

EN47/COMP9

Exploring Computer Science

Lecture 5

Abstraction:

Functions and Constants

Outline

Today's big idea: **Abstraction**

Procedural or behavioral abstraction: **Functions**

Data abstraction: **Constants**

Printing banners

Program specifications:

Display many messages on the screen

Display a banner (two rows of asterisks) to separate sections of the output

```
Starting IM program...
*****
*****
Starting IM conversation...
Number of participants = 2
*****
*****
Start typing at any time.
*****
*****
```

What is the basic code needed to print the banner?

Printing banners

One version:

```
cout << "*****" << endl;  
cout << "*****" << endl;
```

Another version:

```
int count;  
for (count = 0; count < 30; count++) {  
    cout << "*" << endl;  
}  
cout << endl;  
for (count = 0; count < 30; count++) {  
    cout << "*" << endl;  
}  
cout << endl;
```

One complete solution

```
#include <iostream>
int main() {
    // produce some output
    ...
    // print banner lines
    cout << "*****\n";
    cout << "*****\n";

    // produce more output
    ...
    // print banner lines
    cout << "*****\n";
    cout << "*****\n";

    // produce even more output
    ...
    // print banner lines
    cout << "*****\n";
    cout << "*****\n";

    // produce final output
    ...
    return 0 ;
}
```

Is this correct C++ code?

Does it fulfill the program specification?

Are we satisfied?

What if we want to change the banners?

Number of rows, number of asterisks per row, use hyphens instead of asterisks, print the date with each row, ...

Have to edit every “copy” of the code in the program

- Easy to overlook some copies
- Hard to find them all
 - They might not be written identically
 - Code written identically might not serve the same logical purpose

Use a function!

```
#include <iostream>
int main() {
    // produce some output
    ...
    PrintBannerLines();

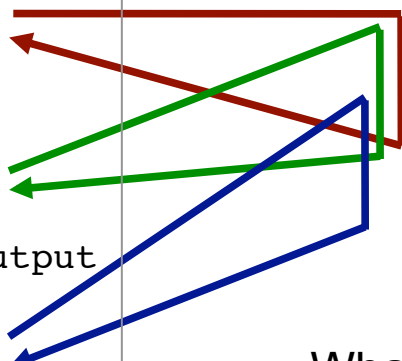
    // produce more output
    ...
    PrintBannerLines();

    // produce even more output
    ...
    PrintBannerLines();

    // produce final output
    ...
    return 0 ;
}
```

Define a function named PrintBannerLines:

```
// print banner lines
cout << "*****\n";
cout << "*****\n";
```



What do we do now if we want to change the banner?

How many places in the program have to be changed?

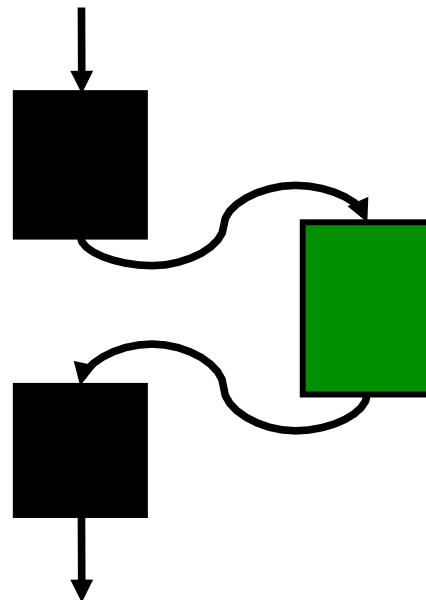
What if we want to print two rows of asterisks for something that isn't a banner?

Another form of control flow

Control flow: the order in which statements are executed

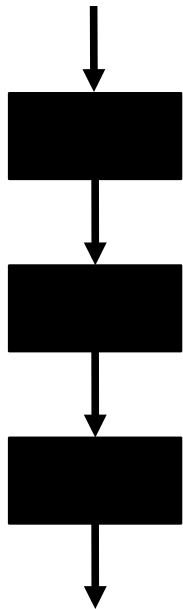
Functions (a.k.a. **procedures** or **subroutines**) allow you to “visit” a chunk of code and then come back

The function may be elsewhere in your own program, or may be code in another file altogether

