24. Default Values

Objectives

- Write functions with default values
- Use default values to extend a function
- Write prototypes for functions with default values and implement the bodies of such functions.

Default Values

Try this:

```
#include <iostream>


void printHeads(int numHeads = 1)
{
  std::cout << "number of heads: " << numHeads
            << std::endl;
}


int main()
{
  printHeads();     // NO ARGUMENT PASSED IN!!!
  printHeads(42);
  return 0;
}
```

(Forget about prototypes for the time being. They will come in a bit ...)

Parameter `numHeads` of function `printHeads()` is called a parameter with a **default value**.

One obvious use of default values is that you can make it easier for other programmers to use your function: You choose the most common value for a parameter and make it the default value for that parameter. Any time someone (including yourself) uses the function in the “most commonly used” manner, he/she need not send in an argument.

Here's another example:

```
#include <iostream>

void printHello(int i = 0)
{
  std::cout << "hello ";
  switch (i)
  {
    case 0: std::cout << "world\n";
            break;
    case 1: std::cout << "galaxy\n";
            break;
    case 2: std::cout << "universe\n";
            break;
    case 3: std::cout << "underverse\n";
            break;
  }
}

int main()
{
  printHello();
```

```
  printHello(1);
  return 0;
}
```

Right now there are only **three rules** to remember: Think of the parameters of a function as parameters without default values and parameters with default values. **All the parameters with default values must be together** and appear **after those without default values**.

In other words this is OK:

```
bool spam(int a, double b, char c = '$', int d = 42)
{
  ...
}
```

But this is WRONG:

```
bool spam(int a, double b = 3.14, char c, int d = 42)
{
  ...
}
```

`c` does not have a default value but appears after `b` which has a default value.

Make sure you try this or you won't remember.

So remember to group up the parameters into those with default values and those without. The ones without (if any) must come first.

Another rule: **default values must be constants** (example: constant literals such as 5 or constant expressions.) In other words this is OK:

```
const int X = 42;
bool spam(int a, double b, char c = '$', int d = X+1)
{
    ...
}
```

But this is WRONG:

```
int X = 42;
bool spam(int a, double b, char c = '$', int d = X)
{
    ...
}
```

Exercise. What is the output (or circle the errors)? (Do this by hand)

```
int f(int a, int b = 4, int c = 5)
{
  return a + b + c;
```

```
}

int main()
{
  std::cout << f(1, 2, 3) << ' ' << f(1, 2) << ' '
            << f(1) << std::endl;
  return 0;
}
```

Now verify using your compiler.

Last one: **You cannot have default values for array parameters.** For instance the following will not compile:

```
#include <iostream>

void print(int x[] = {2, 3})
{
  std::cout << x[0] << ' ' << x[1];
}

int main()
{
  print();
  return 0;
}
```

(See last section for exception.)

Exercise. What is the output (or circle the errors)? (Do this by hand)

```
int f(int a = 3, int b, int c = 5)
{
  return a + b + c;
}

int main()
{
  std::cout << f(1, 2, 3) << std::endl;
  return 0;
}
```

Now verify using your compiler.

Exercise. Write a function `printDateTime()` that prints the date and time data passed in:

```
void printDateTime(int yyyy, int mm, int dd,
                   int hh = 0, int mm = 0, int ss = 0)
{
  ... CODE ...
}

int main()
```

```
{
  printDateTime(2007, 1, 15);
  printDateTime(2007, 1, 15, 14, 15, 59);
  return 0;
}
```

Expected output:

```
2007-01-15 00:00:00
2007-01-15 14:15:59
```

After you are done, make January 1, 1970 the default date.
[Hint: Use `set::fill()` function.]

Default value parameter and nonconstant value

One more example. Recall we have the `isprime()` that returns `true` if the values passed in is a prime number, otherwise `false` is returned.

```
bool isprime(int n)
{
  for (int i = 2; i <= sqrt((double)n); i++)
  {
    if (n % i == 0) return false;
  }
  return true;
}
```

Suppose for some reason you know something about n: if there is a prime divisor of n, then it must be at least 11. You might want to tell `isprime()` to start the search for a divisor from 11. Let's modify the function preserving the older use:

```
#include <isotream>
#include <cmath>

bool isprime(int n, int start = 2)
{
    for (int i = start; i <= sqrt((double)n); i++)
    {
        if (n % i == 0) return false;
    }
    return true;
}

int main()
{
    std::cout << isprime(5) << '\n'
              << isprime(5, 3) << '\n'
              << isprime(169, 11) << std::endl;

    return 0;
}
```

Now let's try to generalize the upper bound of the search. Let's try this:

```
#include <isotream>
#include <cmath>

bool isprime(int n,
             int start = 2,
             int end = sqrt((double)n))
{
    for (int i = start; i <= end; i++)
    {
        if (n % i == 0) return false;
    }
```

```
    return true;
}

int main()
{
    std::cout << isprime(5) << '\n'
              << isprime(5, 3) << '\n'
              << isprime(169, 11) << std::endl;
    return 0;
}
```

Does it work?

