Section 1: Introduction to C

1.1: History and Expressions

Comments

Comments can be added using `//` or adding comments between `/*` and `*/`. The difference is that `/* ... */` can extend over multiples lines.

```
// C comment (one-line only)
/* This is a
multi-line comment */
```

Expressions

C expressions use traditional *infix* algebraic notion: (e.g. 9 + 10)

Operators

C has traditional mathematical as well as non-mathematical operators. Has over **40** operators total.

- C does not have an exponentiation operator
- The `/` operator truncates (rounds toward zero) any intermediate values
- The modulo `%` operator produces the reminder after integer division

1.2: Identifiers and Functions

C Identifiers

C functions, variables, and structures require ***identifiers***. C identifiers:

- Must start with a letter
- Can only contain letters, underscores, and numbers

We use `underscore_style` for identifiers with compound words(e.g. `hst_rate`)

Anatomy of a Function

1. braces `{}` indicate the beginning/end of a function block
2. `return` keyword, followed by an expression, followed by a semicolon(`;`)
3. parameters are separated by comma
4. the function and parameter types are specified

Example:

```
int my_add(int a, int b) {
  return a + b;
}
```

Static Type System

C uses a ***static type system***: all types must be known before the program is run and the type of an identifier cannot change.

Functions

In C, you call a function by *passing* it's *arguments.* A function returns a value.

You can use the keyword `void` as a type to indicate a function has no perimeters.

Example:

```
int my_num(void) {
  return my_add(40, 2);
}
```

In C, functions **cannot** be nested inside of another function(a.k.a local functions)

Example:

```
int outer(int i) {
  int inner(int j) { // INVALID
    // ...
  }
  // ...
}
```

CS 136 Style

You need to provide a purpose and a description of what it does for functions. A **requires** section is needed if appropriate.

Example:

```
// my_divide(x, y) evaluates x/y using
//   integer division
// requires: y is not 0
int my_divide(int x, int y) {
  return x / y;
}
```

C ignores whitespace but is good to include a good use of whitespace.

Example:

```
// The following three functions are equivalent
int my_add(int a, int b) { // GOOD
  return a + b;
}

int my_add(int a,int b){return a+b;} // BAD

int my_add(int a, int b){return a+ // RIDICULOUSLY
b;} // BAD
```

The CS 136 style includes:

- a block start `{` appears at the end of line
- a block end `}` is aligned with the line that started it
- indent a block 2 spaces
- add a space after commas and around arithmetic operators
- for large lines, continue indented on the next line

Example:

```
// my_super_long_function(a, b, c, d, e, f, g) does some
//   amazing things with those parameters...
int my_super_long_function(int a, int b, int c, int d,
                           int e, int f, int g) {
  return a * b + b * c + c * d + d * e + e * f + f * g + g * a;
}
```

1.3: The main function and tracing code

main

A program is typically run by the Operating System(OS) which needs to ***entry point*** to know where to start running the program. In C it is the special function called `main`. There is one and only one `main` function.

`main` has no parameters(optional) and an `int` `return` type. The `return` value tells the OS the “error code”. `main` is a special function and does not require a `return` value so zero(success) is `returned` if no value is present.

Tracing Expressions

Tracing tools are provided and help you see what your code is doing.

Tracing Tools Documentation (cs136.h/cs136-trace.h libraries):

```
/*****************************************************************************
  TRACING TOOLS
```

```
*************************************************************************/

// These tracing tools can be used to help debug your code.
// They will not interfere with Marmoset tests or any I/O testing.


// trace_msg(msg) Displays msg in the console

// trace_X(exp) displays a message in the console of the form:
//   exp => final value

//   X can be one of: int, long, bool, char, double, string, ptr, symbol

// example usage:
//   trace_msg("Hello, World!");
//   trace_int(1 + 1);


// trace_array_Y(arr, len) displays a message in the console of the form:
//   arr => [arr[0], arr[1], ...., arr[len-1]]

//   Y can be one of: int, bool, char, double, ptr, symbol

// example usage:
//   int a[6] = {4, 8, 15, 16, 23, 42};
//   trace_array_int(a, 6);


// trace_printf(str, arg1, arg2, ...) displays a message in the console
//   using the printf format specifier syntax
// note: adds a newline automatically
// WARNING: unlike printf, it does NOT detect errors or mismatches between the
//            number or type of format specifiers so bad combinations such as
//            trace_printf("%s") or trace_printf("%s", 42) may crash

// example usage:
//   trace_printf("the value of x is %d and y is %d", x, y);
// trace_off() Turns off all tracing messages
//   [by default they are turned on]
void trace_off(void);


// trace_sync() "Synchronizes" tracing and printf output by
//   forcing all of the tracing messages to go to the same
//   stream as printf (stdout)
//   NOTE: this may cause your Marmoset and I/O tests to fail
void trace_sync(void);


// trace_version() displays the current version of the cs136 tools library
void trace_version(void);
```

