PYTHONIC C++
CONTROLLING THE FLOW

LECTURE 02-1

JIM FIX, REED COLLEGE CS2-F20

TODAY

We continue looking at C++ program structure

- loops
- procedures and functions

The upshot:

- Programming involves "*controlling the flow of your program's execution.*"
  - Use "control flow" constructs: conditionals, loops, procedures, functions.
- To invent data structures, C++ requires us to think about how these work.
  - ➡ We will look at C++ runtime call stack; stack frames

TO-DOS

- You should have attempted **Lab 01** for today.
- Hand in **Homework 01** by tomorrow morning, 10am.
- Accept **Homework 02**. Out tomorrow morning. Due 9am Tuesday.

LOGISTICS SUMMARY

1. accept my GitClassroom assignment; this creates a private remote repo
2. clone the repo; complete exercises specified in **`README.md`**

```
git clone https://github.com/ReedCS2-F20/hwNN-USER.git
```

3. stage every new/changed file

```
git add ex1.cc ex2.cc ex3.cc
```

4. commit these changes

```
git commit -m "Finished with HWnn"
```

5. push these changes to the remote repo

```
git push origin master
```

6. await feedback from us, within repo "branch" named **`feedback`**

- **NOTE:** you can just upload files into your repo for Homework 01 hand-in (demo).

EXAMPLE: countUp.cc

```
#include <iostream>

int main(void) {

  std::cout << "Enter a positive number: ";
  std::cin >> max;
  if (max < 0) return 0;
  int count = 1;
  while (count <= max) {
    std::cout << count << std::endl;
    count++;
  }
  return 0;
}
```

EXAMPLE: countUp.cc

```
#include <iostream>

int main(void) {

  std::cout << "Enter a positive number: ";
  std::cin >> max;
  if (max < 0) return 0; // needed?
  int count = 1;
  while (count <= max) {
    std::cout << count << std::endl;
    count++;
  }
  return 0;
}
```

EXAMPLE: countUp.cc

```
#include <iostream>

int main(void) {

  std::cout << "Enter a positive number: ";
  std::cin >> max;

  int count = 1;
  while (count <= max) {
    std::cout << count << std::endl;
    count++;
  }
  return 0;
}
```

EXAMPLE: countUp.cc

```
#include <iostream>

int main(void) {

  std::cout << "Enter a positive number: ";
  std::cin >> max;

  int count = 1;                         // init
  while (count <= max) {                 // condition
    std::cout << count << std::endl;     // body
    count++;                             // update
  }
  return 0;
}
```

EXAMPLE: countUp.cc

```
#include <iostream>

int main(void) {

  std::cout << "Enter a positive number: ";
  std::cin >> max;


  for (int count = 1; count <= max; count++) {

    std::cout << count << std::endl;
  }
  return 0;
}
```

EXAMPLE: countUp.cc

```
#include <iostream>

int main(void) {

  std::cout << "Enter a positive number: ";
  std::cin >> max;

  //   .........init ....condition ..update
  for (int count = 1; count <= max; count++) {
    // ........................body
    std::cout << count << std::endl;
  }
  return 0;
}
```

EXAMPLE: countUp.cc

```
#include <iostream>

int main(void) {

  std::cout << "Enter a positive number: ";
  std::cin >> max;

  //    .........init ....condition ..update
  for (int count = 1; count <= max; count++) {
    // ........................body
    std::cout << count << std::endl;
  }
  return 0;
}
```

FOR STATEMENT

- Any time you write a loop like...

```
initialization of state
while (condition on the state to continue looping) {
    body of statements to be repeated
    update of state
}
```

...it can usually be re-written as a C++ **for** statement like so

```
for (initialization ; condition ; update) {
    body
}
```

DO...WHILE STATEMENT

- Any time you write a loop like...

body *of statements to be run at the first time*

while (*condition* *for repeating those*) {

same ***body*** *of statements to be repeated*

}

...it can usually be re-written as a C++ **do-while** statement like so

```
do {
    body
} while (condition)
```

EXAMPLE: LOOP WITHIN guess.cc

```
int guess;
bool success = false;
while (!success) {
  std::cin >> guess;
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else if (guess > number) {
    std::cout << "That's too high. Try again.\n";
  } else {
    success = true;
  }
}
```

EXAMPLE: DIFFERENT LOOP FOR guess.cc

```
int guess;
std::cin >> guess;
while (guess != number) {
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else {
    std::cout << "That's too high. Try again.\n";
  }
  std::cin >> guess;
}
```

EXAMPLE: DO..WHILE FOR guess.cc

```
int guess;
do {
  std::cin >> guess;
  if (guess < number) {
    std::cout << "That's too low. Try again.\n";
  } else if (guess > number) {
    std::cout << "That's too high. Try again.\n";
  }
} while (guess != number);
```