It **won't work** because remember you can only use constants for default values. So how do you set a parameter to "default value" that depends on an expression like the above? This is how you do it:

```
#include <isotream>
#include <cmath>

bool isprime(int n, int start = 2, int end = -1)
{
    if (end == -1) end = sqrt(double(n));
    for (int i = start; i <= end; i++)
    {
        if (n % i == 0) return false;
    }
    return true;
}

int main()
{
    std::cout << isprime(5) << '\n'
              << isprime(5, 3) << '\n'
              << isprime(169, 11) << '\n'
              << isprime(21, 3, 7)
    return 0;
}
```

Of course in this case we must choose a reasonable default value for parameter `end`. In the above case, I choose a value the user won't be using. This is similar to the use of the sentinel values in terminating a while-loop.

Note that a default values can be from a function call as long as the function call involves only constants. Make sure you run this:

```
int g(int x)
{
    return x * x;
}

void f(int x, int y = g(42))
```

```
{ }
```

Function extension

As I've just mentioned, one reason for default values is to make a function convenient to use. You basically set parameter values to the most common values.

There is actually **another use** of default values that's not usually mentioned in some textbooks.

Default values can be used to **extend the functionality** of a function without breaking existing code. Here's what I mean.

Suppose you have a program that computes the average of an array of 3 elements:

```
double avg(int x[])
{
    int s = 0;
    for (int i = 0; i < 3; i++)
    {
        s += a[i];
    }
    return s / 3.0;
}

int main()
{
    int x[] = {1, 2, 3};
    std::cout << avg(x) << std::endl;
    return 0;
}
```

What if you now want to average over 5 elements? You can write a new function that averages over 5 elements. Of course you know now that you should have written a function to average over general lengths.

```
double avg(int x[])
{
    int s = 0;
    for (int i = 0; i < 3; i++)
    {
        s += a[i];
    }
    return s / 3.0;
}

double avg2(int x[], int len)
{
    int s = 0;
    for (int i = 0; i < len; i++)
    {
        s += a[i];
```

BAD! Code duplication! The functions look almost the same.

```
    }
    return (double) s / len;
}

int main()
{
    int x[] = {1, 2, 3};
    std::cout << avg(x) << std::endl;

    int y[] = {1, 2, 3, 4};
    std::cout << avg2(y, 4) << std::endl;
    return 0;
}
```

Another way would be to modify the original function. But of course all uses of the previous version of `avg()` must be changed; you need to add a size to the function call:

```
double avg(int x[], int size)
{
    int s = 0;
    for (int i = 0; i < size; i++)
    {
        s += a[i];
    }
    return s / 3.0;
}

int main()
{
    int x[] = {1, 2, 3};
    std::cout << avg(x, 3) << std::endl;

    int y[] = {1, 2, 3, 4};
    std::cout << avg(y, 4) << std::endl;
    return 0;
}
```

BAD! Each usage of the old function must change.

That's no big deal for a small program. But for a large program with thousands of lines or more, the change can be costly.

Here's another way to modify your `avg()` function without breaking existing usage:

```
double avg(int x[], int size = 3)
{
    int s = 0;
    for (int i = 0; i < size; i++)
    {
        s += a[i];
    }
    return (double) s / size;
}
```

```
int main()
{
    int x[] = {1, 2, 3};
    std::cout << avg(x) << std::endl;

    int y[] = {1, 2, 3, 4};
    std::cout << avg(y, 4) << std::endl;
    return 0;
}
```

Get the point?

One more example.

For some programs it is common to “clip” a value. For instance if you want a value to be positive, after calling the function with the value, the same value is returned if the value is positive. If the value is negative, 0 is returned.