Example:

```
#include "cs136.h"

// my_add(a, b) calculates the sum of a and b
int my_add(int a, int b) {
  return a + b;
}

int main(void) {
  trace_int(1 + 1);
  trace_int(my_add(1, 2));
}
```

Style and Syntax

Document the purpose of a program at the **top** of the file. No need for a documentation of `main`.

Always place function definitions above any other functions that reference them. `main` is at the bottom.

1.4: Testing code

Expressions and Operators

Name	Description
Boolean Expressions	Boolean expressions don't produce true or false, rather they produce zero(`false`) or one(`true`).
Equality Operator(`==`)	Checks if they are equal
Not Equal Operator(`!=`)	Checks if they are not equal
Logical Operators(`!`)	It is the logical operator "not"
Logical Operators(`&&`)	It is the logical operator "and"
Logical Operators(`||`)	It is the logical operator "or"
All non-zero values are true	Operators that expect a Boolean, it will consider all non-zero values to be `true`. Only zero is `false`.

bool Type

The `bool` type is an integer that can only have the value of 0 or 1.

Example:

```c
// A simple program to illustrate boolean expressions
#include "cs136.h"

// is_even(n) returns true(1) is n is even and false(0) otherwise
bool is_even(int n) {
  return (n % 2) == 0;
}

// my_negate(n) returns true(1) is v is false and false(0) if v is true
bool my_negate(bool v) {
  return !v;
}

int main(void){
  trace_int(is_even(3));
  trace_int(is_even(4));
  trace_int(my_negate(true));
  trace_int(my_negate(false));
}
```

Assertions

The `assert` function can formally test code and to check requirements. In `assert(exp)` it stops the program and displays a message if `exp` is false(zero). Otherwise it does nothing. Certain requirements are infeasible.

Example:

```c
#include "cs136.h"

// my_divide(x, y) ....
// requires: y is not 0
// note: an example of infeasable is if x is prime
int my_divide(int x, int y) {
  assert(y != 0);             // assert(y) also works
  return x / y;
}

int main(void) {
  assert(my_divide(0, 1) == 0); //true
```

```
  assert(my_divide(1, 1) == 2); //false and will stop program
}
```

1.5: Statements and Control Flow

Control Flow

The `return` statement is a **control flow** statements which controls the flow by ending the function and returning to the caller.

There are 4 types of control flow:

- compound statements(blocks)
- function calls
- conditionals
- iteration

Compound Statements(Blocks)

Blocks(`{}`) can contain multiple statements which are executed in sequence until you get to the end of the function of reach a `return` statement which evidently ends the program.

Example:

```
#include "cs136.h"

int main(void) {
  trace_int(1 + 1);    // first
  assert(3 > 2);       // second
  assert(0 < 1);       // third
  trace_int(10);       // fourth
  return 0;            // fifth
  trace_int(7);        // unreachable statement
}
```

Functions Calls

When a function is called, the program location "jumps" *from* the current location *to* the start of the function.

The `return` control flow statement changes the program location to go *back* to the **most recent** calling function (where it "jumped from").

