Ve 280

Programming and Introductory Data Structures

Function Call Mechanism;

Enum;

Passing Arguments to Program

Outline

- Function Call Mechanism
- Categorizing Data: `enum`
- Passing Arguments to Program

Call Stacks

How a function call really works

- When we call a function, the program does following steps:

1. Evaluate the actual arguments to the function (order is not guaranteed). Example: y = add(4-1, 5);
2. Create an "**activation record**" (sometimes called a "**stack frame**") to hold the function's formal parameters and local variables.
   - When call function `int add(int a, int b)`, system creates an activation record: a, b (formal), result (local)
3. Copy the actuals' values to the formals' storage space. a=3 b=5
4. Evaluate the function in its local scope.
5. Replace the function call with the result. y=8
6. Destroy the activation record.

Call Stacks

How a function call really works

- It is typical to have multiple function calls. How the activation records are maintained?
  - Answer: stored as a **stack**.
- Stack: a set of objects which modifies as **last in first out**. Example: a stack of plates in a cafeteria
  - Each time you clean a plate, you add it to the top of the stack
  - Each time a new plate is needed, the one at the top is taken **first**

Call Stacks

How a function call really works

- When a function f() is called, its **activation record** is added to the "top" of the stack.
- When the function f() returns, its **activation record** is removed from the "top" of the stack.
- In the meantime, f() may have called other functions.
  - These functions create corresponding activation records.
  - These functions must return (and destroy their corresponding activation records) before f() can return.

Call Stacks

Example

- When a function is called, its **activation record** is added to the "top" of the stack.
- When that function returns, its **activation record** is removed from the "top" of the stack.

double add(double a, double b): a = 1, b = 0, result = 0
double sin(double x): x = 1, result = 0
int main(): x = 1, sinResult = 0

- Note: "top" is placed in quotes, because in reality, stack of activation records grows **down** rather than **up**.

Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Main starts out with an activation record with room only for the local “result”:

main:

result: 0

Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Then, main calls plus_two, passing the literal value "0":

main:

result: 0

plus_two:

x: 0

Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

Which in turn calls plus_one:

main:

result: 0

plus_two:

x: 0

plus_one:

x: 0

Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_one adds one to x, returning the value 1:

main:

result: 0

plus_two:

x: 0

1

plus_one:

x: 0

Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_one's activation record is destroyed:

main:

result: 0

plus_two:

x: 0

1

plus_one:

x: 0

Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_two adds one to the result, and returns the value 2:

main:

result: 2

2

plus_two:

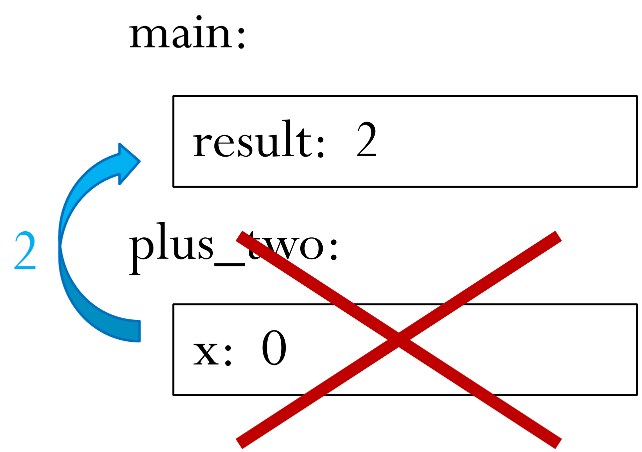
x: 0

Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

plus_two's activation record is destroyed:



Call Stacks

Example

```
int plus_one(int x) {
  return (x+1);
}

int plus_two(int x) {
  return (1 + plus_one(x));
}

int main() {
  int result = 0;

  result = plus_two(0);
  cout << result;
  return 0;
}
```

main then prints the result:

2

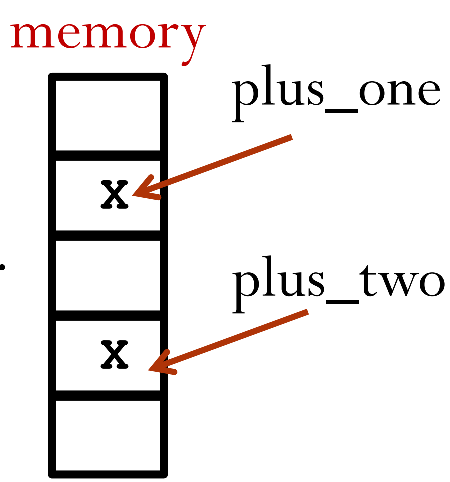
main:

result: 2

Call Stacks

Example: Some things to note

- Even though `plus_one` and `plus_two` both have formal parameters called "x", there is no problem.
  - These two x's are at different locations in memory.
  - `plus_one` cannot see `plus_two`'s x.
  - Instead, the **value** of `plus_two`'s x is passed to `plus_one`, and stored in `plus_one`'s x.

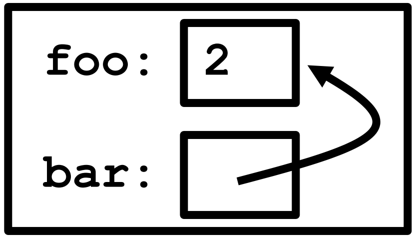


Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

Activation record of main:

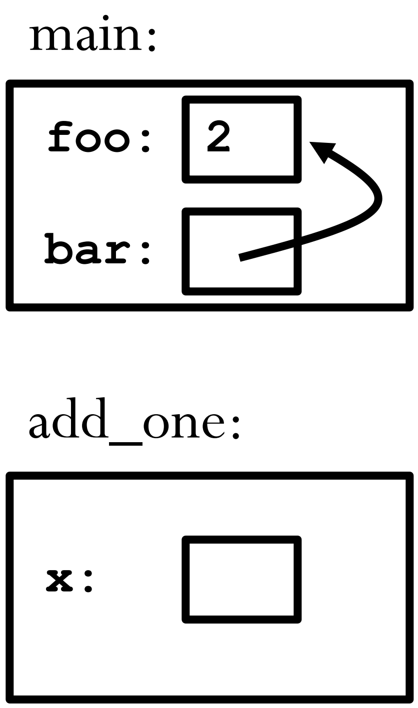


Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

Main calls add_one, creating an activation record for add_one

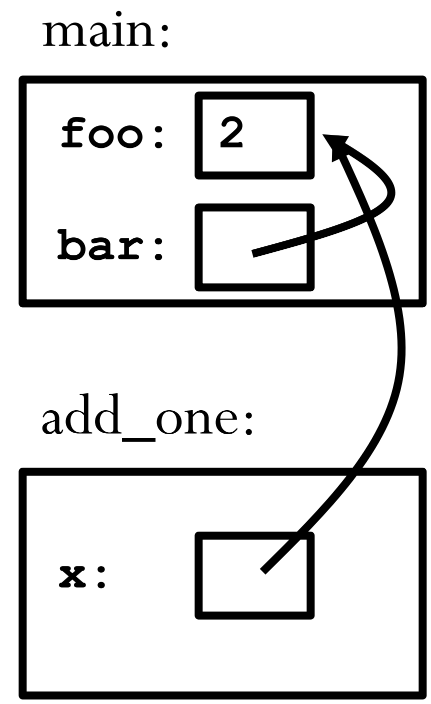


Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

Copy the value of bar to add_one's formal parameter x.



