Unix Programming

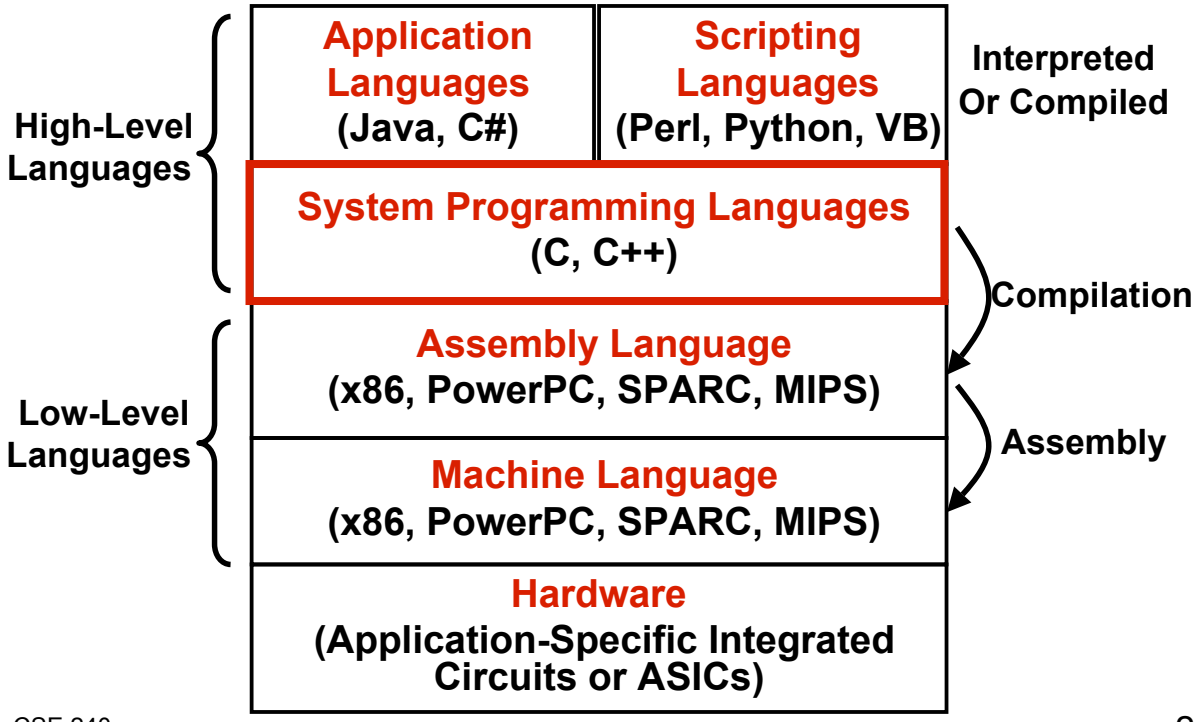
Introduction to C Programming

Nguyen Thanh Hung
Software Engineering Department
Hanoi University of Science and Technology

Outline

- Introduction
- Variables and Operations
- Control Structure
- Functions
- Pointers

Programming Levels



Why High-Level Languages

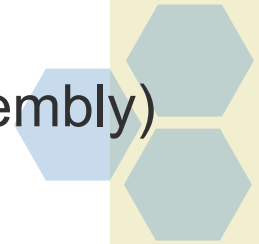
- **Easier than assembly. Why?**
  - Less primitive constructs
  - Variables
  - Type checking
- **Portability**
  - Write program once, run it everywhere
- **Disadvantages**
  - Slower and larger programs (in most cases)
  - Can't manipulate low-level hardware
    - All operating systems have some assembly in them
- **Verdict: assembly coding is rare today**

Our challenge

- **All of you already know Java**
  - We're going to try to cover the basics quickly
  - We'll spend more time on pointers & other C-specific nastiness
- **Created three decades apart**
  - C: 1970s - AT&T Bell Labs
  - C++: 1980s - AT&T Bell Labs
  - Java: 1990s - Sun Microsystems
- **Java and C/C++**
  - Syntactically similar (Java uses C syntax)
  - C lacks many of Java's features
  - Subtly different semantics

C is Similar To Java Without:

- Objects
  - No classes, objects, methods, or inheritance
- Exceptions
  - Check all error codes explicitly
- Standard class library
  - C has only a small standard library
- Garbage collection
  - C requires explicit memory allocate and free
- Safety
  - Java has strong type checking, checks array bounds
  - In C, anything goes
- Portability
  - Source: C code is less portable (but better than assembly)
  - Binary: C compiles to specific machine code



More C and Java differences

- C has a “preprocessor”
  - A separate pre-pass over the code
- Include vs Import
  - Java has *import java.io.*;*
  - C has: *#include <stdio.h>*
  - #include is part of the preprocessor
- Boolean type
  - Java has an explicit boolean type
  - C just uses an “int” as zero or non-zero
  - C’s lack of boolean causes all sorts of trouble
- More differences as we go along…

History of C and Unix

- Unix is the most influential operating system
- First developed in 1969 at AT&T Bell Labs
  - By Ken Thompson and Dennis Ritchie
  - Designed for “smaller” computers of the day
  - Reject some of the complexity of MIT’s Multics
- They found writing in assembly tedious
  - Dennis Ritchie invented the C language in 1973
  - Based on BCPL and B, needed to be efficient (24KB of memory)
- Unix introduced to UC-Berkeley (Cal) in 1974
  - Bill Joy was an early Unix hacker as a PhD student at Cal
  - Much of the early internet consisted of Unix systems Mid-80s • Good, solid TCP/IP for BSD in 1984
- Linux - Free (re)implementation of Unix (libre and gratuit) •
  - Announced by Linus Torvalds in 1991

What is C++?

- **C++ is an extension of C**
  - Also done at AT&T Bell Labs (1983)
  - Backward compatible (good and bad)
  - That is, all C programs are legal C++ programs
- C++ adds many features to C
  - Classes, objects, inheritance
  - Templates for polymorphism
  - A large, cumbersome class library (using templates)
  - Exceptions (not actually implemented for a long time)
  - More safety (though still unsafe)
  - Operator and function overloading
- Thus, many people uses it (to some extent)
  - However, we're focusing on only C, not C++

Program Execution: Compilation vs Interpretation

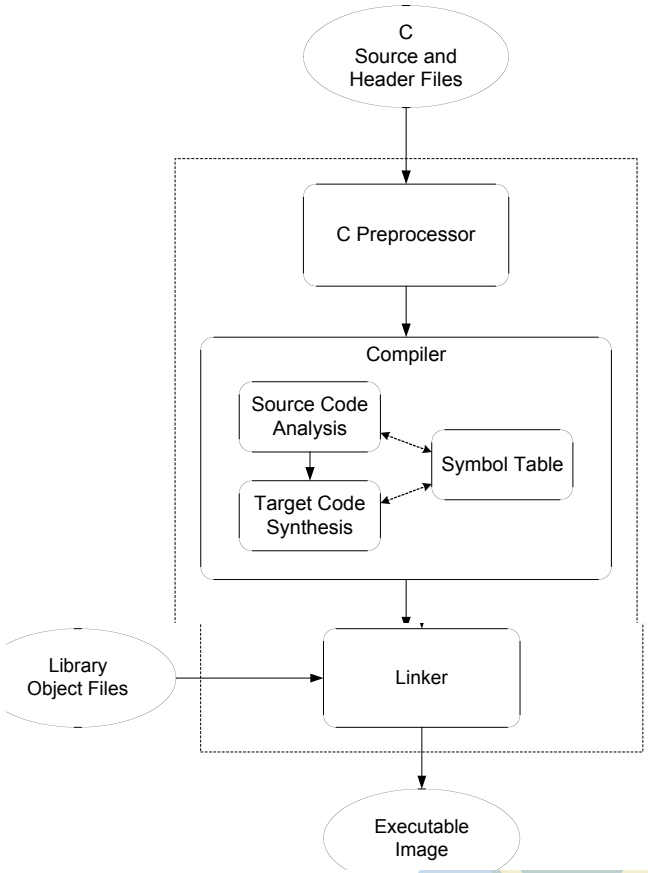
- **Different ways of executing high-level languages**
- **Interpretation**
  - Interpreter: program that executes program statements
    - Directly interprets program (portable but slow)
    - Limited optimization
  - Easy to debug, make changes, view intermediate results
  - Languages: BASIC, LISP, Perl, Python, Matlab
- **Compilation**
  - Compiler: translates statements into machine language
    - Creates executable program (non-portable, but fast)
    - Performs optimization over multiple statements
  - Harder to debug, change requires recompilation
  - Languages: C, C++, Fortran, Pascal
- **Hybrid**
  - Java, has features of both interpreted and compiled languages

