Annotated slides

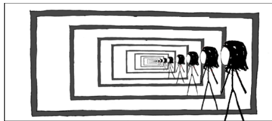
CS319: Scientific Computing (with C++)

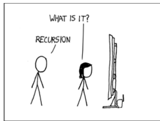
Week 3: More on functions

9am 23 Feb and 4pm 24 Feb, 2021

yesterday

today.





Source: http://xkcdsw.com/1105

1 Part 1: Flow of control – if-blocks
2 Part 2: Loops
3 Part 3: Functions
  - void functions
4 Part 4: Pass-by-value
5 Part 5: Function overloading
6 Part 6: A detailed example
7 Part 7: more details
  - Default values

New class times

	Mon	Tue	Wed	Thu	Fri
9 – 10		LECTURE	✗		
10 – 11		LAB			
11 – 12					
12 – 1					
1 – 2		LAB			
2 – 3					
3 – 4					
4 – 5			LECTURE		

1. The recorded class on Wednesdays at 9.00 moves to **Tuesday at 9.00**.
2. The recorded class on Thursdays at 16.00 stays.
3. **New lab times: Tuesday 10.00-10:50, and 13.00-13.50**. You should try to attend at least one of these.
4. Little, if any, of the "lab" times will be recorded.
5. This may all change again towards the end of the semester.
6. Might switch to Zoom for some classes. Any objections?

CS319 – Week 3
Week 3: More on functions

Start of ...

PART 1
Flow of control – `if`-blocks

Part 1: Flow of control – if-blocks

if statements are used to conditionally execute part of your code.

Structure (i):

```
if ( exprn )
{
  statements to execute if exprn evaluates as
               non-zero ;
}
else
{
   statements if exprn evaluates as 0 ;
}
```

Optional

Note : no semicolon at end of if line or else line

Part 1: Flow of control – if-blocks

The argument to `if()` is a **logical expression**. → something that is true or false.

Example

- `x == 8` (8 is stored in x. same as 8 == x).
- `m == '5'`
- `y <= 1` ← $y \leq 1$
- `y != x` ← $y \neq 1$.
- `y > 0`

! is used for negation

More complicated examples can be constructed using

- **AND** `&&` and
- **OR** `||`.

Eg
```
if ( (y>0) && (y<10))
```
true, eg, for y=1, y=9, but not y=-1.

Eg
```
if ( y>0)
```
same as
```
if ( ! (y<=0) ).
```

Part 1: Flow of control – if-blocks

The argument to `if()` is a **logical expression**.

Example

- `x == 8`
- `m == '5'`
- `y <= 1`
- `y != x`
- `y > 0`

More complicated examples can be constructed using

- **AND** `&&` and
- **OR** `||`.

we can also nest if statements.

Eg

```
if (y > 0) {
    if (y < 10) {
        cout << "y is between 0 and 10";
    }
} else { - }
```

Part 1: Flow of control – if-blocks

01EvenOdd.cpp

```
#include <iostream>

int main(void)
{
  int Number;

  std::cout << "Please enter an integrer: ";
  std::cin >> Number;

  if ( (Number%2) == 0)
    std::cout <<  "That is an even number." << std::endl;
  else
    std::cout <<  "That number is odd." << std::endl;

  return(0);
}
```

[is a number odd or even?]
% gives remainder on division.

Part 1: Flow of control – `if`-blocks

More complicated examples are possible:

Structure (ii):

```
if ( exp1 )
{
    statements to execute if exp1 is "true"
}
else if (exp2)
{
    statements run if exp1 is "false" but exp2 is "true"
}
else
{
    "catch all" statements if neither exp1 or exp2 true.
}
```

Part 1: Flow of control – if-blocks

02Grades.cpp

```
int main(void)
{
   int NumberGrade;
   char LetterGrade;

   std::cout << "Please enter the grade (percentage): ";
   std::cin >> NumberGrade;

   if ( NumberGrade >= 70 )
      LetterGrade = ’A’;
   else if ( NumberGrade >= 60 )
      LetterGrade = ’B’;
   else if ( NumberGrade >= 50 )
      LetterGrade = ’C’;
   else if ( NumberGrade >= 40 )
      LetterGrade = ’D’;
   else
      LetterGrade = ’E’;

   std::cout << "A score of " << NumberGrade << "% cooresponds to
             << LetterGrade << "." << std::endl;
}
```

not use of single = for assignment.

