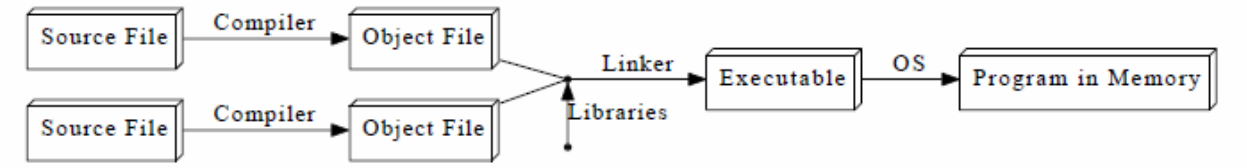
A Very Short tour of C++

High-level Programming languages
Closeness to human languages
Conciseness
Maintainability
Portability

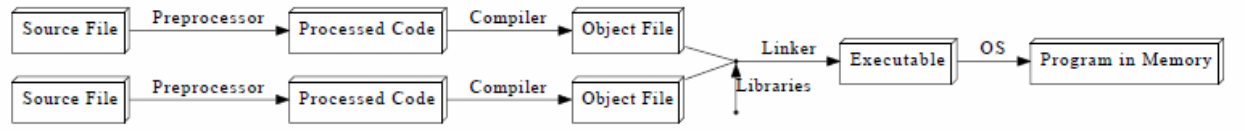
C++ is a *high-level* language: when you write a program in it, the short-hands are sufficiently expressive that you don't need to worry about the details of processor instructions. Just like its predecessor C, C++does give access to some lower-level functionality than other languages (e.g. memory addresses).

C++ is immensely popular, particularly for applications that require speed and/or access to some low-level features. It was created in 1979 by Bjarne Stroustrup, at first as a set of extensions to the C programming language. C++ extends C; our first few lectures will basically be on the C parts of the language.

Compilation and execution process



Object files are intermediate files that represent an incomplete copy of the program: eachsource file only expresses a piece of the program, so when it is compiled into an object file,the object file has some markers indicating which missing pieces it depends on. The linker takes those object files and the compiled libraries of predefined code that they rely on, fills in all the gaps, and spits out the final program, which can then be run by the operating system(OS).
The compiler and linker are just regular programs. The step in the compilation process in which the compiler reads the file is called parsing.
In C++, all these steps are performed ahead of time, before you start running a program. In some languages, they are done during the execution process, which takes time. This is one of the reasons C++ code runs far faster than code in many more recent languages.
C++ actually adds an extra step to the compilation process: the code is run through a preprocessor, which applies some modifications to the source code, before being fed to the compiler. Thus, the modified diagram is:



Please note that everything in C++ is case sensitive: `someName` is not the same as `SomeName`.

Let us begin with the ritualistic first code that writes “Hello, World!” on the terminal

```
// Hello, world
#include <iostream>

int main(void)
{
std::cout << “Hello, world!\n”;

return 0;
}
```

Tokens used
Tokens are the smallest components of a program that have meaning to the compiler – the smallest meaningful symbols in the language.

Keywords: `int, double, for, auto`
Identifiers: `cout, std, x, myFunction`
Literals: `“Hello, world!”, 24.3, 0, ‘c’`
Operators: `+, -, &&, %, <<`
Punctuations / separators: `{ } ( ) , ;`
Whitespace: `spaces, tabs, newlines, comments`

Preprocessor command
A line beginning with a #. #include tells the preprocessor to dump in the contents of another file, here the `iostream` file, which declares the functions for input / output.

Comments
Example

```
/* Old style comment goes till the next */
// new style comment till the end of line
```

Thus, the newline character ‘`\n`’ has a significance in the new style of comments

The old style of commenting is allowed /* `your text here` */
Multiple line comments might as well be done using the older method

A complete code
Example

```
// Hello, world
#include <iostream>

int main(void)
{
std::cout << “Hello, world!\n”;

return 0;
}
```

Must have a `main` function which is called by the OS when a command is executed
Arguments and return value
Each function contains `{}` and whatever is in between
Statement and expression
Non-standard suffixes in C++ (at times, depends on the compiler being used)

A code with simple maths

```
#include <iostream>
using namespace std;

int main(void)
{
int x;      // declaration

x = 4 + 2;  // initialization

cout << x/3 << ' ' << x*2;

return 0;
}
```

Note the `cout` statement that now has a sequence of values to be printed.
Declaration and initialization could have been combined.