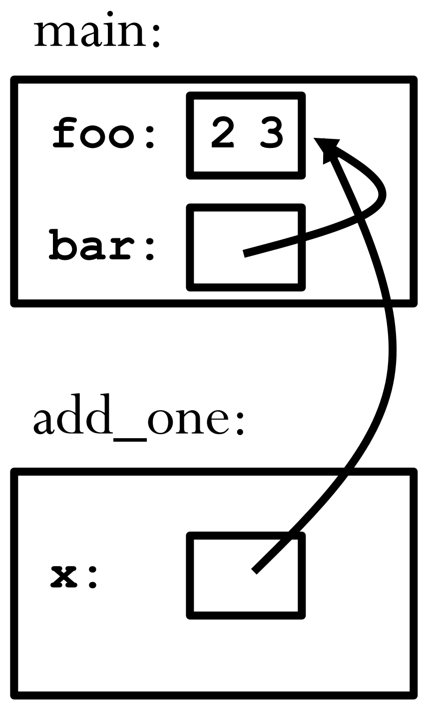
Both x and bar point to foo.

Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

add_one adds 1 to the object pointed to by x.

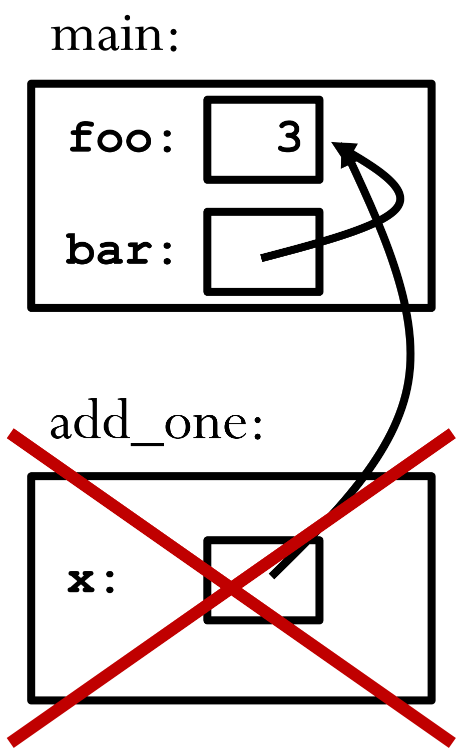


Call Stack

Example: Using Pointers

```
void add_one(int *x) {
  *x = *x + 1;
}

int main() {
  int foo = 2;
  int *bar = &foo;
  add_one(bar);
  return 0;
}
```

add_one's activation record is destroyed.



Call Stack

Example: Recursion

```
main
  x: 
```

- Suppose we call our function as follows:

```
   int main()
{
  int x;
  x = factorial(3);
}
```

```
  int factorial (int n) {
 if (n == 0) return 1;
 else return n*factorial(n-1);
  }
```

Call Stack

Example: Recursion

- main() calls factorial with an argument 3.
- We evaluate the actual argument, create an activation record, and copy the actual value to the formal.

```
main
  x: [ ]
```

```
factorial
  n: [3]
  RA: main line #3
```

RA = "Return Address"

```
   int factorial (int n) {
1.  if (n == 0) return 1;
2.  else return n*factorial(n-1);
   }
```

Call Stack

Example: Recursion

- Now we evaluate the body of factorial:
  - n is not zero, so we evaluate the **else** arm of the if statement:

    return 3 * factorial(2)

  - So, factorial must call factorial. We will create a **new** activation record for a **new** instance of factorial.

```
main
  x: [ ]
```

```
factorial
  n: [3]
  RA: main line #3
```

```
factorial
  n: [2]
  RA: factorial line #2
```

```
  int factorial (int n) {
1. if (n == 0) return 1;
2. else return n*factorial(n-1);
  }
```

Call Stack

Example: Recursion

- Again, n is not zero, so we evaluate the **else** arm again:

  return 2 * factorial(1)

- This creates a new activation record for factorial

```
main
  x: [ ]
```

```
factorial
  n: [3]
  RA: main line #3
```

```
factorial
  n: [2]
  RA: factorial line #2
```

```
factorial
  n: [1]
  RA: factorial line #2
```

```
   int factorial (int n) {
1.  if (n == 0) return 1;
2.  else return n*factorial(n-1);
   }
```

Call Stack

Example: Recursion

- And again, we evaluate the **else** arm:

  return 1*factorial(0)