- clip(5) returns 5
- clip(-3.44) returns 0

In other words the statement `x = clip(x)` will ensure that `x` is positive. Complete this program:

```
double clip(double x)
{
    if (x >= 0)
        return x;
    else
        return 0;
}

int main()
{
    std::cout << clip(5.6) << std::endl    // 5.6
              << clip(0) << std::endl      // 0
              << clip(-1.23) << std::endl; // 0
    return 0;
}
```

Now modify your clip so that it accepts a minimum clipping value. In other words, x = clip(x, 1) ensures that x is at least 1. Furthermore old test code must continue to work.

```
double clip(double x, double min = 0)
{
    if (x >= min)
        return x;
    else
        return min;
}

int main()
{
    std::cout << clip(5.6) << std::endl      // 5.6
```

```
              << clip(0) << std::endl        // 0
              << clip(-1.23) << std::endl    // 0
              << clip(5, 1.2) << std::endl   // 5
              << clip(0.5, 1.2) << std::endl // 1.2
              << clip(-3, 1.2) << std::endl; // 1.2
    return 0;
}
```

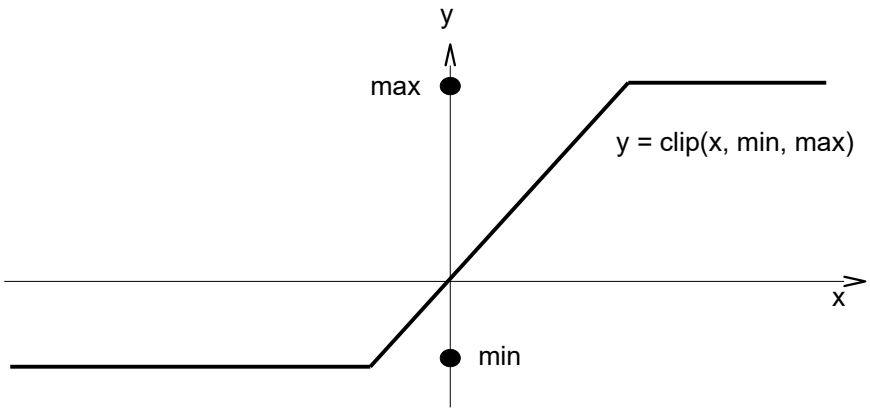
See the use of default values now?

Let's go one step further. Suppose the maximum possible value ever passed into the `clip()` function is 1e100 (mathematically speaking this is 1 x 10^{100}). Let's extend the `clip()` function to accept a maximum clip value.

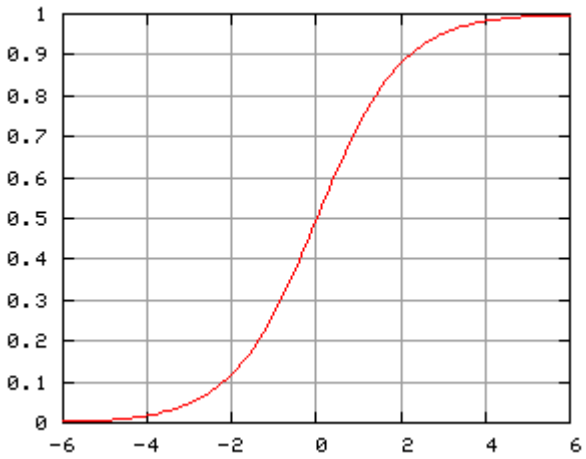
```
double clip(double x,
            double min = 0, double max = 1e100)
{
    if (x < min)
        return min;
    else if (x > max)
        return max;
    return x;
}

int main()
{
    std::cout << clip(5.6) << std::endl      // 5.6
              << clip(0) << std::endl        // 0
              << clip(-1.23) << std::endl    // 0
              << clip(5, 1.2) << std::endl   // 5
              << clip(0.5, 1.2) << std::endl // 1
              << clip(-3, 1.2) << std::endl  // 1
              << clip(3, 0, 5) << std::endl  // 3
              << clip(3, 4, 5) << std::endl  // 4
              << clip(3, 0, 1) << std::endl; // 1
    return 0;
}
```

I hope it's clear that the graph of the clip(x, min, max) (for fixed min and max) function is this:



Such functions that “squeeze” a value to within a range is common in many applications. For instance there's the Sigmoid function:



This is used in artificial intelligence in the study of artificial brain neurons. Maybe you will study that when you take CISS450 Artificial Intelligence.

Exercise. Write a `randrange()` function that accepts integers a and b and returns a random integer n such that a <= n < b. Test your code with this:

```
int randrange(int a, int b)
{
  ... CODE ...
}

int main()
{
  for (int i = 0; i < 10; i++)
  {
    std::cout << randrange(5, 15) << std::endl;
  }
```

```
  return 0;
}
```

Now modify your program so that it accepts a `step` variable. For instance if you call randrange(5, 15, 2), an integer randomly chosen from 5, 7, 9, 11, and 13 is returned; if you call randrange(5, 15, 3), an integer randomly chosen from 5, 8, 11, and 14 is returned;

Exercise. You are a programmer for Wal-art (they sell art supplies ...). Your boss told you to write a function to compute weekly pay. Write a function `getSalary()` that accepts the number of hours (as a double) and computes the wage using this formula:

- For each hour up to 40 hours, the hourly rate is 7.20
- For each overtime hour (i.e. beyond 40 hours), the hourly rate is 8.30.