Compilation vs Interpretation

- Consider the following algorithm:
  - Get W from the keyboard.
  - X=W+W
  - Y=X+X
  - Z=Y+Y
  - Print Z to screen.
- If interpreting, how many arithmetic operations occur?
- If compiling, we can analyze the entire program and possibly reduce the number of operations.
  - Can we simplify the above algorithm to use a single arithmetic operation?

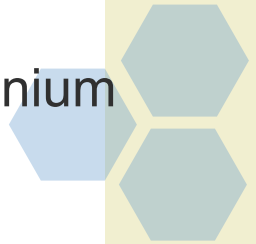
Compiling a C Program

- Entire mechanism is usually called the "compiler"
- **Preprocessor**
  - Macro substitution
  - Conditional compilation
  - "Source-level" transformations
    - Output is still C
- **Compiler**
  - Generates object file
  - Machine instructions
- **Linker**
  - Combine object files (including libraries) into executable image



Compiler

- **Source Code Analysis**
  - "Front end"
  - Parses programs to identify its pieces
    - Variables, expressions, statements, functions, etc.
  - Depends on language (not on target machine)
- **Code Generation**
  - "Back end"
  - Generates machine code from analyzed source
  - May optimize machine code to make it run more efficiently
  - Very dependent on target machine
- **Example Compiler: GCC**
  - The Free-Software Foundation's compiler
  - Many front ends: C, C++, Fortran, Java
  - Many back ends: Intel x86, PowerPC, SPARC, MIPS, Itanium



A Simple C Program

```
#include <stdio.h>
#define STOP 0
void main() {
   /* variable declarations */
   int counter;  /* an integer to hold count values */
   int startPoint; /* starting point for countdown */
   /* prompt user for input */
   printf("Enter a positive number: ");
   scanf("%d", &startPoint); /* read into startPoint */
   /* count down and print count */
   for (counter=startPoint; counter >= STOP; counter--) {
     printf("%d\n", counter);
} }
```

Preprocessor Directives

- **#include <stdio.h>**
  - Before compiling, copy contents of header file (stdio.h) into source code.
  - Header files typically contain descriptions of functions and variables needed by the program.
    - no restrictions -- could be any C source code
- **#define STOP 0**
  - Before compiling, replace all instances of the string "STOP" with the string "0”
  - Called a macro
  - Used for values that won't change during execution, but might change if the program is reused. (Must recompile.)

Comments

- **Begins with /* and ends with */**
  - Can span multiple lines
  - Comments are not recognized within a string
    - example: "my/*don't print this*/string”

    would be printed as: my/*don't print this*/string
- **Begins with // and ends with “end of line”**
  - Single-line comment
  - Introduced in C++, later back-ported to C
- **As before, use comments to help reader, not to confuse or to restate the obvious**

main Function

- Every C program must have a function called main()
  - Starting point for every program
  - Similar to Java's main method
    - public static void main(String[] args)
- The code for the function lives within brackets:

```
void main()
{
   /* code goes here */
}
```

Variable Declarations

- Variables are used as names for data items
- Each variable has a type, tells the compiler:
  - How the data is to be interpreted
  - How much space it needs, etc.
    - *int counter;*
    - *int startPoint;*
- C has similar primitive types as Java
  - int, char, long, float, double
  - More later

Input and Output

- Variety of I/O functions in C Standard Library
  - Must include <stdio.h> to use them
- ***printf("%d\n", counter);***
  - String contains characters to print and formatting directions for variables
  - This call says to print the variable counter as a decimal integer, followed by a linefeed (\n)
- ***scanf("%d", &startPoint);***
  - String contains formatting directions for looking at input
  - This call says to read a decimal integer and assign it to the variable startPoint (Don't worry about the & yet)

More About Output

- Can print arbitrary expressions, not just variables

```
printf("%d\n", startPoint - counter);
```

- Print multiple expressions with a single statement

```
printf("%d %d\n", counter,
        startPoint - counter);
```

- Different formatting options:
  - %d decimal integer
  - %x hexadecimal integer
  - %c ASCII character
  - %f floating-point number

Examples

This code:

```
printf("%d is a prime number.\n", 43);
printf("43 plus 59 in decimal is %d.\n", 43+59);
printf("43 plus 59 in hex is %x.\n", 43+59);
printf("43 plus 59 as a character is %c.\n", 43+59);
```

produces this output:

```
43 is a prime number.
43 plus 59 in decimal is 102.
43 plus 59 in hex is 66.
43 plus 59 as a character is f.
```

Example of Input

- Many of the same formatting characters are available for user input
- scanf("%c", &nextChar);
  - reads a single character and stores it in nextChar
- scanf("%f", &radius);
  - reads a floating point number and stores it in radius
- scanf("%d %d", &length, &width);
  - reads two decimal integers (separated by whitespace), stores the first one in length and the second in width
- Must use ampersand (&) for variables being modified

Compiling and Linking

- Various compilers available
  - cc, gcc
  - includes preprocessor, compiler, and linker
- Lots and lots of options!
  - level of optimization, debugging
  - preprocessor, linker options
  - intermediate files --

  object (.o), assembler (.s), preprocessor (.i), etc.

Outline

- Introduction
- **Variables and Operations**
- Control Structure
- Functions
- Pointers

Basic C Elements

- **Variables**
  - Named, typed data items
- **Operators**
  - Predefined actions performed on data items
  - Combined with variables to form expressions, statements
- **Statements and Functions**
  - Group together operations

Data Types

- C has several basic data types
  - *int* integer (at least 16 bits, commonly 32 bits)
  - *long* integer (at least 32 bits)
  - *float* floating point (at least 32 bits)
  - *double* floating point (commonly 64 bits)
  - *char* character (at least 8 bits)
- Exact size can vary, depending on processor
  - int is supposed to be "natural" integer size
- Signed vs unsigned:
  - Default is 2's complement signed integers
  - Use "unsigned" keyword for unsigned numbers

Variable Names

- Any combination of letters, numbers, and underscore (_)
- Case sensitive
  - "sum" is different than "Sum"
- Cannot begin with a number
  - Usually, variables beginning with underscore are used only in special library routines
- Only first 31 characters are definitely used
  - Implementations can consider more characters if they like

Examples

- Legal
  - i
  - wordsPerSecond
  - words_per_second
  - _green
  - aReally_longName_moreThan31chars
  - aReally_longName_moreThan31characters
- Illegal
  - 10sdigit
  - ten'sdigit
  - done?
  - double

Literals

- Integer
  - 123 /* decimal */
  - -123
  - 0x123 /* hexadecimal */
- Floating point
  - 6.023
  - 6.023e23 /* 6.023 x 1023 */
  - 5E12 /* 5.0 x 1012 */
- Character
  - 'c'
  - '\n' /* newline */