Part 1: Flow of control – `if`-blocks

The other main flow-of-control structures are the `?:` operator, and `switch ... case` structures.

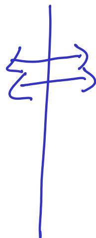
Example (1.)

How to use `?:`

Suppose we want to set $A = |x|$.

We could do this as

```
if ( x<0)
    A = -x;
else
    A = x;
```



$A = (x<0) ? -x : x;$

condition

what to do if condition is true

else.

Example (2.)

How to use `?:` with `std::cout`.

To display |x| try

```
std::cout << "|x| = " <<
    ( (x < 0) ? -x : x) << std::endl;
```

Part 1: Flow of control – `if`-blocks

Exercise 2.1

Find out how `switch... case` construct works, and write a program that uses it.

```
int i = 3, j;
if ( i > 2)
   { j = 2* i;}
else if ( i < 4)
   { j = -i; }.
else
   j=0 ;
```

skipped

At the end of this

j = 6.

Part 1: Flow of control – `if`-blocks

CS319 – Week 3
Week 3: More on functions

END OF PART 1

CS319 – Week 3
Week 3: More on functions

Start of ...

PART 2: Loops

We meet a for-loop briefly in the Fibonacci example. The most commonly used loop structure is for

```
for(initial value; test condition; step)
{
    // code to execute inside loop
}
```

Example: 03CountDown.cpp

```
10 int main(void)
   {
12   int i;

14   for (i=10; i>=1; i--)
       std::cout << i << "... ";

     std::cout << "Zero!\n";

     return(0);
20 }
```

i is initialised to 10

Keep iterating as long as i ≥ 1

at the end of every iteration, set i = i − 1.

1. The syntax of `for` is a little unusual, particularly the use of semicolons to separate the “arguments”.

2. All three arguments are optional, and can be left blank. Example:

The loop on previous slide same as

```
int i=10;
for ( ; i>=1 ; )
{
   std::cout << i << " ... " << std::endl;
   i--;
}
```

3. But it is not good practice to omit any of them, and very bad practice to leave out the middle one (test condition).

4. It is very common to define the increment variable within the for statement, in which case it is "local" to the loop. Example:

```
for (int i= 10; i>=1; i++)
{
    do stuff;
}
```

[Here i is "local" to the for loop]

5. If the body of the loop has only one line, you can omit the { and }.

So, eg.

```
for ( — )
{ cout << "hello";
}
```

optional → { and }

6. There is no semicolon at the end of the `for` line.

The other two common forms of loop in C++ are

- `while` loops
- `do ... while` loops

Exercise 2.2

Rewrite the **count down** example above using a

1. `while` loop.
2. `do ... while` loop.

[Finished here at 10am].

CS319 – Week 3
Week 3: More on functions

END OF PART 2

```
for (int i=0; i<10; i+=2)
{
   ~~~;
}

for (int i=10; i>0; i-=2)
{__
```

← OK: the two "i" variables are different.

CS319 – Week 3
Week 3: More on functions

Start of ...

PART 3: FUNCTIONS

This is Wed (24th Feb) 4pm.

Part 3: Functions

A good understanding of **functions**, and their uses, is of prime importance.
Some functions return/compute a single value. However, many important functions return more than one value, or modify one of its own arguments.
For that reason, we need to understand the difference between **call-by-value** and **call-by-reference** (⟵ later).

. . . .

Every C++ program has at least one function: `main()`

Example

```
#include <iostream>
int main(void )
{
   /* Stuff goes here */
   return(0);
}
```

Part 3: Functions (other than "main").

Each function consists of two main parts:

- Function "header" or **prototype** which gives the function's
  - return value data type, or `void` if there is none, and
  - parameter list data types or `void` if there are none.

  The prototype is often given near the start of the file, before the **main()** section.
  Important: The prototype should be written before the function—perhaps when the program is begin specified.

- **Function definition**. Begins with the function names, parameter list and return type, followed by the body of the function contained within curly brackets.

Part 3: Functions

Syntax

Format:

```
ReturnType FnName ( param1, param2, ...)
{
      statements
}
```

identifier

- ReturnType is the data type of the data returned by the function.
- FnName the identifier by which the function is called.
- Param1, ... consists of
  - the data type of the parameter
  - the name of the parameter will have in the function. It acts within the function as a local variable.
- the statements that form the function's body, contained with braces {...}.

04IsComposite.cpp

```
bool IsComposite(int i)
{
   int k;
   for (k=2; k<i; k++)
     if ( (i%k) == 0)
       return(true);   ← if this is executed, then the function end.

   // If we get to here, then i has no divisors between 2 and i-1
   return(false);
}
```

Takes a single int variable, i, as its argument

Returns either true or false (type bool).

Calling the `IsComposite` function:

04IsComposite.cpp

```
12 int main(void )
   {
14   int i;

16   std::cout << "Enter a natural number: ";
     std::cin >> i;

     std::cout << i << " is a " <<
20     (IsComposite(i) ? "composite" : "prime") << " number."
       << std::endl;

     return(0);
24 }
```

This is same as
if (IsCosposite(i) == true)
cout << i << " is is prime";

Most functions will return some value. In rare situations, they don't, and so have a void argument list.

05Kth.cpp

```
#include <iostream>

void Kth(int i);

int main(void )
{                                                   14
  int i;

  std::cout << "Enter a natural number: ";
  std::cin >> i;                                    18

  std::cout << "That is the ";                      20
  Kth(i);
  std::cout << " number." << std::endl;             22

  return(0);                                        24
}
```

05Kth.cpp (continued)

```
// FUNCTION KTH
// ARGUMENT: single integer
// RETURN VALUE: void (does not return a value)
// WHAT: if input is 1, displays 1st, if input is 2, displays 2nd,
// etc.
void Kth(int i)
{
  std::cout << i;
  i = i%100;
  if ( ((i%10) == 1) && (i != 11))
    std::cout << "st";
  else if ( ((i%10) == 2) && (i != 12))
    std::cout << "nd";
  else if ( ((i%10) == 3) && (i != 13))
    std::cout << "rd";
  else
    std::cout << "th";
}
```

CS319 – Week 3
Week 3: More on functions

END OF PART 2

3

CS319 – Week 3
Week 3: More on functions

Start of ...

PART 4: Pass-by-value

(in functions).

Part 3: Pass-by-value

In C++ we need to distinguish between

- a variable's (unique) memory address
- a variable's identifier (might not be unique) item the value stored in the variable.

The classic example is function that

- takes two `int`eger inputs, `a` and `b`;
- after calling the function, the values of `a` and `b` are swapped.

To understand this example, it is important to understand the difference between a

1. **local variable**, which belongs only to the function (or block) in which it is defined;
2. **global variable**, which belongs to the whole programme, and can be accessed in any function (or block).

(Global variables are very uncommon, but we'll have a look at them in some lab exercises).

Part 3: Pass-by-value

06SwapByValue.cpp

```
4  #include <iostream>
   void Swap(int a, int b);

   int main(void )
8  {
     int a, b;

     std::cout << "Enter two integers: ";
12   std::cin >> a >> b;

14   std::cout << "Before Swap: a=" << a << ", b=" << b
               << std::endl;
16   Swap(a,b);
     std::cout << "After Swap: a=" << a << ", b=" << b
18             << std::endl;
     return(0);
20 }
```

Try this (they won't be swapped)

Part 3: Pass-by-value

```
void Swap(int x, int y)
{
  int tmp;

  tmp=x;
  x=y;
  y=tmp;
}
```

This won't work.
We have passed only the *values stored in the variables a and b*. In the `swap` function these values are copied to local variables x and y. Although the local variables are swapped, they remained unchanged in the calling function.

What we really wanted to do here was to use **Pass-By-Reference** where we modify the contents of the memory space referred to by `a` and `b`. This is easily done...
...we just change the declaration and prototype from

