Control flow in c++

- Flow control is the way a program causes the flow of execution .
- Two types

1.Branching statements

a) if statement

b) if- else statement

c) switch statement

d) goto statement

2. Looping statements

a) For statement.

a) While statement.

a) Do – while statement.

If statement

- Decision making statement
- Syntax

```
  if(test_expression)
   statements
```

Example:-

```
Int age
Cout<<“enter ur age”;
Cin>>age;
If(age > 12 && age < 20)
Cout<<“ u r teen aged”;
}
```

If – else statement

- **Nested if – else statement**

Example :-

```
 int age;
Cout<<“ enter ur age”;
Cin>>age;
If( age >12 && age <20)
Cout<<“u r a teen aged”;
Else
   if (age < 13 )
Cout<<“u will surely reach teen age”;
Else
 cout<<“u have crossed teen age”;
}
```

C++ Functions

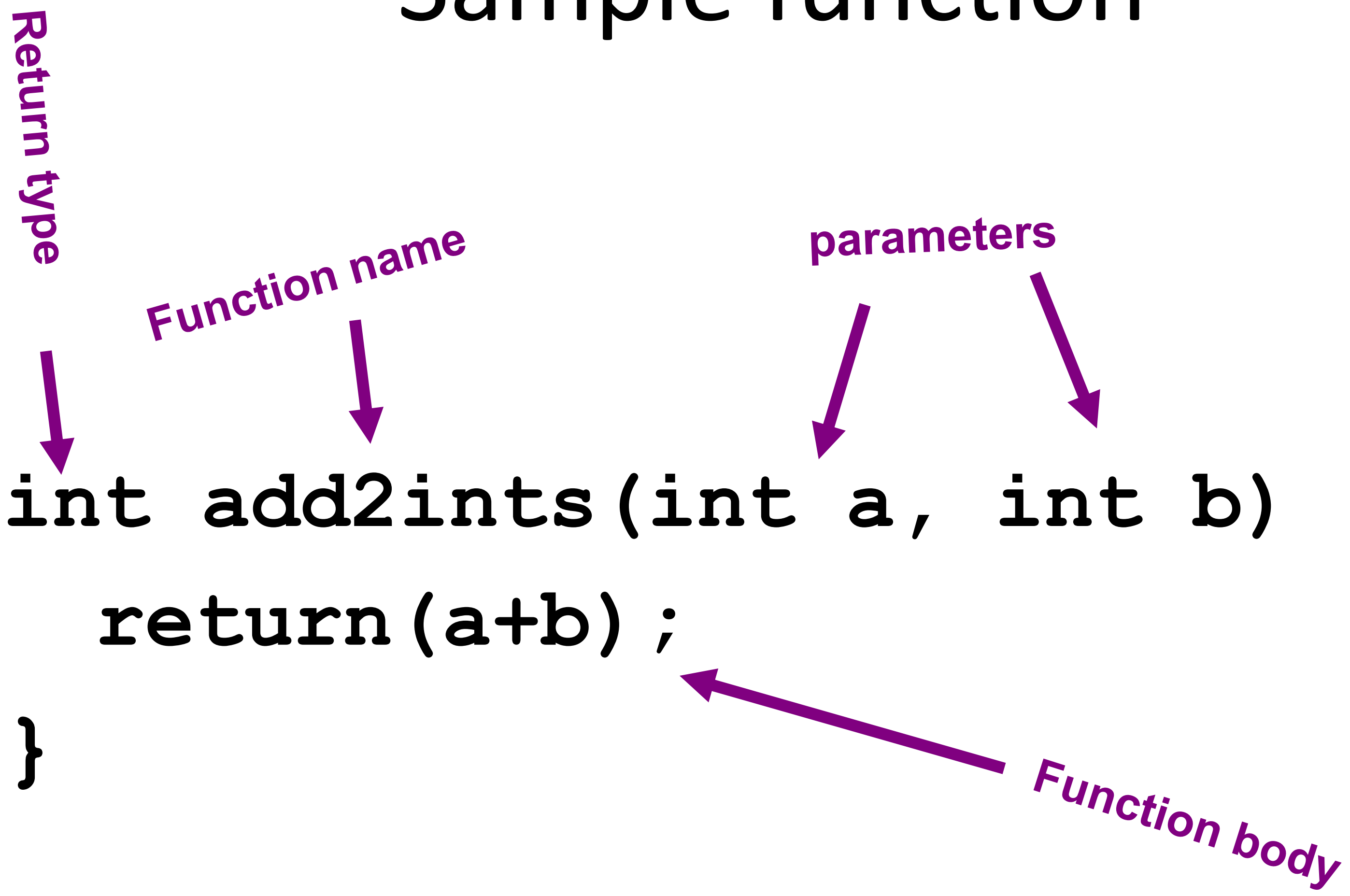
C++ Functions

- In other languages called subroutines or procedures.
- C++ functions all have a *type*.
  - Sometimes we don’t need to have a function return anything – in this case the function can have type **`void`**.

C++ Functions (cont.)

- C++ functions have a list of *parameters*.
  - Parameters are the things we give the function to operate on.
    - Each parameter has a *type*.
  - There can be zero parameters.

Sample function



Return type

Function name

parameters

```
int add2ints(int a, int b) {
   return(a+b);
}
```

Function body

Using functions – Math Library functions

- C++ includes a library of Math functions you can use.
- You have to know how to *call* these functions before you can use them.
- You have to know what they return.

`double sqrt( double )`

- When *calling* **`sqrt`**, we have to give it a **`double`**.
- The **`sqrt`** function returns a **`double`**.
- We have to give it a **`double`**.

```
x = sqrt(y);
x = sqrt(100);
```

`x = sqrt(y);`

- The stuff we give a function is called the argument(s). **`Y`** is the argument here.

Telling the compiler about `sqrt()`

- How does the compiler know about **`sqrt`** ?

- You have to tell it:

```
#include <math.h>