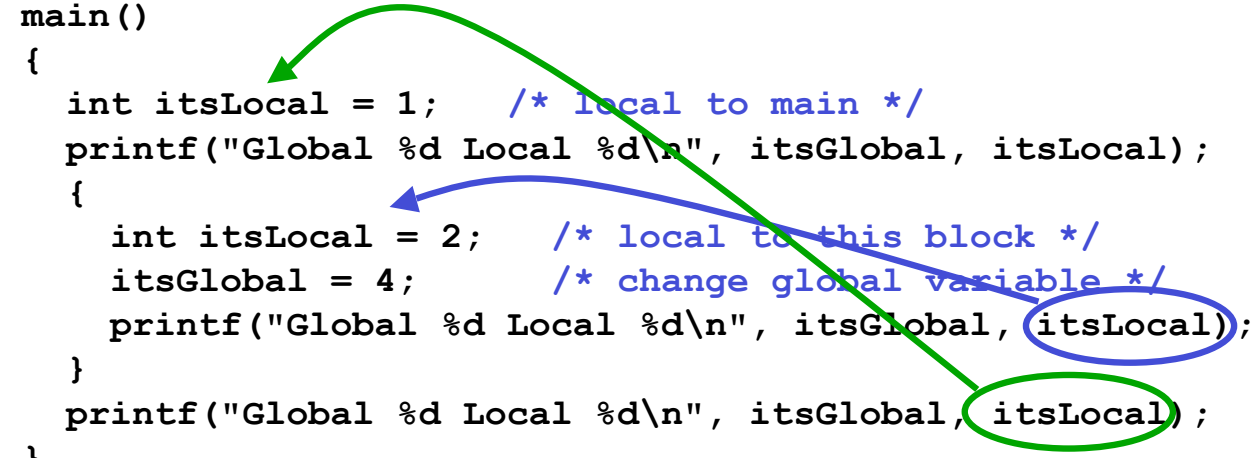
Scope: Global and Local

- Where is the variable accessible?
- **Global**: accessed anywhere in program
- **Local**: only accessible in a particular region
- Compiler infers scope from where variable is declared
  - Programmer doesn't have to explicitly state
- Variable is local to the block in which it is declared
  - Block defined by open and closed braces { }
  - Can access variable declared in any "containing" block
- Global variable is declared outside all blocks

Example

```
#include <stdio.h>
int itsGlobal = 0;

main()
{
  int itsLocal = 1;    /* local to main */
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
  {
    int itsLocal = 2;    /* local to this block */
    itsGlobal = 4;       /* change global variable */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
  }
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
}
```



Output

```
Global 0 Local 1
Global 4 Local 2
Global 4 Local 1
```

Expression

- Any combination of variables, constants, operators, and function calls
  - Every expression has a type, derived from the types of its components (according to C typing rules)
- Examples:

  counter >= *STOP*

  x + *sqrt(y)*

  x & *z* + 3 || 9 - *w*-- % 6

Statement

- Expresses a complete unit of work
  - Executed in sequential order
- Simple statement ends with semicolon

```
z = x * y; /* assign product to z */
y = y + 1; /* after multiplication */
;  /* null statement */
```

- Compound statement formed with braces
  - Syntactically equivalent to a simple statement

```
{ z = x * y; y = y + 1; }
```

Operators

- Three things to know about each operator
- **(1) Function**
  - What does it do?
- **(2) Precedence**
  - In which order are operators combined?
  - Example:

  "a * b + c * d" is the same as "(a * b) + (c * d)" because multiply (*) has a higher precedence than addition (+)
- **(3) Associativity**
  - In which order are operators of the same precedence combined?
  - Example:
  - "a - b - c" is the same as "(a - b) - c" because add/sub associate left-to-right

Assignment Operator

- Changes the value of a variable

```
x = x + 4;
```

1. Evaluate right-hand side.

2. Set value of left-hand side variable to result.

Assignment Operator

- All expressions evaluate to a value, even ones with the assignment operator
- For assignment, the result is the value assigned
  - Usually (but not always) the value of the right-hand side
    - Type conversion might make assigned value different than computed value
- Assignment associates right to left.
  - y = x = 3;
- y gets the value 3, because (x = 3) evaluates to the value 3
  - y = (x = 3);

Arithmetic-Logical-Relational Operators

Symbol	Operation	Usage
*	multiply	`x * y`
/	divide	`x / y`
%	modulo	`x % y`
+	addition	`x + y`
–	subtraction	`x - y`

All associate left to right

`* / %` **have higher precedence than** `+ -`

Example

- **2 + 3 * 4 versus**
- **(2 + 3) * 4**

Symbol	Operation	Usage
!	logical NOT	`!x`
&&	logical AND	`x && y`
||	logical OR	`x || y`

Symbol	Operation	Usage
>	greater than	`x > y`
>=	greater than or equal	`x >= y`
<	less than	`x < y`
<=	less than or equal	`x <= y`
==	equal	`x == y`
!=	not equal	`x != y`

Arithmetic Expressions

- If mixed types, smaller type is "promoted" to larger

  x + 4.3

  if x is int, converted to double and result is double

- Integer division -- fraction is dropped

  x /3

  if x is int and x=5, result is 1 (not 1.666666...)

- Modulo -- result is remainder

  x %3

  if x is int and x=5, result is 2

Assignment vs Equality

- Don't confuse equality (==) with assignment (=)

```
int x = 9;
int y = 10;
if (x == y) {
        printf("not executed\n");
}
if (x = y) {
        printf("%d %d", x, y);
}
```

- Result: "10 10" is printed. Why?
- Compiler will not stop you! (What happens in Java?)

Special Operators: ++ and --

- Changes value of variable before (or after) its value is used in an expression

Symbol	Operation	Usage
++	postincrement	`x++`
--	postdecrement	`x--`
++	preincrement	`++x`
--	predecrement	`--x`

- Pre: Increment/decrement variable before using its value
- Post: Increment/decrement variable after using its value

Using ++ and --

```
x = 4;
y = x++;
```

Results: x = 5, y = 4

(because x is incremented after assignment)

```
x = 4;
y = ++x;
```

Results: x = 5, y = 5

(because x is incremented before assignment)

Special Operators: +=, *=, etc.

- Arithmetic and bitwise operators can be combined with assignment operator

Statement	Equivalent assignment
`x += y;`	`x = x + y;`
`x -= y;`	`x = x - y;`
`x *= y;`	`x = x * y;`
`x /= y;`	`x = x / y;`
`x %= y;`	`x = x % y;`
`x &= y;`	`x = x & y;`
`x |= y;`	`x = x | y;`
`x ^= y;`	`x = x ^ y;`

Outline

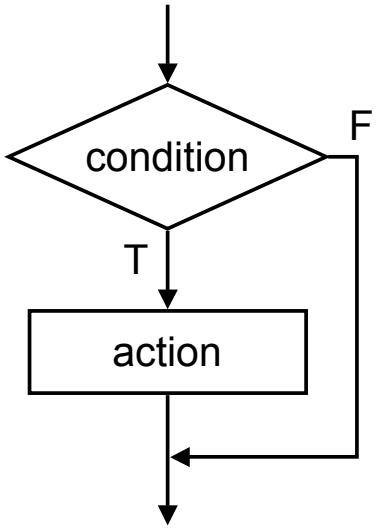
- Introduction
- Variables and Operations
- **Control Structure**
- Functions
- Pointers

Control Structures

- **Conditional**
  - Making decision about which code to execute, based on evaluated expression
  - if
  - if-else
  - switch
- **Iteration**
  - Executing code multiple times, ending based on evaluated expression
  - while
  - for
  - do-while

If

```
if (condition)
  action;
```



Condition is a C expression,
which evaluates to TRUE (non-zero) or FALSE (zero).
Action is a C statement,
which may be simple or compound (a block).