```
void Swap(int x, int y)   // Pass by value
```

to

```
void Swap(int &x, int &y)   // Pass by Reference
```

Part 3: Pass-by-value

CS319 – Week 3
Week 3: More on functions

END OF PART 3 4

CS319 – Week 3
Week 3: More on functions

Start of ...

PART 4: Function overloading

Part 4: Function overloading

C++ has certain features of **polymorphism** – for example, where two different functions can have the same name, so long as they have different argument lists.

This is called **function overloading**.

As a simple example, we'll write two functions with the same name: one that swaps the values of a pair of `int`s, and that other that swaps a pair of `float`s. (Later in the course, we'll see how to do this with `template`s.)

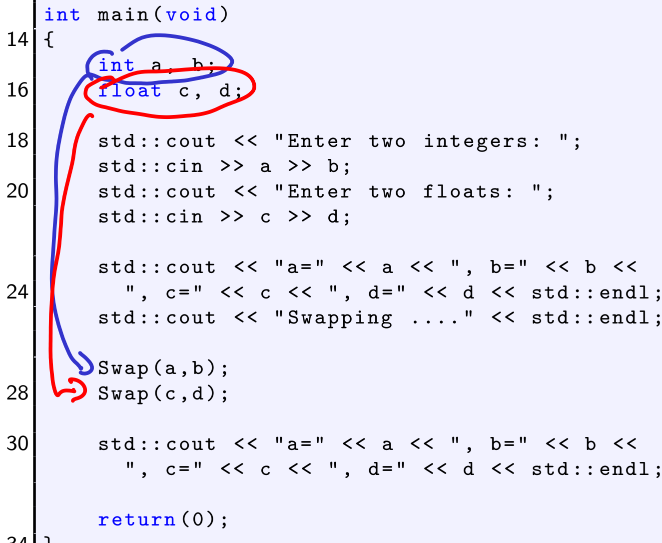
07Swaps.cpp

```
#include <iostream>

// We have two function prototypes!
void Swap(int &a, int &b);
void Swap(float &a, float &b);
```

Part 4: Function overloading

07Swaps.cpp (continued)



```
int main(void)
{
    int a, b;
    float c, d;

    std::cout << "Enter two integers: ";
    std::cin >> a >> b;
    std::cout << "Enter two floats: ";
    std::cin >> c >> d;

    std::cout << "a=" << a << ", b=" << b <<
      ", c=" << c << ", d=" << d << std::endl;
    std::cout << "Swapping ...." << std::endl;

    Swap(a,b);
    Swap(c,d);

    std::cout << "a=" << a << ", b=" << b <<
      ", c=" << c << ", d=" << d << std::endl;

    return(0);
}
```

Part 4: Function overloading

07Swaps.cpp (continued)

```
    void Swap(int &a, int &b)
40  {
      int tmp;

      tmp=a;
44    a=b;
      b=tmp;
46  }

48  void Swap(float &a, float &b)
    {
50    float tmp;

52    tmp=a;
      a=b;
54    b=tmp;
    }
```

Part 4: Function overloading

What does the compiler take into account to distinguish between overloaded functions?

C++ takes the following into account: (the "function signature")

- **Type of arguments**. So `void Sort(int, int)` is different from `void Sort(char, char)`.
- **The number of arguments**. So `int Add(int a, int b)` is different from `int Add(int a, int b, int c)`.

But not

- **Return values**. For example, we cannot have two functions `int Convert(int)` and `float Convert(int)` since they have the same argument list.
- **user-defined types** (using `typedef`) that are in fact the same. See, for example, `10OverloadedConvert.cpp`.

Part 5: Function overloading

Say A & B are arrays of same length: The setting
C = A+B; is very useful.

CS319 – Week 3
Week 3: More on functions

END OF PART 5

in C, to compute absolute value
of a int, use abs()
of a float, use fabs().
In C++, can just use abs() in all cases

CS319 – Week 3
Week 3: More on functions

Start of ...

PART 6: A detailed example

Part 5: A detailed example

In the following example, we combine two features of C++ functions:

- Pass-by-reference,
- Overloading,

We'll write two functions, both called `Sort`:

- `Sort(int &a, int &b)` – sort two integers in ascending order.
- `Sort(int list[], int n)` – sort the elements of a list of length n.

The program will make a list of length 8 of random numbers between 0 and 39, and then sort them using **bubble sort**.
(See video for full description).

after Sort (a,b)
should find that $a \leq b$.

Part 5: A detailed example

09Sort.cpp (i)