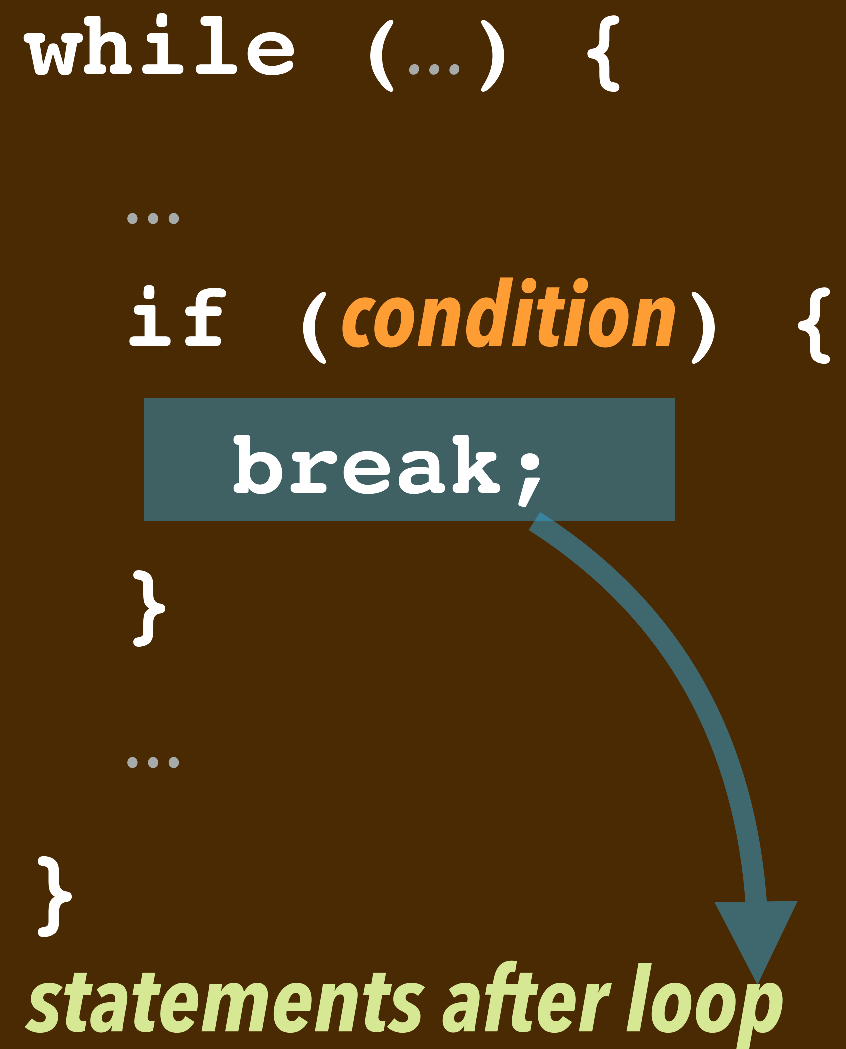
EXITING A LOOP WITH BREAK STATEMENT

- You can jump out of a loop using **break**

```
while (…) {
   …
   if (condition) {
      break;
   }
   …
}
statements after loop
```

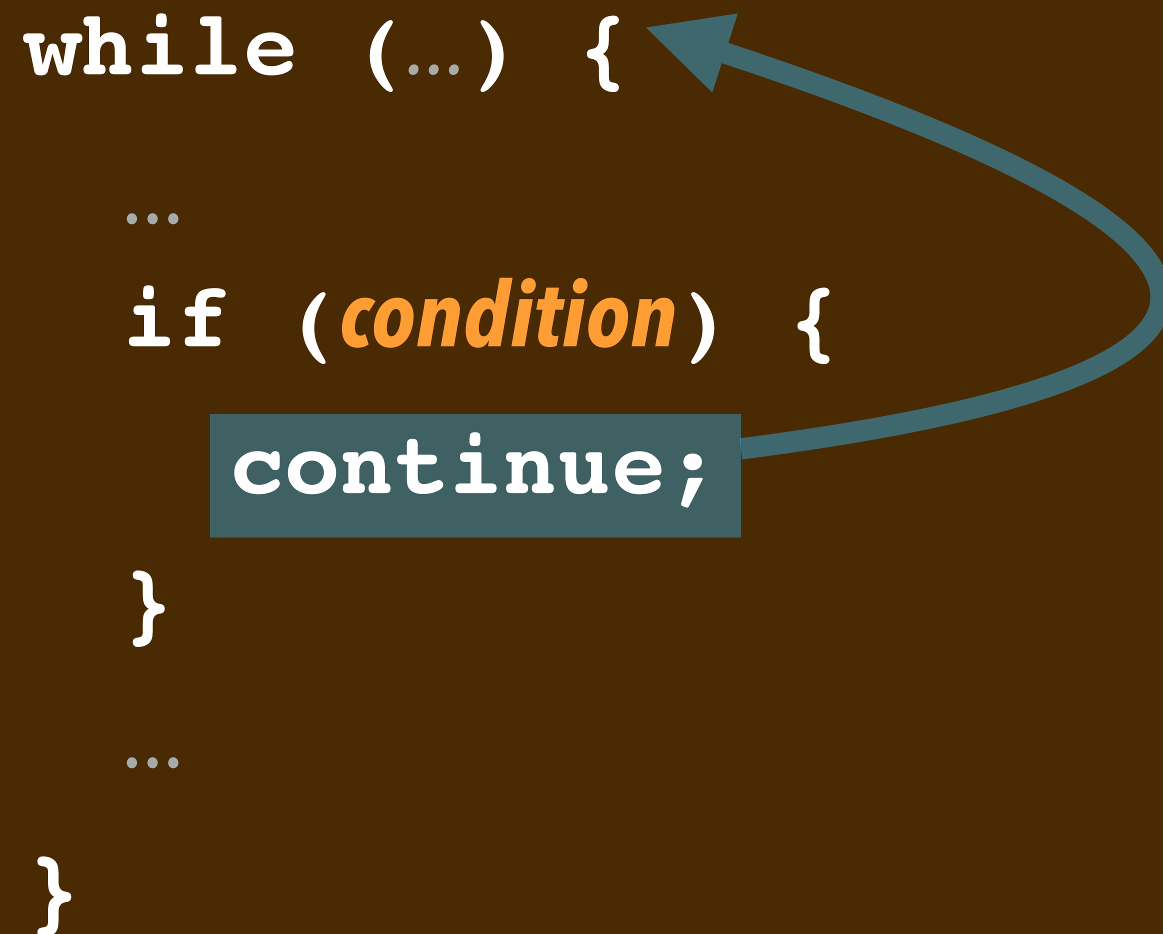
EXITING A LOOP WITH BREAK STATEMENT

- You can jump out of a loop using **break**



CONTINUING A LOOP WITH CONTINUE STATEMENT

- You can loop back within a loop using **`continue`**



NESTED FOR LOOPS: stars.cc

```
std::cin >> rows;
std::cin >> columns;
for (int r = 1; r <= rows; r++) {
  for (int c = 1; c <= columns; c++) {
    std::cout << "*";
  }
  std::cout << std::endl;
}
```

