Hello World\n");  
    return 0;  
}
```

```
int puts() {  
    ...  
}
```

Error: puts already defined

```
int  
_IO_puts (const char *str)  
{  
    int result = EOF;  
    size_t len = strlen (str);  
    _IO_acquire_lock (stdout);  
    if ((_IO_vtable_offset (stdout) != 0  
        || _IO_fwide (stdout, -1) == -1)  
        && !_IO_sputn (stdout, str, len) == len  
        && !_IO_putc_unlocked ('\n', stdout) != EOF)  
        result = MIN (INT_MAX, len + 1);  
    _IO_release_lock (stdout);  
    return result;  
}  
weak_alias (_IO_puts, puts)  
libc_hidden_def (_IO_puts)
```

Namespace pollution

Preprocessor Directives

C++ uses namespace

```
#include <iostream>
```

```
int main() {  
    cout << "Hello World!" << endl;  
    return 0;  
}
```

Error: cout not defined

Error: endl not defined

Preprocessor Directives

C++ uses namespace

```
#include <iostream>
```

```
int main() {  
    std::cout << "Hello World!" << std::endl;  
    return 0;  
}
```

Can co-exist in different namespace

```
void cout() { ... }
```

Preprocessor Directives

C++ uses namespace

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {  
    cout << "Hello World!" << endl;  
    return 0;  
}
```