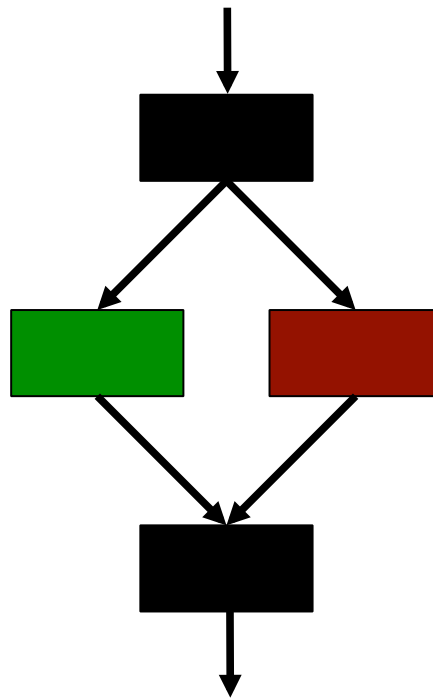


Recall other forms of control flow

Sequential

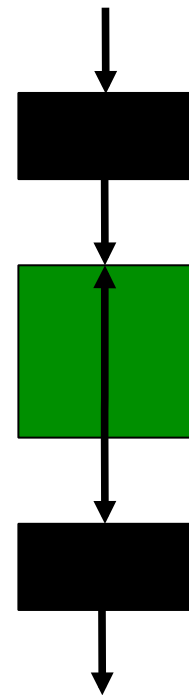


Conditional



if, else

Loop



while, for

Big idea for code: Functions

1. Identify the goal: “Print a banner.”

(More abstract than “Print two rows of asterisks.”)

2. Give the function that does that a name: PrintBannerLines

3. Define the solution by writing the code

```
// print banner lines
cout << "*****\n";
cout << "*****\n";
```

4. Whenever you want to print a banner, use the function name

```
PrintBannerLines();
```

Abstraction


“one name, one definition, many uses”

Functions

A familiar C++ function

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

Function definition
for main()



Parameters and returns values

The function does not return a value

The function has no parameters

```
// Function to print banner lines  
void PrintBannerLines()  
{  
    cout << "*****\n";  
    cout << "*****\n";  
}
```

Parameters

Suppose we want to change the program:

It should now print 5 rows when it starts and when it finishes,
but print the original 2-row banner everywhere else

We could write an additional function that prints 5 rows of
asterisks, or...

Parameters

Can we somehow generalize PrintBannerLines?

Can we modify the function so that it will print N rows of asterisks?

N is the number of rows that we want “this time” when we call it

N is information that is required to write the code of the function

Passing arguments

```
#include <iostream>
int main() {
    // produce some output
    ...
    PrintBannerLines(5);

    // produce more output
    ...
    PrintBannerLines(2);

    // produce even more output
    ...
    PrintBannerLines(5);

    // produce final output
    ...
    return 0 ;
}
```

argument of the call

parameter of the function

5

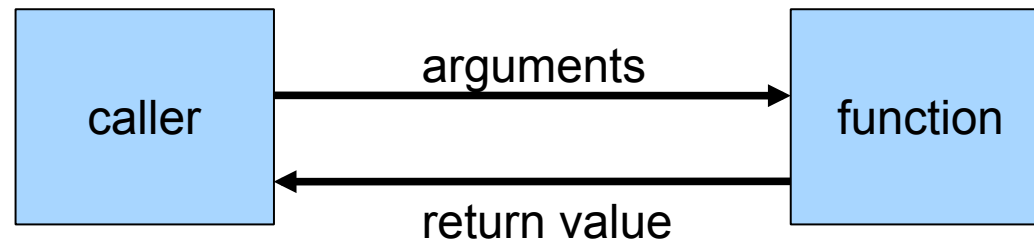
```
// Function to print banner lines
void PrintBannerLines (int numLines)
{
    int i;
    for (i=0; i< numLines; i++) {
        cout << "*****\n";
    }
}
```

numLines is the parameter of the function PrintBannerLines

numLines can be used inside the function just like a variable

Return value

A way for the function to send back data to the calling routine:
Opposite of parameters/arguments, which send data from the calling routine to the function



```
area=DiskArea(1.7);
```

```
return 3.14*radius*radius;
```

A function can have multiple parameters but only one return value

Example: DiskArea Function

Specification:

Write a function that returns the area of a disk of given radius

Example: DiskArea Function

Specification:

Write a function that returns the area of a disk of given radius

```
// Return area of disk with radius r
double DiskArea(double r)
{
    return (3.14159265364 * r * r);
}
```