- This creates a new activation record for factorial

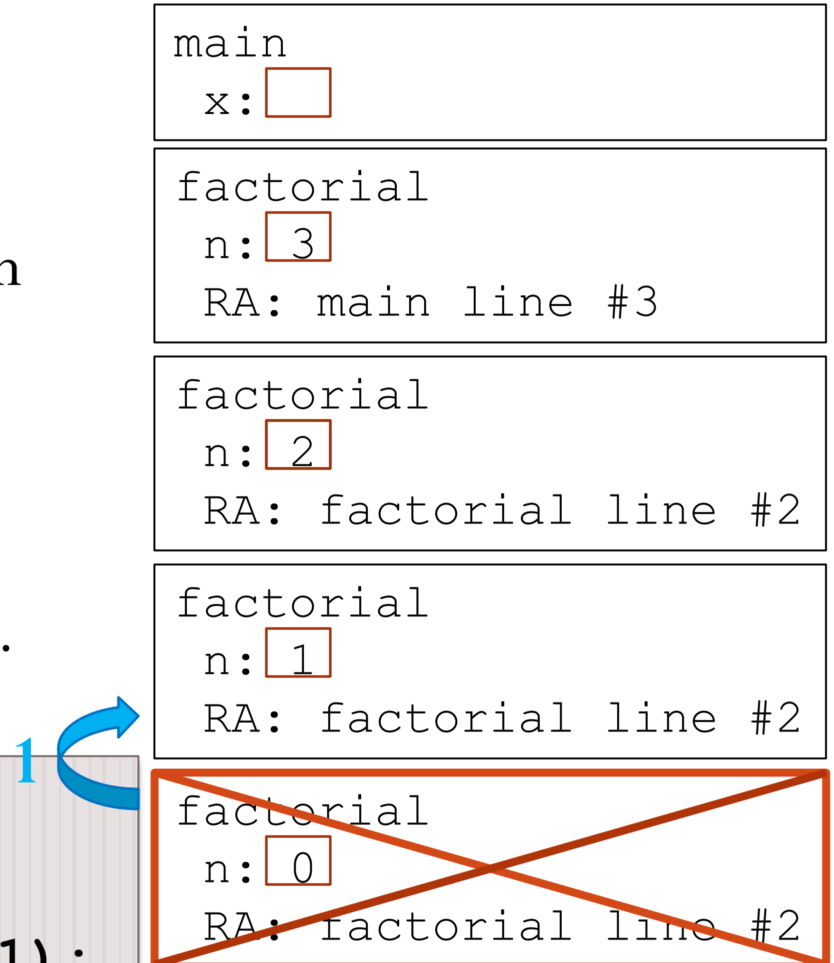
```
   int factorial (int n) {
1.  if (n == 0) return 1;
2.  else return n*factorial(n-1);
   }
```

```
main
  x: [ ]

factorial
  n: [3]
  RA: main line #3

factorial
  n: [2]
  RA: factorial line #2

factorial
  n: [1]
  RA: factorial line #2

factorial
  n: [0]
  RA: factorial line #2
```

Call Stack

Example: Recursion

- In evaluating factorial(0), n is zero, so we evaluate the **if** arm rather than **else** arm.
- Return the value "1"
- Popping the most recent activation record off the stack.




```
   int factorial (int n) {
1.  if (n == 0) return 1;
2.  else return n*factorial(n-1);
   }
```