Interaction:

```
mylaptop % ./stars
3
5
*****
*****
*****
mylaptop %
```

STRING CONCATENATION: stars_string.cc

```
#include <iostream>
#include <string>

int main(void) {
  int rows, columns;
  std::cin >> rows;
  std::cin >> columns;
  std::string s = "";
  for (int r = 1; r <= rows; r++) {
    for (int c = 1; c <= columns; c++) {
      s = s + "*";
    }
    s = s + "\n";
  }
  std::cout << s;
}
```

STRING CONCATENATION: stars_string.cc

```
#include <iostream>
#include <string>

int main(void) {
  int rows, columns;
  std::cin >> rows;
  std::cin >> columns;
  std::string s = "";
  for (int r = 1; r <= rows; r++) {
    for (int c = 1; c <= columns; c++) {
      s = s + "*";
    }
    s = s + "\n";
  }
  std::cout << s;
}
```

MODULAR DECOMPOSITION: FUNCTIONS AND PROCEDURES

It's good to break up a program into smaller, testable code components.

- Define meaningful ***functions*** for making calculations, return a result.
- Define meaningful ***procedures*** for performing actions.
  - ➡ in **Python**: a function that returns `None`
- In each case, we write them in a useful, general way: use ***abstraction***.

Their syntax/use is different...

- We call ***functions*** within an expression to calculate a value.
- We call ***procedures*** as a statement line to perform some action.
- When defined, use *formal parameters* to define the calculation/action.
- When called, these take actual parameters to calculate/act with at call site.

WRITING FUNCTIONS: cToF.cc

```
#include <iostream>

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  int f = c * 9 / 5 + 32;
  std::cout << "That's " << f << " degrees fahrenheit.\n";

  return 0;
}
```

WRITING FUNCTIONS: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
  int degreesF = degreesC * 9 / 5 + 32;
  return degreesF;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  int f = cToF(c);
  std::cout << "That's " << f << " degrees fahrenheit.\n";

  return 0;
}
```

WRITING FUNCTIONS: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  std::cout << "That's " << cToF(c) << " degrees fahrenheit.\n";

  return 0;
}
```

WRITING FUNCTIONS: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  std::cout << "That's " << cToF(c) << " degrees fahrenheit.\n";

  return 0;
}
```

WRITING PROCEDURES: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

void reportTemp(int value, std::string units) {
  std::cout << "That's " << value
  std::cout << " degrees " << units;
  std::cout << "." << std::endl;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  reportTemp(cToF(c), "fahrenheit");

  return 0;
}
```

WRITING PROCEDURES: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

void reportTemp(int value, std::string units) {
  std::cout << "That's " << value
  std::cout << " degrees " << units;
  std::cout << "." << std::endl;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  reportTemp(cToF(c), "fahrenheit");

  return 0;
}
```

FUNCTION DEFINITION

WRITING PROCEDURES: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}
```

FUNCTION DEFINITION

```
void reportTemp(int value, std::string units) {
  std::cout << "That's " << value
  std::cout << " degrees " << units;
  std::cout << "." << std::endl;
}
```

PROCEDURE DEFINITION

```
int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  reportTemp(cToF(c), "fahrenheit");

  return 0;
}
```

WRITING PROCEDURES: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

void reportTemp(int value, std::string units) {
  std::cout << "That's " << value
  std::cout << " degrees " << units;
  std::cout << "." << std::endl;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  reportTemp(cToF(c), "fahrenheit");

  return 0;
}
```

FUNCTION DEFINITION

PROCEDURE DEFINITION

FUNCTION CALL SITE

WRITING PROCEDURES: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

void reportTemp(int value, std::string units) {
  std::cout << "That's " << value
  std::cout << " degrees " << units;
  std::cout << "." << std::endl;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  reportTemp(cToF(c), "fahrenheit");

  return 0;
}
```

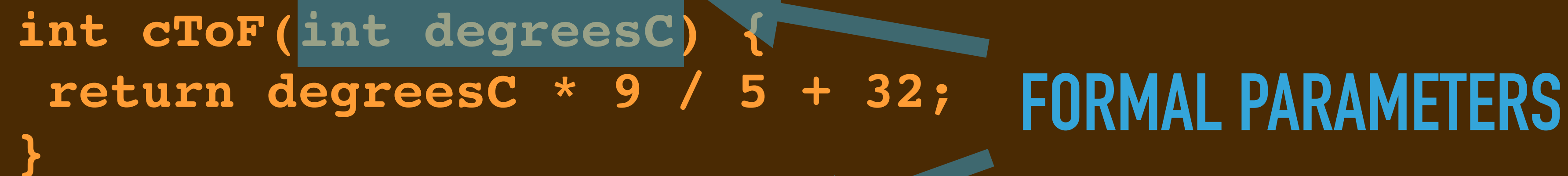
FUNCTION DEFINITION

PROCEDURE DEFINITION

PROCEDURE CALL LINE

FUNCTION CALL SITE

WRITING PROCEDURES: cToF.cc




```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

void reportTemp(int value, std::string units) {
  std::cout << "That's " << value
  std::cout << " degrees " << units;
  std::cout << "." << std::endl;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  reportTemp(cToF(c), "fahrenheit");

  return 0;
}
```