A code with input and output

```
#include <iostream>
using namespace std;

int main(void)
{
int x;      // declaration

cin >> x;   // initialization

cout << x/3 << ' ' << x*2;

return 0;
}
```

We can also have multiple inputs using a single `cin`, just as `cout` is used to print a sequence of values.

Input and Output

Example

```
#include <iostream> // preprocessor command
using namespace std;
int main()
{
// Read and print three floating point numbers
float a, b, c;
cin >> a >> b >> c; // input
```

```
// output
cout << a << ", " << b << ", " << c << endl;
return 0;
}
```

Points to be noted:
`namespace`
`iostream`
`float` variables
`cin` reads in from `stdin`
`cout` prints out to `stdout`
`endl` is a special variable (same as `\n`); Unlike fortran

A detour on namespaces
`cout <<` : This is the syntax for outputting some piece of text to the screen.
Namespaces: In C++, identifiers can be defined within a context – sort of a directory of names – called a `namespace`. When we want to access an identifier defined in a namespace, we tell the compiler to look for it in that namespace using the *scope resolution operator* (`::`). Here, we're telling the compiler to look for `cout` in the `std` namespace, in which many standard C++ identifiers are defined.
A cleaner alternative is to add the following line below line 2:
`using namespace std;`
This line tells the compiler that it should look in the `std` namespace for any identifier we haven't defined. If we do this, we can omit the `std::` prefix when writing `cout`. This is the recommended practice.

Use of the math library
```
#include <iostream>
#include <cmath>

using namespace std;

int main(void)
{
float angle; // Angle, in degrees
cin >> angle;
cout << cos(angle * M_PI / 180.0 ) << endl;
// M_PI is from <cmath>
return 0;
}
```

Declaration and definition
A *declaration* tells compilers about the name and type of an object, function, class, or template, but it omits certain details. These are declarations:
```
extern int x;                           // object declaration
int numDigits(int number);              // function declaration
class Clock;                            // class declaration
template<class T>
class SmartPointer;                     // template declaration
```

A *definition*, on the other hand, provides compilers with the details. For an object, the definition is where compilers allocate memory for the object. For a function or a function template, the definition provides the code body. For a class or a class template, the definition lists the members of the class or template:

```
int x;                                  // object definition

int numDigits(int number)               // function definition
{                                       // (this function returns
  int digitsSoFar = 1;                  // the number of digits in
                                        // its parameter)
  if (number < 0) {
    number = -number;
    ++digitsSoFar;
  }
  while (number /= 10) ++digitsSoFar;
  return digitsSoFar;
}

class Clock {                            // class definition
public:
  Clock();
  ~Clock();
  int hour() const;
  int minute() const;
  int second() const;
  ...
};

template<class T>
class SmartPointer {                     // template definition
public:
  SmartPointer(T *p = 0);
  ~SmartPointer();
  T * operator->() const;
  T& operator*() const;
  ...
};
```

Declaration of types and initializing

Example

```
int i = 3;
float x = 10.0;
i = i + 1;
int j = i;
```

Type declaration is mandatory and must be done before first use

Usually the type declaration is done just before first use, so that chance of not initializing a variable is minimized and unnecessary variables are not declared

Variable names start with a letter or "_", and are case sensitive

Initialization can occur on the same line

Multiple declarations are allowed

```
int x, y; float i, j;
```

`limits.h` gives valid range of integer types, `float.h` gives valid range, precision
`bool` type is supported in C++
Characters
Escape sequences
`const` declaration; once initialized, cannot be changed

```
const float pi 3.14159
```

#define is obsolete, which is nothing more than string substitution

Expressions and operators
For relational operators, zero is false, non-zero is true
Remember difference between relational and assignment operators! Do not mix up = and ==
C has a condition operator:

```
y = (x < 0) ? -x : x; // y = abs(x)
```

Logical operators (Boolean operands)
Example

Bit-wise operators, including shifts
Table

Assignment operators
Many in C/C++

Scope resolution operator
Operator precedence
Left to right or right to left, as defined
Don't make clever use of precedence, rather use parentheses

