C++

Variables, types: **`int, float, double, bool`**, ...
Assignments
Expressions

Basic control structures: **`if, else`**
Loops **`while, do-while, for, ...`**

Type casting: **`static_cast`**

Basic mathematical functions
Functions, Recursion

Glen Cowan, RHUL Physics
Rob Miller, University College London
Adapted by Martin Erdmann

C++ building blocks

All of the words in a C++ program are either:

Reserved words: cannot be changed, e.g.,

`if, else, int, double, for, while, class`, ...

Library identifiers: default meanings usually not changed, e.g., **`cout, sqrt`** (square root), ...

Programmer-supplied identifiers:

e.g. variables created by the programmer,

`x, y, probeTemperature, photonEnergy`, ...

Valid identifier must begin with a letter or underscore ("_") , and can consist of letters, digits, and underscores.

Try to use meaningful variable names.

Reserved words

asm	continue	float	new	signed	try
auto	default	for	operator	sizeof	typedef
break	delete	friend	private	static	union
case	do	goto	protected	struct	unsigned
catch	double	if	public	switch	virtual
char	else	inline	register	template	void
class	enum	int	return	this	volatile
const	extern	long	short	throw	while
using	namespace	bool	static_cast	const_cast	
dynamic_cast		true	false		

Organization: Computer Memory

000110011111101000111

8 bits = 1 Byte

e.g.
32 bits = 1 word

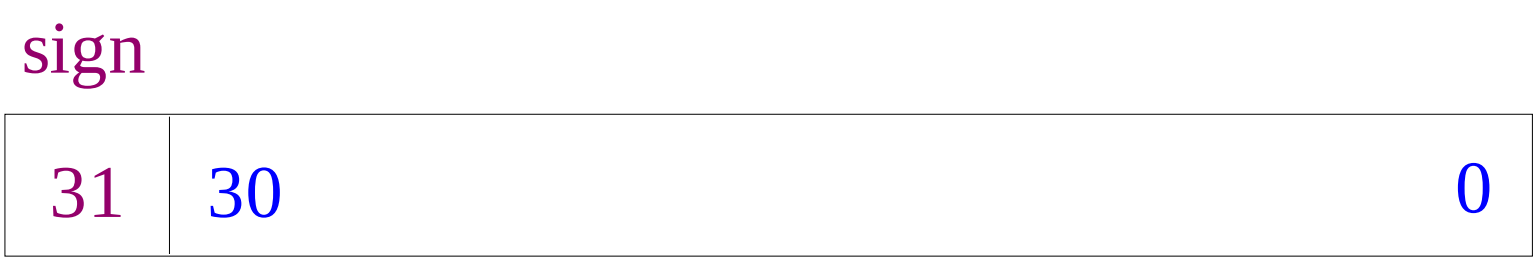
1024	0	0	0	1	1	0	1	1
1032	0	1	1	1	0	0	0	1
1040	0	1	1	0	0	1	0	0
1048	0	0	1	1	1	0	1	1

machine dependent!

Data Representation: Integer (IEEE)

Decimal → Binary

0 → 0...000
1 → 0...001
2 → 0...010
3 → 0...011
...



Range for 32-bit machines: $-2^{31} \rightarrow 2^{31} - 1$

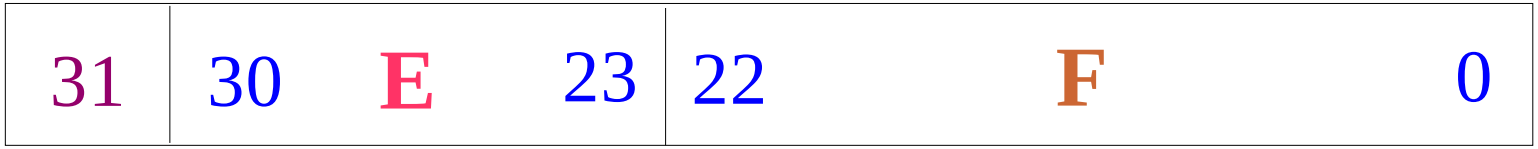
or **-2147483648 → 2147483647**

Data Representation: Real (IEEE)

$$Z = (1 - 2 * b_{sign}) * F * 2^{E}$$

$$F = b_1 2^{-1} + b_2 2^{-2} \ldots \quad (b_i = 0,1)$$

sign: b_{sign}



$-3.4\ 10^{+38} \rightarrow -1.17\ 10^{-38}$ 0 $1.17\ 10^{-38} \rightarrow 3.4\ 10^{+38}$

traps: range, rounding errors, not exact:Z=1/3

C++ Data types

Data values can be stored in variables of several types.