Lookup namespace std for unknowns


What is Scope

The region or section of code

- where the variable can be accessed

The scope of a variable

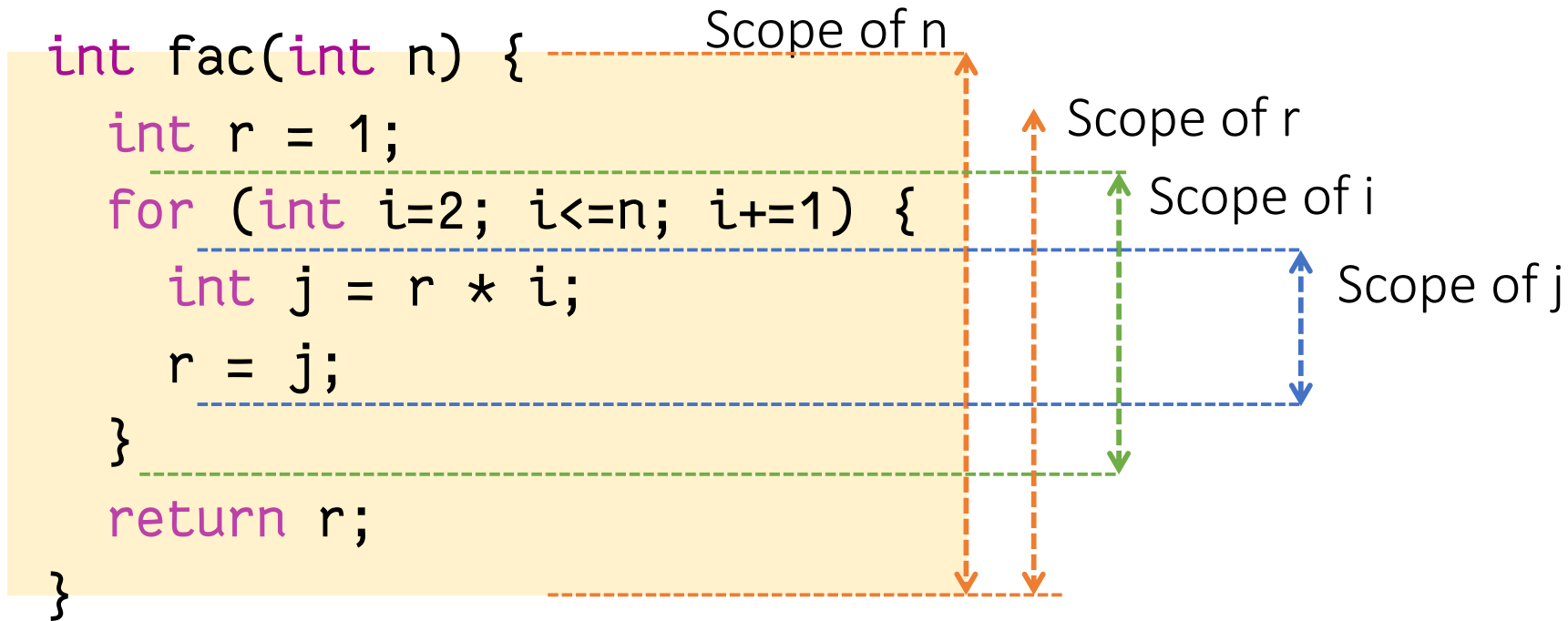
- is the block from when it is declared



A block is denoted by open
and close braces { }


Does not exist before
the declaration

Variable Scope



Shadow Variables

Variables with the same name as one in an outer scope



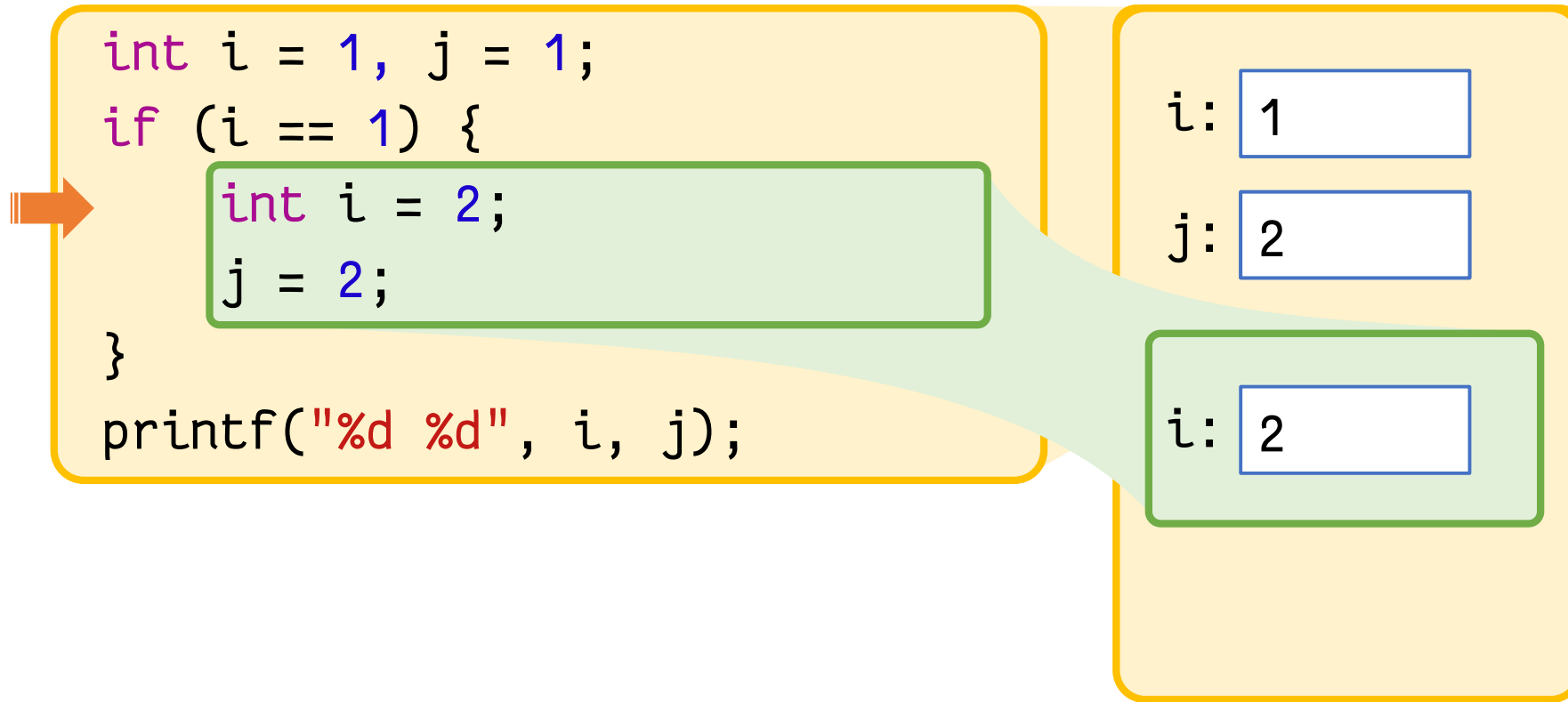
```
int i = 1, j = 1;  
if (i == 1) {  
    int i = 2;  
    j = 2;  
}  
printf("%d %d", i, j);
```

i: 1

j: 1


Variable Scope

Variables with the same name as one in an outer scope



Variable Scope

Variables with the same name as one in an outer scope



```
int i = 1, j = 1;  
if (i == 1) {  
    int i = 2;  
    j = 2;  
}  
printf("%d %d", i, j);
```

i: 1
j: 2

output

1 2

Scope of C/C++ Functions

Order of Definition

Functions have to be defined before they can be used

```
int square(int x) {  
    return x + x;  
}
```

```
int sum_of_squares(int x, int y) {  
    return square(x) + square(y);  
}
```

```
double hypotenuse(int a, int b) {  
    return sqrt(sum_of_squares(a, b));  
}
```

What if it's not possible?

```
int function_A() {  
    ...  
    x = function_B();  
    ...  
}
```

```
int function_B() {  
    ...  
    y = function_A();  
    ...  
}
```

Function Prototype

Declare the function before the definition

- The name
- number and type of inputs
- return type

```
int sum_of_squares(int, int) ;
```

- Almost identical to a function header
- Parameter names are optional
- Note the semicolon at the end

Using a prototype

```
int function_B();
```

```
int function_A() {  
    ...  
    x = function_B();  
    ...  
}
```

```
int function_B() {  
    ...  
    y = function_A();  
    ...  
}
```

