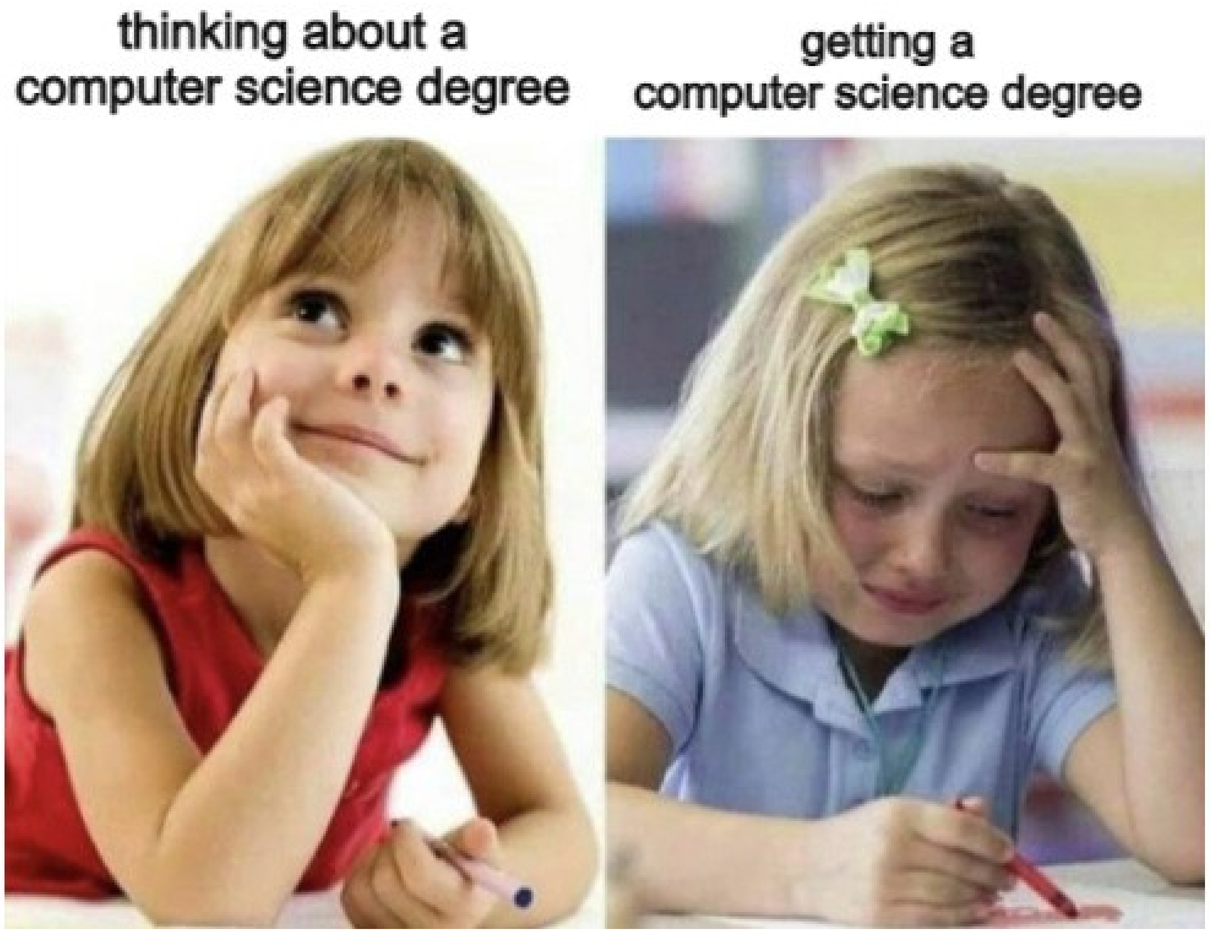
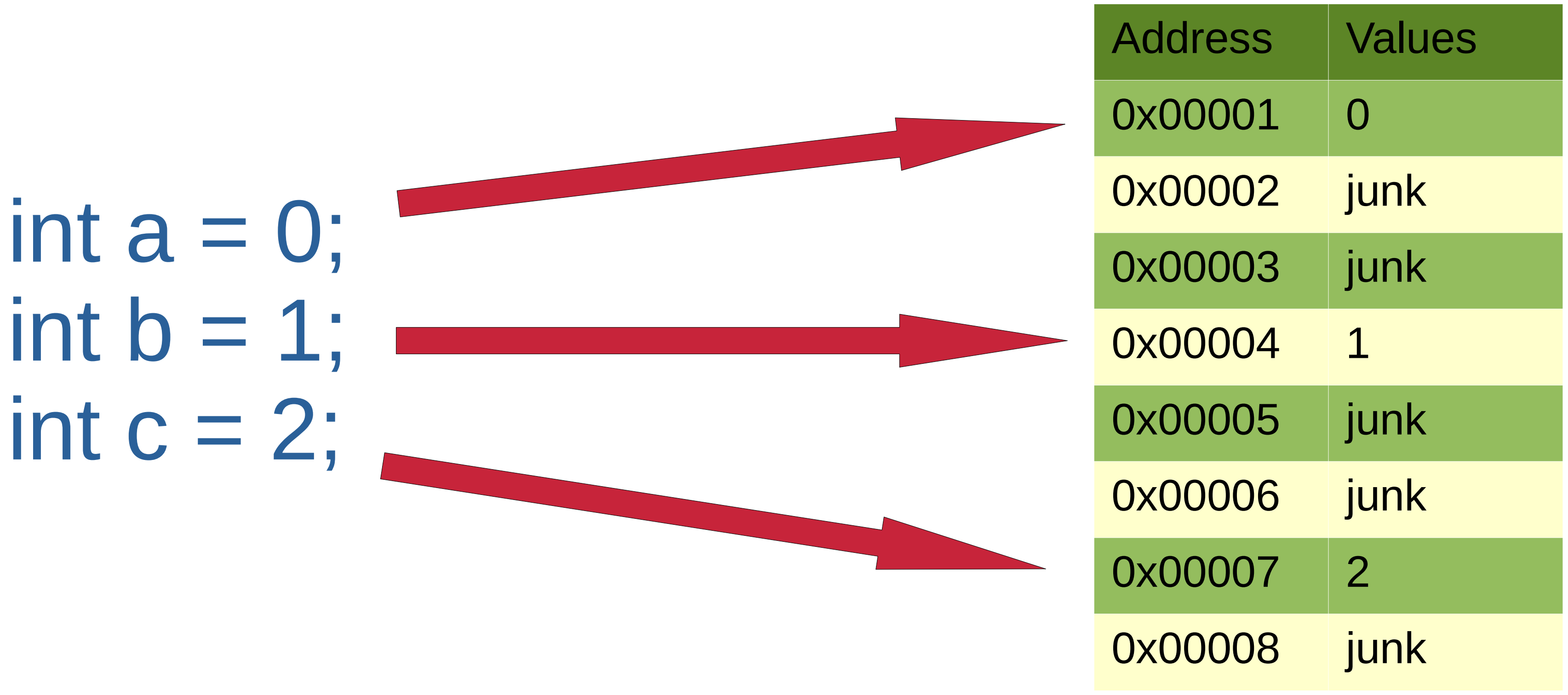

thinking about a computer science degree
getting a computer science degree

CS1428
Foundation of Computer Science

Lecture 8: Passing by Reference

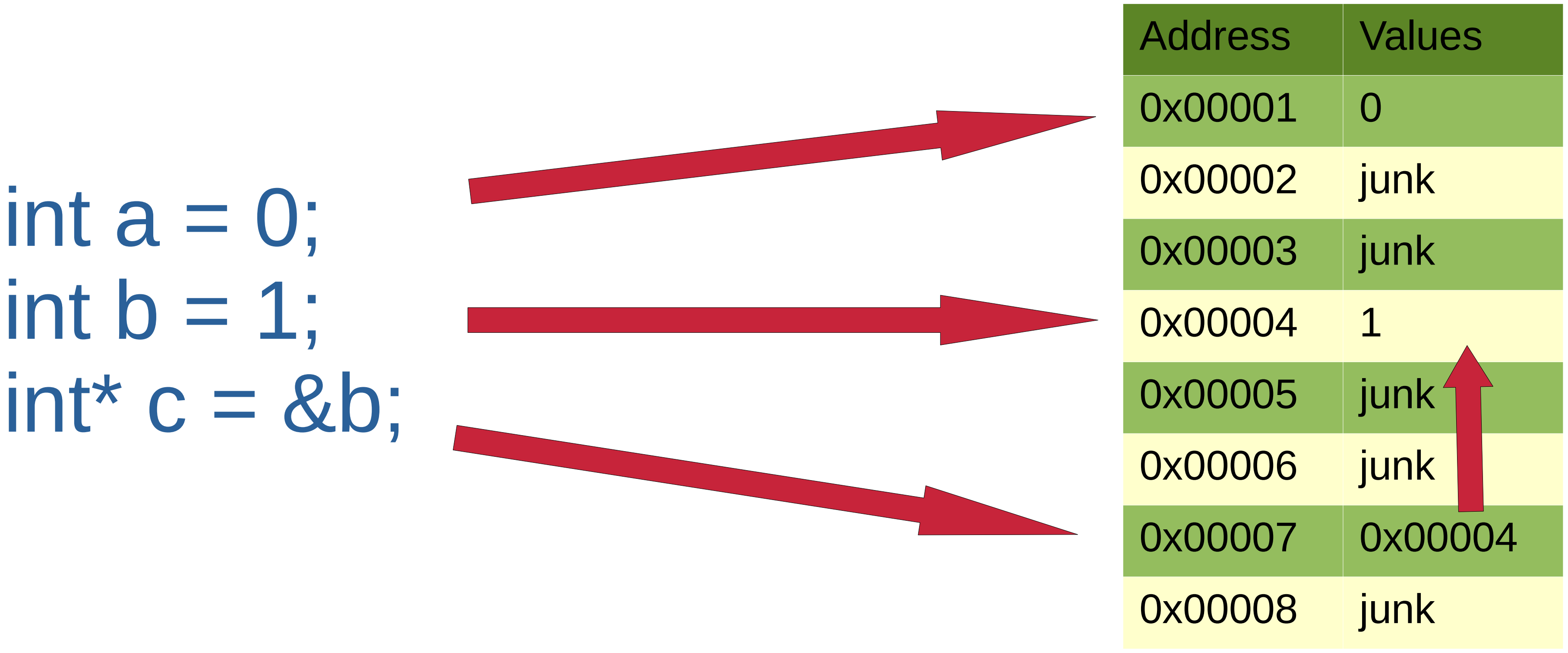
Variables in Memory:



```
int a = 0;
int b = 1;
int c = 2;
```