Flow Control
If else if example

```
#include <iostream>
using namespace std;
int main(void)
{
int x, y;

cout << "Supply values of two integers to be compared: ";
cin >> x >> y;

if(x > y)
      cout << "x is greater than y" << endl;
else if(y > x)
      cout << "y is greater than x\n";
else
      cout << "x and y are equal" << endl;
return 0;
}
```

Possible pitfalls in if

```
if ( x < 0 )
```

```
      x = -x; // abs(x)
      y = -y; // always executed
```
Orphans

Assignment vs relational operators
Common styles of coding
switch case example

```
#include <iostream>
using namespace std;

int main(void)
{
int x = 6;

switch(x)
      {
      case 1:
            cout << "x is 1\n";
            break;
      case 2:
      case 3:
            cout << "x is 2 or 3";
            break;
      default:
            cout << "x is not 1, 2, or 3";
      }

return 0;
}
```

Goto
Example
Pit-falls of using goto

Loop statements
While
Example with terminal end-of-file (Ctrl-d) character (part 3, slide 9)
Loop can take place zero or more times
Do-while
Example (Newton's method, slide 2, lecture 4)
Loop can take place once or more times
For
Example (nested loops, two running indices, slide 4, part 4)
Loop can take place zero or more times

Break and continue statements
Examples (avoid using goto!)

Array
Collection of elements of the same type

Declaration
Initialization (automatic dimension)
Elements appear row-wise in C/C++ (while in Fortran, it is column-wise)
Example code (with row-wise and column-wise multiplications)

Functions
Must be declared, and declared correctly, before use
Name of the function
Return value
Arguments (names don't matter, but the order does)

```
#include <iostream>

using namespace std;

int raiseToPower(int base, int exponent)  // function declaration
{                                         // function definition begins
int result = 1;
for (int i = 0; i < exponent; i = i + 1)
      {
      result = result * base;
      }
return result;
}                                         // function definition ends

int main(void)
{
int threeExpFour = raiseToPower(3, 4);    // function invocation
cout << "3^4 is " << threeExpFour << endl;
return 0;
}
```

Use of header files (MIT notes)
Function prototypes are generally put into separate header files
– Separates specification (declaration) of the function from its implementation (definition)

```
// myLib.h - header file
// contains prototypes / declaration
int square(int);
int cube (int);

// myLib.cpp - definition / implementation file
#include "myLib.h"
int cube(int x)
{
return x*square(x);
}
int square(int x)
{
return x*x;
}
```

Body of the function
Extern functions

Static function (static keyword means local in scope of file, Paul implies it keeps a function local to a file)
Explanation of header files, possibility of multiple inclusion of header files: examples (slide 1, tutorial 6)
Scope of variables, external variables, local, auto, file scope, static data is explained correctly, good example on slide 9, part 5
Default function arguments: It is possible to specify the value of the arguments not given in the call of a function, strong typing advantages

```
#include <math.h>
extern double log_of(double x, double base = M_E);
// M_E in <math.h>
```

Can be used as follows:

```
x = log_of(y); // base e
z = log_of(y, 10); // base 10
```

Limitations of assumptions in C compilation: importance of –Wall
Use of libraries
Libraries are generally distributed as the header file containing the prototypes, and a binary .dll/.so file containing the (compiled) implementation
– Don't need to share your .cpp code

```
// myLib.h – header
// contains prototypes
double squareRoot(double num);
```

Library user only needs to know the function prototypes (in the header file), not the implementation source code (in the .cpp file)
– The Linker (part of the compiler) takes care of locating the implementation of functions in the .dll file at compile time

```
// myLib.h – header
// contains prototypes
double squareRoot(double num);

// libraryUser.cpp – some other guy's code
#include "myLib.h"
double fourthRoot(double num)
{
return squareRoot(squareRoot(num));
}
```

You don't actually need to implement `raiseToPower` and `squareRoot` yourself; `cmath` (part of the standard library) contains functions `pow` and `sqrt`

```
#include <cmath>
double fourthRoot(double num)
{
return sqrt(sqrt(num));
}
```

Recursion is allowed except for the main function

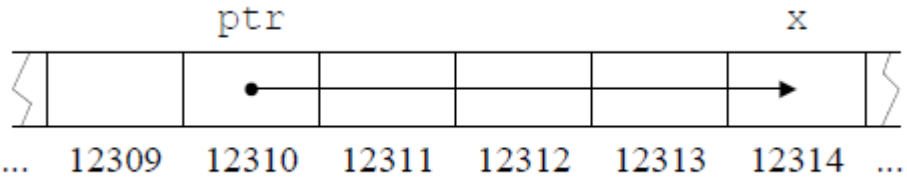
Pointer

Ideas and Explanations

When you declare a variable, the computer associates the variable name with a particular location in memory and stores a value there.