When function is called

```
int square(int x) { return x + x; }
```

```
int sum_of_squares(int x, int y)  
{ return square(x) + square(y); }
```

```
double hypotenuse(int x, int y)  
{ return sqrt(sum_of_squares(x, y)); }
```

```
int main(void) {  
    int x = 3, y = 4;  
    double hyp = hypotenuse(x, y);  
    printf("%d\n", hyp);  
    return 0;  
}
```

Parameter Passing

Inputs to the function

Parameter Passing

Three ways of doing it in C++

1. Pass by Value
2. Pass by Reference
3. Pass by Pointer

Important to have a mental model

- of what is going on in the computer (execution state)

Pass by Value

The value is copied over

Passing by value

During a function call

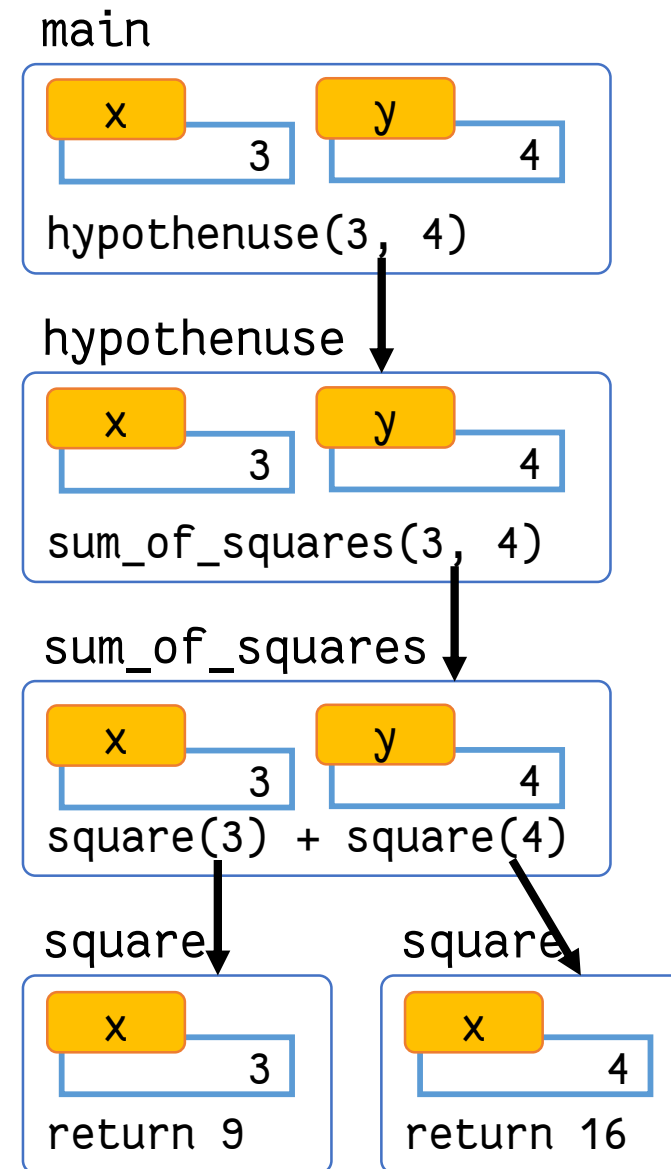
- each argument expression is evaluated
- value is passed to the function by copying
- to the input parameter

Recall, in C/C++ variables are just symbols

- that alias a memory location
- e.g. a postal code is just an alias of a physical address