Basic integer type: **`int`** (also **`short`**, **`unsigned`**, **`long int`**, ...)

Number of bits used depends on compiler; typically 32 bits, check number of Bytes on computer using function sizeof(int)

Basic floating point types (i.e., for real numbers):

`float` usually 32 bits

`double` usually 64 bits best for our purposes

Boolean: **`bool`** (equal to **`true`** or **`false`**)

Character: **`char`** (single ASCII character only, can be blank), no native 'string' type; more on C++ strings later.

Declaring variables

All variables must be declared before use.

```
int main(){
  int numPhotons;        // Use int to count things
  double photonEnergy;   // Use double for reals
  bool goodEvent;        // Use bool for true or false
  int minNum, maxNum;    // More than one on line
  int n = 17;            // Can initialize value
  double x = 37.2;       // when variable declared.
  char yesOrNo = 'y';    // Value of char in ' '
      ...
}
```

Assignment of values to variables

Declaring a variable establishes its name; value is undefined (unless done together with declaration).

Value is assigned using = (the assignment operator):

```
int main(){
  bool aOK = true;  // true, false
predefined constants
  double x, y, z;
  x = 3.7;
  y = 5.2;
  z = x + y;
  cout << "z = " << z << endl;
     ...
}
```

Constants

Sometimes we want to ensure the value of a variable doesn't change.

Useful to keep parameters of a problem in an easy to find place, where they are easy to modify.

Use keyword **`const`** in declaration:

```
const int numChannels = 12;
const double PI = 3.14159265;

// Attempted redefinition by Indiana State Legislature
PI = 3.2;            // ERROR will not compile
```

Old C style retained for compatibility (avoid this):

```
#define PI 3.14159265
```

Expressions

C++ has obvious(?) notation for mathematical expressions:

operation	symbol
addition	`+`
subtraction	`-`
multiplication	`*`
division	`/`
modulus	`%`

Note division of **`int`** values is truncated:

```
int n, m;  n = 5;  m = 3;
int ratio = n/m;      // ratio has value of 1
```

Modulus gives remainder of integer division:

```
int nModM = n%m;          // nModM has value 2
```

Operator precedence

`*` and `/` have precedence over `+` and `-`, i.e.,

```
x*y + u/v  means  (x*y) + (u/v)
```

`*` and `/` have same precedence, carry out left to right:

```
x/y/u*v means ((x/y) / u) * v
```

Similar for `+` and `-`

```
x - y + z  means  (x - y) + z
```

Many more rules (google for C++ operator precedence).

Easy to forget the details, so use parentheses unless it's obvious.

Boolean expressions and operators

Boolean expressions are either true or false, e.g.,

```
int n, m; n = 5; m = 3;
bool b = n < m;              // value of b is false
```

C++ notation for boolean expressions:

greater than	`>`
greater than or equals	`>=`
less than	`<`
less than or equals	`<=`
equals	`==`
not equals	`!=`

not =

Can be combined with `&&` (“and”), `||` (“or”) and `!` (“not”), e.g.,

```
(n < m) && (n != 0)          (false)
(n%m >= 5) || !(n == m)      (true)
```

Precedence of operations not obvious; if in doubt use parentheses.