Example If Statement

```
if (x <= 10)
   y = x * x + 5;
```

```
 if (x <= 10) {
   y = x * x + 5;
   z = (2 * y) / 3;
}
```

```
if (x <= 10)
  y = x * x + 5;
  z = (2 * y) / 3;
```

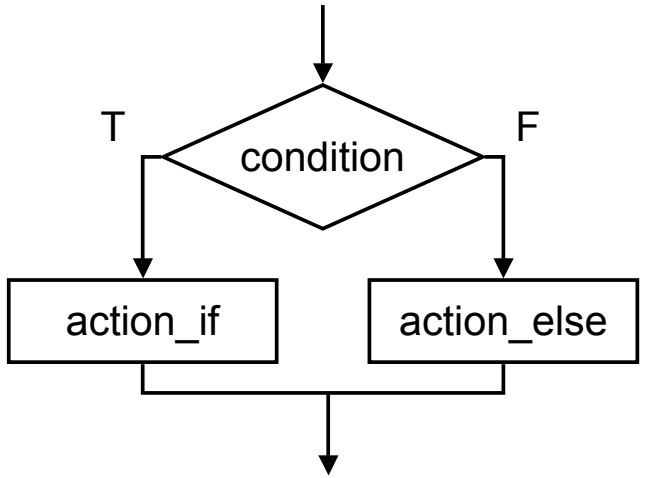
```
if (0 <= age && age <= 11) {
  kids = kids + 1;
}
if (month == 4 || month == 6 ||
    month == 9 || month == 11) {
  printf("The month has 30 days.\n");
}
if (x = 2) {
  y = 5;
}
```

Common C error, assignment (=) Versus equality (==)

This is a common programming error (= instead of ==), not caught by compiler because it's syntactically correct.

if-else

```
if (condition)
  action_if;
else
  action_else;
```



Else allows choice between two mutually exclusive actions without re-testing condition.

Matching else with if

- Else is always associated with closest unassociated if

```
if (x != 10)
  if (y > 3)
    z = z / 2;
  else
    z = z * 2;
```

is the same as...

```
if (x != 10) {
  if (y > 3)
    z = z / 2;
  else
    z = z * 2;
}
```

is NOT the same as...

```
if (x != 10) {
  if (y > 3)
    z = z / 2;
}
else
  z = z * 2;
```

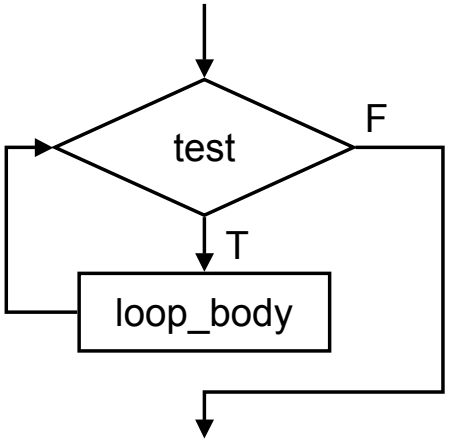
Solution: *always* use braces (avoids the problem entirely)

if and else

```
if (month == 4 || month == 6 || month == 9 ||
    month == 11) {
  printf("Month has 30 days.\n");
} else if (month == 1 || month == 3 ||
           month == 5 || month == 7 ||
           month == 8 || month == 10 ||
           month == 12) {
  printf("Month has 31 days.\n");
} else if (month == 2) {
  printf("Month has 28 or 29 days.\n");
} else {
  printf("Don't know that month.\n");
}
```

While

```
while (test)
  loop_body;
```



Executes loop body as long as test evaluates to TRUE (non-zero)
*Note: Test is evaluated **before** executing loop body*

Infinite Loops

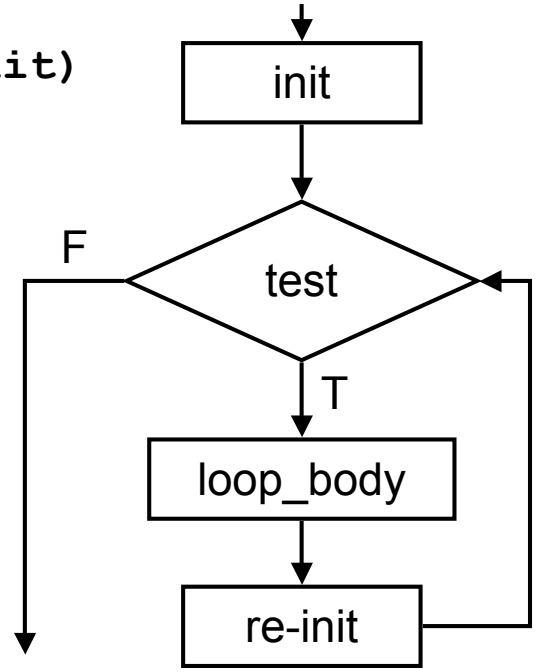
The following loop will never terminate:

```
x = 0;
while (x < 10) {
  printf("%d ", x);
}
```

- Loop body does not change condition...
  - ...so test is never false
- Common programming error that can be difficult to find

For

```
for (init; end-test; re-init)
  statement
```



Executes loop body as long as test evaluates to TRUE (non-zero). Initialization and re-initialization code included in loop statement.

*Note: Test is evaluated **before** executing loop body*

Nested Loops

- **Loop body can (of course) be another loop**

```
/* print a multiplication table */
 for (mp1 = 0; mp1 < 10; mp1++) {
  for (mp2 = 0; mp2 < 10; mp2++) {
    printf("%d\t", mp1*mp2);
  }
  printf("\n");
 }
```

Another Nested Loop

- Here, test for the inner loop depends on counter variable of outer loop

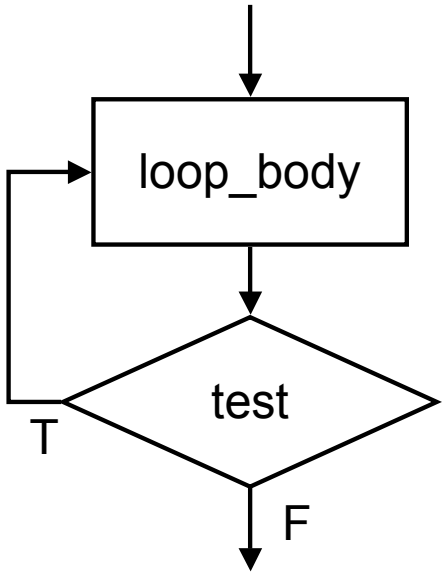
```
for (outer = 1; outer <= input; outer++) {
  for (inner = 0; inner < outer; inner++) {
     sum += inner;
   }
}
```

For vs. While

- For loop is preferred for counter-based loops
  - Explicit counter variable
  - Easy to see how counter is modified each loop
- While loop is preferred for sentinel-based loops
  - Test checks for sentinel value.
- Either kind of loop can be expressed as other, so really a matter of style and readability

Do-While

```
do
  loop_body;
while (test);
```



Executes loop body as long as test evaluates to TRUE (non-zero).

*Note: Test is evaluated **after** executing loop body*

Break and Continue

- **break;**
  - used only in switch statement or iteration statement
  - passes control out of the “nearest” (loop or switch) statement containing it to the statement immediately following
  - usually used to exit a loop before terminating condition occurs (or to exit switch statement when case is done)
- **continue;**
  - used only in iteration statement
  - terminates the execution of the loop body for this iteration
  - loop expression is evaluated to see whether another iteration should be performed
  - If for loop, also executes the re-initializer

Example

What does the following loop do?