WRITING PROCEDURES: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

void reportTemp(int value, std::string units) {
  std::cout << "That's " << value
  std::cout << " degrees " << units;
  std::cout << "." << std::endl;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  reportTemp(cToF(c), "fahrenheit");

  return 0;
}
```

FORMAL PARAMETERS

ACTUAL PARAMETERS

WRITING PROCEDURES: cToF.cc

```
#include <iostream>

int cToF(int degreesC) {
 return degreesC * 9 / 5 + 32;
}

void reportTemp(int value, std::string units) {
  std::cout << "That's " << value
  std::cout << " degrees " << units;
  std::cout << "." << std::endl;
}

int main(void) {

  int c;
  std::cout << "Enter a temperature in degrees celsius: ";
  std::cin >> c;
  reportTemp(cToF(c), "fahrenheit");

  return 0;
}
```

FORMAL PARAMETERS

ACTUAL PARAMETERS

DECOMPOSITION: temp.cc

```
#include <iostream>

int getInt(std::string prompt, bool oneline) {
  std::cout << prompt << " ";
  if (!oneline) {
    std::cout << "\n";
  }
  int response;
  std::cin >> response;
  return response;
}

int fToC(int f) {
  return (f - 32) * 5 / 9;
}

...
```

DECOMPOSITION: temp.cc (CONT'D)

```
...

int cToF(int c) {
  return c * 9 / 5 + 32;
}

void reportConversion(int amount, std::string scale,
                      std::string toScale) {
  std::cout << "If in " << scale;
  std::cout << ", that would be " << amount;
  std::cout << " degrees " << toScale << ".\n";
}

int main() {
  int t = getInt("Enter a temperature as an integer:",true);
  reportConversion(fToC(t),"fahrenheit","celsius");
  reportConversion(cToF(t),"celsius","fahrenheit");
  return 0;
}
```

ANATOMY OF A C PROGRAM

a preamble of `#include` lines for needed *header files*

a *procedure* or *function* definition

a *procedure* or *function* definition

...

a *procedure* or *function* definition

definition of the `main` function

EXAMPLE: ANATOMY OF temp.cc

```
#include <iostream>
return-type fToC( formal-parameter-declarations )  { func-body }
return-type cToF( formal-parameter-declarations )  { func-body }
return-type getInt( formal-parameter-declarations )  { func-body }
void reportConversion( formal-param... )  { proc-body }
int main(void) { program-body }
```

ANATOMY OF A FUNCTION DEFINITION

Every **function** definition has

- The function's name
- The function's formal parameter declarations.
- The body of its calculation, including use of return.
- The type of value it returns.

Syntax:

return-type name `(` *formal-parameter-declarations* `)` `{` *func-body* `}`

- A parameter declaration is of the form: *type-name variable-name*
  - Where *type-name* is `int`, `bool`, `char`,`double`, `std::string`, etc.
- Needed so the compiler can lay out bytes in memory to store each variable.

PROCEDURES VERSUS FUNCTIONS

A procedure is a callable "routine" that performs a series of statements, too.

- Unlike a function, procedures do not return a value.
- Remember Python functions that always return the `None` value?

Syntax:

`void` *name* `(` *formal-parameter-declarations* `)` `{` *proc-body* `}`

- Like functions, every **procedure** definition has
  - The procedure's name
  - The procedure's formal parameter declarations.
  - The body of its activity, including use of `return`.
- Since they don't return values, use `void` in place of the return type.

EXAMPLE: guessDecomp.cc

```
#include ...
void initialize() { ... }
int randomInt(int low, int high) { ... }
bool assessGuess(int guess, int target) { ... }
void giveInstructions() { ... }
void promptForGuess(int tries, int bound) { ... }
bool playGame(int number, int bound) { ... }
bool checkPlayAgain(void) { ... }
int main(void) {
  initialize();
  do {
    int number = randomInt(1,100);
    giveInstructions();
    bool theyWon = playGame(number,6);
    if (theyWon) {
      std::cout << "Well done! ";
    } else {
      std::cout << "Sorry, you are out of guesses...\n";
    }
    std::cout << number << " was the number I chose.\n";
  } while (checkPlayAgain());
}
```

EXTENDING THE C++ LANGUAGE

- Procedures/functions allow you to extend C++, make your own constructs...
  - A function is like a programmer-defined expression/operation.
  - A procedure is like a programmer-defined statement.

- C++ also allows you to define your own data structures...
  - The building blocks will be (primitive) C *arrays* and C *structs*.
  - When we cover object-orientation in C++, *classes* will let us define new data structures (new object types) as well as the operations on them (their *methods*)

But first...

...we need to start thinking more carefully about the C++ runtime system.

...we need to be aware of where variables and data are placed in memory.

THE C++ RUNTIME SYSTEM

Let's begin examining the run-time system of a compiled C++ program.

Q: Where are variables and data structures placed in memory? How are they laid out?

A: The *call stack* memory holds a called procedure's (or function's) local variables and passed parameters. (There are other areas of memory, too...)

➡ We'll soon investigate it using the *address-of* operator **&**.

EXAMPLE: foo.cc

```
int g1(double y, double z) {
... // uses n, z0
}
int g2(...) { ... }
int f(double x) {
... // uses i, j; calls g1,g2
}
void P(int a, int b) {
... // uses d; calls f
}
int h(int n) { ... /* calls h */ }
void Q(int c) {
... // uses e; calls recursive h
}
int main() {
... // calls P, Q
}
```

EXAMPLE: foo.cc