Shorthand assignment statements

full statement	shorthand equivalent
`n = n + m`	`n += m`
`n = n - m`	`n -= m`
`n = n * m`	`n *= m`
`n = n / m`	`n /= m`
`n = n % m`	`n %= m`

Special case of increment or decrement by one:

full statement	shorthand equivalent
`n = n + 1`	`n++` **(or `++n`)**
`n = n - 1`	`n--` **(or `--n`)**

`++` or `--` before variable means first increment (or decrement), then carry out other operations in the statement (more later).

if and else

Simple flow of control is done with **`if`** and **`else`**:

```
if ( boolean test expression ){
  Statements executed if test expression true
}
```

or

```
if (expression1 ){
  Statements executed if expression1 true
}
else if ( expression2 ) {
  Statements executed if expression1 false
  and expression2 true
}
else {
  Statements executed if both expression1 and
  expression2 false
}
```

more on if and else

Note indentation and placement of curly braces:

```
if ( x > y ){
  x = 0.5*x;
}
```

Some people prefer

```
if ( x > y )
{
  x = 0.5*x;
}
```

If only a single statement is to be executed, you can omit the curly braces -- this is usually a bad idea:

```
if  ( x > y )  x = 0.5*x;
```

Putting it together -- checkArea.cc

```
#include <iostream>
using namespace std;
int main() {
  const double maxArea = 20.0;
  double width, height;
  cout << "Enter width" << endl;
  cin >> width;
  cout << "Enter height" << endl;
  cin >> height;
  double area = width*height;
  if ( area > maxArea ){
    cout << "Area too large" << endl;
  }
  else {
    cout << "Dimensions are OK" << endl;
  }
  return 0;
}
```

“while” loops

A **while** loop allows a set of statements to be repeated as long as a particular condition is true:

```
while( boolean expression ){
  // statements to be executed as long as
  // boolean expression is true

}
```

For this to be useful, the boolean expression must be updated upon each pass through the loop:

```
while (x < xMax){
  x = x + y;
  ...
}
```

Possible that statements never executed, or that loop is infinite.

“do-while” loops

A **do-while** loop is similar to a **while** loop, but always executes at least once, then continues as long as the specified condition is true.

```
do {
  // statements to be executed first time
  // through loop and then as long as
  // boolean expression is true

} while ( boolean expression )
```

Can be useful if first pass needed to initialize the boolean expression.

"for" loops

A **for** loop allows a set of statements to be repeated a fixed number of times. The general form is:

```
for ( initialization action ;
      boolean expression ; update action ){
  // statements to be executed

}
```

Often this will take on the form:

```
for (int i=0; i<n; i++){
  // statements to be executed n times

}
```

Note that here **i** is defined only inside the **{ }.**

Examples of loops

A for loop:

```
int sum = 0;
for (int i = 1; i<=n; i++){
  sum += i;
}
cout << "sum of integers from 1 to " << n <<
     " is " << sum << endl;
```

A do-while loop:

```
int n;
do {
  bool gotValidInput = false;
  cout << "Enter a positive integer" << endl;
  cin >> n;
  gotValidInput = n > 0;
} while ( !gotValidInput )
```

Nested loops

Loops (as well as if-else structures, etc.) can be nested, i.e., you can put one inside another:

```
// loop over pixels in an image

for (int row=1; row<=nRows; row++){
  for (int column=1; column<=nColumns; column++){
    int b = imageBrightness(row, column);
    ...

  }     // loop over columns ends here
}       // loop over rows ends here
```

We can put any kind of loop into any other kind, e.g., **`while`** loops inside **`for`** loops, vice versa, etc.

More control of loops

`continue` causes a single iteration of loop to be skipped (jumps back to start of loop).

`break` causes exit from entire loop (only innermost one if inside nested loops).

```
while ( processEvent ) {
  if ( eventSize > maxSize ){
     continue;
  }
  if ( numEventsDone > maxEventsDone ){
     break;
  }
//  rest of statements in loop ...
}
```

Usually best to avoid **`continue`** or **`break`** by use of **`if`** statements.

Type casting

Often we need to interpret the value of a variable of one type as being of a different type, e.g., we may want to carry out floating-point division using variables of type **`int`**.