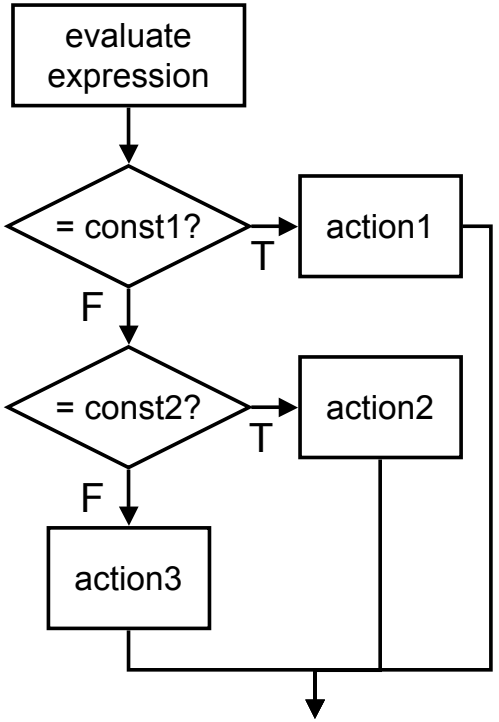
```
for (i = 0; i <= 20; i++) {
  if (i%2 == 0) {
    continue;
  }
  printf("%d ", i);
}
```

What would be an easier way to write this?

What happens if `break` instead of `continue`?

Switch

```
switch (expression) {
 case const1:
   action1;
   break;
 case const2:
   action2;
   break;
 default:
   action3;
 }
```



Alternative to long if-else chain.
If break is not used, then
case "falls through" to the next.

Switch Example

```
/* same as month example for if-else */
switch (month) {
 case 4:
 case 6:
 case 9:
 case 11:
   printf("Month has 30 days.\n");
   break;
 case 1:
 case 3:
 /* some cases omitted for brevity...*/
   printf("Month has 31 days.\n");
   break;
 case 2:
   printf("Month has 28 or 29 days.\n");
   break;
 default:
   printf("Don't know that month.\n");
}
```

More About Switch

- Case expressions must be constant

```
case i: /* illegal if i is a variable */
```

- If no break, then next case is also executed

```
switch (a) {
 case 1:
   printf("A");
 case 2:
   printf("B");
 default:
   printf("C");
}
```

If a is 1, prints "ABC".
If a is 2, prints "BC".
Otherwise, prints "C".

Enumerations

- Keyword enum declares a new type
  - enum colors { RED, GREEN, BLUE, GREEN, YELLOW, MAUVE };
  - RED is now 0, GREEN is 1, etc.
  - Gives meaning to constants, groups constants

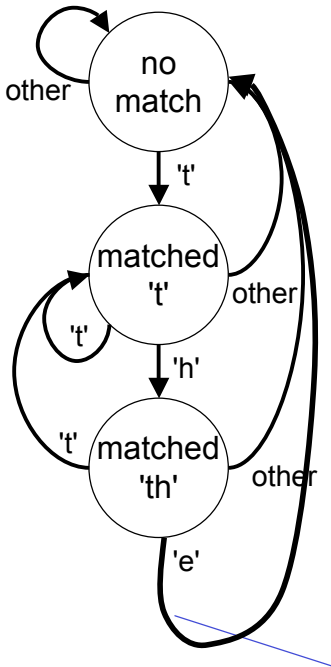
```
enum colors house_color;
house_color = get_color();
switch (house_color) {
        case RED:
                /* code here */
        break;
                /* more here... */
}
```

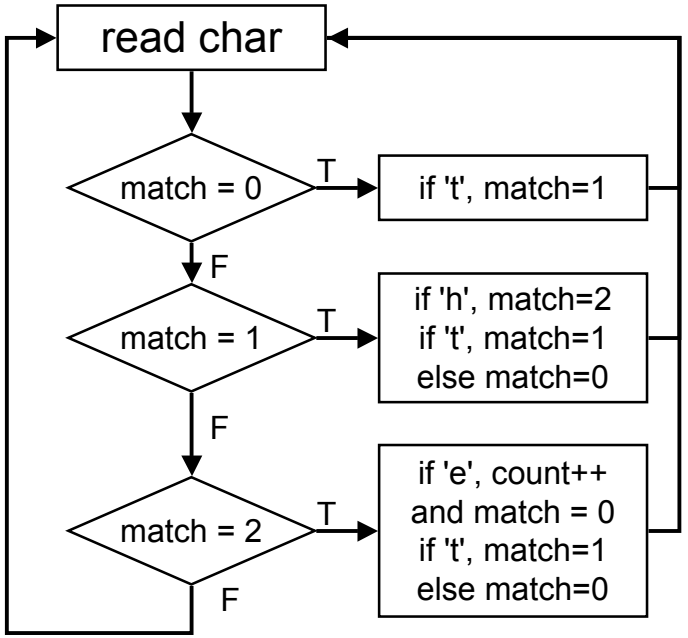
Example: Searching for Substring

- Have user type in a line of text (ending with linefeed) and print the number of occurrences of "the"
- Reading characters one at a time
  - Use the getchar() function—returns a single character
- Don't need to store input string; look for substring as characters are being typed
  - Similar to state machine:

  based on characters seen, move toward success state or move back to start state
  - Switch statement is a good match to state machine

Substring: State machine to flow chart





Substring: Code (Part 1)

```
#include <stdio.h>
 enum state { NO_MATCH, ONE_MATCH, TWO_MATCHES };
 main()
 {
   char key;       /* input character from user */
   int match = NO_MATCH ; /* state of matching */
   int count = 0; /* number of substring matches */
   /* Read character until newline is typed */
   key = getchar();
   while (key != '\n') {
     /* Action depends on number of matches so far */
     switch (match) {
        …..

     }
     key = getchar();
   }
   printf("Number of matches = %d\n", count);
 }
```

Substring: Code (Part 2)

```
case NO_MATCH:  /* starting - no matches yet */
        if (key == 't') {
          match = ONE_MATCH;
        } else {
          match = NO_MATCH;
}
break;
case ONE_MATCH:  /* 't' has been matched */
        if (key == 'h') {
          match = TWO_MATCHES;
        } else if (key == 't') {
          match = ONE_MATCH;
        } else {
          match = NO_MATCH;
        }
break;
```

Substring: Code (Part 3)

```
case TWO_MATCHES:  /* 'th' has been matched */
        if (key == 'e') {
          count++;   /* increment count */
          match = NO_MATCH; /* go to starting point */
        } else if (key == 't') {
          match = ONE_MATCH;
        } else {
          match = NO_MATCH;
}
break;
```

Outline

- Introduction
- Variables and Operations
- Control Structure
- **Functions**
- Pointers

Function

- Smaller, simpler, subcomponent of program
- Provides abstraction
  - Hide low-level details
  - Give high-level structure to program, easier to understand overall program flow
  - Enables separable, independent development
- C functions
  - Zero or multiple arguments (or parameters) passed in
  - Single result returned (optional)
  - Return value is always a particular type

Example of High-Level Structure

```
void main() {
  setup_board();  /* place pieces on board */
  determine_sides(); /* choose black/white */
   /* Play game */
   while (no_outcome_yet()){
     whites_turn();
     blacks_turn();
   }
}
```

Structure of program is evident, even without knowing implementation

Function in C

Definition

type of return value

name of function

types of all arguments

exits function with specified return value

```
int factorial(int n)
{
  int i;
  int result = 1;
  for (i = 1; i <= n; i++) {
    result = result * i;
  }
  return result;
}
```

Function call -- used in expression

```
a = x + factorial(f + g);
```

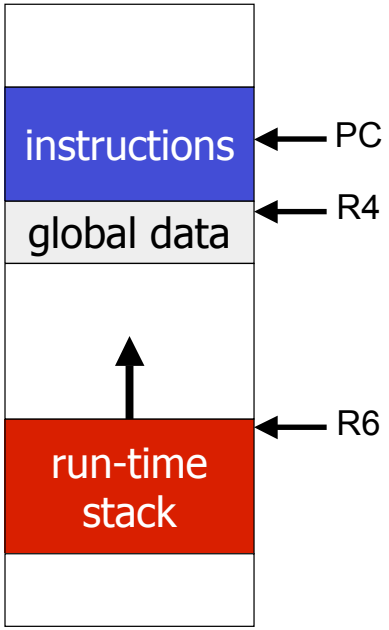
1. evaluate arguments

2. execute function

3. use return value in expression

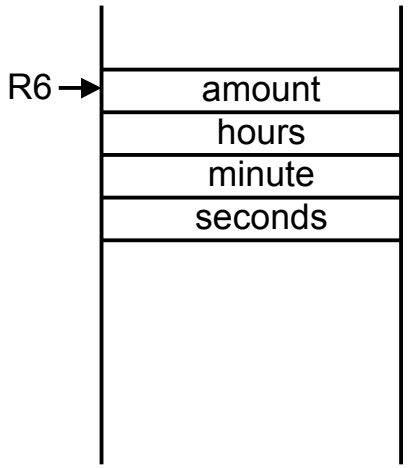
Allocating Space for Variables

- Global data section
  - All global variables stored here (actually all static variables)
  - R4 points to beginning
- Run-time stack
  - Used for local variables
  - R6 points to top of stack
  - New frame for each block (goes away when block exited)
- Offset = distance from beginning of storage area