Pointers ar ejust variables storing integers – but those integers happen to be memory addresses, usually addresses of other variables. A pointer that stores the address of some variable `x` is said to point to `x`. We can access the value of `x` by dereferencing the pointer.

In the following figure, the dot-arrow notation indicates that `ptr` "pointsto" `x` – that is, the value stored in `ptr` is12314, `x`'s memory address.



Pointer is an object that refers to another object (or, stores the address of another variable; self pointing pointers?)

Simple example

```
#include <iostream>
using namespace std;

int main(void)
{
int* p;
int j = 4;

p = &j;
cout << *p << endl;

*p = 5;
cout << *p << " " << j << endl;

if (p != 0)
      {
      cout << "Pointer p points at " << *p << endl;
      }

return 0;
}
```

Type of the variable a pointer is pointing to is important since that allows correct dereferencing.

I prefer declaration of pointers with the idea that when I dereference this pointer gives a `float` or `int` value, etc. The pointer itself is of pointer data type.

Null pointer: Never use an uninitialized pointer. Dereferencing a null pointer causes a core dump.

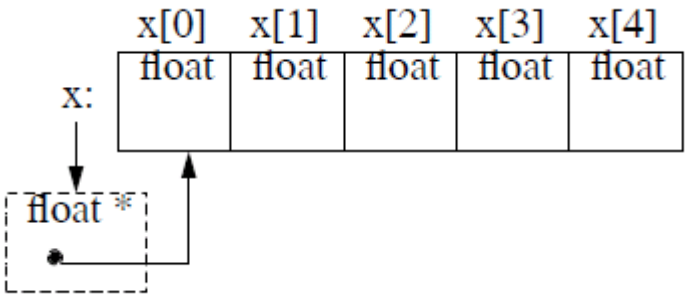
Arrays and pointers

Name of an array is the address of the first array element.

Consider

```
float x[5];
```

In computer memory the variables are placed as shown below:



In C/C++, `x` is a pointer to the first element

`*x` and `x[0]` are the same

`x` and `&x[0]` are the same

Elements of an array can be accessed either way, i.e., by using the array operator, or pointer operation.

`x` is a label to an array of object, not a pointer Object

Intro to pointer arithmetic, slide 4, lecture 6

```
float x[5];
float *y = &x[0];
```

`y` is a pointer to `x[0]`

`y+1` is pointer to `x[1]`

Thus `*(y+1)` and `x[1]` access the same object

`y[1]` is shorthand for `*(y+1)`

Integer add, subtract and relational operators are allowed on pointers

Summing an array C++ style

```
float x[5];
// type in your code to fill x

float *y = x;
double sum = 0.0;
for (int i = 0; i < 5; i++)
      {
      sum += *y++;
      }
```

It takes some time to get use to writing in this style
Be prepared to read code written by others in this style
We are not worrying about performance issues, as yet.

Dynamic allocation of arrays: new and delete in C++, malloc and free in C

Examples

```
float* x = new float[n];
```

`new` is an operator that returns a pointer to the newly created array
Note use of `n`; a variable
In C, one does

```
float *x = (float *)malloc( n*sizeof(float) );
```