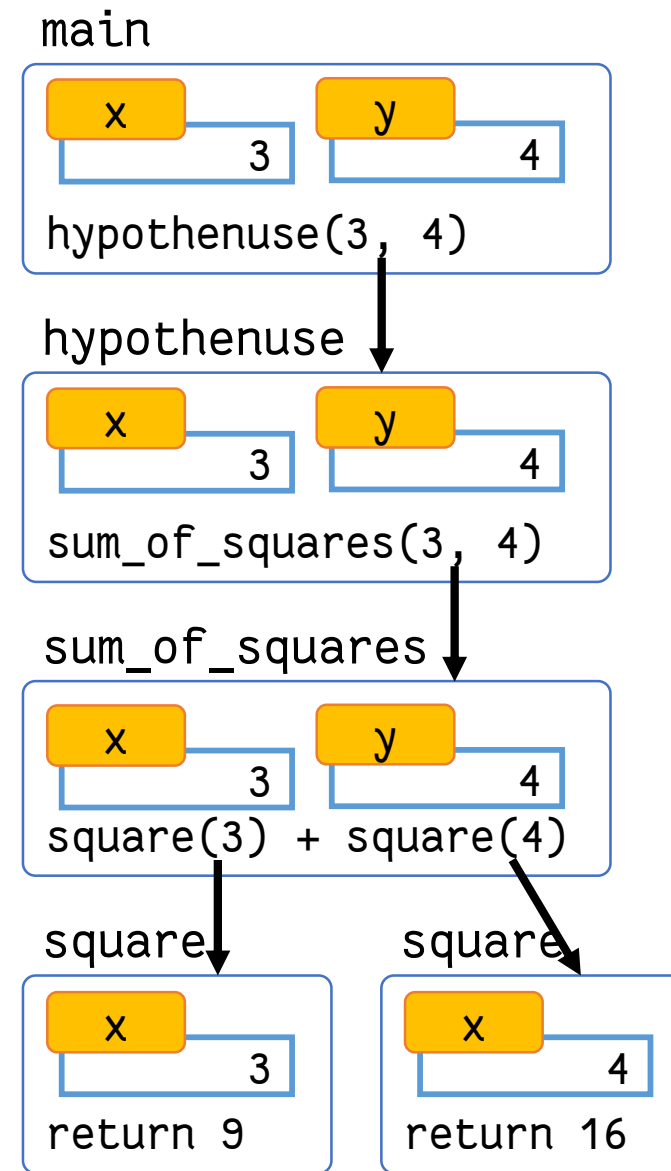
Pass by Value

- hypotenuse **activates**, while main **suspends**
- Value of input arguments are **copied** into the parameters



Lexical Scoping

- The **scope of a variable** is within the block that is declared
- Variables within a block are **local** to that block
- Variables of same name can co-exists **across** functions



Nested Functions

Not allowed in C11 standards

- No ambiguity on variable scoping

Allowed in GNU extension

- lexical scoping rules
- not covered in this class

Functions



- 0, 1 or many inputs
- 0 or 1 output

What if we want more than one output?

- Output a compound object
 - e.g. array or struct
- Modify the inputs using **reference** or **pointers**

Multiple Function Output

Given a time duration in seconds, compute the number of hours, minutes and seconds

```
void split_time(int t, int h, int m, int s) {  
    h = t / 3600;  
    m = (t % 3600) / 60;  
    s = t % 60;  
}
```

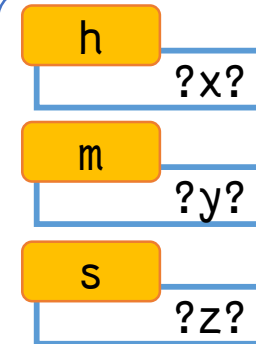
```
int main(void) {  
    int h, m, s;  
    split_time(5000, h, m, s);  
    printf("Duration: %d:%d:%d\n", h, m, s);  
}
```

Recall Pass-by-Value

```
void split_time(int t, int h,  
                int m, int s) {  
    h = t / 3600;  
    m = (t % 3600) / 60;  
    s = t % 60;  
}
```

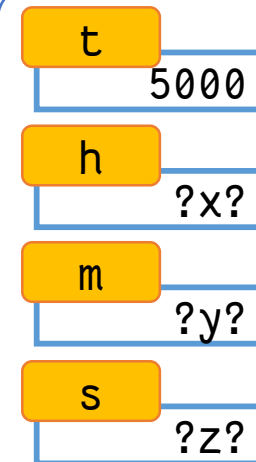
```
int main(void) {  
    int h, m, s;  
    split_time(5000, h, m, s);  
}
```

main



`split_time(5000, h, m, s)`

split_time



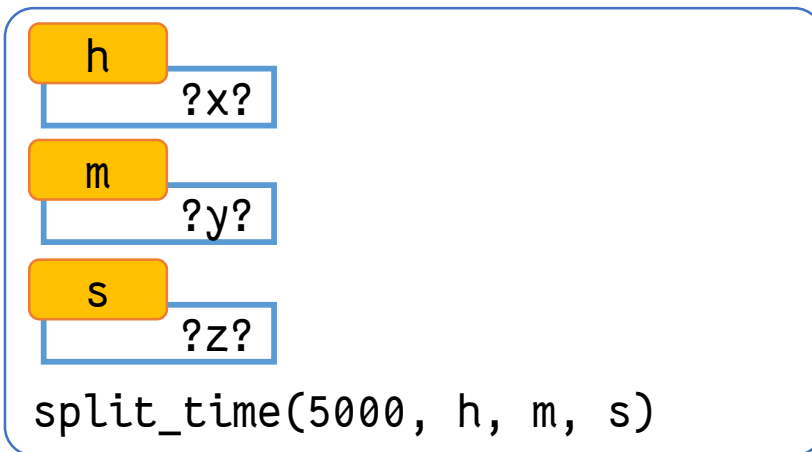
Recall Pass-by-Value

```
void split_time(int t, int h,  
               int m, int s) {  
    h = t / 3600;  
    m = (t % 3600) / 60;  
    t = t % 60;  
}
```