Suppose we have: **`int n, m; n = 5; m = 3;`** and we want to know the real-valued ratio of **`n/m`** (i.e. not truncated). We need to "type cast" **`n`** and **`m`** from **`int`** to **`double`** (or **`float`**):

```
double x = static_cast<double>(n) /
           static_cast<double>(m);
```

will give **`x = 1.666666...`**

Will also work here with **`static_cast<double>(n)/m;`** but **`static_cast<double>(n/m);`** gives **`1.0.`**

Similarly we can use **`static_cast<int>(x)`** to turn a **`float`** or **`double`** into an **`int`**, etc.

Standard mathematical functions

Simple mathematical functions are available through the standard C library **`cmath`** (previously **`math.h`**), including:

```
abs   acos  asin  atan  atan2 cos   cosh  exp
fabs  fmod  log   log10 pow   sin   sinh  sqrt
tan   tanh
```

Most of these can be used with **`float`** or **`double`** arguments; return value is then of same type.

To use these functions we need: **`#include <cmath>`**

Google for C++ cmath or see **`www.cplusplus.com`** for more info.

A simple example

Create file **testMath.cc** containing:

```
// Simple program to illustrate cmath library
#include <iostream>
#include <cmath>
using namespace std;
int main() {

  for (int i=1; i<=10; i++){
    double x = static_cast<double>(i);
    double y = sqrt(x);
    double z = pow(x, 1./3.);    // note decimal pts
    cout << x << "  " << y << "  " << z << endl;
  }
...
}
```

Note indentation and use of blank lines for clarity.

Running testMath

Compile and link: `g++ -o testMath testMath.cc`

Run the program: `./testMath`

```
1  1  1
2  1.41421  1.25992
3  1.73205  1.44225
4  2  1.5874
...
```

Often it is useful to save output directly to a file. Unix allows us to redirect the output:

```
./testMath > outputFile.txt
```

Similarly, use `>>` to append file, `>!` to insist on overwriting.

These tricks work with any Unix commands, e.g., `ls, grep`, ...

Scope basics

The scope of a variable is that region of the program in which it can be used.

If a block of code is enclosed in braces `{ }`, then this delimits the scope for variables declared inside the braces. This includes braces used for loops and if structures:

```
int x = 5;
for (int i=0; i<n; i++){
  int y = i + 3;
  x = x + y;
}
cout << "x = " << x << endl;  // OK
cout << "y = " << y << endl;  // BUG -- y out of scope
cout << "i = " << i << endl;  // BUG -- i out of scope
```

Variables declared outside any function, including `main`, have 'global scope'. They can be used anywhere in the program.

More scope

The meaning of a variable can be redefined in a limited ‘local scope’:

```
int x = 5;
{
  double x = 3.7;
  cout << "x = " << x << endl;// will print x = 3.7
}
cout << "x = " << x << endl;  // will print x = 5
```

(This is bad style; example is only to illustrate local scope.)

In general try to keep the scope of variables as local as possible. This minimizes the chance of clashes with other variables to which you might try to assign the same name.

Functions

Up to now we have seen the function **main**, as well as mathematical functions such as **sqrt** and **cos**. We can also define other functions, e.g.,

```
const double PI = 3.14159265;     // global constant
double ellipseArea(double, double);  // prototype
int main() {
  double a = 5;
  double b = 7;
  double area = ellipseArea(a, b);
  cout << "area = " << area << endl;
  return 0;
}

double ellipseArea(double a, double b){
  return PI*a*b;
}
```

The usefulness of functions

Now we can ‘call’ **`ellipseArea`** whenever we need the area of an ellipse; this is modular programming.

The user doesn’t need to know about the internal workings of the function, only that it returns the right result.

‘Procedural abstraction’ means that the implementation details of a function are hidden in its definition, and needn’t concern the user of the function.

A well written function can be re-used in other parts of the program and in other programs.

Functions allow large programs to be developed by teams.

Return type of a function

The prototype must also indicate the return type of the function,

e.g., **`int, float, double, char, bool`**.

```
double ellipseArea(double, double);
```

The function's return statement must return a value of this type.

```
double ellipseArea(double a, double b){
  return PI*a*b;
}
```

When calling the function, it must be used in the same manner as an expression of the corresponding return type, e.g.,

```
double volume = ellipseArea(a, b) * height;
```

Return type **`void`**

The return type may be ‘void’, in which case there is no return statement in the function:

```
void showProduct(double a, double b){
  cout << "a*b = " << a*b << endl;
}
```

To call a function with return type void, we simply write its name with any arguments followed by a semicolon:

```
showProduct(3, 7);
```

Recursion

A function is recursive (or has a recursive definition)
if the definition includes a call to itself.
A familiar mathematical example of a recursive function
is the factorial function "!".