Address	Values
0x00001	0
0x00002	junk
0x00003	junk
0x00004	1
0x00005	junk
0x00006	junk
0x00007	2
0x00008	junk

Pointers in Memory:



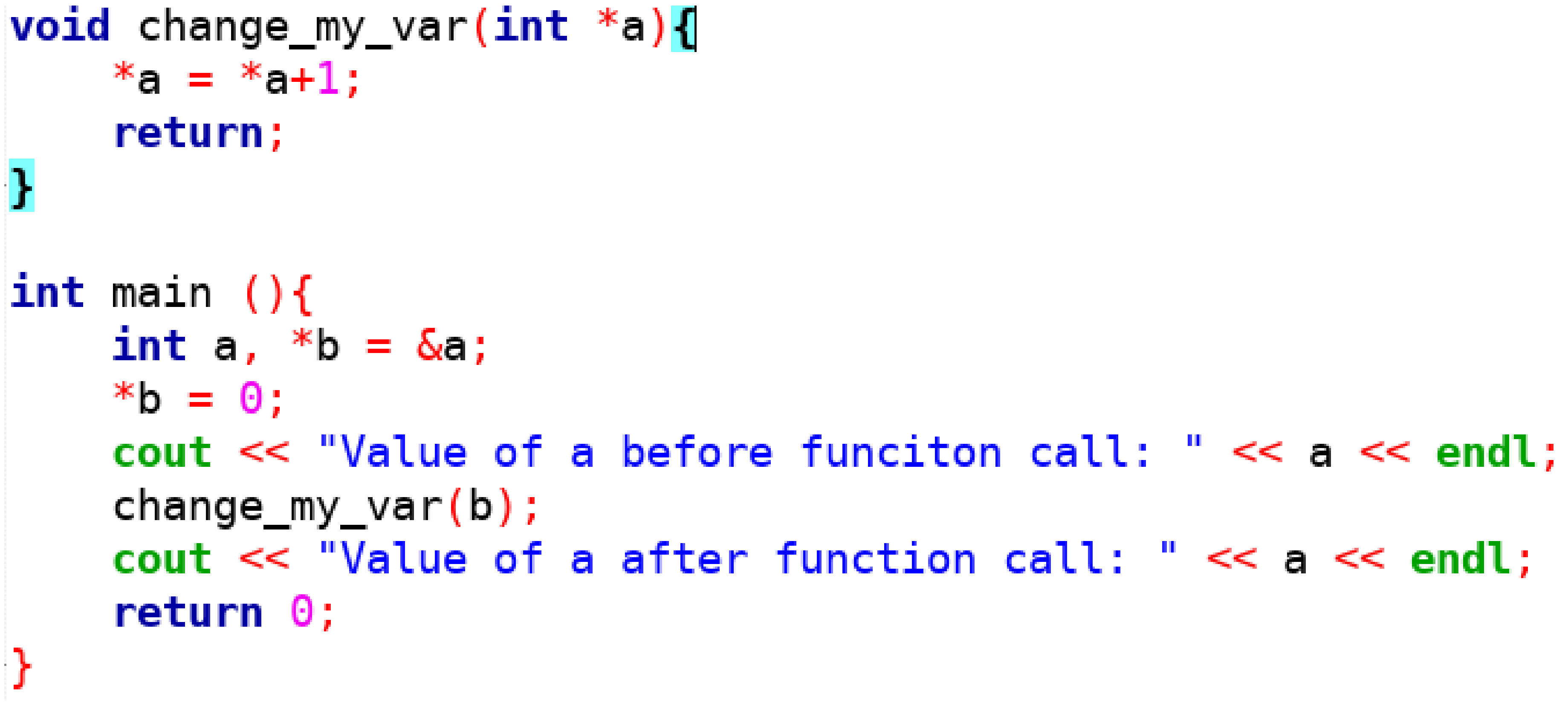
```
int a = 0;
int b = 1;
int* c = &b;
```

Address	Values
0x00001	0
0x00002	junk
0x00003	junk
0x00004	1
0x00005	junk
0x00006	junk
0x00007	0x00004
0x00008	junk

Pointers in C++:

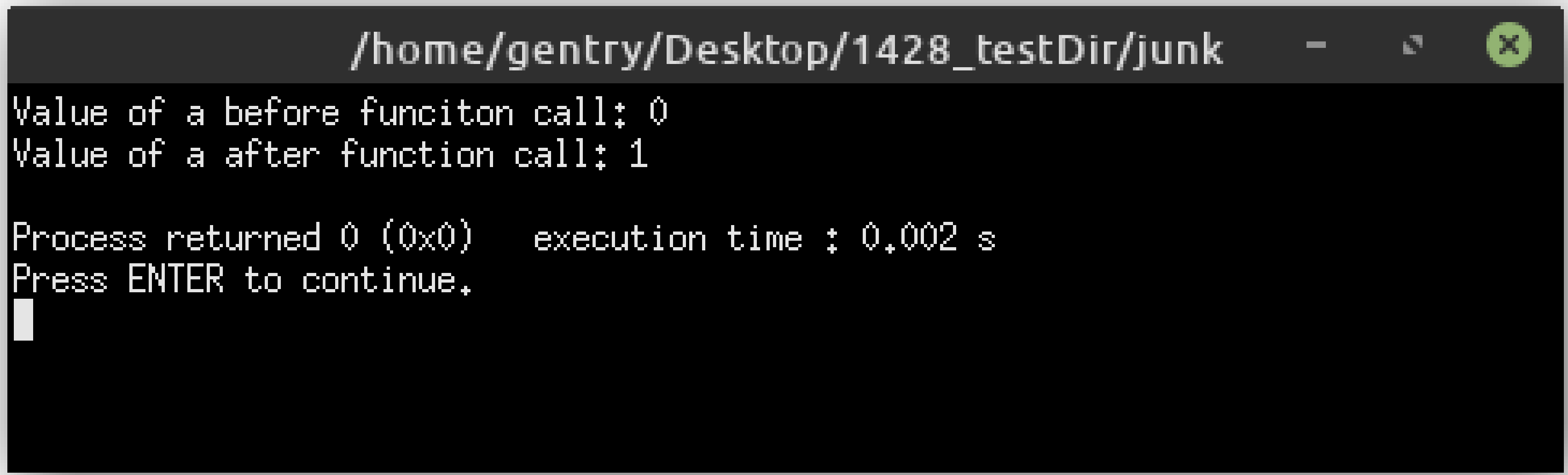
- Imagine a variable that stored the **address** of a memory location, rather than the **value** stored there.
- We call this kind of variable a “pointer” because it points to a specific memory location.
- Pointers are created by putting an asterisk * in front of a variable name.
- Passing a pointer to a function allows the function alter the value of a variable created in the calling functions.

