**A) What is the difference between passing arguments by arguments and passing argument by reference?**

When you call a function, you can pass arguments to it in two different ways: by value and by reference.

Passing arguments by value means that the function creates a copy of the argument value and works with that copy inside the function. Any changes made to the copy do not affect the original value of the argument outside of the function.

Passing arguments by reference means that the function receives a reference or a pointer to the original argument, rather than a copy. Any changes made to the referenced value inside the function are also reflected in the original value outside of the function.

Here is an example code snippet in C++ that demonstrates the difference between passing arguments by value and by reference:

For example

```cpp
#include <iostream>

using namespace std;

void squareByValue(int x) {
    x = x * x;
}

void squareByReference(int& x) {
    x = x * x;
}

int main() {
    int num = 5;
    cout << "Original value of num: " << num << endl;

    squareByValue(num);
    cout << "Value of num after squareByValue: " << num << endl;

    squareByReference(num);
    cout << "Value of num after squareByReference: " << num << endl;

    return 0;      You, 8 seconds ago • Uncommitted changes
}
```

In this example, the squareByValue() function receives an integer x by value and creates a copy of num inside the function. Therefore, any changes made to x inside the function do not affect the original value of num outside of the function. The output of the program is:

For example:

```
Original value of num: 5
Value of num after squareByValue: 5
Value of num after squareByReference: 25
```

On the other hand, the squareByReference() function receives an integer reference x and modifies the original value of num through that reference.

Therefore, the changes made to x inside the function are reflected in the original value of num outside of the function. The output of the program shows that the value of num is updated to the square of its original value after calling squareByReference().

In summary, passing arguments by value creates a copy of the original value and works with that copy inside the function, while passing arguments by reference allows the function to modify the original value directly.

**B) What values does the RAND function generate?**

The rand() function is a standard library function in C and C++ that generates a sequence of pseudo-random integer values. The range of values generated by the rand() function depends on the implementation and the system on which the program is running, but the minimum value is guaranteed to be 0, and the maximum value is guaranteed to be at least 32767.

In C++, the maximum value returned by rand() is defined in the RAND_MAX constant, which is typically a large integer value such as 32767. To generate a random integer between 0 and a specific upper bound, you can use the modulo operator (%) to reduce the range of the output values.

For example, to generate a random integer between 0 and 99, you can use the following code snippet

```cpp
#include <cstdlib> // for rand() and srand()
#include <ctime> // for time()

int main() {
   // seed the random number generator with the current time
   srand(time(NULL));

   // generate a random integer between 0 and 99
   int random_num = rand() % 100;

   return 0;
}
```

### C) How do you randomize a program? How do you scale or shift the values produced by the rand function?

To randomize a program, you can use a random number generator function to generate a sequence of random numbers. One way to do this is by using the rand() function in your programming language.

The rand() function generates a pseudo-random integer between 0 and the maximum value supported by the programming language. For example, in C++, the maximum value returned by rand() is defined in the RAND_MAX constant. To generate a random integer between 0 and a specific upper bound, you can use the modulo operator (%) to reduce the range of the output values. Here is an example code snippet in C++:

For example

```cpp
#include <cstdlib> // for rand() and srand()
#include <ctime> // for time()

int main() {
   // seed the random number generator with the current time
   srand(time(NULL));

   // generate a random integer between 0 and 99
   int random_num = rand() % 100;

   return 0;
}
```

To scale or shift the values produced by the rand() function, you can multiply or add a constant value to the output of the function. For example, to generate a random floating-point number between 0 and 1, you can divide the output of the rand() function by the maximum value and add a shift value. Here is an example code snippet in C++:

For Example

```cpp
#include <cstdlib> // for rand() and srand()
#include <ctime> // for time()

int main() {
   // seed the random number generator with the current time
   srand(time(NULL));

   // generate a random floating-point number between 0 and 1
   double random_num = static_cast<double>(rand()) / RAND_MAX;

   // scale the random number to a different range
   double scaled_num = random_num * 10.0 + 5.0;

   return 0;
}
```

In this example, the static_cast<double>(rand()) / RAND_MAX expression generates a random number between 0 and 1, and the scaled_num = random_num * 10.0 + 5.0 expression scales and shifts the random number to the range of 5 to 15.

**D) What is a recursive function? What is a base case?**

A recursive function is a function that calls itself within its own code. Recursive functions are used when a problem can be broken down into smaller and smaller sub-problems until the solution becomes trivial. Each recursive call handles a smaller part of the problem, and the results of the smaller sub-problems are combined to solve the larger problem.

A recursive function typically consists of two parts: the base case and the recursive case. The base case is a condition that is checked at the beginning of

each recursive call, and if the condition is met, the function returns a specific value without making any further recursive calls. The base case serves as a stopping condition to prevent the function from calling itself indefinitely and causing a stack overflow.

The recursive case, on the other hand, is the part of the function that makes one or more recursive calls to itself with a smaller version of the original problem. The function continues to call itself until it reaches the base case and returns a value.

Here is an example of a simple recursive function to calculate the factorial of a non-negative integer:

```
int factorial(int n) {
   // base case: if n is 0 or 1, return 1
   if (n == 0 || n == 1) {
      return 1;
   }
   // recursive case: call factorial(n-1) and multiply by n
   else {
      return n * factorial(n-1);
   }
}
```

In this example, the base case is when n is 0 or 1, and the function returns 1 without making any further recursive calls. The recursive case is when n is greater than 1, and the function calls itself with n-1 as the argument, multiplies the result by n, and returns the value. The function continues to call itself with smaller values of n until it reaches the base case and returns a value.

In summary, a recursive function is a function that calls itself to solve a problem by breaking it down into smaller sub-problems. The base case is a stopping

condition that prevents the function from calling itself indefinitely, while the recursive case is the part of the function that makes one or more recursive calls to solve the problem.