Functions

Chapter 7

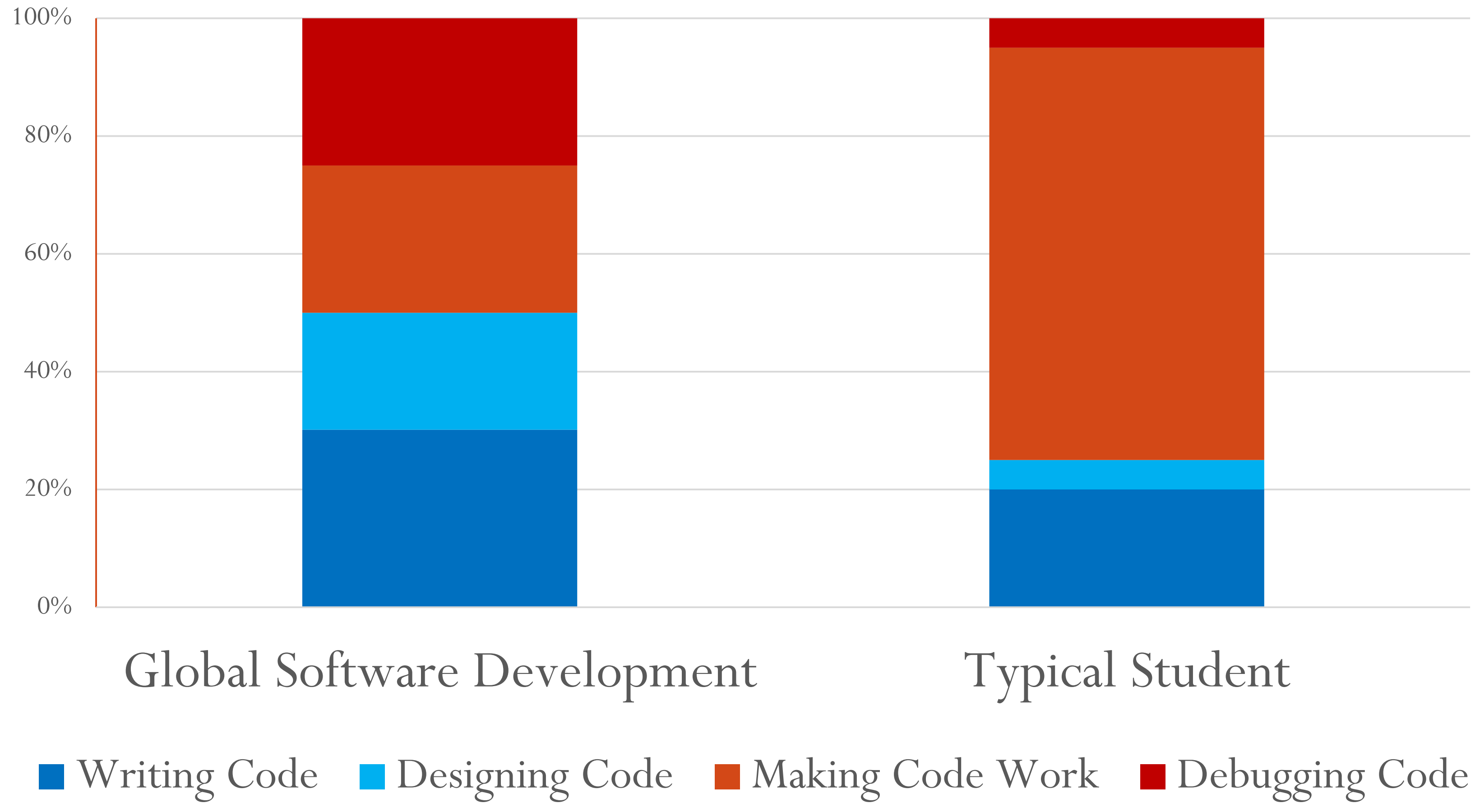
Functions, Pass by Value, Pass by Reference

Functions in Review

- Realistic Programs have the Following Properties
  - Multiple Source Files (sometimes hundreds)
  - Millions of Lines of Code (And you thought the homeworks were long)
  - Multiple Programmers coding

- As such organization of a program is of the up-most importance

Software Development Time



Sources

Left Bar: Reversible Debugging Software Cambridge University

Right Bar: Professor’s Non-Scientific Anecdotal Evidence

Scenario:

Programmer timothy is generating a lengthy program for his company that requires using a sorting algorithm in 30 unique places in the code. Due to an irrational fear of functions timothy decides to write the code once, and copy and paste the code each time it is needed.

On verification of his code, timothy discovers a problem with his sorting algorithm.

How soon will timothy be collecting unemployment?