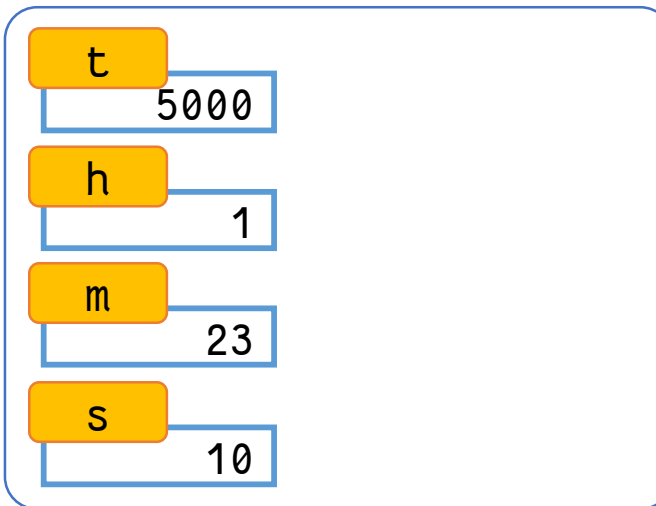
```
int main(void) {  
    int h, m, s;  
    split_time(5000, h, m, s);  
}
```

What happens to the variables in main after `split_time` returns?

main



split_time



Pass by Reference

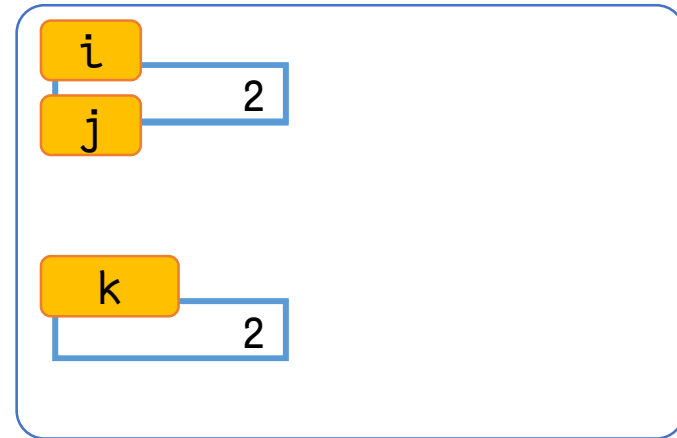
Parameters are references

C++: Aliasing

Variables that refer to the same memory

```
int i = 1, k;  
int& j = i;  
i = i + 1  
k = j  
cout << i << j << k << endl;
```

main



output

222



C++: Parameter Passing

Passing by reference

- In C++, the function parameter can accept a **reference**

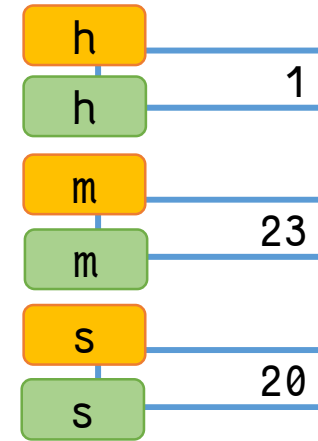
<https://goo.gl/zrhUw4>

Recall split_time

```
void split_time(int t, int &h,  
               int &m, int &s) {  
    h = t / 3600;  
    m = (t % 3600) / 60;  
    s = t % 60;  
}
```

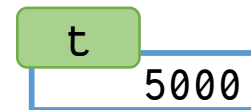
```
int main(void) {  
    int h, m, s;  
    split_time(5000, h, m, s);  
}
```

main



split_time(5000, h, m, s)

split_time



Pointers

Pointers are **variables** that contain the **memory addresses** of another variable

Analogy

Suppose we think of variables as boxes

- name is the label given to the box

```
int age;
```

- value is stored in the box

```
age = 40;
```



But...

Variables are stored in memory

- Think of memory as a shelf
- Variables are not boxes, but a slot in the shelf

Declaring a variable

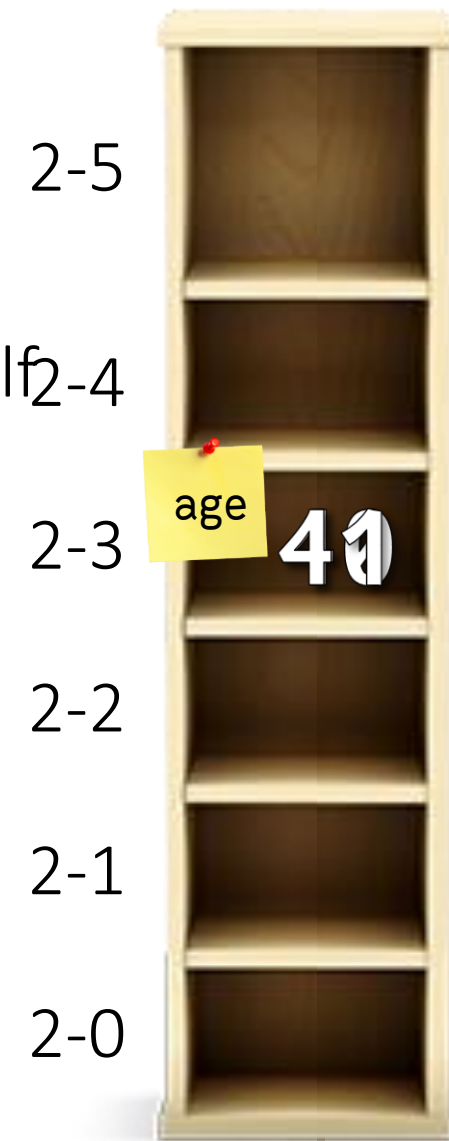
- Assigns a shelf to the variable

```
int age = 40;
```

```
age = age + 1;
```