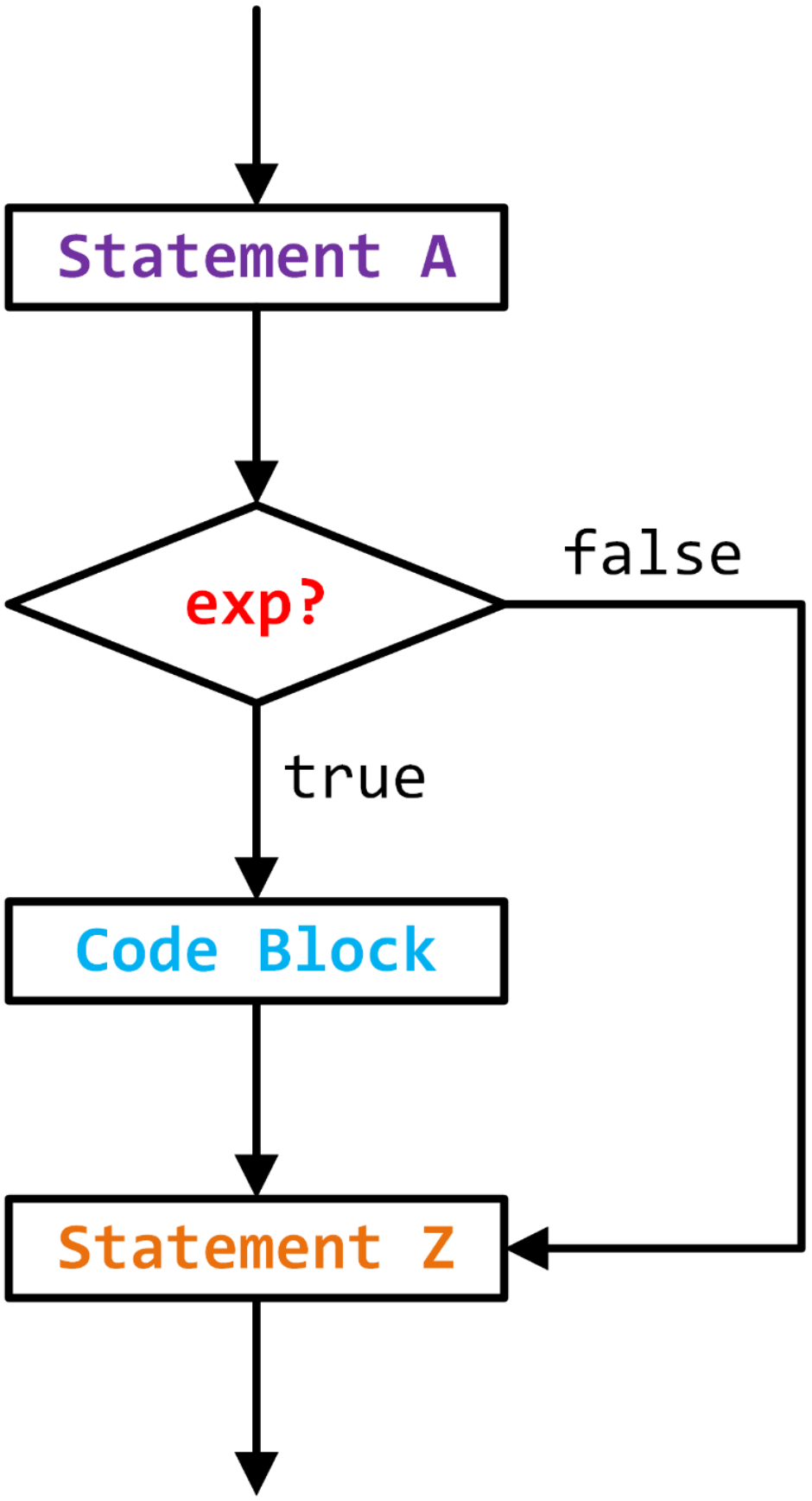
Example:

```
#include "cs136.h"

// blue() is just a demo function
void blue(void) {
  printf("three\n");
  return;
}

// green() is just a demo function
void green(void) {
  blue();
  printf("fourCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  return;
}

// red() is just a demo function
void red(void) {
  printf("twoCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  green();
  printf("fiveCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  return;
}

int main(void) {
  printf("oneCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  red();
  printf("sixCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
}
```

Conditionals

The `if` control flow statement allows us to have functions with conditional behaviour. In other words, it allows us to execute statements only if an `expression` is true (non-zero).