```
int g1(double y, double z) {
... // uses n, z0
}
int g2(...) { ... }
int f(double x) {
... // uses i, j; calls g1,g2
}
void P(int a, int b) {
... // uses d; calls f
}
int h(int n) { ... /* calls h */ }
void Q(int c) {
... // uses e; calls recursive h
}
int main() {
... // calls P, Q
}
```

main:

call stack

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

main:

call stack

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

P: a, b, d

main:

call stack

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

f: x, i, j

P: a, b, d

main:

call stack

EXAMPLE: foo.cc

g1(y,z):
- uses n, z0

g2():
-

f(x): calls g1,g2
- uses i,j

P(a,b): calls f
- uses d

h(n): calls h
-

Q(c): calls recursive h
- uses e

main(): calls P, Q
-

g1: y, z, n, z0

f: x, i, j

P: a, b, d

main:

call stack

EXAMPLE: foo.cc

g1(y,z):
- uses n, z0

g2():
-

f(x): calls g1,g2
- uses i,j

P(a,b): calls f
- uses d

h(n): calls h
-

Q(c): calls recursive h
- uses e

main(): calls P, Q
-

f: x, i, j

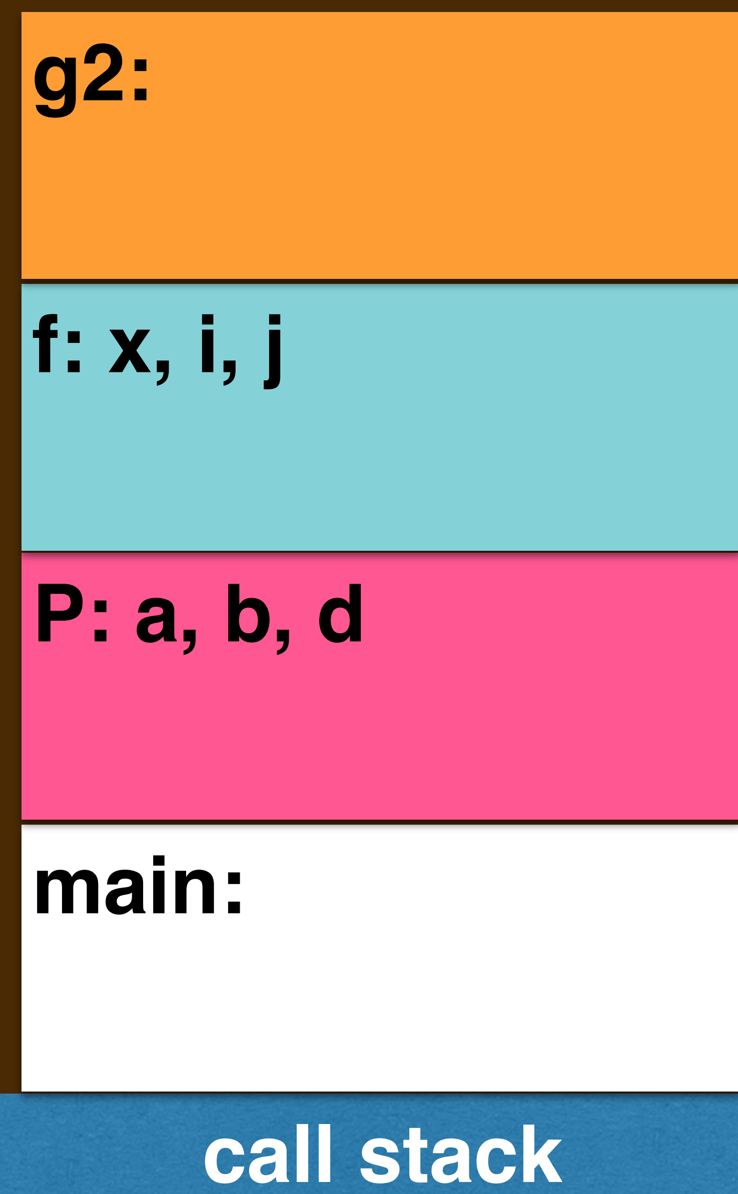
P: a, b, d

main:

call stack