Local Variable Storage

- Local variables stored in activation record (stack frame)
- Symbol table "offset" gives the distance from the base of the frame
  - A new frame is pushed on the run-time stack each time block is entered
  - R6 is the stack pointer – holds address of current top of run-time stack
  - Because stack grows downward, stack pointer is the smallest address of the frame, and variable offsets are >= 0.



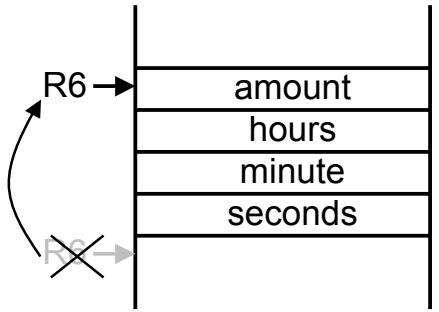
Symbol Table

- Compiler tracks each symbol (identifiers) and its location
  - In assembler, all identifiers were labels
  - In compiler, identifiers are variables
- Compiler keeps more information
  - Name (identifier)
  - Type
  - Location in memory
  - Scope

Name	Type	Offset	Scope
amount	double	0	main
hours	int	1	main
minutes	int	2	main
seconds	int	3	main

Symbol Table Example

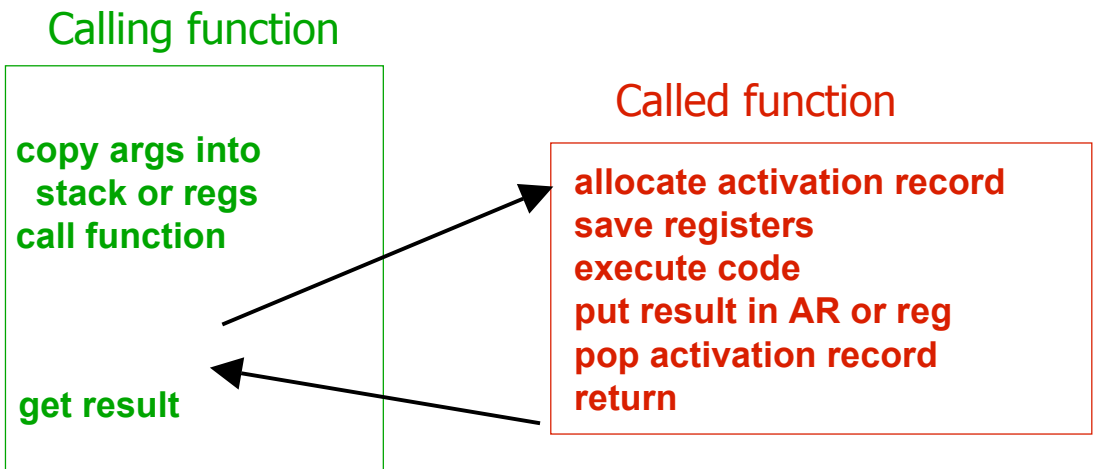
```
int main() {
    int seconds;
    int minutes;
    int hours;
    double amount;
    ...
}
```



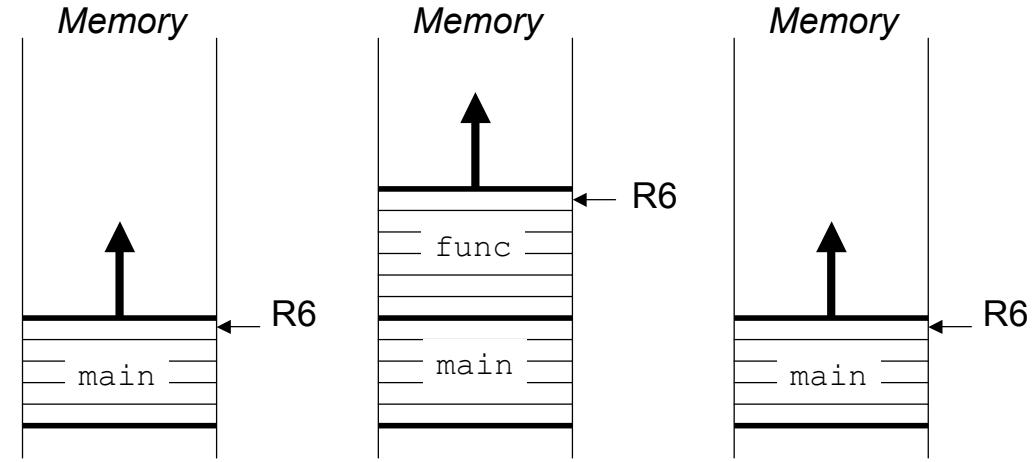
Name	Type	Offset	Scope
amount	double	0	main
hours	int	1	main
minutes	int	2	main
seconds	int	3	main

Implementing Functions

- Activation record
  - Information about each function, including arguments and local variables
  - Also stored on run-time stack



Run-Time Stack for Functions

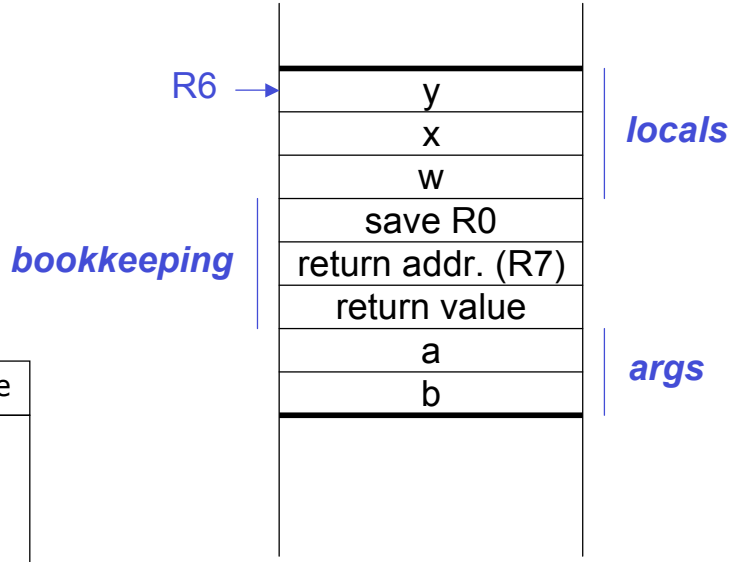


Before call ***During call*** ***After call***

Activation Record

```
int func(int a, int b)
{
  int w, x, y;
  .
  .
  .
  return y;
}
```

Name	Type	Offset	Scope
b	int	7	func
a	int	6	func
"ret. value"	int	5	func
w	int	2	func
x	int	1	func
y	int	0	func