Scenario - Conclusion

- Since the code was copied and pasted, timothy will now have to fix his code in 30 separate locations
- Timothy runs the risk of fixing it in only 29 of 30 locations leaving behind a bug
- If a function was used
  - The fix would only have to be applied to the function itself
  - The fix would guarantee all 30 of 30 locations to exhibit the corrected behavior

- Reuse code as often as you can through functions.
- Never copy and paste code

Functions - Purpose

- Functions provide two primary advantages
  - Code Reuse
    - Generalize the functions as much as possible allowing for greatest reuse
    - Less Code, Less Problems
    - A problem that occurs in a function only requires fixing once
  - Organization
    - A large piece of code can be hidden away in a function. When the code is called ("used") the name of the function is used as one line of code.

Code Organization – Closer Look

- Without functions all code would be placed in the main function
  - The main function could then be millions of lines long
- A typical program is:
  - Complex
  - Made up of several sub-problems
  - Requires a strategy to tackle
- Functions allow for sub-problems to be abstracted as a concept
  - i.e. the sub-problem of sorting can be abstracted as a function called sort. Allowing the programmer to quickly use and understand the code

Concept of Variable Scope

- Rule: Every variable's scope is limited to
  1. the nearest curly braces which encapsulates the variable's declaration
  2. all subsequent curly braces nested within the encapsulating curly braces

```
int main()
{
    {
       int x;
       // Exists inside these curly braces
    }
    // No longer exists referencing the variable here will result
    // in a syntax error
    return 0;
}
```

Concept of Variable Scope

```
int main()
{
    {
        int x;
        // Exists inside these curly braces (Rule 1)
        {
           // Exists inside these sub-curly braces (Rule 2)
        }
    }

    // No longer exists
    // referencing the variable here will result
    // in a syntax error

    return 0;
}
```

Scoping In Practice

```
for(i = 0; i < 8; i++)
{
    bool b = true;
    for(j = 0; j < 4; j++)
    {
        ...
    }
}
```

- Often variables are declared inside a loop or conditional if they are temporarily needed
- In the above case the boolean *b* exists inside both loops
- Every time a new iteration of the loop occurs, the variable is destroyed and recreated

Variable Naming Conflicts

- Syntax Error

```
{
     int x;
     int x;
}
```

- Allowed

```
{
     int x;
     {
          int x;
     }
}
```

- Upper example violates the syntax of C++ since the same variable is declared twice in the same scope
  - Generates re-declaration error
- Lower example is allowed since the variable is declared twice in the same scope

- Since both Xs are present in the inner curly braces isn't there a conflict?

Naming Conflicts

- Rule: If two or more variables with the same name exist in a given scope, the variable most local to the scope is used.

```
int main()
{
    int x = 1;
    {
       int x = 2;
       cout << x  << endl; // Prints 2 since x = 2 is more local
       {
             cout << x << endl; // Prints 2 since x = 2
       }
    }
    cout << x << endl; // Prints 1, only one x now exists
}
```

Scope and Functions

```
int main()
{
    int x = 4;
    // Variable y is outside the scope
    // therefore ii cannot be accessed
    return 0;
}

void someFunction()
{
    int y = 10;
    // Variable x is outside the scope
    // therefore it cannot be accessed
}
```

- Only globals, variables passed as parameters, or local variables to the function are accessable

Final Note on Variable Scope

- Having multiple variables with the same name at a given scope should be avoided for the following reasons:
  - Easy to miss a declaration of a variable inside a scope
  - Increases the chances of writing code referring to the wrong variable
  - Decreases the readability of your code for other programmers

- There are 26 letters in the alphabet + 10 digits
  - For a four letter word there are 36^4 equaling 1,679,616 combinations alone
  - There is no reason to repeat variable names within a scope

Globals

- When a variable is declared outside of curly braces i.e:

```
int x;
int main()
{
    x = 2;
    return 0;
}
```

It is a called a global and is available in:

- Every function in the program
- Every source file in the program

Issues with Globals

- Globals are difficult to track
  - A global can be modified in any function, or line of code
  - If a bug arises from its use prepare, to spend hours tracking it down

- Local variables with the same name can inadvertently mask the global

- Naming conflicts present issues when merging programs
  - Program 1 has global *int x*
  - Program 2 has global *int x* but stores different data
  - If the source files of the two programs are merged, one of the declared Xs will have to be renamed and every reference to it updated

Issues with Globals

- Globals create race conditions in multi-threaded environments

- Globals Violates Open-Closed Design Principle of OOP

- Globals are created the moment a program begins it's execution
  - Always consume memory whether being used or not
  - Especially an issue for large arrays or other data structures

Final Notes on Globals

- There places where globals should be used:
  - Avoiding passing of parameters from deep function stacks
  - If the program written is small and will never be merged into another program
- The cases in which the advantages of using a global outweighs the drawbacks are rare
- Novices tend to use globals to avoid passing data in a function
  - This is a poor programming practice

- **Since students almost always misuse globals, you will not be allowed to use them in this class**

Default Values

- Some parameters can have default values which allow them to be omitted when called.
- Example, find from string library:

```
size_t find (const string& str, size_t pos = 0) const;
```

Here the *pos* parameter, represents start of search, defaults to zero. Therefore it can be omitted.