Test your code with this `main()`

```
double getSalary(int hours)
{
    ... CODE ...
}


int main()
{
    std::cout << getSalary(30) << std::endl;
    std::cout << getSalary(45.5) << std::endl;
    return 0;
}
```

[... time passes ...nostalgic music ... scenes fade in and out ...] It's been 6 months and your `getSalary()` function has been used extensively in the information system. Your boss told you that the higher-ups have decided to create 3 types of hourly wage workers. The previous computation for the salary refers to an hourly wage worker of type 0. For type 1, the hourly wage is 8.50/hour for hours up to 40 hours/week and 9.30 for each hour after 40 hours. For employee of type 2, the hourly wage is 10.50/hour; overtime hourly rate is the same as the regular hourly rate.

Modify your `getSalary()` function so that previous usage is not broken. Test your code with this `main()`:

```
// *** NEED TO CHANGE ***
double getSalary(int hours)
{
  ... OLD CODE ...
}


int main()
{
    std::cout << getSalary(30) << std::endl;
    std::cout << getSalary(45.5) << std::endl;
    std::cout << getSalary(45.5, 0) << std::endl;
```

```
    std::cout << getSalary(45.5, 1) << std::endl;
    std::cout << getSalary(45.5, 2) << std::endl;
    return 0;
}
```

Default values of basic types are all the same. However you cannot use default values for arrays in the “obvious” way:

```
void f(int x[] = {1, 2, 3})
{
    ... code ...
}
```

The program will not even compile. In other words, the **array initializer list cannot be used as a default value**.

Prototypes for default values

First try this:

```
#include <iostream>

void printHeads(int = 1);

int main()
{
    printHeads();
    printHeads(42);
    return 0;
}

void printHeads(int numHeads = 1)
{
    std::cout << "number of heads: " << numHeads
              << std::endl;
}
```

It does **not** work. Remember that!!!

Now try this:

```
#include <iostream>

void printHeads(int = 1);

int main()
{
    printHeads();
    printHeads(42);
    return 0;
}

void printHeads(int numHeads)
{
    std::cout << "number of heads: " << numHeads
              << std::endl;
}
```

See the difference?

The **default values appear only in the prototype if you want to have a prototype**

and not in the function header when you define the function bodies.

(This is not arbitrary. There's a reason for this. But you would need to know how compilers work to understand why this is the case. So for now you just have to remember this fact.)

Of course if you don't have a prototype, then you can include the default

value in the function header – but NOT if you have a prototype.

That's the only gotcha you should remember. This is a very common mistake for beginners.

Multi-file compilation

I have nothing to add here. Just remember that the **default values appear only in the function prototypes**, not in the function headers when defining function bodies (see previous section) if you want to give that function a prototype. In other words, this **does not work**:

```
#include <iostream>
#include "test.h"

int main()
{
    printHeads();
    printHeads(42);
    return 0;
}
```

```
// test.h
#ifndef TEST_H
#define TEST_H

void printHeads(int numHeads = 1);

#endif
```

```
// test.cpp
#include <iostream>
#include "test.h"

void printHeads(int numHeads = 1)
{
    std::cout << "number of heads: " << numHeads
              << std::endl;
}
```

Modify your test.cpp:

```
// test.cpp
#include <iostream>
#include "test.h"

void printHeads(int numHeads)
{
    std::cout << "number of heads: " << numHeads
              << std::endl;
}
```

and it works. Make sure you run this.

Exercise. Identify all the errors in this header file:

```
#ifndef BLAH_H
#define BLAH_H

int[] spam(int = 0, char, int[] = {2, 3, 5, 7},
           int a = 1, int b = 2, int c = a + b);

#end
```

(There are 5).

Default value for array parameter

Recall that You cannot have default values for array parameters. For instance the following will not compile:

```
#include <iostream>

void print(int x[] = {2, 3})
{
    std::cout << x[0] << ' ' << x[1];
}

int main()
{
  print();
  return 0;
}
```

There's an exception: **The only exception is an array of characters.** In this case, **your array parameter must be an array of constant characters**:

```
#include <iostream>

void printHello(const char s[] = "world")
{
  std::cout << "hello " << s << std::endl;
}

int main()
{
  printHello();
  printHello("underverse");
  return 0;
}
```

This means that in the `printHello()` function you cannot change the characters in parameter `s`.

Exercise. Remove `const` from the above and try to compile.

Exercise. Write a function `print_dollar_amt()` such that

```
print_dollar_amt(123);     // prints $1.23
print_dollar_amt(123, 45); // prints $123.45
```