In C++, to delete a dynamically allocated array one uses the `delete` operator

```
delete [] x;
```

in C one uses the `free()` function

```
free(x);
```

Line fit example

```
#include <iostream>
using namespace std;

void linefit(void)
{
// Create arrays with the desired number of elements
int n;
cin >> n;

float* x = new float[n];
float* y = new float[n];

// Read the data points
for (int i = 0; i < n; i++)
      {
      cin >> x[i] >> y[i];
      }

// Accumulate sums Sx and Sy in double precision
double sx = 0.0;
double sy = 0.0;
for (int i = 0; i < n; i++)
      {
      sx += x[i];
      sy += y[i];
      }
// Compute coefficients
double sx_over_n = sx / n;
double stt = 0.0;
double b = 0.0;
for (i = 0; i < n; i++)
      {
      double ti = x[i] - sx_over_n;
      stt += ti * ti;
      b += ti * y[i];
      }

b /= stt;
double a = (sy - sx * b) / n;
delete [] x;
delete [] y;
cout << a << " " << b << endl;
}

int main(void)
```

```
{
linefit();
return 0;
}
```

Discussion on memory leak: Remember to free the dynamically allocated memory at the correct place. If not done correctly, this can either lead to memory leak, or can lead to attempts to access memory areas that have already been freed.

What happens if we forget to put the [] in using delete? – it deletes only the first objet.

Character arrays

Why that '\0' at the end; example of walking through arrays

Here is an example to illustrate the cctype library:

```
#include <iostream>
#include <cctype>
using namespace std;

int main(void)
{
char messyString[] = "t6H0I9s6.iS.999a9.STRING";
char current = messyString[0];
for(int i = 0; current != '\0'; current = messyString[++i])
      {
      if(isalpha(current))
            cout << (char)(isupper(current) ? tolower(current) : current);
      else if(ispunct(current))
            cout << ' ';
      }

cout << endl;
return 0;
}
```

This example uses the `isalpha`, `isupper`, `ispunct`, and `tolower` functions from the cctype library. The `is`-functions check whether a given character is an alphabetic character, an uppercase letter, or a punctuation character, respectively. These functions return a Boolean value of either `true` or `false`. The `tolower` function converts a given character to lowercase.

The for loop takes each successive character from `messyString` until it reaches the null character. On each iteration, if the current character is alphabetic and uppercase, it is converted to lowercase and then displayed. If it is already lowercase it is simply displayed. If the character is a punctuation mark, a space is displayed. All other characters are ignored. The resulting output is `this is a string`.

The notable character / string manipulation functions are available through the use of:

`cctype (ctype.h)`: character handling

`cstdio (stdio.h)`: input/output operations

`cstdlib (stdlib.h)`: general utilities

`cstring (string.h)`: string manipulation

What is `(char)`?

Old and new ways of type casting:

C++ is a strong-typed language. Many conversions, specially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion: functional and c-like casting:

```
short a=2000;
int b;
b = (int) a;    // c-like cast notation
b = int (a);    // functional notation
```

The functionality of these explicit conversion operators is enough for most needs with fundamental data types. However, these operators can be applied indiscriminately on classes and pointers to classes, which can lead to code that while being syntactically correct can cause runtime errors. For example, the following code is syntactically correct:

```
// class type-casting
#include <iostream>
using namespace std;

class CDummy {
    float i,j;
};

class CAddition {
      int x,y;
  public:
      CAddition (int a, int b) { x=a;
y=b; }
      int result() { return x+y;}
};

int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

The program declares a pointer to `CAddition`, but then it assigns to it a reference to an object of another incompatible type using explicit type-casting:

```
padd = (CAddition*) &d;
```

Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to. The subsequent call to member `result` will produce either a run-time error or a unexpected result.

Opt for casting forms `static_cast`, `const_cast`, `dynamic_cast`, and `reinterpret_cast`. Conventional C-style casts look like this: ¤

```
(type) expression                          // cast expression to be of
                                           // type type
```

The new casts look like this: ¤