Example calls

```
size_t find (const string& str, size_t pos = 0) const;
```

```
string example = "Hello World";
// Using Default value
cout << example.find("o") << endl;
// Overriding default value
cout << example.find("o", 5) << endl;
```

User Functions and default param.

```
// Set the parameters equal to the default value in
// the prototype
float rectArea(float width = 0, float height = 0);


// Function definition remains the same
float rectArea(float width , float height) {
return width * height;
}
```

- Here width and height will default to zero

Example Calls

```
// Set the parameters equal to the default value in
// the prototype
float rectArea(float width = 0, float height = 0);

int main()
{

   rectArea();// Width = 0, height = 0
   rectArea(2);// Width = 2, height = 0
   rectArea(2, 4); // Width = 2, Height = 4
   return 0;
}
```

- Default parameters are overwritten in the order in which they appear in the prototype

Rule of Default Values

- For a given prototype, all parameters after a default parameter appears must also be default parameters.

- Valid Prototype
  - `float add(int a, int b = 1, int c = 1);`

- Invalid Prototype
  - `float add(int a, int b = 1, int c);`

Other Basics of Functions

- Multiple variables can be passed as parameters using the comma operator

- **Only one variable can be returned!!!**
  - **return a,b;   will compile but will not give the desired effect**

Make a Swap Function

```
void swap(int x, int y)
{
   int tmp = x;
   x = y;
   y = tmp;
}
```

- Here both the values of x and y need to change, but until now there is no known method to return back two values
  - Excluding arrays and classes

Passing by Reference

- By default parameters are passed by value, meaning a copy of the passed variable is used
- Problem: Make a function that takes two input variables and swaps their values
- The output of this program will be: 1, 2

The swap function is unable to modify the original a, b variables

```
void swap(int x, int y);
int main()
{
   int a = 1, b = 2;
   swap(a, b);
   cout << a << ", " << b << endl;
   return 0;
}

void swap(int x, int y)
{
   int tmp = x;
   x = y;
   y = tmp;
}
```

C++ Style Pass by Reference

- Operator &:
  - When used as in a declaration it defines an alias to another variable

- Example:

```
int a = 1;
int &b = a;
b = 2;
cout << a << ", " << b << endl;
```

- The above code outputs: 2, 2
- *b* is an alias of *a*, any modifications to *b* are reflected into *a* and vice versa

Using References in Functions

- Defining the parameters as references means *x* becomes a reference for *a* on the swap call
- *a* is no longer passed by value
- Any modifications to *x* are reflected back to the original passed variable *a*
- Therefore the output is:

  2, 1

```
void swap(int &x, int &y);
int main()
{
    int a = 1, b = 2;
    swap(a, b);
    cout << a << ", " << b << endl;
    return 0;
}

void swap(int &x, int &y)
{
    int tmp = x;
    x = y;
    y = tmp;
}
```

Passing by Reference

- If passing by reference exists, why not pass all variables by reference?
  - Passing by reference allows for accidental modification to the variable because of a programmer's mistake.
  - Naively you may assume that if you are careful this will never happen, but even the most veteran programmer can make mistakes.

Returns

- Pass by Value:
  - the return is the only output
  - therefore can only output one piece of data
- Pass by Reference:
  - an input can act also as an output
  - allows for outputting more than one piece of data
  - Note: you can still use returns when passing by reference

- Golden rule on Use:
  - Pass by Value: when returning a single piece of data
  - Pass by Reference: when returning two or more pieces of data

Guided - Example 10.1

- Write a function which takes a radius of a circle as an input and returns both the area and circumference of the circle. Then call the function in the main using various test data.

Independent Example 10.2

- Create a function which takes seconds as an input, and returns back the number of hours and minutes the seconds represents.

- Test the function by calling it in the main using some sample data.

Passing by Reference and Structures

- You may want to pass a structure (class) by reference even if the function has no intention of modifying it. Why?
  - The amount of memory allocated to a structure can be very large
  - Passing by reference prevents an expensive memory copy
  - Saves computations and memory space

Example of Passing Struct by Ref

```
struct Rectangle {
   double x, y;
   double width;
   double height;
};


double calcArea(Rectangle &rect)
{
   double area = rect.width * rect.height;
   return area;
}
```

- The above example avoids an unnecessary memory copy of the Rectangle structure
- In practice this function would be considered unsafe. Why?

Example of Passing Struct by Ref

```
struct Rectangle {
   double x, y;
   double width;
   double height;
};


double calcArea(Rectangle &rect)
{
   double area = rect.width * rect.height++;
   return area;
}
```

- It is unsafe because the user can accidentally modify a variable

Safe Pass by Reference

```
struct Rectangle {
   double x, y;
   double width;
   double height;
};


double calcArea(const Rectangle &rect)
{
   double area = rect.width * rect.height;
   return area;
}
```

- Placing *const* in front of the structure name ensures that a compiler error will be thrown if the programmer attempts to modify the structure

Safe Passage

- If you intend on passing a structure/class by reference but do not intend on altering any of the fields pass using *const*