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```



EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

call stack
f: x, i, j
P: a, b, d
main:

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

P: a, b, d

main:

call stack

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

main:

call stack

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

Q: c, e

main:

call stack

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

h: n

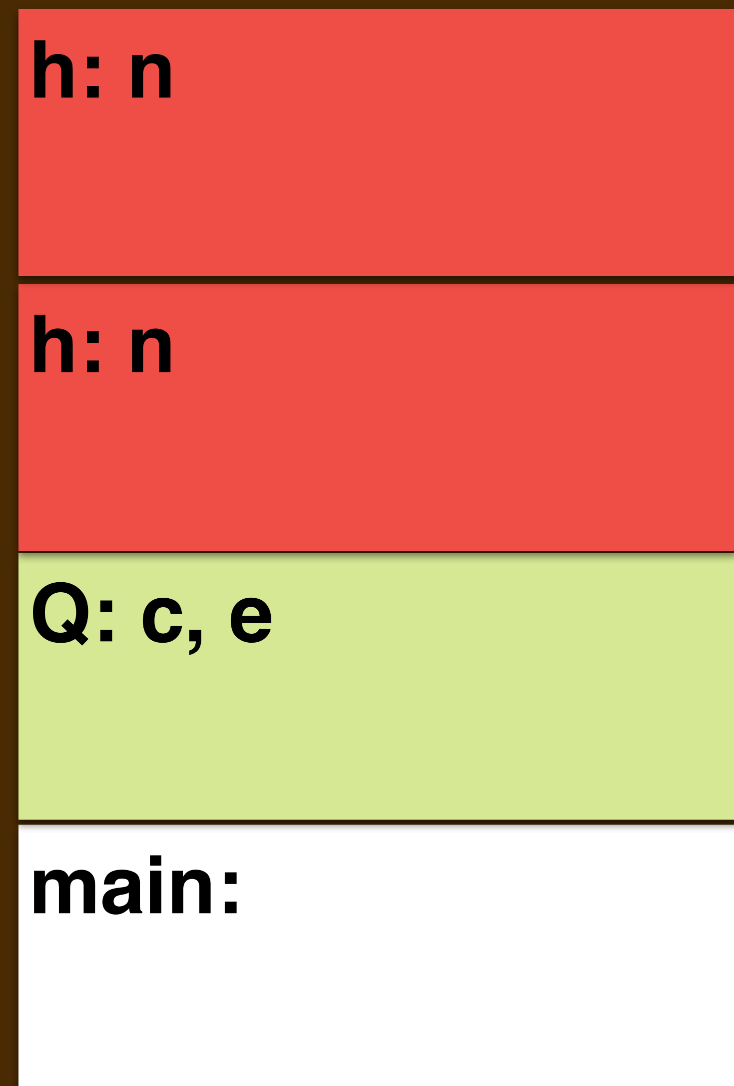
Q: c, e

main:

call stack

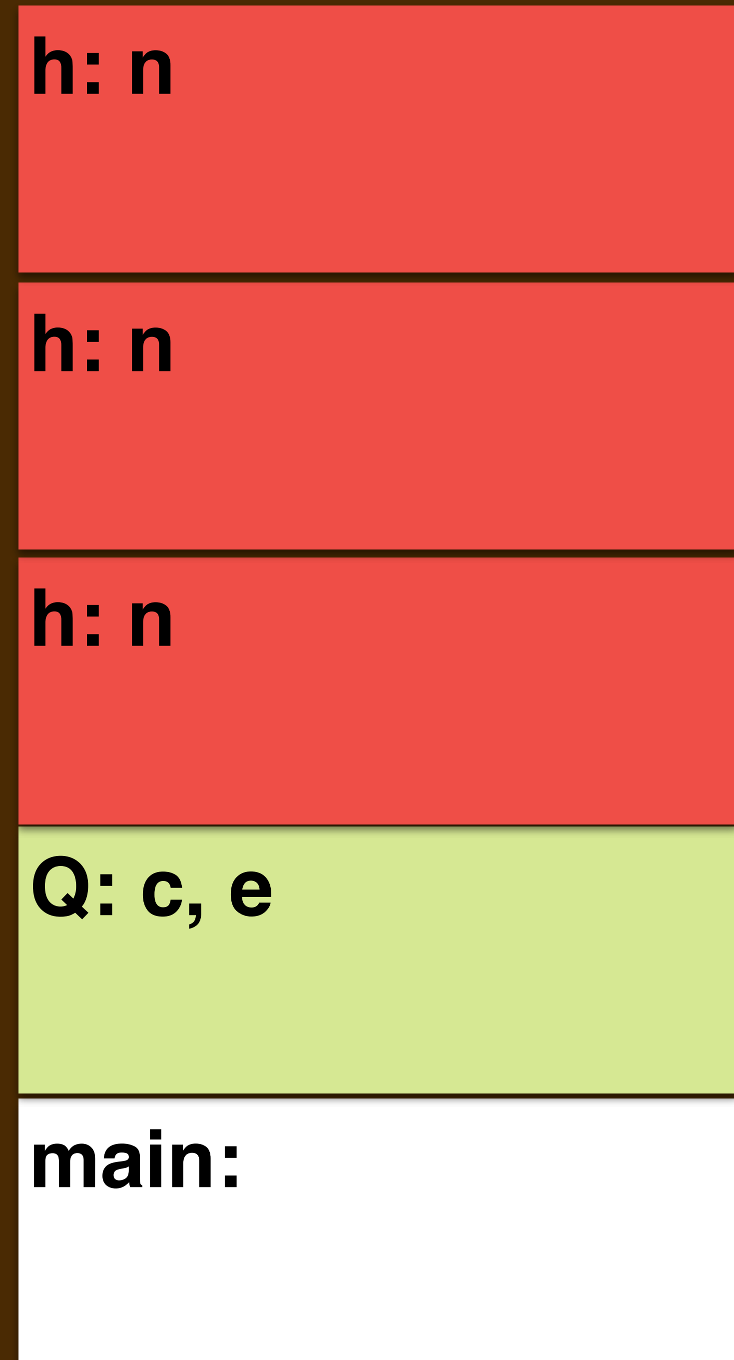
EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```



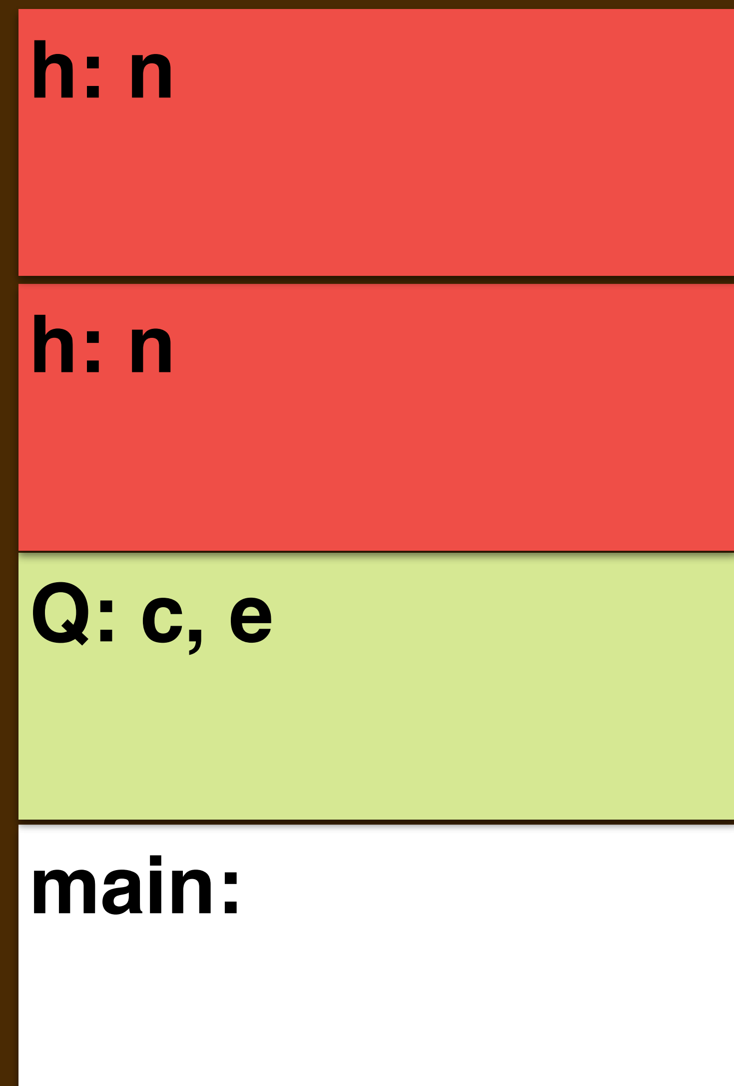
EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```



EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```



EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

call stack
h: n
Q: c, e
main:

EXAMPLE: foo.cc

g1(y,z):
- uses n, z0

g2():
-

f(x): calls g1,g2
- uses i,j

P(a,b): calls f
- uses d

h(n): calls h
-

Q(c): calls recursive h
- uses e

main(): calls P, Q
-

Q: c, e

main:

call stack

EXAMPLE: foo.cc