```

Other Math Library Functions

`ceil` `floor`

`cos` `sin` `tan`

`exp` `log` `log10` `pow`

`fabs` `fmod`

Writing a function

- You have decide on what the function will *look* like:
  - Return type
  - Name
  - Types of parameters (number of parameters)
- You have to write the body (the actual code).

Function parameters

- The parameters are *local variables* inside the body of the function.
  - When the function is called they will have the values *passed in*.
  - The function gets *a copy* of the values passed in

Sample Function

```
int add2nums( int firstnum, int secondnum ) {
  int sum;

  sum = firstnum + secondnum;

  // just to make a point
  firstnum = 0;
  secondnum = 0;

  return(sum);
}
```

Testing **add2nums**

```
int main(void) {
  int y,a,b;

  cout << "Enter 2 numbers\n";
  cin >> a >> b;

  y = add2nums(a,b);

  cout << "a is " << a << endl;
  cout << "b is " << b << endl;
  cout << "y is " << y << endl;
  return(0);
}
```

What happens here?

```
int add2nums(int a, int b) {
  a=a+b;
  return(a);
}
…
int a,b,y;
…
y = add2nums(a,b);
```

Local variables

- Parameters and variables declared inside the definition of a function are *local*.
- They only exist inside the function body.
- Once the function returns, the variables no longer exist!
  - That's fine! We don't need them anymore!

Block Variables

- You can also declare variables that exist only within the *body* of a compound statement *(a block*)*:*

```
{
int foo;
      …
      …
}
```

Global variables

- You can declare variables outside of any function definition – these variables are *global variables*.
- Any function can access/change global variables.
- Example: flag that indicates whether debugging information should be printed.

Function Prototypes

- A Function prototype can be used to *tell* the compiler what a function looks like
  - So that it can be called even though the compiler has not yet seen the function definition.
- A function prototype specifies the function name, return type and parameter types.