```
static_cast<type>(expression)              // cast expression to be of
                                           // type type
const_cast<type>(expression)
dynamic_cast<type>(expression)      <type>
reinterpret_cast<type>(expression)  <type>
```

These different casting forms serve different purposes: ¤

- `const_cast` is designed to cast away the constness of objects and pointers. This type of casting manipulates the constness of an object, either to be set or to be removed. For example, in order to pass a const argument to a function that expects a non-constant parameter
- `dynamic_cast` is used to perform "safe downcasting," The dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class. Therefore, dynamic_cast is always successful when we cast a class to one of its base classes.
- `reinterpret_cast` is engineered for casts that yield implementation-dependent results, e.g., casting between function pointer types. Itconverts any pointer type to any other pointer type, even of unrelated classes. The operation result is a simple binary copy of the value from one pointer to the other. All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked.
- `static_cast` is sort of the catch-all cast. It's what you use when none of the other casts is appropriate. It's the closest in meaning to the conventional C-style casts. It can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived. This ensures that at least the classes are compatible if the proper object is converted, but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type. Therefore, it is up to the programmer to ensure that the conversion is safe. On the other side, the overhead of the type-safety checks of `dynamic_cast` is avoided.

Conventional casts continue to be legal, but the new casting forms are preferable. They're much easier to identify in code (both for humans and for tools like `grep`), and the more narrowly specified purpose of each casting form makes it possible for compilers to diagnose usage errors. For example, only `const_cast` can be used to cast away the constness of something. If you try to cast away an object's or a pointer's constness using one of the other new casts, your cast expression won't compile.

Strings in C++

String class

String objects are a special type of container, specifically designed to operate with sequences of characters.

Unlike traditional c-strings, which are mere sequences of characters in a memory array, C++ string objects belong to a class with many built-in features to operate with strings in a more intuitive way and with some additional useful features common to C++ containers.
The `string` class is an instantiation of the `basic_string` class template, defined in `<string>` as:

```
typedef basic_string<char> string;
```

Member functions

(constructor)	Construct string object (constructor member)
operator=	String assignment (public member function)

Iterators:

begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)

Capacity:

size	Return length of string (public member function)
length	Return length of string (public member function)
max_size	Return maximum size of string (public member function)
resize	Resize string (public member function)
capacity	Return size of allocated storage (public member function)
reserve	Request a change in capacity (public member function)
clear	Clear string (public member function)
empty	Test if string is empty (public member function)

Element access:

operator[]	Get character in string (public member function)
at	Get character in string (public member function)

Modifiers:

operator+=	Append to string (public member function)
append	Append to string (public member function)
push_back	Append character to string (public member function)
assign	Assign content to string (public member function)
insert	Insert into string (public member function)
erase	Erase characters from string (public member function)
replace	Replace part of string (public member function)
swap	Swap contents with another string (public member function)

String operations:

c_str	Get C string equivalent (public member function)

data	Get string data (public member function)
get_allocator	Get allocator (public member function)
copy	Copy sequence of characters from string (public member function)
find	Find content in string (public member function)
rfind	Find last occurrence of content in string (public member function)
find_first_of	Find character in string (public member function)
find_last_of	Find character in string from the end (public member function)
find_first_not_of	Find absence of character in string
find_last_not_of	Find absence of character in string from the end (public member function)
substr	Generate substring (public member function)
compare	Compare strings (public member function)

Example:

```
#include <iostream>
#include <string>
using namespace std;

int main ()
{
    char *line = "short line for testing";

    // with no arguments
    string s1;
    s1 = "Anatoliy";
    cout << "s1  is: " << s1 << endl;

    // copy constructor
    string s2 (s1);
    cout << "s2  is: " << s2 << endl;

    // one argumen
    string s3 (line);
    cout << "s3  is: " << s3 << endl;

    // first argumen C string
    // second number of characters
    string s4 (line,10);
    cout << "s4  is: " << s4 << endl;

    // 1 - C++ string
    // 2 - start position
    // 3 - number of characters
    string s5 (s3,6,4); // copy word 'line' from s3
    cout << "s5  is: " << s5 << endl;

    // 1 - number characters
    // 2 - character itself
    string s6 (15,'*');
    cout << "s6  is: " << s6 << endl;

    // 1 - start iterator
```

```
    // 2 - end iterator
    string s7 (s3.begin(),s3.end()-5);
    cout << "s7  is: " << s7 << endl;

    // you can instantiate string with assignment
    string s8 = "Anatoliy";
    cout << "s8  is: " << s8 << endl;

    return 0;
}
```

OUTPUT:

```
// s1  is: Anatoliy
// s2  is: Anatoliy
// s3  is: short line for testing
// s4  is: short line
// s5  is: line
// s6  is: ***************
// s7  is: short line for te
// s8  is: Anatoliy
```

Variable scope, initialization and lifetime

Every pair of { } defines a new scope

Even a pair with `if`, `for`, *etc* defines a new scope.

Variables declared in a scope are deleted when execution leaves scope