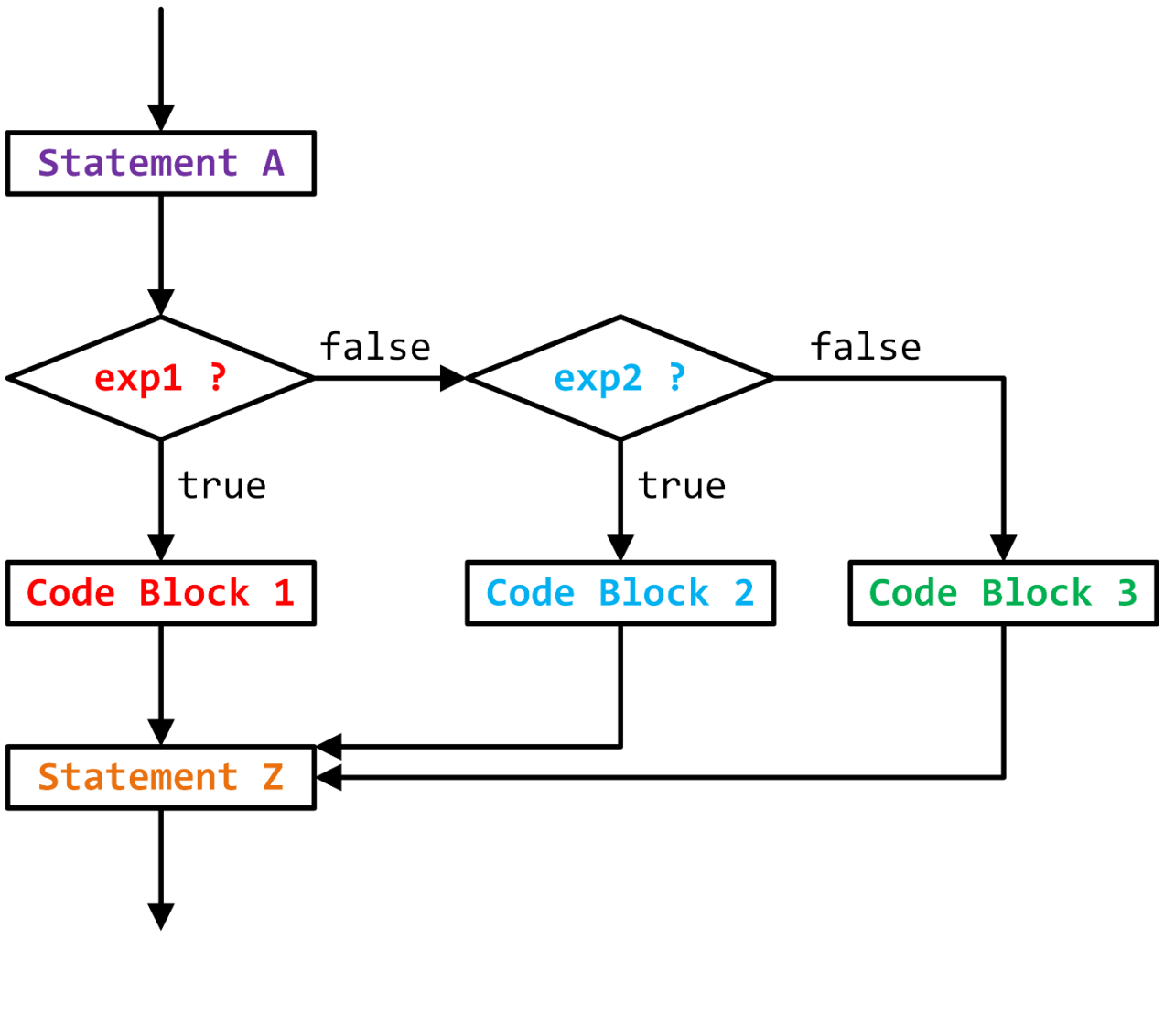
The syntax of if is `if(expression) statement` where the statement is only executed if the expression is true (non-zero).




```
Statement A;
if (exp) {
   Code Block;
}
Statement Z;
```

The `if` statement can be combined with `else` if there are two conditions. If an `if` statement contains a return statement, there may be no need for an else block.

If there are more than two possible results, use `else if`.




```
Statement A;
if (exp1) {
  Code Block 1;
} else if (exp2) {
  Code Block 2;
} else {
  Code Block 3;
}
Statement Z;
```