Pointer Parameters:



```
void change_my_var(int *a){
    *a = *a+1;
    return;
}

int main (){
    int a, *b = &a;
    *b = 0;
    cout << "Value of a before funciton call: " << a << endl;
    change_my_var(b);
    cout << "Value of a after function call: " << a << endl;
    return 0;
}
```

Pointer Output:


/home/gentry/Desktop/1428_testDir/junk

```
Value of a before funciton call: 0
Value of a after function call: 1

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

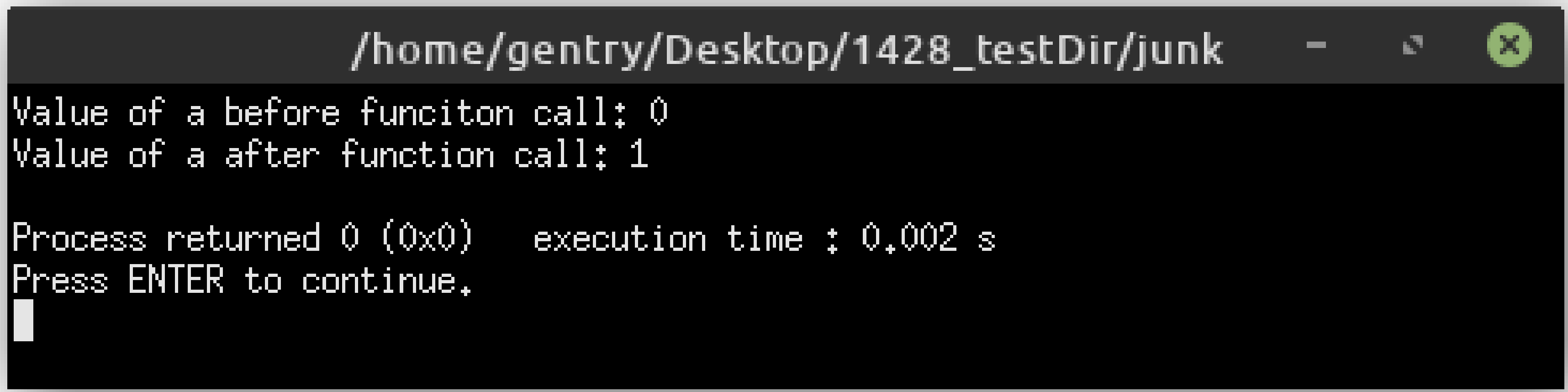
The Problem with Pointers:

- Pointers give us the ability to directly control the values written at particular memory addresses.
- This is very powerful but also very easy to misuse. Referencing an illegal location can cause a **segmentation fault**.
- C++ let’s us declare that a parameter is “passed by reference” which has the same effect without worrying about pointers.

By-reference Parameters:

```
void change_my_var(int &a){
    a = a+1;
    return;
}

int main (){
    int a= 0;
    cout << "Value of a before funciton call: " << a << endl;
    change_my_var(a);
    cout << "Value of a after function call: " << a << endl;
    return 0;
}
```

By-reference Output:


/home/gentry/Desktop/1428_testDir/junk

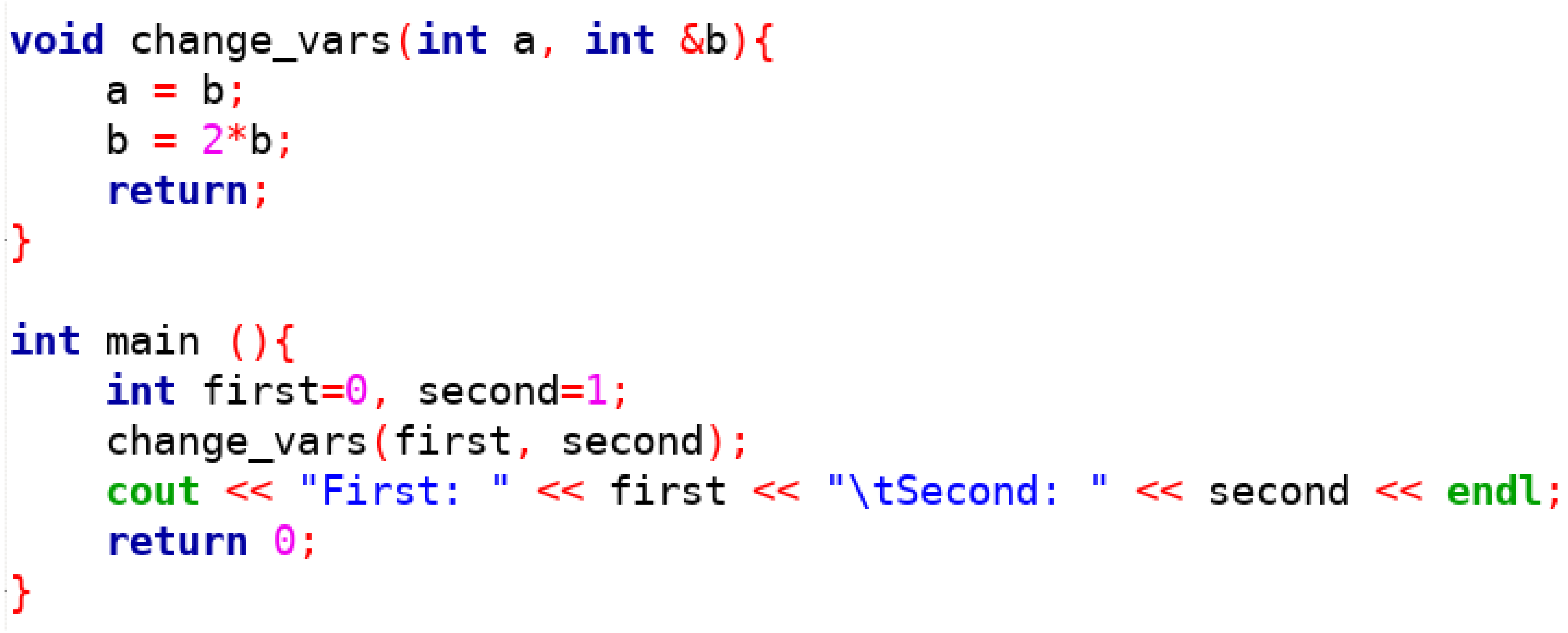
```
Value of a before funciton call: 0
Value of a after function call: 1

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

By Value vs By Reference:

- Function arguments are copied into new variables.
- Function parameters are only visible to functions.
- Changes to variables passed as arguments only affect the function.

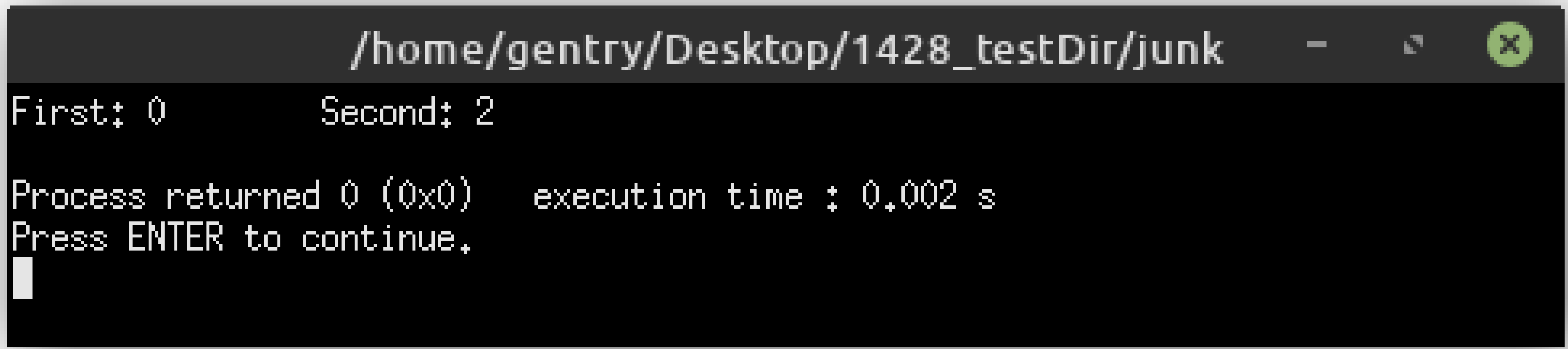
- Functions are passed the memory location of arguments.
- Functions can alter the value of variables <u>in other scopes</u>.
- By reference parameters use &.

By-reference vs. By-value:



```
void change_vars(int a, int &b){
    a = b;
    b = 2*b;
    return;
}

int main (){
    int first=0, second=1;
    change_vars(first, second);
    cout << "First: " << first << "\tSecond: " << second << endl;
    return 0;
}
```

By-reference vs. By-value:

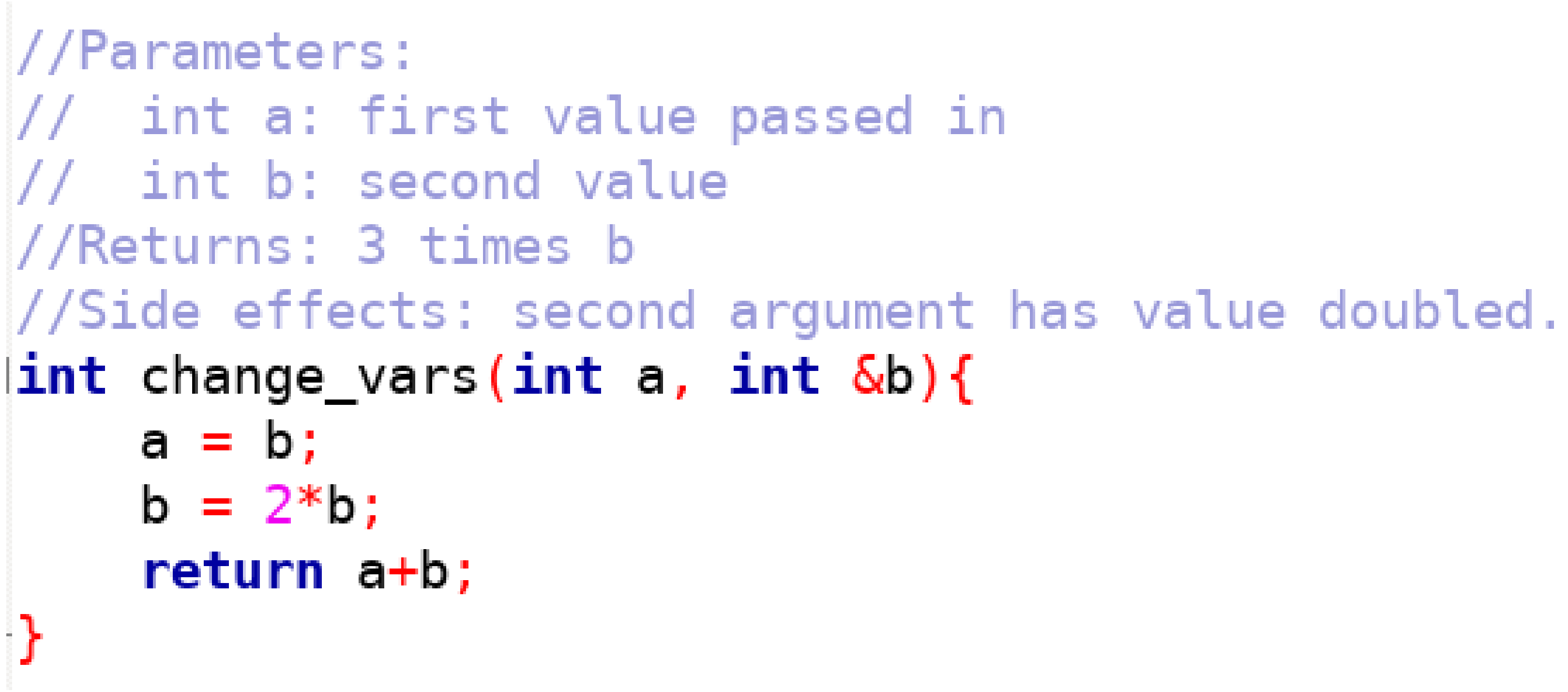

```
/home/gentry/Desktop/1428_testDir/junk
```

```
First: 0        Second: 2

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

Side Effects:

- We have been discussing functions only in terms of inputs(parameters) and outputs(return).
- Functions can also affect programs in other ways. We call these other ways “side effects”
- Examples of Side effects:
  - Changes to by-reference variables.
  - Console and file output.
  - Changes to global variables.

Function Comments:



```cpp
//Parameters:
//  int a: first value passed in
//  int b: second value
//Returns: 3 times b
//Side effects: second argument has value doubled.
int change_vars(int a, int &b){
    a = b;
    b = 2*b;
    return a+b;
}
```

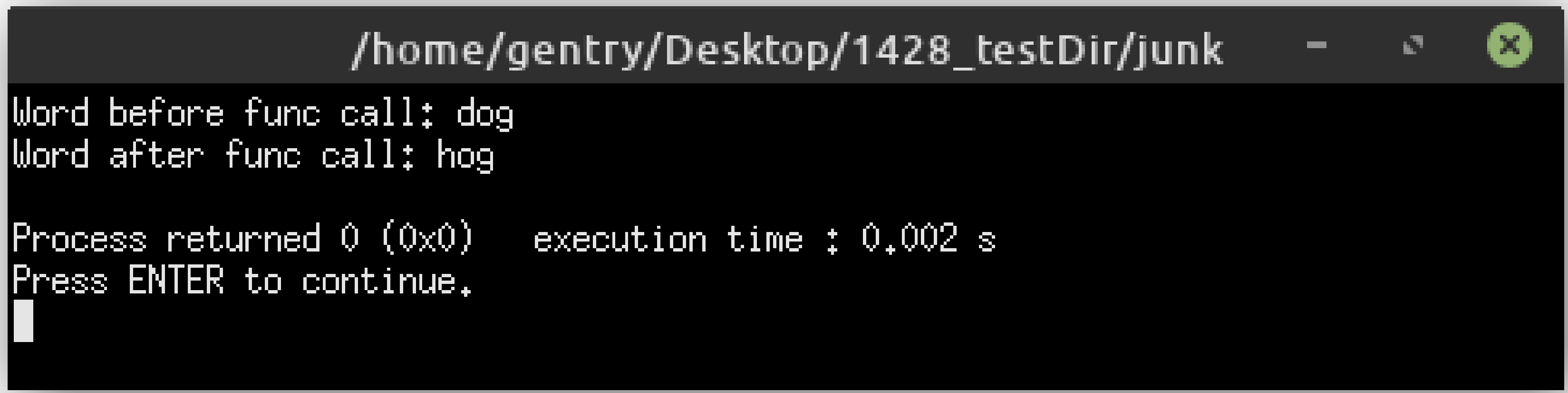
Passing Arrays to Functions:

- Arrays are <u>always</u> passed by reference.
- Arrays can be very “large” in memory, so copying them into a function parameter would be expensive.
- You do not have to use & to pass an array by reference.

Passing an Array:

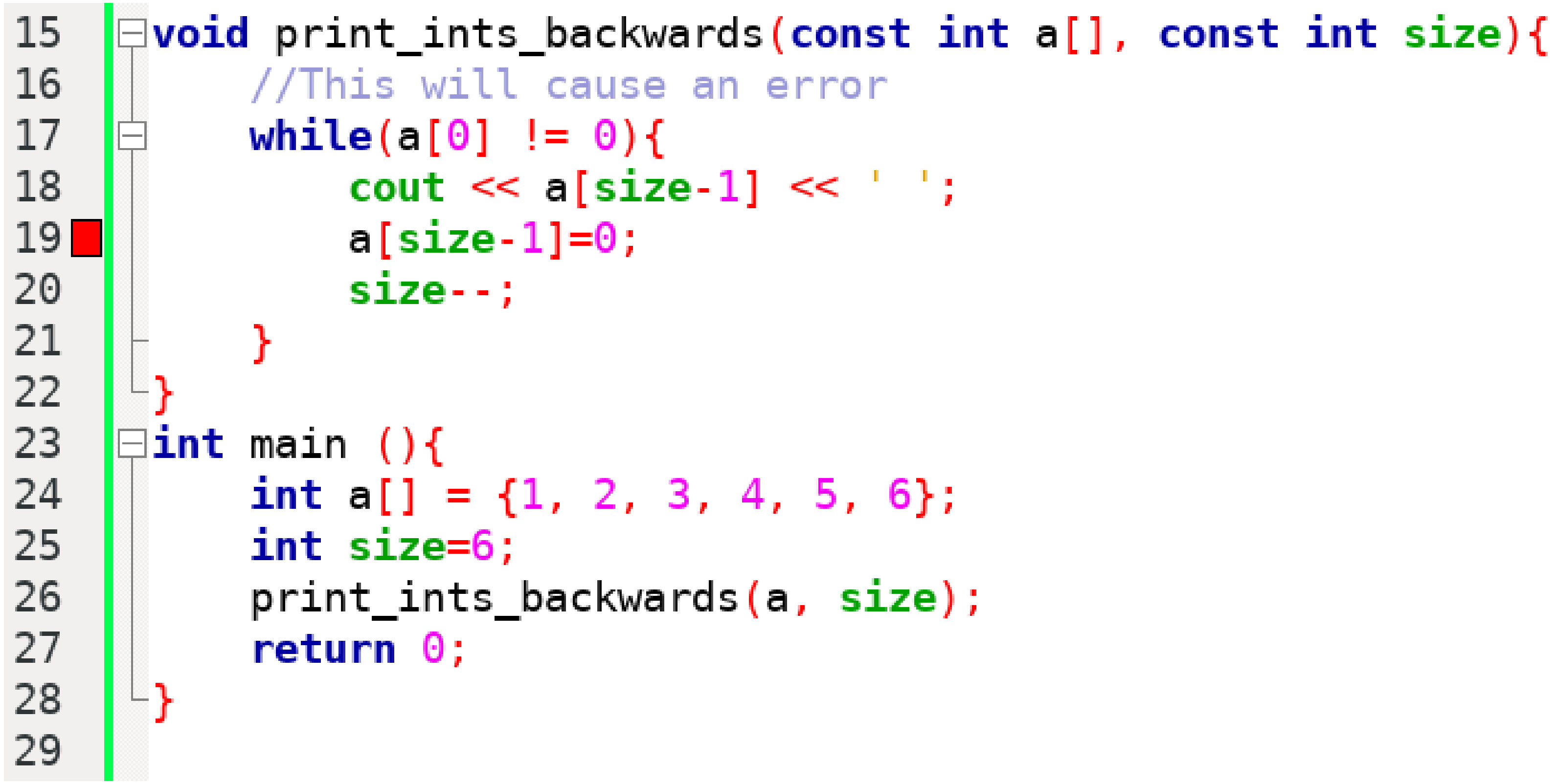
```
void change_word(char word[]){
    word[0] = 'h';
    return;
}