Example prototypes

```
double sqrt( double);

int add2nums( int, int);

int counter(void);
```

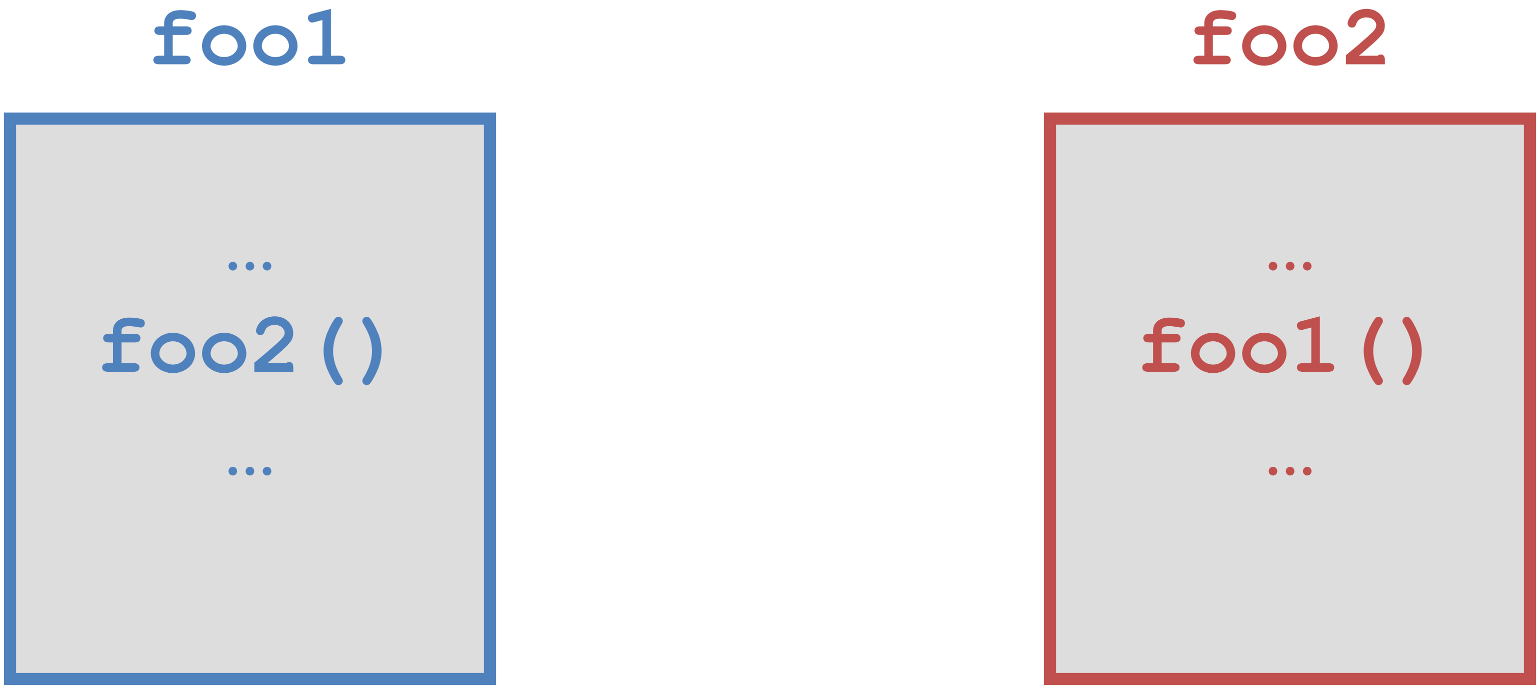
Using a prototype

```
int counter(void);

int main(void) {
  cout << counter() << endl;
  cout << counter() << endl;
  cout << counter() << endl;
}

int counter(void) {
  static int count = 0;
  count++;
  return(count);
}
```

Functions that call each other



foo1

…

foo2()

…

foo2

…

foo1()

…

Call-by-value vs. Call-by-reference

- So far we looked at functions that get a copy of what the *caller* passed in.
  - This is call-by-value, as the value is what gets passed in (the value of a variable).
- We can also define functions that are passed a *reference* to a variable.
  - This is call-by-reference, the function can change a callers variables directly.