Call Stack

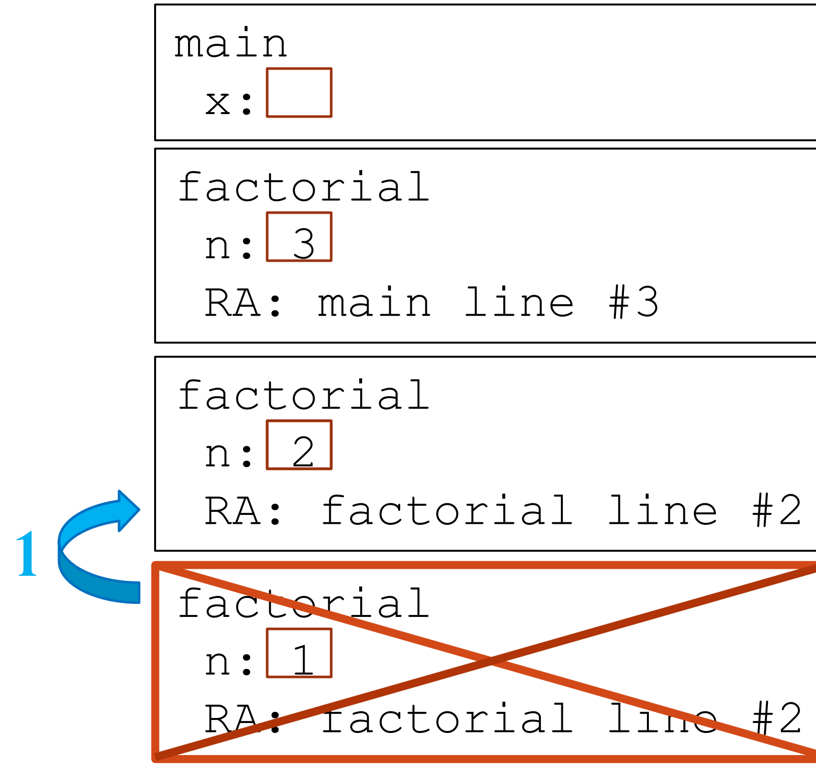
Example: Recursion

- In factorial(1), we called factorial(0) as follows:
  return 1 * factorial(0)

- Now we know the value of factorial(0), so we complete factorial(1):

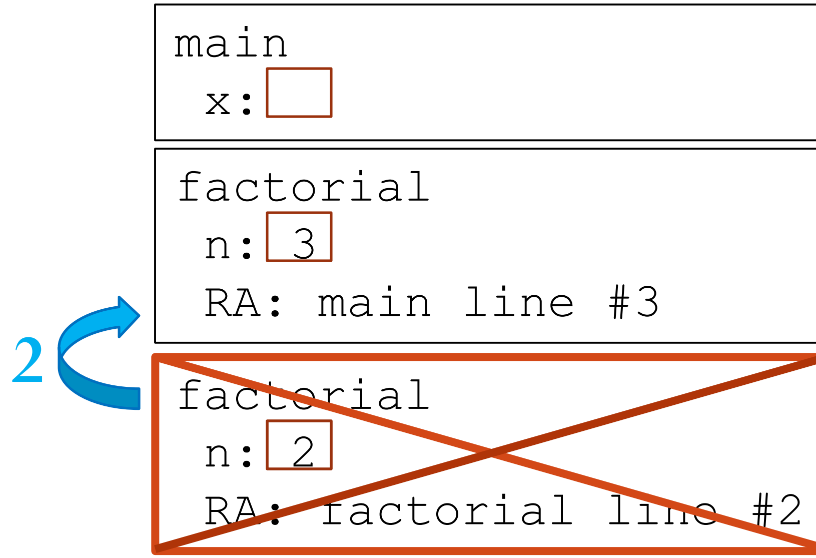
  return 1 * 1 => return 1;
  from factorial(1)

- This pops another activation record off the stack



Call Stack

Example: Recursion



- Now it allows us to complete evaluating factorial(2):

  return 2 * factorial(1) =>

  return 2 * 1 =>

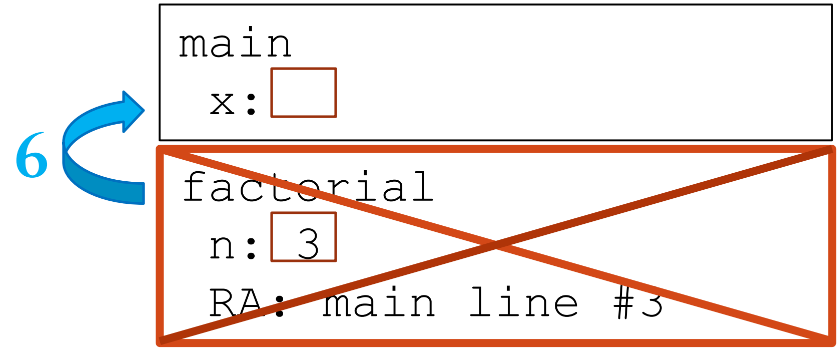
  return 2

  from factorial(2)

- Now pop off another activation record.