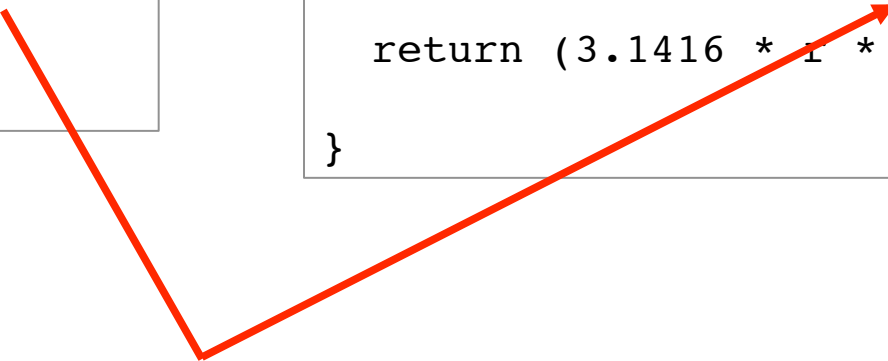
Matching up the arguments

The **function call** must include a matching argument for each parameter

When the function is executed, the value of the argument becomes the initial value of the parameter

```
int main (void) {  
    ...  
    z = 98.76;  
    x = 34.575 * Area ( z/2.0 );  
    ...  
    return 0;  
}
```

```
// Return area of disk  
// with radius r  
double DiskArea(double r) {  
    return (3.1416 * r * r);  
}
```



Multiple parameters

A function may have more than one parameter

Arguments must match parameters in number, order, and type

```
...  
double gpt, gpa;  
gpt = 3.0 + 3.3 + 3.9;  
gpa = Avg ( gpt, 3 );  
...
```

```
double Avg (double total, int count) {  
    return total / (double) count ;  
}
```

2 arguments that match the 2 parameters



Passing Arguments

```
#include <iostream>
int main(){
    // produce some output
    ...

    PrintBannerLines(5, '&', 21);

    // produce more output
    ...

    PrintBannerLines(2, '@', 15);

    // produce even more output
    ...

    PrintBannerLines(5, '1', 1);

    // produce final output
    ...
    return 0 ;
}
```

```
void PrintBannerLines(int numLines,
                      char mychar,
                      int numChars) {

    int i, j;

    for (i = 0; i < numLines; i++) {

        for (j = 0; j < numChars; j++) {
            cout << mychar;
        }

        cout << endl;
    }
}
```

More about calling functions

Empty () are required when making a call to a parameter-less (a.k.a. **void**) function

A function must be **declared** before it is called

```
#include <iostream>

void PrintBannerLines ( void ) {
    cout<<"*****\n";
    cout<<"*****\n";
}

int main () {
    ...

    PrintBannerLines( );
    ...

    return 0;
}
```

More about calling functions

The definition is an implicit declaration

Another way is to declare a **function prototype**

- Declares number and types of parameters, and type of return value

```
#include <iostream>

void PrintBannerLines ( void );

int main () {
    ...

    PrintBannerLines( );
    ...

    return 0;
}

void PrintBannerLines ( void ) {
    cout<<"*****\n";
    cout<<"*****\n";
}
```

Library functions

Pre-written functions are commonly packaged in **libraries**

Every C++ compiler comes with a set of standard libraries

There are many useful functions in other libraries

```
#include <iostream>
#include <string>
...
#include <cmath>
...
```

Summary: Functions

Functions may take several parameters, or none

Functions may return one value, or none

Functions are valuable!

- A tool for program structuring
- Provide *abstract* services: The caller cares what the function does, but not *how* it does it
- Make programs easier to write, debug, and understand

Symbolic constants

Symbolic constants

The big idea for *concrete data*

```
if (myMoney > 80.0)
{
    myShoes = myShoes + 1;
    myMoney = myMoney - 80.0;
}
```

Symbolic constants

The big idea for *concrete data*

```
if (myMoney > 80.0)
{
    myShoes = myShoes + 1;
    myMoney = myMoney - 80.0;
}
```

```
const double COST_OF_SHOES = 80.0;
...
if (myMoney > COST_OF_SHOES ) {
    myShoes = myShoes + 1;
    myMoney = myMoney - COST_OF_SHOES;
}
```

Sound familiar?

Functions abstract *behavior* (“procedural information”)

Symbolic constants abstract *data*

- Can use a variable:

```
double COST_OF_SHOES = 80.0;
```

- Even better, use a const

```
const double COST_OF_SHOES = 80.0;
```

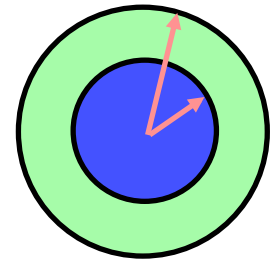
Example: Washer Area Function

Specification:

Write a function to find the area of a washer given the inner radius and outer radius.