References

- A *reference* variable is an alternative name for a variable. A *shortcut*.
- A reference variable must be initialized to *reference* another variable.
- Once the reference is initialized you can treat it just like any other variable.

Reference Variable Declarations

- To declare a reference variable you precede the variable name with a “&”:

```
int &foo;
double &blah;
char &c;
```

Reference Variable Example

```
int count;
int &blah = count;
// blah is the same variable as count

count = 1;
cout << “blah is “ << blah << endl;
blah++;
cout << “count is “ << count << endl;
```

Reference Parameters

- You can declare reference parameters:

```
void add10( int &x) {
  x = x+10;
}
…
add10(counter);
```

The parameter is a reference

Useful Reference Example

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

Recursion

- Functions can call themselves! This is called recursion.

- Recursion is very useful – it’s often very simple to express a complicated computation recursively.

Example - Computing Factorials

```
int factorial( int x ) {
  if (x == 1)
    return(1);
  else
    return(x * factorial(x-1));
}
```

Designing Recursive Functions

- Define “Base Case”:
  - The situation in which the function does **not** call itself.
- Define “recursive step”:
  - Compute the return value the help of the function itself.

Recursion Base Case

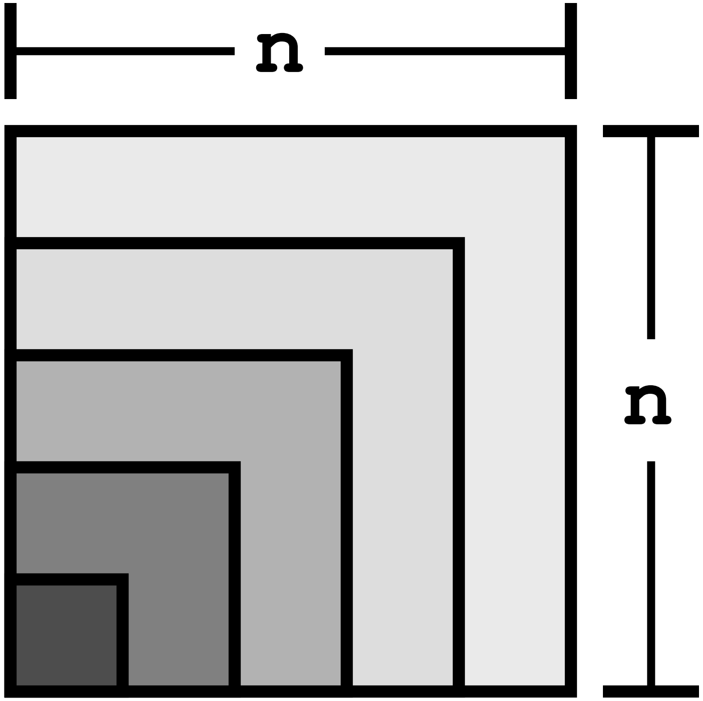
- The base case corresponds to a case in which you know the answer (the function returns the value immediately), or can easily compute the answer.
- If you don’t have a base case you can’t use recursion! (and you probably don’t understand the problem).

Recursive Step

- Use the recursive call to solve a **sub-problem.**
  - The parameters must be different (or the recursive call will get us no closer to the solution).
  - You generally need to do something besides just making the recursive call.

Recursion is a favorite test topic

- Write a recursive C++ function that computes the area of an **nxn** square.



Base case:

n=1 area=1

Recursive Step:

area = n+n-1+area(n-1)

Recursive area function

```
int area( int n) {
  if (n == 1)
    return(1);
  else
    return( n + n - 1 + area(n-1) );
}
```

Recursion Exercise

- Write a function that prints a triangle:

```
triangle(4);            triangle(5);

   *                        *
  ***                      ***
 *****                    *****
*******                  *******
                        *********
```