Call Stack

Example: Recursion



- Now we can complete evaluating factorial(3):

  return 3 * factorial(2) =>

  return 3 * 2 =>

  return 6

- That is the correct answer.

- Don’t forget that last pop!

Outline

- Function Call Mechanism
- Categorizing Data: `enum`
- Passing Arguments to Program

Categorizing Data

Introducing enums

- In addition to single constants, we may need to categorize data.

- For example, there are four different suits in cards:
  - Clubs ♣
  - Diamonds ♦
  - Hearts ♥
  - Spades ♠

- You could encode each of these as a separate integer like:

```
const int CLUBS = 0;
const int DIAMONDS = 1;
// and so on...
```

Categorizing Data

Introducing enums

```
const int CLUBS = 0;
const int DIAMONDS = 1;
```

- Unfortunately, encoding information this way is not very convenient.

- For example, consider the predicate `isRed()`

```
bool isRed(int suit);
 // REQUIRES: suit is one of Clubs,
 //           Diamonds, Hearts,
 //           or Spades
 // EFFECTS:  returns true if the color
 //           of this suit is red.
```

Categorizing Data

Introducing enums

```
const int CLUBS = 0;
const int DIAMONDS = 1;

bool isRed(int suit);
// REQUIRES: suit is one of Clubs,
//           Diamonds, Hearts, or Spades
// EFFECTS:  returns true if the color
//           of this suit is red.
```

- This is annoying, since we need this REQUIRES clause; not all integers encode a suit.
- There is a better way: the **enumeration** (or **enum**) type.

Categorizing Data

enums

- You can define an enumeration type as follows:

```
enum Suit_t {CLUBS, DIAMONDS,
             HEARTS, SPADES};
```

- To define variables of this type you say:

```
enum Suit_t suit;
```

- You can initialize them as:

```
enum Suit_t suit = DIAMONDS;
```

- Once you have such an enum type defined, you can use it as an argument, just like anything else.
- Enums are passed by-value, and can be assigned.

Categorizing Data

enums

- With enum, the specification for the function `isRed()` can be simplified by removing the REQUIRES clause.

```
bool isRed(enum Suit_t s);
// EFFECTS:  returns true if the color
//           of this suit is red.
```

Categorizing Data

enums

```
bool isRed(enum Suit_t s) {
  switch (s) {
      case DIAMONDS:
      case HEARTS:
            return true;
            break;
      case CLUBS:
      case SPADES:
            return false;
            break;
      default:
            assert(0);
            break;
  }
}
```

Categorizing Data

enums

- If you write

```
enum Suit_t {CLUBS, DIAMONDS,
             HEARTS, SPADES};
```

then numerically

```
CLUBS = 0, DIAMONDS = 1,
HEARTS = 2, SPADES = 3
```

- Using this fact, it will sometimes make life easier

```
enum Suit_t s = CLUBS;
const string suitname[] = {"clubs",
    "diamonds", "hearts", "spades"};
cout << "suit s is " << suitname[s];
```

Outline

- Function Call Mechanism
- Categorizing Data: `enum`
- Passing Arguments to Program

Passing Arguments to Program

Introduction

- So far, we have considered programs that take no arguments
  - You run your program like: ./program

- However, programs can take arguments.

- For example, many Linux commands are programs and they take arguments!
  - **`diff file1 file2`**
  - **`rm file`**
  - …

Passing Arguments to Program

Introduction

```
diff file1 file2
```

- The first word, `diff`, is the **name** of the program to run.
- The second and third words are **arguments** to the `diff` program.
- These arguments are passed to `diff` for its consideration, like arguments are passed to functions.
- The operating system collects arguments and passes them to the program it executes.