int main (){
    char word[] = {'d', 'o', 'g', '\0'};
    cout << "Word before func call: " << word << endl;
    change_word(word);
    cout << "Word after func call: " << word << endl;
}
```

Passing Arrays:


/home/gentry/Desktop/1428_testDir/junk

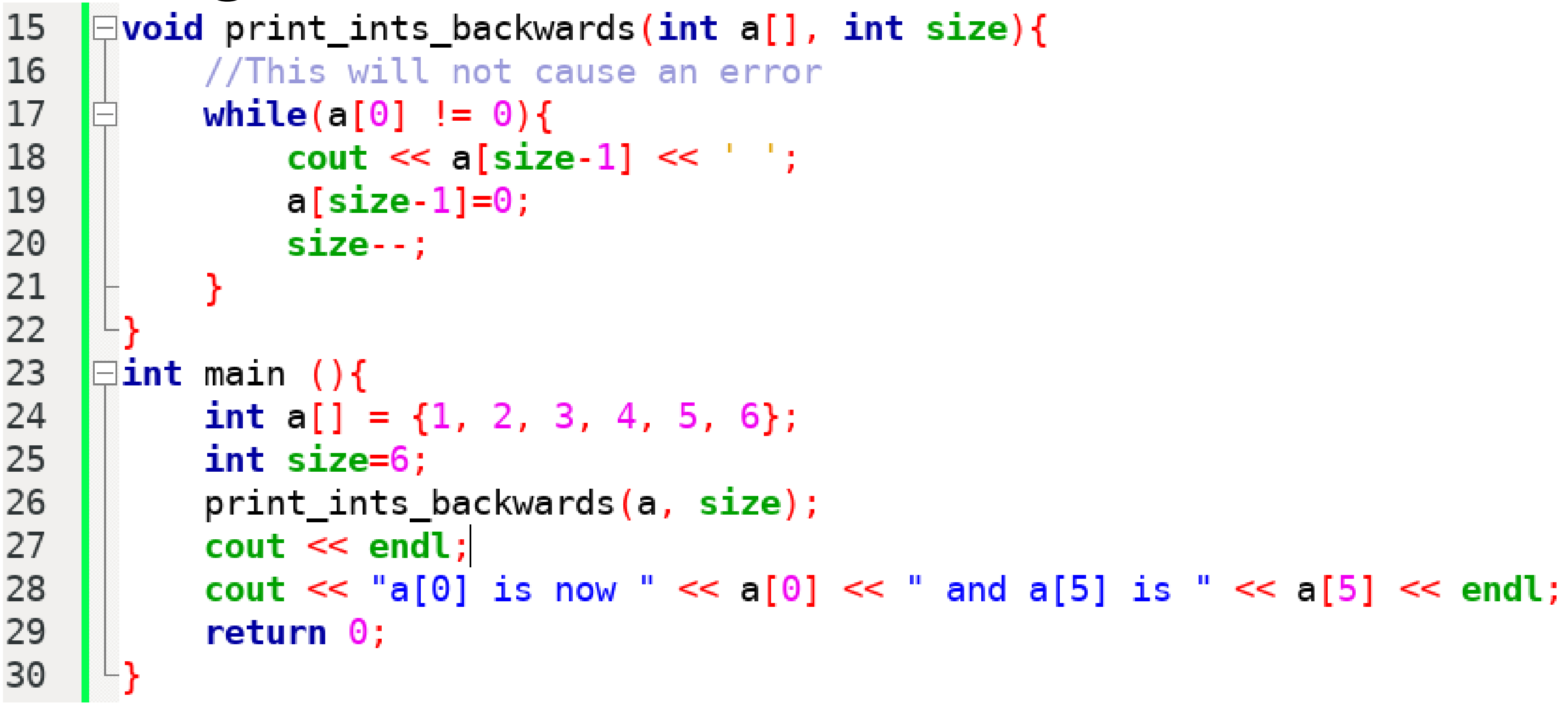
```
Word before func call: dog
Word after func call: hog

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

Avoiding Side Effects:



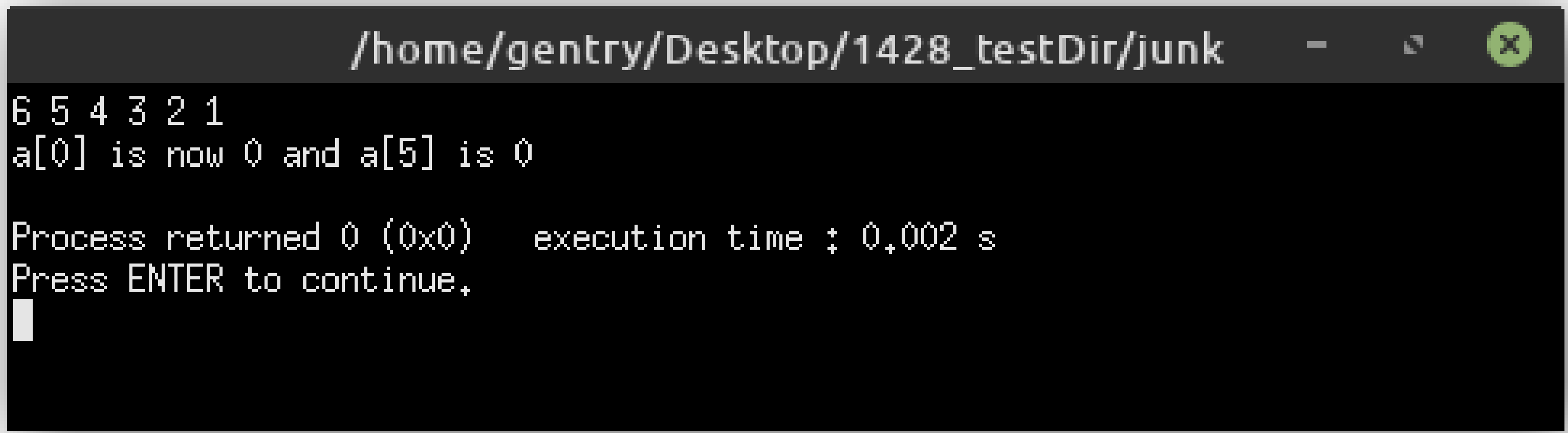
```
void print_ints_backwards(const int a[], const int size){
    //This will cause an error
    while(a[0] != 0){
        cout << a[size-1] << ' ';
        a[size-1]=0;
        size--;
    }
}
int main (){
    int a[] = {1, 2, 3, 4, 5, 6};
    int size=6;
    print_ints_backwards(a, size);
    return 0;
}
```

Avoiding Side Effects:



```cpp
void print_ints_backwards(int a[], int size){
    //This will not cause an error
    while(a[0] != 0){
        cout << a[size-1] << ' ';
        a[size-1]=0;
        size--;
    }
}
int main (){
    int a[] = {1, 2, 3, 4, 5, 6};
    int size=6;
    print_ints_backwards(a, size);
    cout << endl;
    cout << "a[0] is now " << a[0] << " and a[5] is " << a[5] << endl;
    return 0;
}
```

Side Effects Output:


/home/gentry/Desktop/1428_testDir/junk

```
6 5 4 3 2 1
a[0] is now 0 and a[5] is 0

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

const Parameters:

- The **const** keyword can be used with any parameter to keep its value from being changed.
- Pass-by-value parameters that are marked **const** act just like local constants.
- Pass-by-reference **const** parameters are “protected” by the function and cannot be changed.
- **const** is part of the “contract” that teams of coders can use to make their segments of code work together.

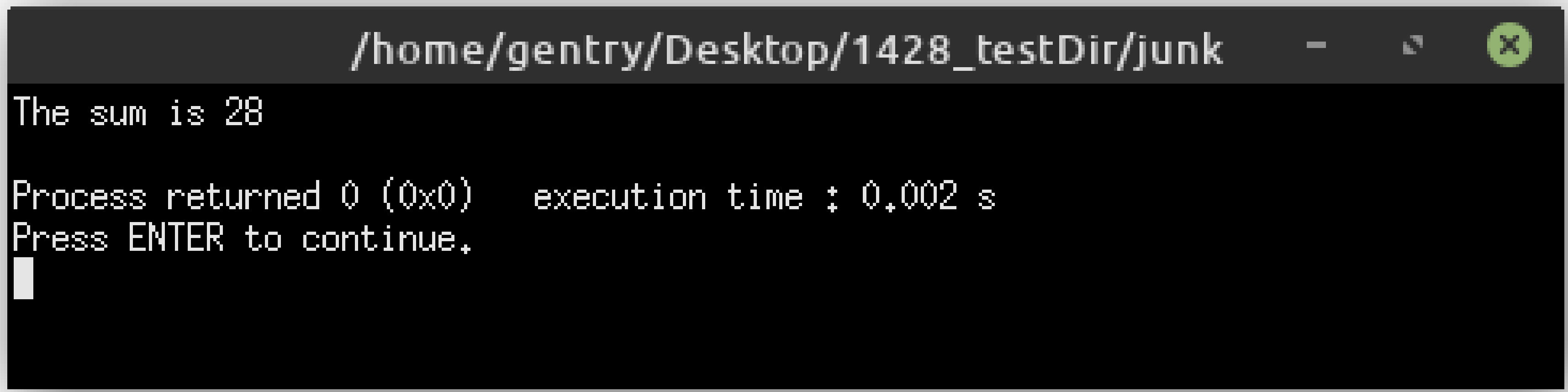
What is the Size of an Array:

- Functions must know the size of an array argument to avoid out-of-bound errors.
- It is a good practice to always pass in the size of an array that is being processed by a function.
- **sizeof** can be used to measure the size of arrays.
- **strlen** can be used to measure the length of c-strings.

Passing in an Array Size:

```
int sum_array(const int a[], int size){
    int sum = 0;
    for(int i = 0; i<size; i++)sum+=a[i];
    return sum;
}

int main() {
    int a[] = {4, 6, 8, 10};
    cout << "The sum is " << sum_array(a, 4) << endl;
    return 0;
}
```

Passing in an Array Size:

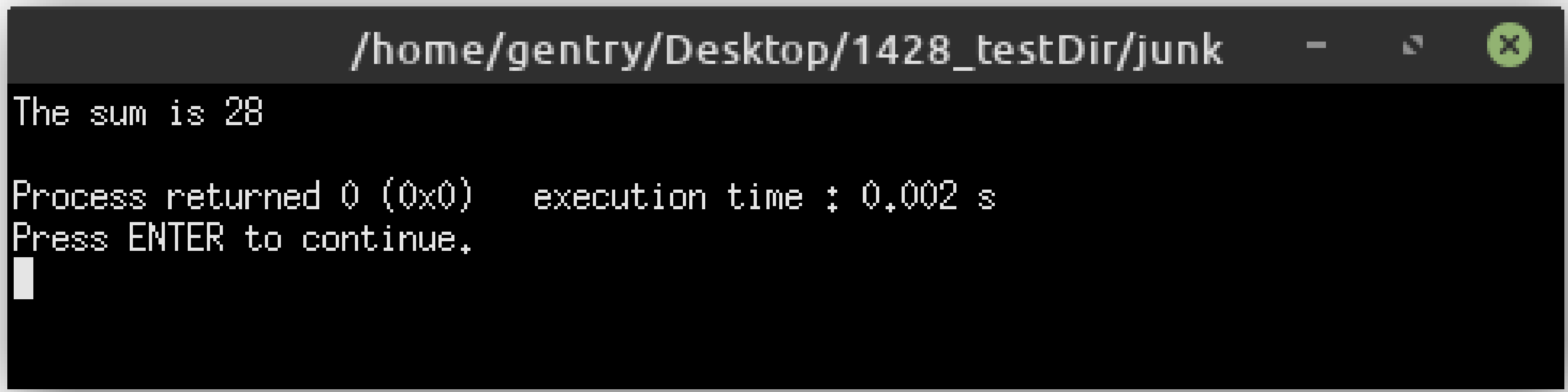


```
The sum is 28

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

Using sizeof:

```
int main() {
    int a[] = {4, 6, 8, 10};
    int sum = 0;
    for(int i = 0; i<(sizeof(a)/sizeof(int)); i++)sum+=a[i];
    cout << "The sum is " << sum << endl;
    return 0;
}
```

Using sizeof:

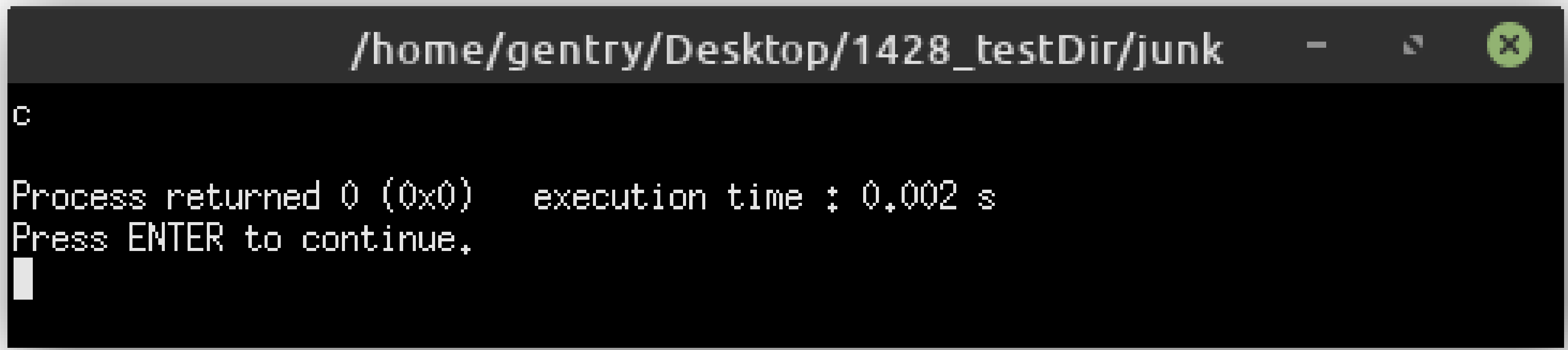

/home/gentry/Desktop/1428_testDir/junk

```
The sum is 28

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

Using strlen:

```
void print_middle_letter(char str[]){
    cout << str[strlen(str)/2] << endl;
}

int main() {
    char str[] = "Bobcats";
    print_middle_letter(str);
    return 0;
}
```

Using strlen:



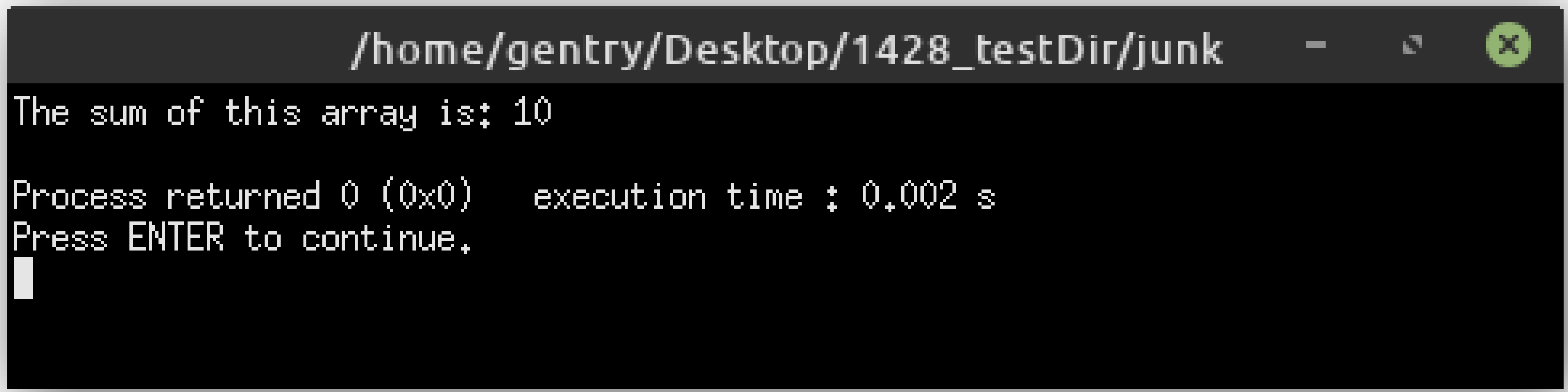
/home/gentry/Desktop/1428_testDir/junk

```
c

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

Using sizeof with a Function Parameter:

```
int sum_array(const int a[]){
    int sum = 0;
    //This will cause an error
    for(int i = 0; i<(sizeof(a)/sizeof(int)); i++)sum+=a[i];
    return sum;
}


int main() {
    int a[] = {4, 6, 8, 10};
    cout << "The sum of this array is: " << sum_array(a) << endl;
    return 0;
}
```

Using sizeof with a Function Parameter:


/home/gentry/Desktop/1428_testDir/junk

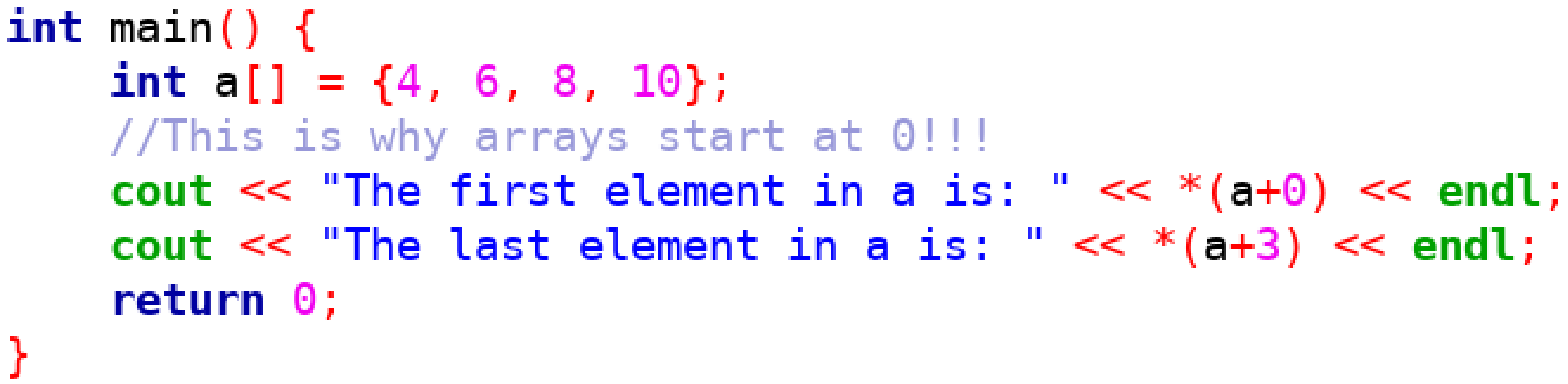
```
The sum of this array is: 10

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

4+6+8+10=10???

Why Doesn’t sizeof Work with Parameters:

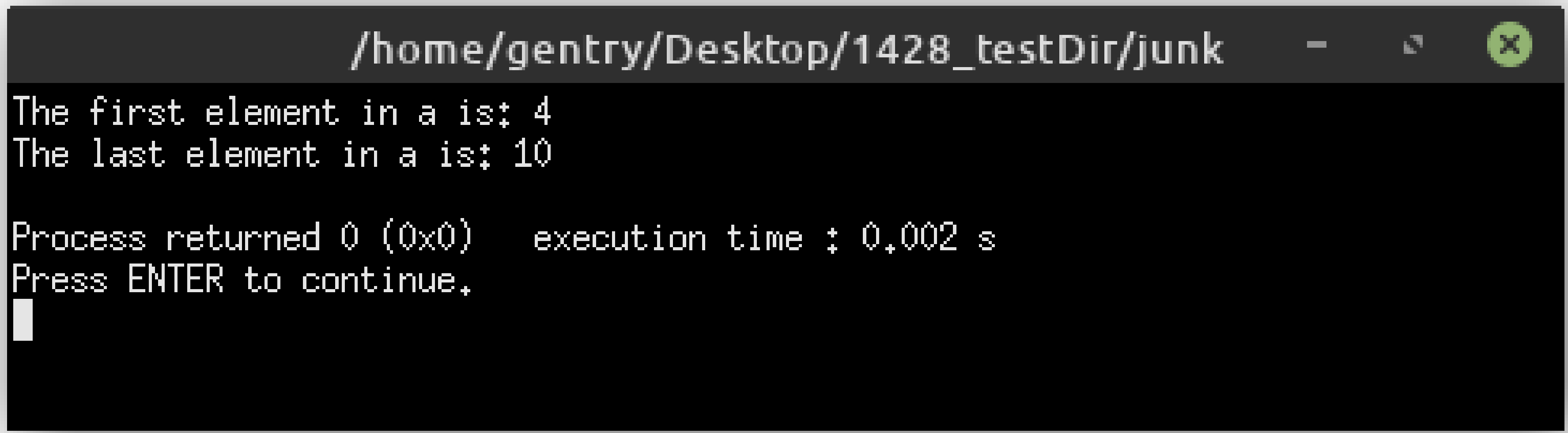
- Remember that by-reference parameters in C++ are passed using pointers.
- AND arrays are always passed by reference.
- So an array in C++ is always passed as a pointer to the first value in the array.
- This causes **sizeof** to return the size of the pointer, <u>not</u> the size of the array.
- On my machine a memory address is 64bits, so sum_array only counted two values.

Arrays are Secretly Pointers :



```
int main() {
    int a[] = {4, 6, 8, 10};
    //This is why arrays start at 0!!!
    cout << "The first element in a is: " << *(a+0) << endl;
    cout << "The last element in a is: " << *(a+3) << endl;
    return 0;
}
```

Arrays are Secretly Pointers :



```
The first element in a is: 4
The last element in a is: 10

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```

Pointer Arithmetic:

- We can apply basic operators to pointer variables just like other variables.
- Changing the value of a pointer makes it point to a different address.
- Incrementing or decrementing a pointer moves by a number of bytes determined by the <u>data type.</u>

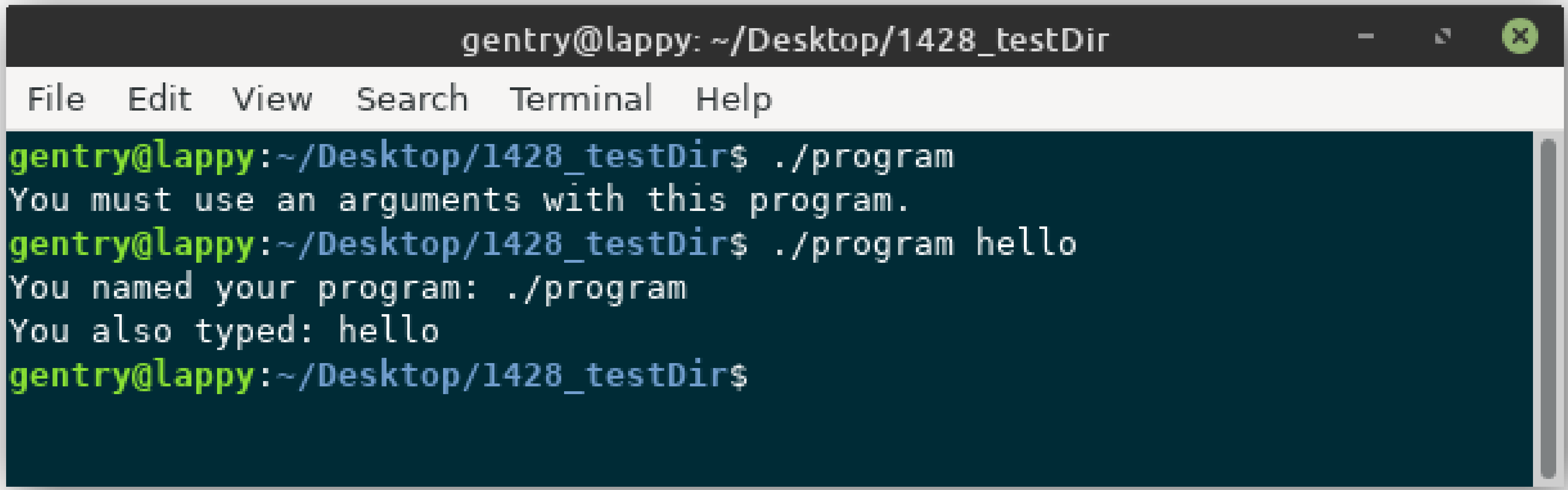
Arguments to the Main Function:

- Functions can have parameters and the **main** function is a function, so does it have parameters?
- We often ignore the parameters of **main** but they do exist and we can use them.
- The parameters of main are:
  - int argc: a count of the arguments
  - char* argv[]: an array of C-strings

Arguments to main :

```
int main(int argc, char* argv[]) {
    if(argc == 1){
        cout << "You must use an arguments with this program." << endl;
    }
    else{
        cout << "You named your program: " << argv[0] << endl;
        cout << "You also typed: " << argv[1] << endl;
    }
    return 0;
}
```

Arguments to main :

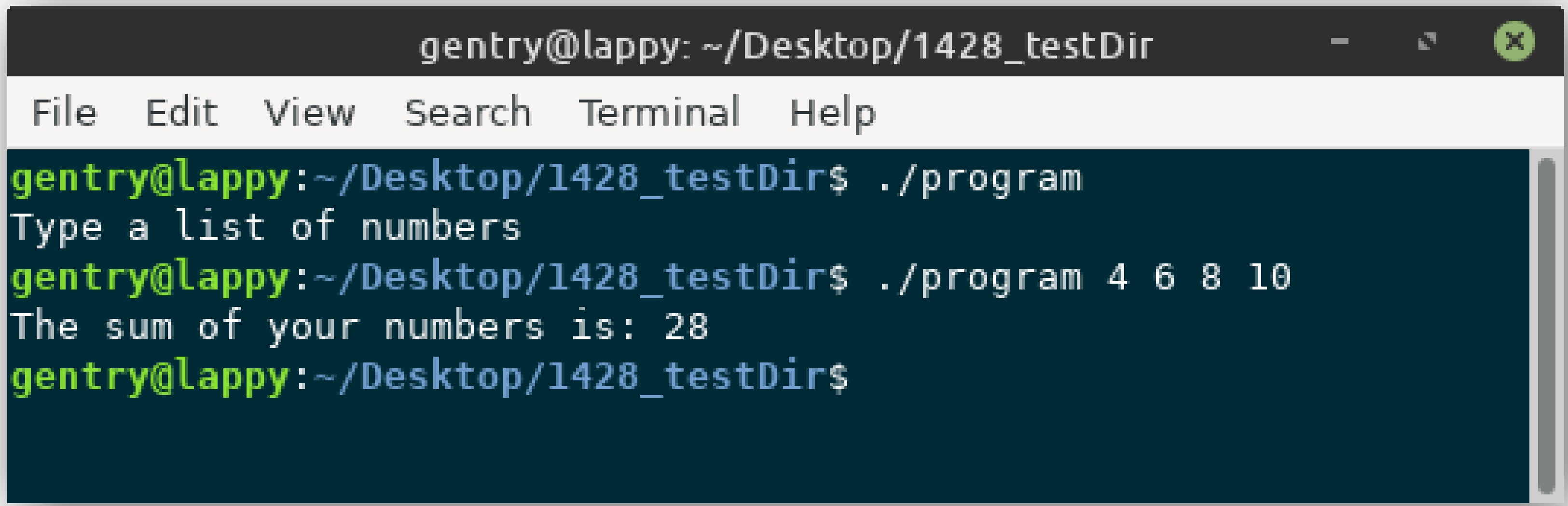

gentry@lappy: ~/Desktop/1428_testDir
File Edit View Search Terminal Help

```
gentry@lappy:~/Desktop/1428_testDir$ ./program
You must use an arguments with this program.
gentry@lappy:~/Desktop/1428_testDir$ ./program hello
You named your program: ./program
You also typed: hello
gentry@lappy:~/Desktop/1428_testDir$
```