0! = 1
for all n > 0, n! = n*(n-1)!

Thus, by repeatedly using the definition, we can work out that
6! = 6x5! = 6x5x4! = 6x5x4x3! = 6x5x4x3x2! = 6x5x4x3x2x1!
= 6x5x4x3x2x1x1 = 720

Again, notice that the definition of "!" includes both
a base case (the definition of 0!) and a recursive part

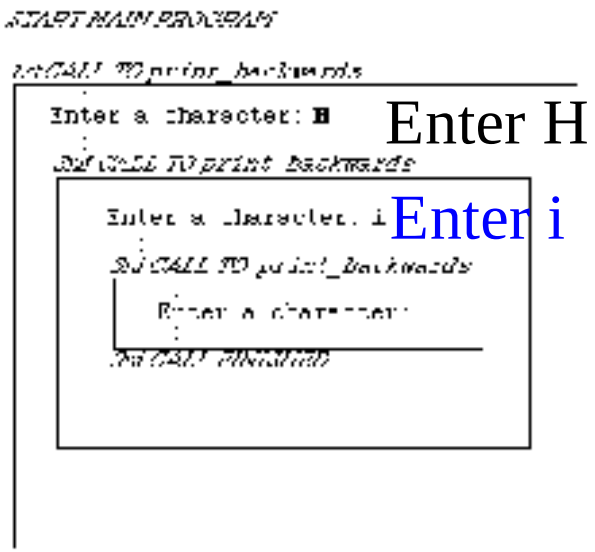
Example: print backwards

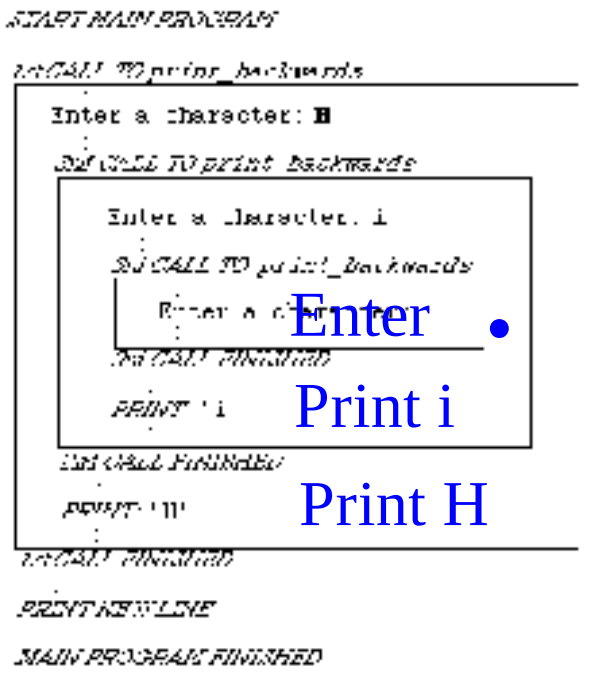
```
#include<iostream>
using namespace std;
void print_backwards();

int main()
{
        print_backwards();
        cout << "\n";
        return 0;
}

void print_backwards()
{
        char character;
        cout << "Enter a character ('.' to end program): ";
        cin >> character;
        if (character != '.')
        {
                print_backwards();
                cout << character;
        }
}
```