```
g1(y,z):
• uses n, z0
g2():
•
f(x): calls g1,g2
• uses i,j
P(a,b): calls f
• uses d
h(n): calls h
•
Q(c): calls recursive h
• uses e
main(): calls P, Q
•
```

main:

call stack

EXAMPLE: fib.cc

```
int fib(int n, std::string pfx) {
  if (n == 0 || n == 1) {
    return n;
  } else {
    return  fib(n-2) + fib(n-1);
  }
}

int main(void) {
  int blah = fib(5);
  std::cout << blah << std::endl;
}
```

EXAMPLE: instrumented_fib.cc

```
int fib(int n, std::string pfx) {
  unsigned long a_n = (unsigned long)&n;

  std::cout << pfx;
  printf("> Entering fib(%d). Address of n is %lx.\n",n,a_n);

  int return_value;
  if (n == 0 || n == 1) {
    return_value = n;
  } else {
    return_value = fib(n-2,pfx+"==") + fib(n-1,pfx+"==");
  }

  std::cout << pfx;
  printf("> Exiting fib(%d). Returning %d.\n",n,return_value);

  return return_value;
}
int main() {
  int blah = fib(5,"=");
  std::cout << blah << std::endl;
}
```

EXAMPLE: instrumented_fib.cc

```
int fib(int n, std::string pfx) {
  unsigned long a_n = (unsigned long)&n;

  std::cout << pfx;
  printf("> Entering fib(%d). Address of n is %lx.\n",n,a_n);

  int return_value;
  if (n == 0 || n == 1) {
    return_value = n;
  } else {
    return_value = fib(n-2,pfx+"==") + fib(n-1,pfx+"==");
  }

  std::cout << pfx;
  printf("> Exiting fib(%d). Returning %d.\n",n,return_value);

  return return_value;
}
int main() {
  int blah = fib(5,"=");
  std::cout << blah << std::endl;
}
```

EXAMPLE: instrumented_fib.cc

```
int fib(int n, std::string pfx) {
  unsigned long a_n = (unsigned long)&n;

  std::cout << pfx;
  printf("> Entering fib(%d). Address of n is %lx.\n",n,a_n);

  int return_value;
  if (n == 0 || n == 1) {
    return_value = n;
  } else {
    return_value = fib(n-2,pfx+"==") + fib(n-1,pfx+"==");
  }

  std::cout << pfx;
  printf("> Exiting fib(%d). Returning %d.\n",n,return_value);

  return return_value;
}
int main() {
  int blah = fib(5,"=");
  std::cout << blah << std::endl;
}
```

EXAMPLE: instrumented_fib.cc

```
int fib(int n, std::string pfx) {
  unsigned long a_n = (unsigned long)&n;

  std::cout << pfx;
  printf("> Entering fib(%d). Address of n is %lx.\n",n,a_n);

  int return_value;
  if (n == 0 || n == 1) {
    return_value = n;
  } else {
    return_value = fib(n-2,pfx+"==") + fib(n-1,pfx+"==");
  }

  std::cout << pfx;
  printf("> Exiting fib(%d). Returning %d.\n",n,return_value);

  return return_value;
}
int main() {
  int blah = fib(5,"=");
  std::cout << blah << std::endl;
}
```

EXAMPLE: instrumented_fib.cc OUTPUT

```
=> Entering fib(5). Address of n is 7fff5622568c.
===> Entering fib(3). Address of n is 7fff562255ec.
=====> Entering fib(1). Address of n is 7fff5622554c.
=====> Exiting fib(1). Returning 1.
=====> Entering fib(2). Address of n is 7fff5622554c.
=======> Entering fib(0). Address of n is 7fff562254ac.
=======> Exiting fib(0). Returning 0.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=====> Exiting fib(2). Returning 1.
===> Exiting fib(3). Returning 2.
===> Entering fib(4). Address of n is 7fff562255ec.
=====> Entering fib(2). Address of n is 7fff5622554c.
=======> Entering fib(0). Address of n is 7fff562254ac.
=======> Exiting fib(0). Returning 0.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=====> Exiting fib(2). Returning 1.
=====> Entering fib(3). Address of n is 7fff5622554c.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=======> Entering fib(2). Address of n is 7fff562254ac.
=========> Entering fib(0). Address of n is 7fff5622540c.
=========> Exiting fib(0). Returning 0.
=========> Entering fib(1). Address of n is 7fff5622540c.
=========> Exiting fib(1). Returning 1.
=======> Exiting fib(2). Returning 1.
=====> Exiting fib(3). Returning 2.
===> Exiting fib(4). Returning 3.
=> Exiting fib(5). Returning 5.
```

EXAMPLE: instrumented_fib.cc OUTPUT