Each shelf has an address

- Like shelf number



Pointers

Like variables, they occupy space in memory

```
int *ptr;
```

- Declared with an *

But their values are addresses

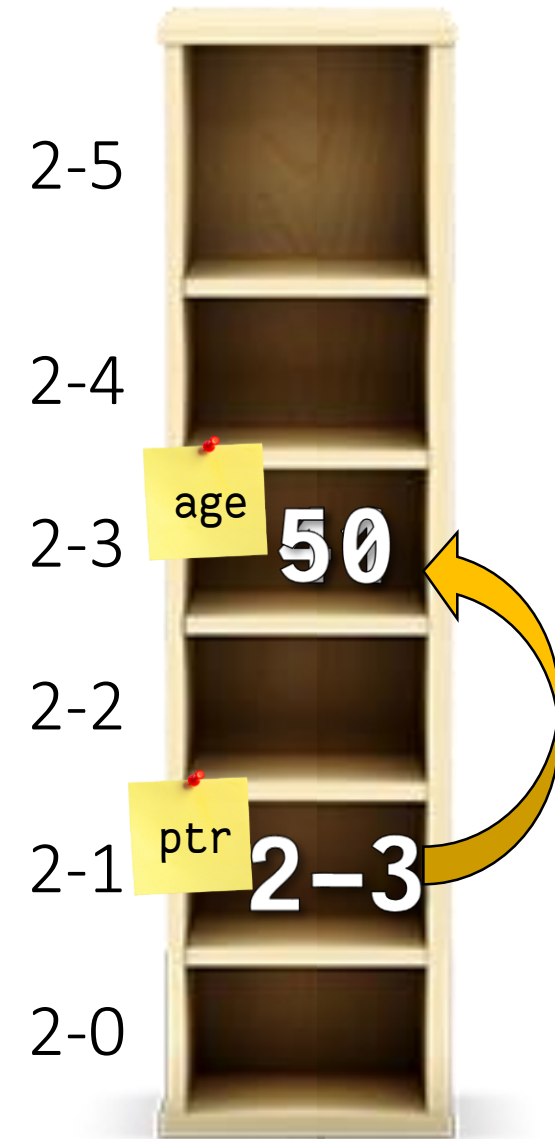
```
ptr = &age;
```

- & means “address of”