Mechanics of a Recursive Call

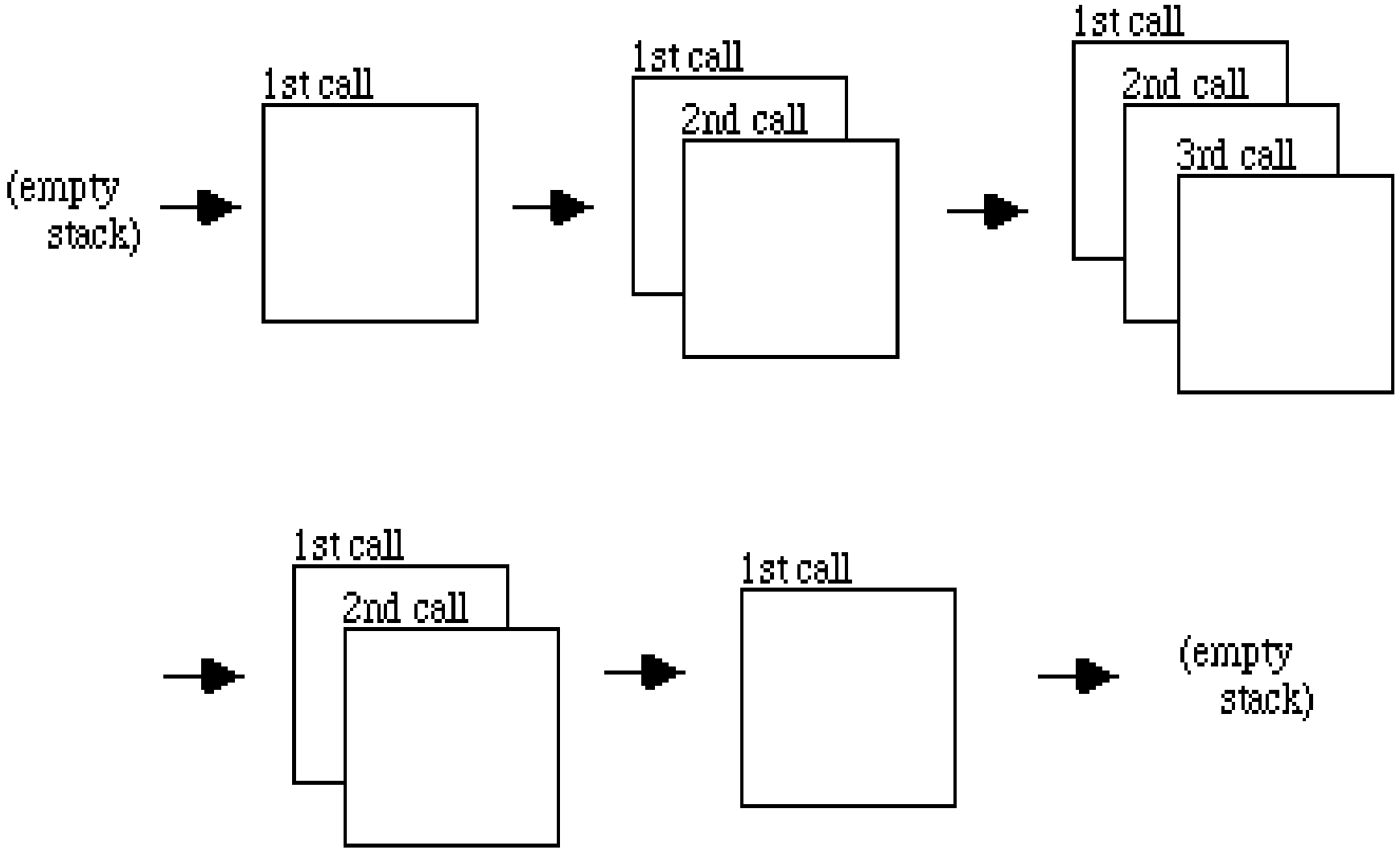
A typical input/output session is:

```
Enter a character ('.' to end program): H
Enter a character ('.' to end program): i
Enter a character ('.' to end program): .
iH
```





C++ Memory: Stack



Take away

We've seen some basic elements of a C++ program:

variables, e.g., **`int`**, **`double`**, **`bool`**, etc.;

how to assign values and form expressions;

how to control the flow of a program with **`if`**, **`else`**, and loops **`for`**, **`while`**, **`...`**

how to reinterpret e.g. a **`double`** as an **`int`** (type casting)

standard C library of mathematical functions (**`cmath`**).

declaring and defining our own functions, recursion