```
=> Entering fib(5). Address of n is 7fff5622568c.
===> Entering fib(3). Address of n is 7fff562255ec.
=====> Entering fib(1). Address of n is 7fff5622554c.
=====> Exiting fib(1). Returning 1.
=====> Entering fib(2). Address of n is 7fff5622554c.
=======> Entering fib(0). Address of n is 7fff562254ac.
=======> Exiting fib(0). Returning 0.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=====> Exiting fib(2). Returning 1.
===> Exiting fib(3). Returning 2.
===> Entering fib(4). Address of n is 7fff562255ec.
=====> Entering fib(2). Address of n is 7fff5622554c.
=======> Entering fib(0). Address of n is 7fff562254ac.
=======> Exiting fib(0). Returning 0.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=====> Exiting fib(2). Returning 1.
=====> Entering fib(3). Address of n is 7fff5622554c.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=======> Entering fib(2). Address of n is 7fff562254ac.
=========> Entering fib(0). Address of n is 7fff5622540c.
=========> Exiting fib(0). Returning 0.
=========> Entering fib(1). Address of n is 7fff5622540c.
=========> Exiting fib(1). Returning 1.
=======> Exiting fib(2). Returning 1.
=====> Exiting fib(3). Returning 2.
===> Exiting fib(4). Returning 3.
=> Exiting fib(5). Returning 5.
```

EXAMPLE: instrumented_fib.cc OUTPUT

```
=> Entering fib(5). Address of n is 7fff5622568c.
===> Entering fib(3). Address of n is 7fff562255ec.
=====> Entering fib(1). Address of n is 7fff5622554c.
=====> Exiting fib(1). Returning 1.
=====> Entering fib(2). Address of n is 7fff5622554c.
=======> Entering fib(0). Address of n is 7fff562254ac.
=======> Exiting fib(0). Returning 0.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=====> Exiting fib(2). Returning 1.
===> Exiting fib(3). Returning 2.
===> Entering fib(4). Address of n is 7fff562255ec.
=====> Entering fib(2). Address of n is 7fff5622554c.
=======> Entering fib(0). Address of n is 7fff562254ac.
=======> Exiting fib(0). Returning 0.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=====> Exiting fib(2). Returning 1.
=====> Entering fib(3). Address of n is 7fff5622554c.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=======> Entering fib(2). Address of n is 7fff562254ac.
=========> Entering fib(0). Address of n is 7fff5622540c.
=========> Exiting fib(0). Returning 0.
=========> Entering fib(1). Address of n is 7fff5622540c.
=========> Exiting fib(1). Returning 1.
=======> Exiting fib(2). Returning 1.
=====> Exiting fib(3). Returning 2.
===> Exiting fib(4). Returning 3.
=> Exiting fib(5). Returning 5.
```

EXAMPLE: instrumented_fib.cc OUTPUT

```
=> Entering fib(5). Address of n is 7fff5622568c.
===> Entering fib(3). Address of n is 7fff562255ec.
=====> Entering fib(1). Address of n is 7fff5622554c.
=====> Exiting fib(1). Returning 1.
=====> Entering fib(2). Address of n is 7fff5622554c.
=======> Entering fib(0). Address of n is 7fff562254ac.
=======> Exiting fib(0). Returning 0.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=====> Exiting fib(2). Returning 1.
===> Exiting fib(3). Returning 2.
===> Entering fib(4). Address of n is 7fff562255ec.
=====> Entering fib(2). Address of n is 7fff5622554c.
=======> Entering fib(0). Address of n is 7fff562254ac.
=======> Exiting fib(0). Returning 0.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=====> Exiting fib(2). Returning 1.
=====> Entering fib(3). Address of n is 7fff5622554c.
=======> Entering fib(1). Address of n is 7fff562254ac.
=======> Exiting fib(1). Returning 1.
=======> Entering fib(2). Address of n is 7fff562254ac.
=========> Entering fib(0). Address of n is 7fff5622540c.
=========> Exiting fib(0). Returning 0.
=========> Entering fib(1). Address of n is 7fff5622540c.
=========> Exiting fib(1). Returning 1.
=======> Exiting fib(2). Returning 1.
=====> Exiting fib(3). Returning 2.
===> Exiting fib(4). Returning 3.
=> Exiting fib(5). Returning 5.
```

INTERPRETING instrumented_fib.cc'S OUTPUT

The base 16 (hexadecimal) representation of the addresses where n live are the following:

```
7fff5622568c
7fff562255ec
7fff5622554c
7fff562254ac
7fff5622540c
```

These successive addresses of **`n`** within the call stack hint that:

- ➡The call stack grows downward with each function call.
- ➡The call stack retreats upward with each **`return`**.
- ➡It is laid out from higher addresses to lower addresses in memory.
- ➡Each stack frame for **`fib`** is 160 bytes. (`0c` to `ac` is 10x16 bytes)

INTERPRETING instrumented_fib.cc'S OUTPUT

The base 16 (hexadecimal) representation of the addresses where n live are the following:

- `7fff5622568c` Depth 0 call. This is a higher memory address.
- `7fff562255ec` Depth 1 call.
- `7fff5622554c` Depth 2 call.
- `7fff562254ac` Depth 3 call.
- `7fff5622540c` Depth 4 call. This is a lower memory address.

These successive addresses of `n` within the call stack hint that:

- ➡The call stack grows downward with each function call.
- ➡The call stack retreats upward with each `return`.
- ➡It is laid out from higher addresses to lower addresses in memory.
- ➡Each stack frame for `fib` is 160 bytes. (`0c` to `ac` is 10x16 bytes)

NEXT TIME

C++ has "aggregate" data structures that were originally in C.

- An **array** is a primitive form of list/sequence/vector.
- A **struct** is a primitive form of object.
- (Incidentally, a C++ **object** is a richer form of a struct.)

- To use these we need to understand how they are laid out in memory.
- We'll look at stack-allocated arrays and structs first.
  - Later we'll allocate them on the **heap** also/instead.
- We'll play with the **&** operator some more too.