```
#include <iostream>
#include <stdlib.h>

const int N=8;

void Sort(int &a, int &b);
void Sort(int list[], int length);
void PrintList(int x[], int n);
```

Example of Global variable.

Function prototypes.

means N cannot change.

Part 5: A detailed example

09Sort.cpp (ii)

```
14  int main(void )
    {
16    int i, x[N];

18    for (i=0; i<N; i++)
        x[i]=rand()%40;

      std::cout << "The list is:\t\t";
22    PrintList(x, N);
      std::cout << "Sorting..." << std::endl;

      Sort(x,N);

      std::cout << "The sorted list is:\t";
28    PrintList(x, N);
      return(0);
30  }
```

Generating 8 random numbers

This is calling the "list" version of Sort.

Part 5: A detailed example

09Sort.cpp (iii)

```
32 // Arguments:  two integers
   // return value:  void
34 // Does:  Sorts a and b so that a<=b.
   void Sort(int &a, int &b)
36 {
     if (a>b)
38   {
       int tmp;
40     tmp=a;  a=b;   b=tmp;
     }
42 }

44 // Arguments:  an integer array and its length
   // return value:  void
46 // Does:  Sorts the 1st n elements of x
   void Sort(int x[], int n)
48 {
     int i, k;
50   for (i=n-1; i>1; i--)
       for (k=0; k<i; k++)
52       Sort(x[k], x[k+1]);
   }
```

Eg Suppose list is

7, 5, 3, 1.

So n=4.

i=3, k=0
Sort(7,5)
[5, 7, 3, 1]

i=3, k=1. Sort(7,3)
[5, 3, 7, 1]
i=3, k=2, Sort(7,1)
[5, 3, 1, 7].

Part 5: A detailed example

```
62 void PrintList(int x[], int n)
   {
64   for (int i=0; i<n; i++)
       std::cout << x[i] << "  ";
66   std::cout << std::endl;
   }
```

Part 5: A detailed example

CS319 – Week 3
Week 3: More on functions

END OF PART 5

Finished here for the week.

CS319 – Week 3
Week 3: More on functions

Start of ...

PART 6: More details

In C++, one can also define functions that have assigned default values:

```
int mult(int a, int b=1, int c=1) // from 10Mult.cpp
{
   return(a * b * c);
}
```

This means that, if the user fails to provide the second and third arguments to the function, it is assumed that they are both 1.

Example

```
std::cout << "mult(1) = " << mult(1);
std::cout << "mult(1,2) = " << mult(1,2);
std::cout << "mult(1,2,3) = " << mult(1,2,3);
```

Our next example is in 11Binary.cpp, and uses some *bitwise operators*.
These relate to the logical operators you may have seen in CS304.
First we'll look at a function to convert from decimal to binary:

11Binary.cpp

```
   std::string Int_to_Binary(int a)
46 {
     std::string A="";
48   for (int i=(int)log2(a); i>=0; i--)
     {
50     if ( a >= pow(2,i))
       {
52       A=A+"1";
         a=a-pow(2,i);
54     }
       else
56       A=A+"0";
     }
58   return(A);
   }
```

We'll return to a recursion-based implementation later...

Next, the calling part (modified from the actual code to simplify formatting):

11Binary.cpp (main function)

```
int a, b, c;

std::cout << "Input two integers: ";
std::cin >> a >> b;
std::cout << "You entered: " << a << " and " << b;

std::cout << a << " = " << Int_to_Binary(a) << std::endl;
std::cout << b << " = " << Int_to_Binary(b) << std::endl;

c = a^b;
std::cout << "XOR: a^b = " << c << " = " << Int_to_Binary(c);
c = a&b;
std::cout << "AND: a&b = " << c << " = " << Int_to_Binary(c);
c = a|b;
std::cout << " OR: a|b = " << c << " = " << Int_to_Binary(c);
```