Passing Arguments to Program

- Arguments are passed to the program through main() function.

- We need to change the argument list of main():
  - Old: `int main()`
  - New: `int main(int argc, char *argv[])`

Passing Arguments to Program

```
int main(int argc, char *argv[])
```

- Each argument is just a sequence of characters.
- All the arguments (including program name) form an array of C-strings.
- `int argc:` the number of strings in the array
  - E.g., `diff file1 file2: argc = 3`
  - The name `argc` is by convention and it stands for "argument count".

Passing Arguments to Program

```
int main(int argc, char *argv[])
```

- `argv` stores the array of C-strings.
  - Remember, a C-string is itself an array of char and it can be thought of as a pointer to char.
  - Thus, an array of C-strings can be thought of as an array of pointers to char.
  - Thus, `argv` is an array of pointers to char: `char *argv[]`
  - The name `argv` is again by convention and it is short for "argument vector" or "argument values".

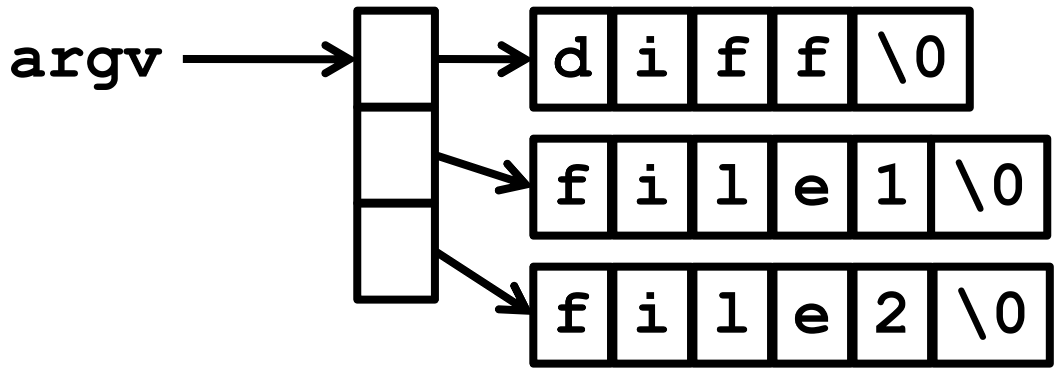
Passing Arguments to Program

argv

`diff file1 file2`

`char *argv[]`

- Pictorially, this would look like the following in memory:



<u>Note</u>: argv[0] is the name of the program being executed.

Passing Arguments to Program

Example

- Suppose we wanted to write a program that is given a list of integers as its arguments, and prints out the sum of that list.

- Before we can write this program we need a way to convert from C-strings to integers.
- We use predefined "standard library" function called `atoi()`.
- It's specification is

```
int atoi(const char *s);
// EFFECTS: parses s as a number and
//          returns its int value
```

  - Need to `#include <cstdlib>`

Passing Arguments to Program

Example

- The problem we are examining can be solved as:

```
int main (int argc, char *argv[])
{
  int sum = 0;
  for (int i = 1; i < argc; i++) {
      sum += atoi(argv[i]);
  }
  cout << "sum is " << sum;
  return 0;
}
```

Passing Arguments to Program

Example

```
int main (int argc, char *argv[]) {
        int sum = 0;
        for (int i = 1; i < argc; i++) {
                sum += atoi(argv[i]);
        }
        cout << "sum is " << sum;
        return 0;
}
```

- Finally, we save it to `sumIt.C`, compile, and run it:

```
$ g++ -o sumIt sumIt.C
$ ./sumIt 3 10 11 12 19
```

- It will print "sum is 55".

Question: What is argc? What is argv[0]?

References

- enum
  - C++ Primer, 4th Edition, Chapter 2.7

- Command-Line Arguments
  - Absolute C++, 4th Edition, Page 373