Dereference with *

```
*ptr = 50;
```

```
printf("%d\n", age);
```



Simply put

Pointers are declared with a `*`

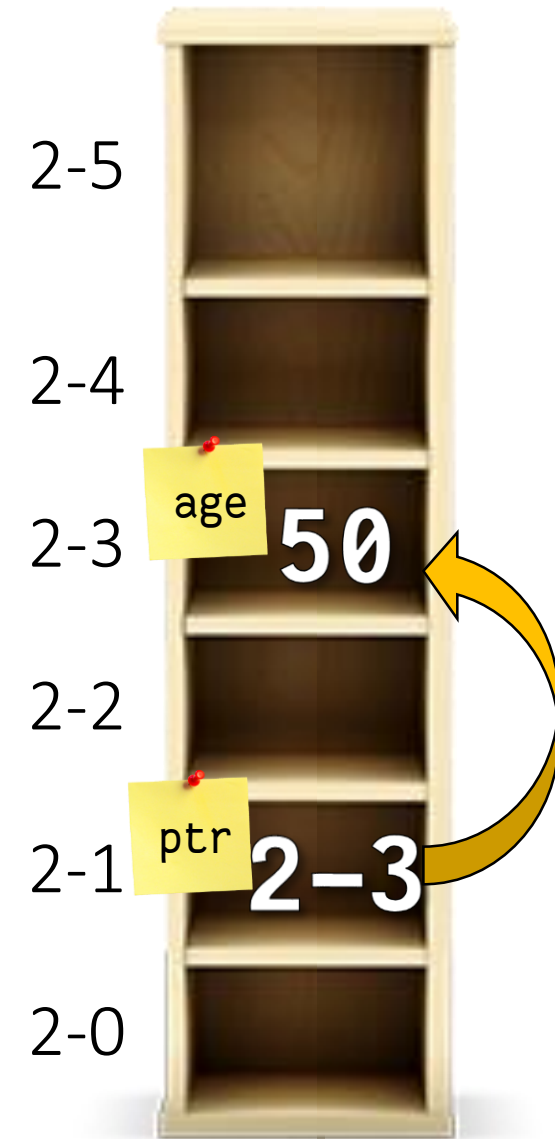
```
int *ptr;
```

`&` means address of variable

```
ptr = &age;
```

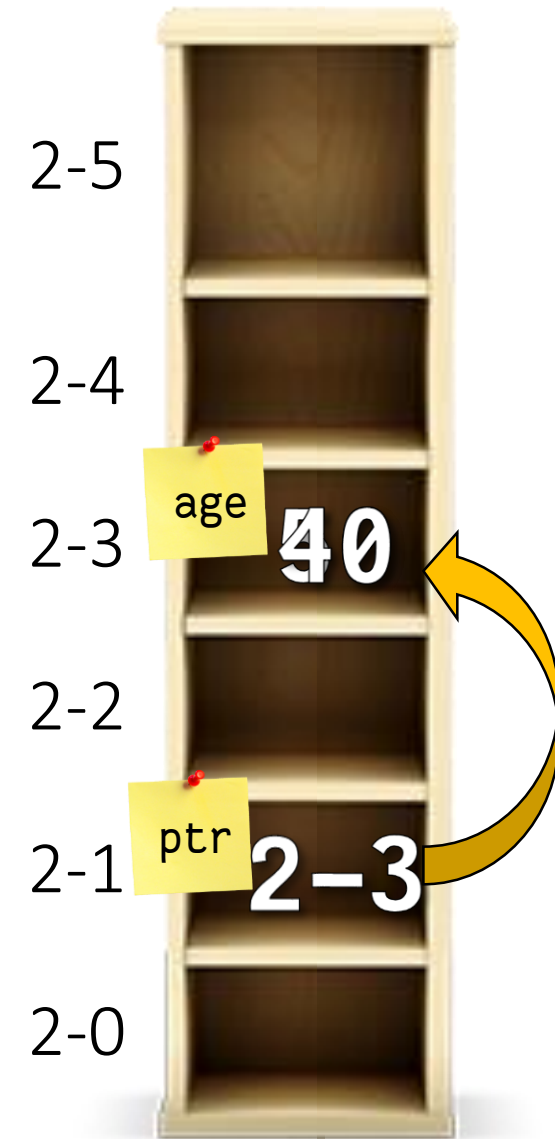
`*` means deference, i.e., go to the address

```
*ptr = 50;
```



Example in code

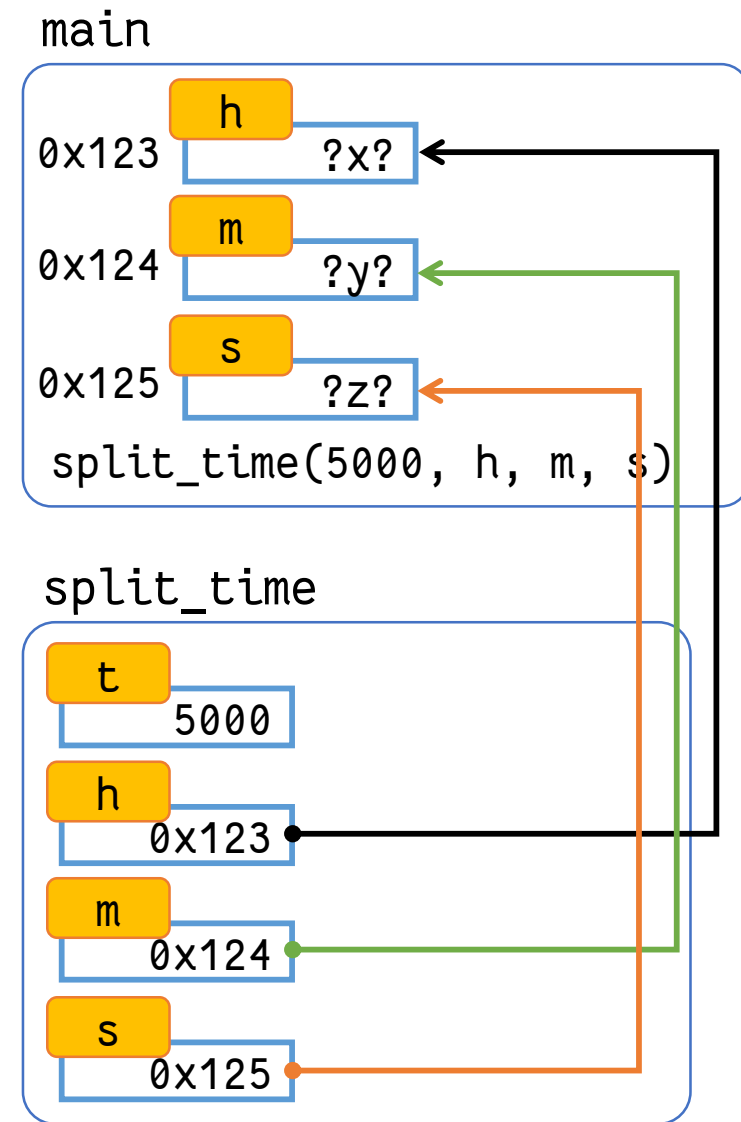
```
int age;      // create a variable age
age = 40;     // assign value 40
int *ptr;     // create pointer to an int
ptr = &age;   // assign address of age
*ptr = 50;    // set pointed location to 50
printf("%d\n", age);
```



Recall split_time

```
void split_time(int t, int *h,  
                int *m, int *s) {  
    *h = t / 3600;  
    *m = (t % 3600) / 60;  
    *s = t % 60;  
}
```