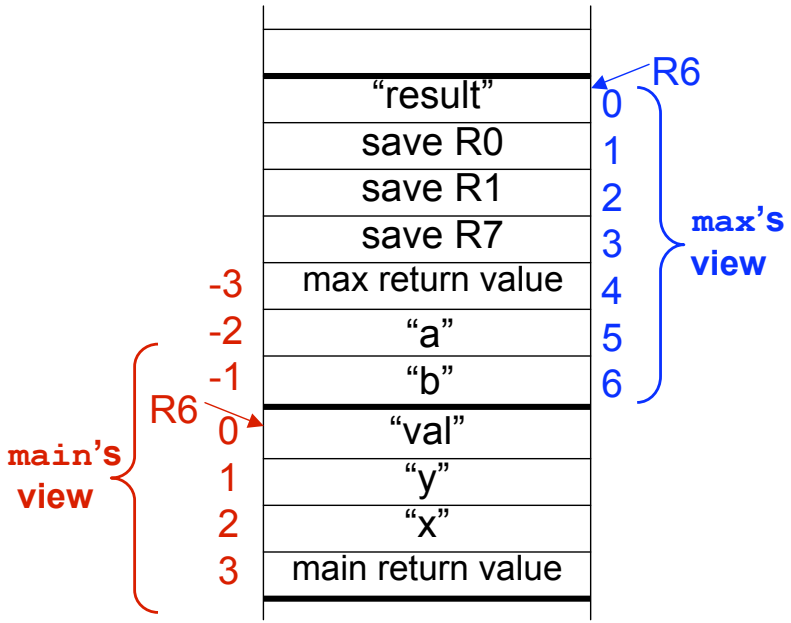
Activation Record Bookkeeping

- Return value
  - Space for value returned by function
  - Allocated even if function does not return a value
- Return address
  - Save pointer to next instruction in calling function
  - Convenient location to store R7
    - in case another function is called
- Save registers
  - Save all other registers used (but not R6, and often not R4)

Function Call Example

```
int main()
{
  int x, y, val;
  x = 10;
  y = 11;
  val = max(x + 10, y);
  return val;
}

int max(int a, int b)
{
  int result;
  result = a;
  if (b > a) {
    result = b;
  }
  return result;
}
```



Outline

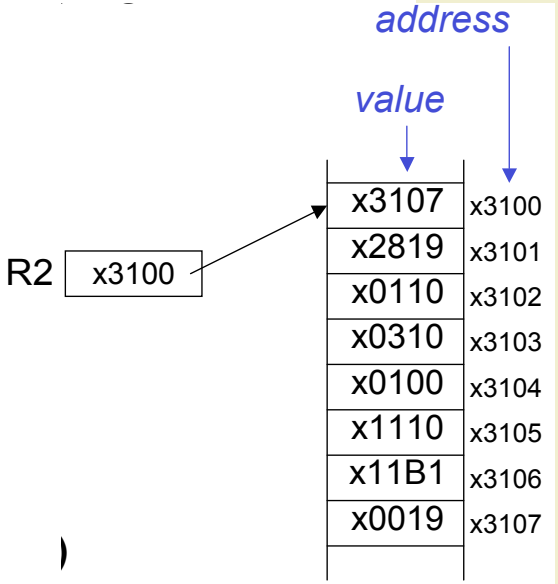
- Introduction
- Variables and Operations
- Control Structure
- Functions
- **Pointers and Arrays**

Pointers and Arrays

- **Pointer**
  - Address of a variable in memory
  - Allows us to indirectly access variables
    - In other words, we can talk about its address rather than its value
- **Array**
  - A list of values arranged sequentially in memory
  - Expression a[4] refers to the 5the lement of the array a

Address vs. Value

- Sometimes we want to deal with the address of a memory location, rather than the value it contains
- Adding a column of numbers:
  - R2 contains address of first location
  - Read value, add to sum, and increment R2 until all numbers R2 have been processed
- R2 is a pointer
  - It contains the address of data
  - (It's also an array, but more on that later)



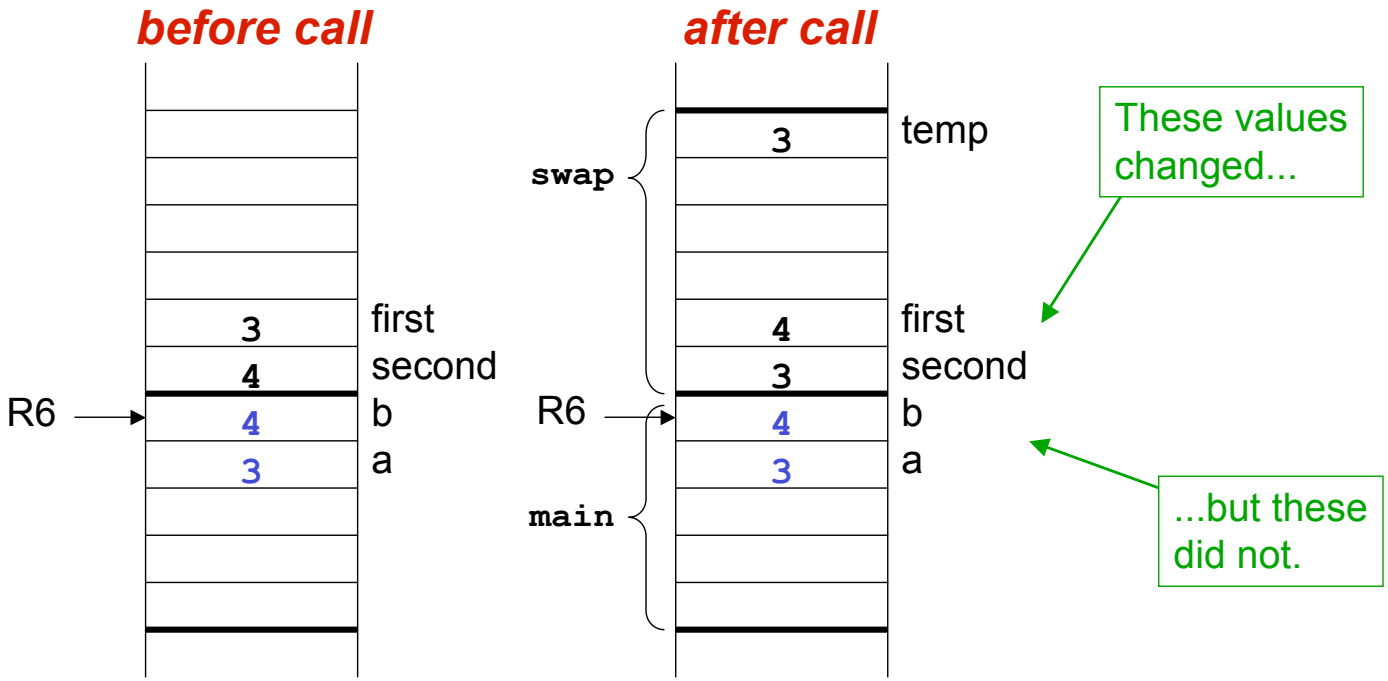
Another Need for Addresses

- Consider the following function that's supposed to swap the values of its arguments.

```
void swap_wrong(int first, int second) {
  int temp = first;
  first = second;
  second = temp;
}
int main() {
int a = 3, b = 4;
  swap_wrong(a, b);
}
```

- What's wrong with this code?

Executing the Swap Function



Swap needs addresses of variables outside its own activation record

Pointers in C

- Declaration

  Int *p; /*p is a pointer to an int*/

  A pointer in C is always a pointer to a particular data type: int*, double*, char*, etc.

- Operators
  - *p -- returns the value pointed to by p
  - &z -- returns the address of variable z

Example

```
int i;
int *ptr;

i = 4;
ptr = &i;
*ptr = *ptr + 1;
printf("%d\n", i);
```

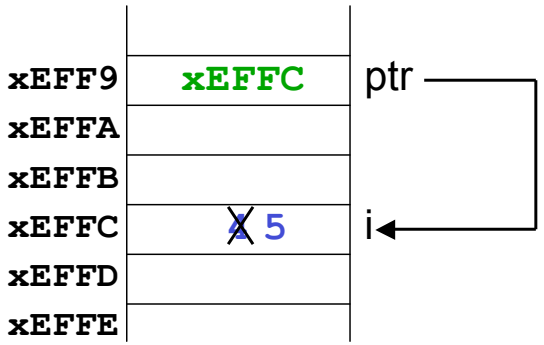
store the value 4 into the memory location associated with i

store the address of "i" into the memory location associated with ptr

read the contents of memory at the address stored in ptr

print the value "5", becaue "i" was modified indirectly via ptr

store the result into memory at the address stored in ptr



Pointers vs Arguments

Passing a pointer into a function allows the function to read/change memory outside its activation record

```
void swap(int *first, int *second)
{
  int temp = *first;
  *first = *second;
  *second = temp;
}
```

How would you do this in Java?