```
int main(void)
{
float temp = 1.1;
int a;
int b;

cout << "Supply two integers: ";
cin >> a >> b;

if (a < b)
      {
      int temp = a; // This "temp" hides other one
      cout << 2 * temp << endl;
      }// Block ends; local "temp" deleted.
else
      {
      int temp = b; // Another "temp" hides other one
      cout << 3 * temp << endl;
      }

{     // my block
int a;
cout << "Supply value of a in my block: ";
cin >> a;
cout << "a in my block: " << a << endl;
}

cout << "a outside my block: " << a << endl;
cout << a * b + temp << endl;
```

```
return 0;
}
```

Passing arguments by value

Stack and heap:
Stack frames for different functions

Address, pointer, reference
References are the most confusing issue in C++
Ampersand here is not the address
When we write `void f(int &x) {...}` and call `f(y)`, the reference variable `x` becomes another name – an alias – for the value of `y` in memory. We can declare a reference variable locally, as well:

```
int y;
int &x = y; // Makes x a reference to, or alias of, y
```

Consider the following example:

```
float x = 12.1;
float& a = x;
float &b = x;
```

`a`, `b`, and `x` all represent the same object

Don't confuse a reference and a pointer

```
int i = 3; // data object
int &j = i; // reference to i
int *p = &i; // pointer to i
```

Example of the use of reference:

```
void swap(int &a, int &b)
{
int t = a;

a = b;
b = t;
}

int main(void)
{
int q = 3;
int r = 5;

swap(q, r);

cout << "q " << q << endl; // q 5
cout << "r " << r << endl; // r 3
}
```

C does not have reference; instead you have to write

```
extern void swap(int *a, int *b);
swap(&q, &r);
```

Returning Multiple values

The return statement only allows you to return 1 value. Passing output variables by reference overcomes this limitation.

```
int divide(int numerator, int denominator, int &remainder)
{
remainder = numerator % denominator;
return numerator / denominator;
}

int main(void)
{
int num = 14;
int den = 4;
int rem;

int result = divide(num, den, rem);

cout << result << "*" << den << "+" << rem << "=" << num << endl;
// 3*4+2=12
}
```

Better examples will come with classes

Recursion

A function can call itself. Write two functions- one to compute the Fibonacci sequence and the other two compute factorials. Can the `main` function call itself?

Constant pointer

```
const float pi = 3.1415;
float pdq = 1.2345;
const float *p = &pi;
float* const d = &pi; // WRONG
float* const q = &pdq;
const float *const r = &pi;

*p = 3.0; // WRONG
p = &pdq; // OK
*p = 3.0; // still WRONG

*q = 3.0; // OK
q = &pdq; // WRONG

*r = 3.0; // WRONG
r = &pdq; // WRONG AGAIN
```

`const` qualifier can refer to what is pointed at (frequent usage)
`const` qualifier can refer to pointer itself (rare usage)
`const` qualifier can refer to both (infrequent usage)

Usefulness?

Now, we have a way of maintaining the original array elements despite passing the array to the function – very useful! Implementation of client / server applications becomes lot easier.

```
void f(int i, float x, const float *a) {
i = 100;
x = 101.0*a[0]; // OK
a[0] = 0.0; // WRONG!
}
int j = 1;
int k = 2;
float y[] = {3.0, 4.0, 5.0};
f(j, k, y);
```

A `const` argument tells user of function that his data wouldn't be changed
The `const` is enforced when attempting to compile function.
First aspect of spirit of client/server interface

Function Name Overloading
Allowed for both for library and user functions

```
void printOnNewLine(int x)
{
cout << "Integer: " << x << endl;
}
void printOnNewLine(char *x)
{
cout << "String: " << x << endl;
}
void printOnNewLine(int x, int y)
{
cout << "2 Integers: " << x << " and " << y << endl;
}
```

• Many functions with the same name, but different arguments
• The function called is the one whose arguments match the invocation

So, in C++ one can have

```
int sqr(int i);
float sqr(float x);
double sqr(double x);
```

Separate functions with same name
Functions distinguished by their name, and the number and type of arguments
Name mangling occurs to create the external symbol seen by the linker

`nm` command gives all symbol name of an object code or a library