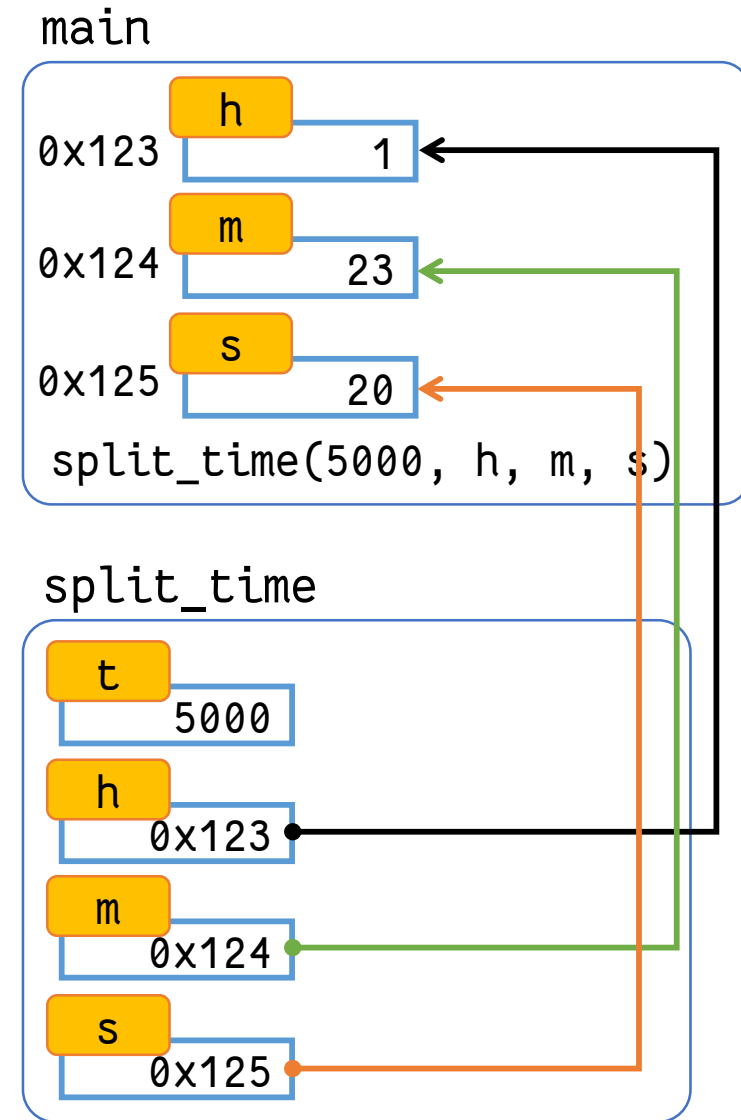
```
int main(void) {  
    int h, m, s;  
    split_time(5000, &h, &m, &s);  
}
```



Passing by Pointer

```
void split_time(int t, int *h,  
               int *m, int *s) {  
    *h = t / 3600;  
    *m = (t % 3600) / 60;  
    *s = t % 60;  
}
```

```
int main(void) {  
    int h, m, s;  
    split_time(5000, &h, &m, &s);  
}
```



Passing by Pointer

Use pointers as paramters (inputs)

```
void split_time(int t, int *h, int *m, int *s);
```

Manipulate pointers in function

```
*h = t / 3600;  
*m = (t % 3600) / 60;  
*s = t / 60;
```

Call function using address of variable

```
split_time(5000, &h, &m, &s);
```

Example: Swapping Variables

Pass by Value

```
void swap(int x, int y) {  
    int temp = x;  
    x = y;  
    y = temp;  
}  
  
int main(void) {  
    int x = 1, y = 5;  
    printf("x=%d, y=%d\n", x, y);  
    swap(x, y);  
    printf("x=%d, y=%d\n", x, y);  
}
```

Example: Swapping Variables

Pass by Reference

```
void swap(int &x, int &y) {  
    int temp = x;  
    x = y;  
    y = temp;  
}  
  
int main(void) {  
    int a = 1, b = 5;  
    printf("a=%d, b=%d\n", a, b);  
    swap(a, b);  
    printf("a=%d, b=%d\n", a, b);  
}
```

Example: Swapping Variables

Pass by Pointer

```
void swap(int *x, int *y) {  
    int temp = *x;  
    *x = *y;  
    *y = temp;  
}  
  
int main(void) {  
    int x = 1, y = 5;  
    printf("x=%d, y=%d\n", x, y);  
    swap(&x, &y);  
    printf("x=%d, y=%d\n", x, y);  
}
```

Summary

Functional Abstraction

- Break large problem into smaller sub-problems
- Modular design of code

Function Call

- Passing by value
- Passing by reference
- Passing by pointer

Alias/Reference

- Variables that alias another
- `&var` creates an reference

Pointers

- Variables that contain addresses
- `*var` creates a pointer
- `&var` gives address of var

Weekly Activities

Saturday

- Tutorial 2
- Lab 2

Just open

- Lecture Training 4
- Tutorial 3
- Problem Set 3
- Bonus Challenge 3.1