All arguments in C are pass-by-value. Also true in Java, but Java has reference types

Arguments are integer pointers. Caller passes addresses of variables that it wants function to change

Using Arguments for Results

- Pass address of variable where you want result stored
  - Useful for multiple results
  - Example:
    - Return value via pointer
    - Return status code as function result
- This solves the mystery of the '&' for calling scanf():

```
scanf("%d %d", &data1, &data2);
```

read decimal integers into `data1` and `data2`

Null Pointer

- Sometimes we want a pointer that points to nothing.
- In other words, we declare a pointer, but we're not ready to actually point to something yet.

```
int *p;
p = NULL;  /* p is a null pointer */
```

- NULL is a predefined macro that contains a value that a non-null pointer should never hold.
  - Often, NULL = 0, because Address 0 is not a legal address for most programs on most platforms
  - Dereferencing a NULL pointer: program crash!

```
int *p = NULL; printf("%d", *p); // CRASH!
```

Pointer Problems

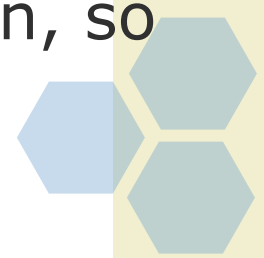
- What does this do?

```
int *x;
*x = 10;
```

- Answer: writes "10" into a random location in memory
- What's wrong with:

```
int* func() {
        int x = 10;
        return &x;
}
```

- Answer: storage for "x" disappears on return, so the returned pointer is dangling



Arrays

- How do we allocate a group of memory locations?
  - Character string
  - Table of numbers
- How about this?

```
int num0;
int num1;
int num2;
int num3;
```

- Not too bad, but...
  - What if there are 100 numbers?
  - How do we write a loop to process each number?
- Fortunately, C gives us a better way -- the array.

  int num[4];
- Declares a sequence of four integers, referenced by:

  num[0], num[1], num[2], num[3].

Array Syntax

Declaration

```
type   variable[num_elements];
```

all array elements are of the same type

number of elements must be known at compile-time

Array Reference

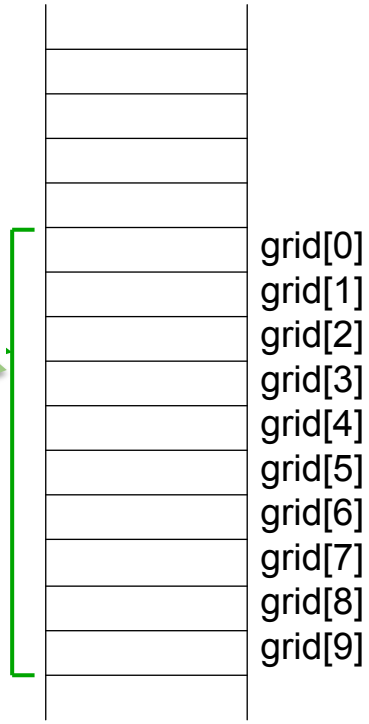
```
variable[index];
```

i-th element of array (starting with zero);
no limit checking at compile-time or run-time

Array as a Local Variable

- Array elements are allocated as part of the activation record

```
int grid[10];
```



- First element (grid[0]) is at lowest address of allocated space

Passing Arrays as Arguments

- C passes arrays by address
  - the address of the array (i.e., of the first element) is written to the function's activation record
  - otherwise, would have to copy each element

```
int main() {
   int numbers[MAX_NUMS];
   ...
   mean = average(numbers, MAX_NUMS);
   ...
}
int average(int values[], int size) {
   int index, sum = 0;
   for (index = 0; index < size; index++) {
      sum = sum + values[index];
   }
   return (sum / size);
}
```

This must be a constant, e.g.,
`#define MAX_NUMS 10`

More on Passing Arrays

- No run-time length information
  - C doesn't track length of arrays
  - No Java-like values.length construct
  - Thus, you need to pass length or use a sentinel

```
int average(int values[], int size)
 {
  int index, sum;
  for (index = 0; index < size; index++) {
    sum = sum + values[index];
  }
  return (sum / size);
}
```

Relationship between Arrays and Pointers

An array name is essentially a pointer to the first element in the array

```
char data[10];
char *cptr;
cptr = data; /* points to data[0] */
```

Difference:

Can change the contents of cptr, as in

```
cptr = cptr + 1;
```

Correspondence between Ptr and Array Notation

- Given the declarations on the previous page, each line below gives three equivalent expressions:

`cptr`	`data`	`&data[0]`
`(cptr + n)`	`(data + n)`	`&data[n]`
`*cptr`	`*data`	`data[0]`
`*(cptr + n)`	`*(data + n)`	`data[n]`

String Length - Array Style

```
int strlen(char str[])
{
   int i = 0;
   while (str[i] != '\0') {
     i++;
   }
   return i;
}
```

String Length - Pointer Style

```
int strlen(char* str)
{
   int i = 0;
   while (*str != '\0') {
      i++;
      str++;
   }
   return i;
}
```

String Copy - Array Style

```
void strcpy(char dest[], char src[])
{
   int i = 0;
   while (src[i] != '\0') {
      dest[i] = src[i];
      i++;
   }
   dest[i] = '\0';
}
```

String Copy - Array Style #2

```
void strcpy(char dest[], char src[])
{
   int i = 0;
   while ((dest[i] = src[i]) != '\0') {
      i++;
   }
}
```

String Copy - Pointer Style

```
void strcpy(char* dest, char* src)
{
   while ((*dest = *src) != '\0') {
      dest++;
      src++;
   }
}
```

String Copy - Pointer Style #2

```
void strcpy(char* dest, char* src)
{
    while ((*dest++ = *src++) != '\0') {
        // nothing
    }
}
```

C String Library

- C has a limited string library
  - All based on null-terminated strings
  - #include <string.h> to use them
- Functions include
  - int strlen(char* str)
  - void strcpy(char* dest, char* src)
  - int strcmp(char* s1, char* s2)
    - Returns 0 on equal, -1 or 1 if greater or less
    - Remember, 0 is false, so equal returns false!
  - strcat(char* dest, char* src)
    - string concatenation (appending two strings)
  - strncpy(char* dest, char* src, int max_length)
  - strncmp(char* s1, char* s2, int max_length)
  - strncat(char* dest, char* src, int max_length)
  - Plus some more...

String Declaration Nastiness

- What's the difference between:
  - char amessage[] = "message"
  - char *pmessage = "message"
- Answer:

char amessage[] = "message" **// single array**

char *pmessage = "message" **// pointer and array**

Main(), revisited

- Main supports command line parameters
  - Much like Java's

public static void main(String[] args)

- Main supports command line parameters:

```
int main(int argc, char *argv[])
 {
   int i;
   for (i = 0; i<argc; i++) {
     printf("%s\n", argv[i]);
   }
}
```

- Displays each command-line argument
  - Zero-parameter is the program name

Exercises

- Write a function countEven(int*, int) which receives an integer array and its size, and returns the number of even numbers in the array.
- Write a function contains(char*, char) which returns true if the 1st parameter cstring contains the 2nd parameter char, or false otherwise.
- Write a function revString(char*) which reverses the parameter cstrin
- Write a C program to find the max of an integer set entered by users.