

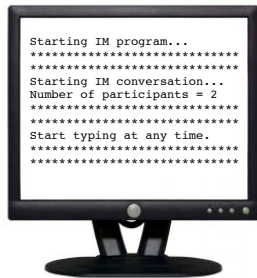
COMP 10
EXPLORING COMPUTER SCIENCE

Lecture 4
Abstraction:
Functions and Constants



Program specifications:

- Display many messages on the screen
- Display a two rows of asterisks to separate sections of the output



PRINTING BANNERS

One version:

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

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 ;
}
```

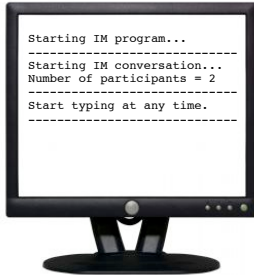
Is this correct C++ code?

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

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 to change the banner?

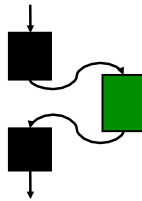
How many places in the code have to be changed?

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

ANOTHER FORM OF CONTROL FLOW

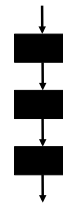
Control flow: the order in which statements are executed

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

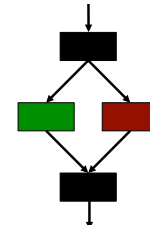


RECALL OTHER FORMS OF CONTROL FLOW

Sequential

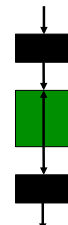


Conditional



if, else

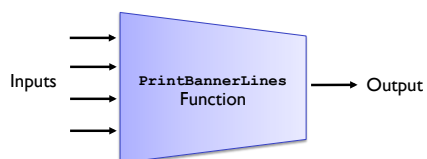
Loop



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



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"

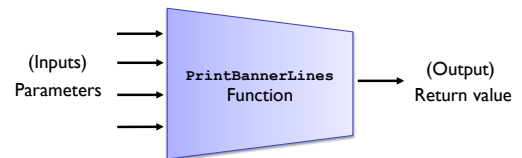
FUNCTIONS

A FAMILIAR C++ FUNCTION

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

Function definition for main()

PARAMETERS AND RETURN VALUES



PARAMETERS AND RETURN 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...

CAN WE GENERALIZE?

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";
    }
}
```

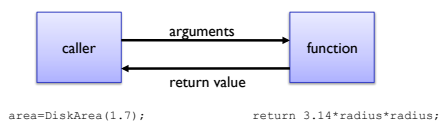
numLines is the parameter of the function PrintBannerLines

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



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

```
// Return area of disk with radius r
float DiskArea(float 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 * DiskArea ( z/2.0 );
    ...
    return 0;
}
```

```
// Return area of disk
// with radius r
float DiskArea(float 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

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

```
float Avg (float total, int count) {
    return total / (float) 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

```
#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

BIG IDEA FOR DATA:
SYMBOLIC CONSTANTS

One way to compute area:

```
float radius = 0;
cin >> radius;
float area = 3.14159 * radius * radius;
```

BIG IDEA FOR DATA:
SYMBOLIC CONSTANTS

One way to compute area:

```
float radius = 0;
cin >> radius;
float area = 3.14159 * radius * radius;
```

An alternative:

```
const float PI = 3.14159;
float radius = 0;
cin >> radius;
float area = PI * radius * radius;
```

SOUND FAMILIAR?

Functions abstract *behavior* (“procedural information”)Symbolic constants abstract *data*

Can use a variable:

```
float PI = 3.14159;
```

Even better, use a const

```
const float PI = 3.14159;
```

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:



```
float DiskArea (float 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:

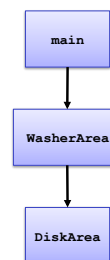


```
float DiskArea (float r);
```

```
float WasherArea (float inner, float outer) {
    float 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
float DiskArea(float r) {
    return PI * r * r;
}

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

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

    return areaOfWasher;
}

// read washer info and print area
int main() {
    float 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**

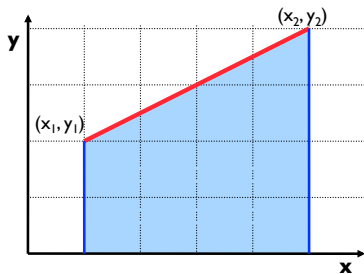
Procedural or behavioral abstraction: **Functions**

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