Using Arguments to main :

```
int main(int argc, char* argv[]) {
    int sum = 0;
    if(argc == 1){
        cout << "Type a list of numbers" << endl;
    }
    else{
        for(int i = 1; i < argc; i++) sum += atoi(argv[i]);
        cout << "The sum of your numbers is: " << sum << endl;
    }
    return 0;
}
```

Using Arguments to main :

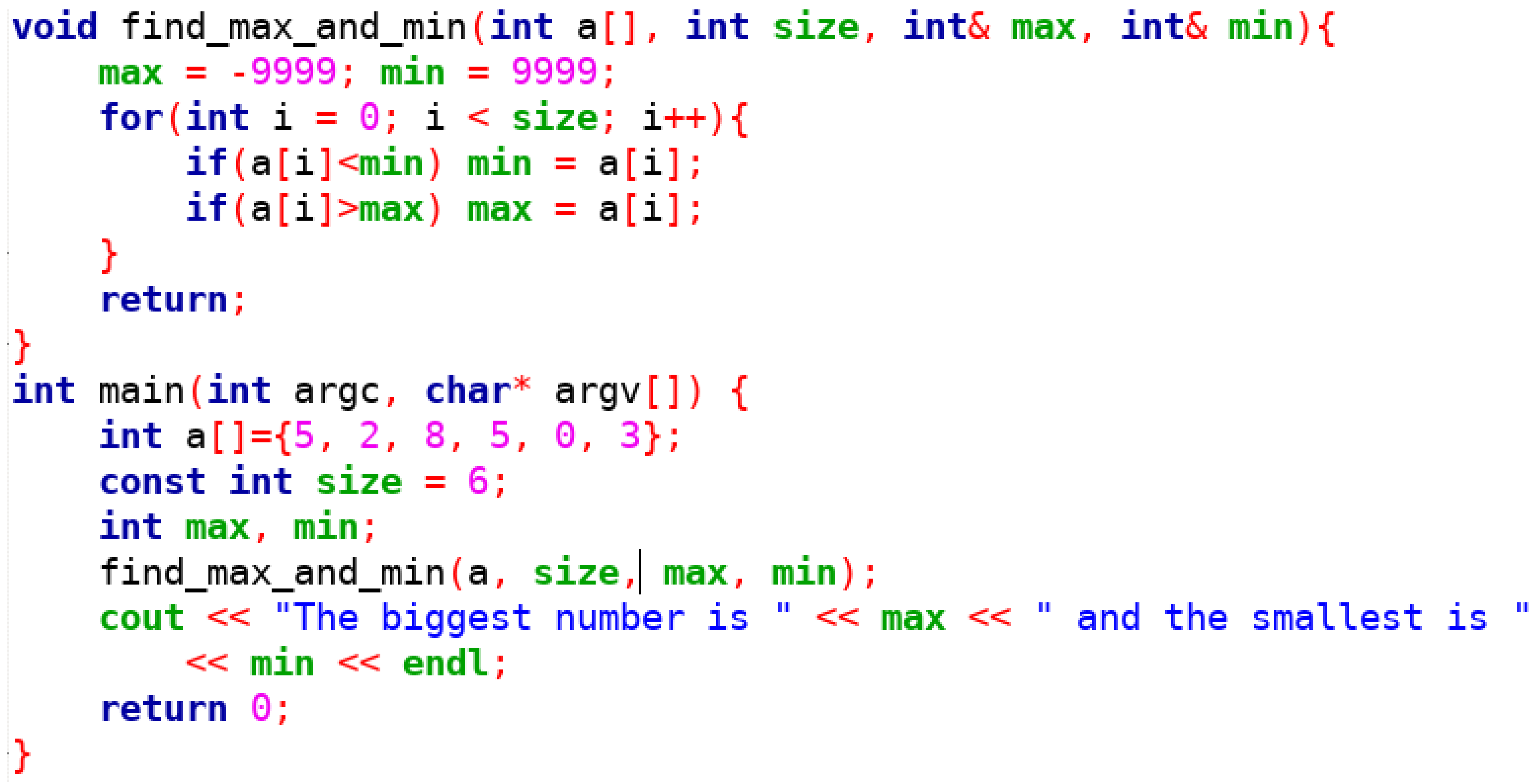

gentry@lappy: ~/Desktop/1428_testDir
File Edit View Search Terminal Help

```
gentry@lappy:~/Desktop/1428_testDir$ ./program
Type a list of numbers
gentry@lappy:~/Desktop/1428_testDir$ ./program 4 6 8 10
The sum of your numbers is: 28
gentry@lappy:~/Desktop/1428_testDir$
```

When to Pass by Reference:

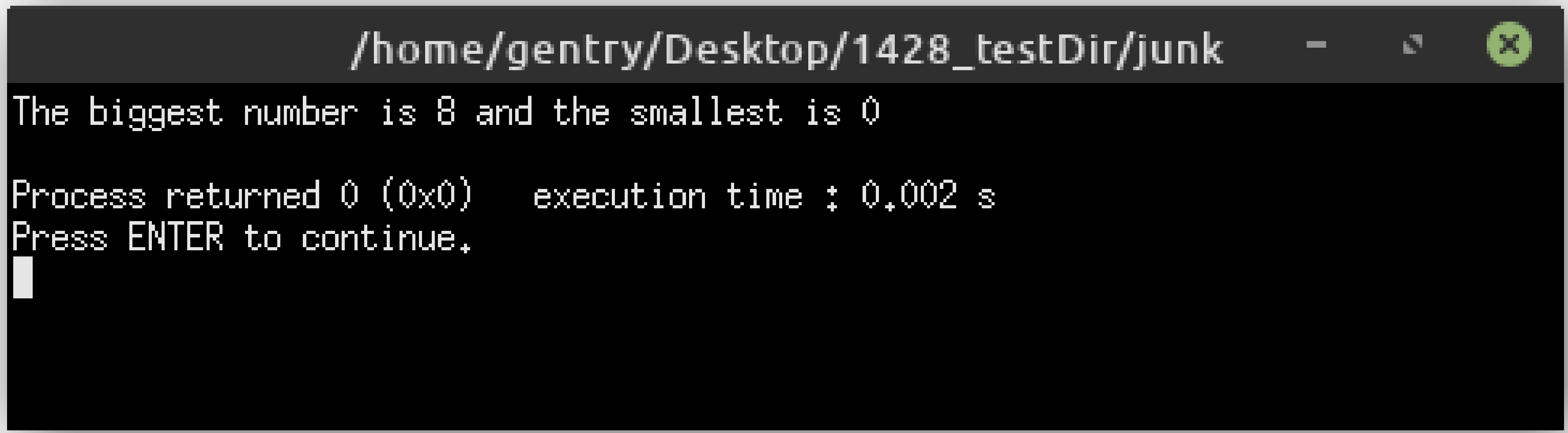
- Passing by value is safer because the variables of the calling function cannot be changed.
- Passing by reference can be more efficient when large blocks of memory are being passed as arguments.
- Passing by reference can let a function have more output than just a return.

Reference Parameters as Outputs:



```
void find_max_and_min(int a[], int size, int& max, int& min){
    max = -9999; min = 9999;
    for(int i = 0; i < size; i++){
        if(a[i]<min) min = a[i];
        if(a[i]>max) max = a[i];
    }
    return;
}
int main(int argc, char* argv[]) {
    int a[]={5, 2, 8, 5, 0, 3};
    const int size = 6;
    int max, min;
    find_max_and_min(a, size, max, min);
    cout << "The biggest number is " << max << " and the smallest is "
        << min << endl;
    return 0;
}
```

Reference Parameters as Outputs:


/home/gentry/Desktop/1428_testDir/junk

```
The biggest number is 8 and the smallest is 0

Process returned 0 (0x0)   execution time : 0.002 s
Press ENTER to continue.
```