Functions (2)

Anna Simon

September 25, 2015

- No class next week
- Two projects due October 7th
- Next class: October 7th

ROOT:

- If you are using your own computer for the class, install ROOT 5.34/32 that you can download from: https://root.cern.ch/content/release-53432.
- The required prerequisites can be found here: https://root.cern.ch/build-prerequisites.
- Follow the instruction to install ROOT:
  - https://root.cern.ch/build-root-old-method - to used "old" "\configure;\make" method
  - https://root.cern.ch/building-root - to use cmake for building ROOT
- MAKE SURE TO INSTALL version 5.34/32!

Functions: arguments vs. parameters

Parameters

Function parameter is a variable declared in the prototype or declaration of a function

```
1 int Add(int x, int y); //prototype
2
3 int Add(int x, int y) //declaration
4 {
5 }
```

Arguments

Argument is a value passed to the function in place of the parameter when the function is called

```
1 Add(3,4);
2
3 int nVal=3;
4 Add(nVal, 4);
```

When the function is called all the parameters of the function are created as variables and the values of the parameters are copied to these arguments. The parameters are destroyed when the function block terminates.

Passing arguments by value

When the arguments are **passed by value**, a copy of the argument is passed to the function.

```
1  int Add(int x, int y)
2  {
3      return x+y;
4  }
5
6  int main()
7  {
8      Add(5,4);
9
10     int z = 5;
11     Add(z,4); //variables or
           literals
12     Add(z+2,4);
13
14     return 0;
15 }
```

The function cannot change the original argument.

```
1  int Add(int x, int y)
2  {
3      x=3;
4      return x+y;
5  }
6
7  int main()
8  {
9      int z = 5;
10     cout << z << endl;
11
12     Add(z,4);
13    cout << z << endl;
14
15     return 0;
16 }
```

NOTE: Copying large structs or classes takes a lot of time and can compromise the program performance.

Passing arguments by address

Passing an argument by address involves passing the address of the argument variable rather than the argument variable itself. Because the argument is an address, the function parameter must be a pointer.

```
1  void SetToSix(int *pValue)
2  {
3    *pValue = 6;
4  }
5
6  int main()
7  {
8    int nValue = 5;
9
10   cout << "nValue = " << nValue <<
         endl;
11   SetToSix(&nValue);
12   cout << "nValue = " << nValue <<
         endl;
13   return 0;
14 }
```

It is usually used to pass an array or dynamically allocated memory

```
1  void PrintArray(int *pnArray, int
       nLength)
2  {
3    // make sure it's not a NULL
         pointer
4    if (!pnArray)
5      return;
6    for (int iii=0; iii < nLength;
         iii++)
7      cout << pnArray[iii] << endl;
8  }
9
10 int main()
11 {
12   int anArray[6] = { 6, 5, 4, 3, 2,
         1 };
13   PrintArray(anArray, 6);
14 }
```

Array length is passed as an argument, as the arrays do not keep track of their size.

Returning values by value

- the simplest
- a copy of the value is returned to the caller
- can return literals, variables or expressions
- returns variables passed by value as arguments or variables defined within the function
- slow, as it has to create a copy

```
1 int DoubleValue (int x)
2 {
3     int nValue = x * 2;
4     return nValue;
5 }
```

Returning by address

- can only return variables
- if returned is an address of a variable defined within a function, it will point to an unallocated memory
- often used to return newly allocated memory

```
1  int* DoubleValue(int *x)
2  {
3      *x = *x * 2;
4      return x;
5  }
6
7  int main()
8  {
9      int y=3;
10     int *pY = DoubleValue(&y);
11
12     cout <<*pY << endl;
13     return 0;
14 }
```

```
1  int* AllocateArray(int nSize)
2  {
3    return new int[nSize];
4  }
5
6  int main()
7  {
8    int *pnArray = AllocateArray(25);
9    // do stuff with pnArray
10
11   delete[] pnArray;
12   return 0;
13 }
```

Combining both together

```
1  float *Calibrator(float *data, float aa, float bb, int detNo = 4){
2
3      //allocate a dynamic memory for the calibrated energy
4      float *energy = new float [detNo];
5
6      for (int i=0;i<detNo;i++){
7          energy[i] = data[i] * aa + bb;
8      }
9
10     //return the address of the memory allocation, the pointer goes
           out of scope, but the memory is not deleted
11     return energy;
12 }
13
14 int main(){
15
16     float data[4] = {1, 2, 3, 4};
17     float *ene = Calibrator(data, 2, 0, 4);
18
19     cout << ene[0] << " " << ene[1] << " " << ene[2] << " " << ene[3]
           << endl;
20
21     //delete the dynamic memory
22     delete [] ene;
23 }
```

Default parameters

A **default parameter** has a default value provided to it that is used if user does not supply a value for this parameter.

```
1  int GetSeconds(int hrs, int mins = 0, int sec = 0)
2  {
3      return hrs *60 *60 + mins * 60 + sec;
4  }
5
6  int main()
7  {
8      GetSeconds(10);
9      GetSeconds(3, 20);
10     GetSeconds(5, 7, 14);
11 }
```

Passing arguments to `main()` function

```
1  #include <iostream>
2  #include <stdlib.h> /* for atoi() and atof() */
3  using namespace std;
4
5  int main(int argc,char *argv[]) {
6      int m;
7      float n;
8      char errorMsg[100], outMsg[100];
9
10     if (argc != 4) {
11         sprintf(errorMsg,"Usage: %s m n filename\n",argv[0]);
12         cout << errorMsg;
13         return 1;
14     }
15
16     m = atoi(argv[1]); /* convert strings to integers */
17     n = atof(argv[2]); /* convert strings to floats */
18
19     sprintf(outMsg,"%s received m=%i n=%f filename=%s\n",argv[0],m,n
           ,argv[3]);
20     cout << outMsg;
21
22     return 0;
23 }
```