Assume you already have another function that calculates the area of a circle:

```
double DiskArea (double r);
```

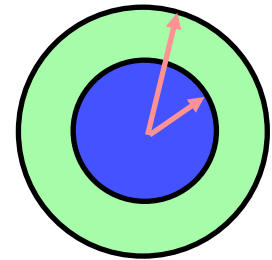


Example: Washer Area Function

Specification:

Write a function to find the area of a washer given the inner radius and outer radius.

Assume you already have another function that calculates the area of a circle:

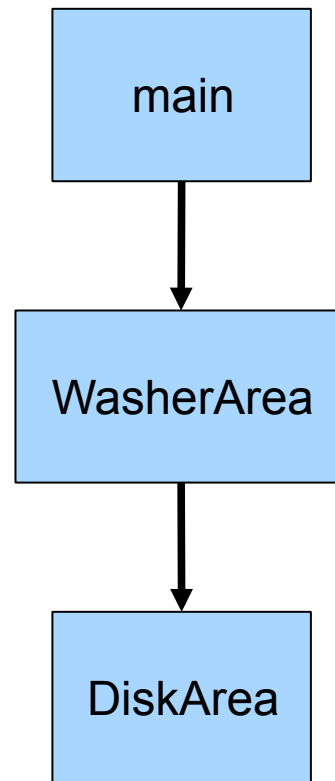


```
double DiskArea (double r);
```

```
double WasherArea (double inner, double outer) {  
    double innerArea, outerArea, areaOfWasher ;  
  
    innerArea = DiskArea (inner) ;  
    outerArea = DiskArea (outer) ;  
    areaOfWasher = outerArea - innerArea;  
  
    return areaOfWasher ;  
}
```

Static call graph

Shows which function calls which...



The Whole Program

```
// return area of disk
// with radius r
double DiskArea(double r) {
    return 3.1416 * r * r;
}

// Find area of washer
// with given inner and outer radius.
// calls double DiskArea (double r)
double WasherArea (double inner,
                   double outer) {
    double innerArea, outerArea;
    double areaOfWasher ;

    innerArea = DiskArea (inner) ;
    outerArea = DiskArea (outer) ;
    areaOfWasher = outerArea - innerArea;

    return areaOfWasher ;
}
```

```
// read washer info and print area
int main() {
    double inner, outer, area ;

    cout << "Input inner radius "
          << "and outer diameter:"
          << endl ;
    cin >> inner;
    cin >> outer;
    area = WasherArea (inner, outer/2.0) ;

    cout << area << endl ;

    return 0 ;
}
```

Summary

Today's big idea: **Abstraction**

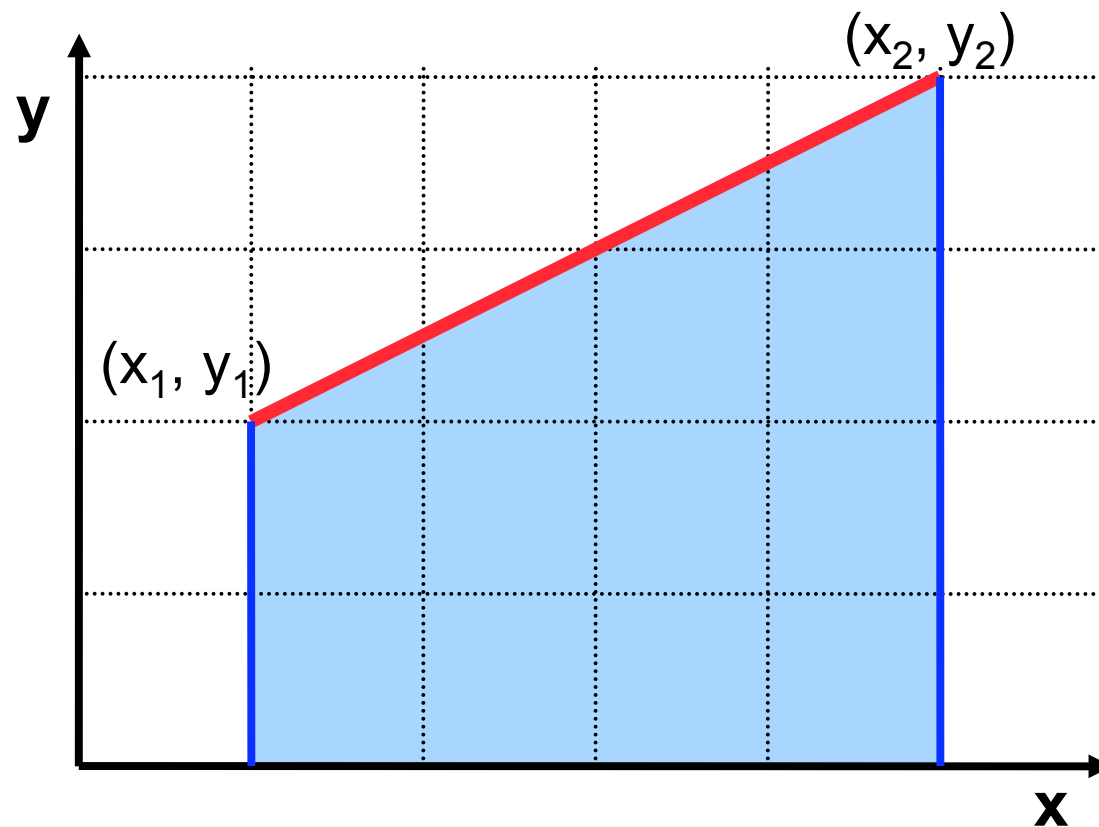
Procedural or behavioral abstraction: **Functions**

Data abstraction: **Constants**

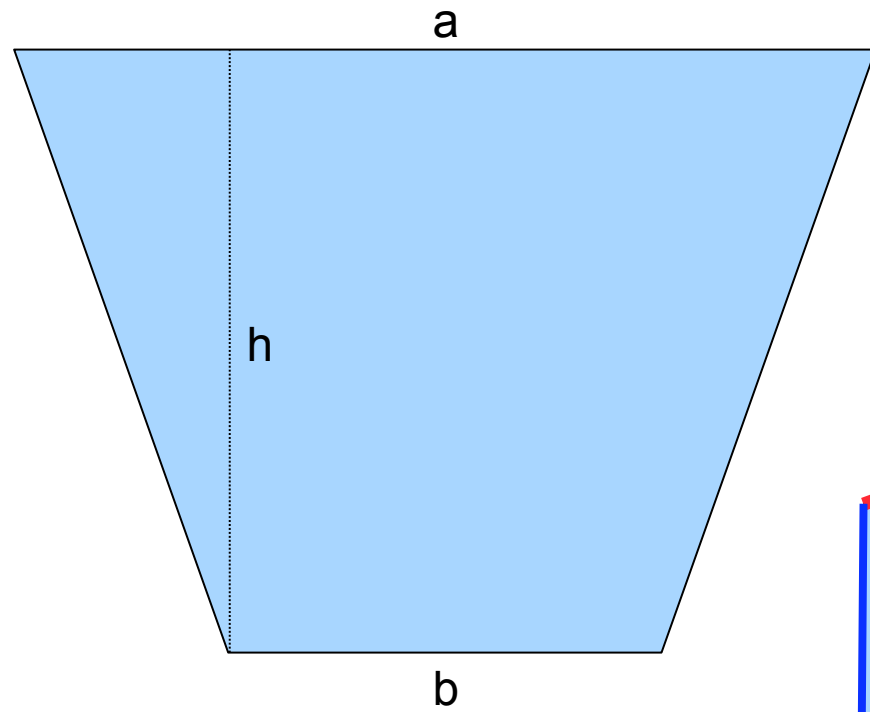
Lab 3 prep: Geometry review

What shape is formed between the line and the x axis?

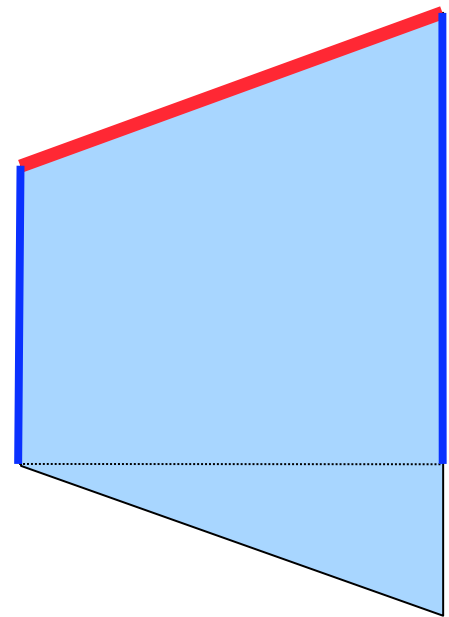
What is the area under the line?



Trapezoid



$$A = h(a+b)/2$$



For Thursday

Bring a write-up of your algorithm
for computing the area under a line!