

C++ Programming

July 16th, 2019

Simon Scheidegger

Roadmap of today

1. Basics in C++
2. Preprocessing/Compiling/Linking

1. Basics in C++



A first C++ Program

See [demo_0/hello.cpp](#)

- `/*` and `*/` are the delimiters for comments
- `includes` declarations of I/O streams
- `declares` that we want to use the standard library (`std`)
- the main program is always called `main`
- `cout` is the standard output stream.
- `<<` is the operator to write to a stream
- statements end with a `;`
- `//` starts one-line comments
- A **return value of 0** means that everything went OK

```
/* A first program */
#include <iostream>

using namespace std;
int main()
{
    cout << "Hello students!\n";

    // std::cout without the using
    // declaration
    return 0;
}
```

How to compile and execute a program

Compile = **translate** a program from a programming language to machine language.

CPP Compilers from different vendors (some non-free):

g++, **icpc**, ...

How do we compile e.g. hello.cpp

```
compiler [options] program-name [-o executable-name]
```

concretely:

```
g++ hello.cpp -o helloworld.x
```

The compiled program can then be executed by:

```
./helloworld.x
```

How a Computer Engineering Student feels



When the PROGRAM Compiles

More about the `std` namespace

```
#include <iostream>
using std::cout;
int main()
{
    cout << "Hello\n";
}
```

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello\n";
}
```

- All these versions are equivalent
- Feel free to use any reasonable style in your program
- Never use `using` statements globally in libraries!

```
#include <iostream>
int main()
{
    std::cout << "Hello\n";
}
```

A first calculation

See [demo_0/calc.cpp](#)

- `<cmath>` is the header for mathematical functions
- Output can be connected by `<<`
- Expressions can be used in output statements
- What are these constants?
 - 5.
 - 0
 - `"\n"`

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

int main()
{
    cout << "The square root of 5 is"
    << sqrt(5.) << "\n";
    return 0;
}
```


Integral data types

- ♦ Signed data types
 - ♦ `short`, `int`, `long`, `long long`
- ♦ Unsigned data types
 - ♦ `unsigned short`, `unsigned int`,
 - ♦ `unsigned long`, `unsigned long long`
- ♦ Are stored as binary numbers
 - ♦ `short`: usually 16 bit
 - ♦ `int`: usually 32 bit
 - ♦ `long`: usually 32 bit on 32-bit CPUs and 64 bit on 64-bit CPUs
 - ♦ `long long`: usually 64 bits

Unsigned

- The `unsigned` keyword is a data type specifier, that makes a variable only represent **positive numbers and zero**.
- It can be applied only to the **char, short, int and long types**.
- For example, if an **int** typically holds values from -32768 to 32767, an `unsigned int` will hold values from 0 to 65535.
- **You can use this specifier** when you know that your variable will **never need to be negative**.
- For example, if you declared a variable '`myHeight`' to hold your height, you could make it unsigned because you know that you would never be negative inches tall.

Characters

- ♦ Character types
 - ♦ Single byte: **char**, unsigned char, signed char
 - ♦ Uses ASCII standard
 - ♦ Multi-byte (e.g. for Japanese:): wchar_t
 - ♦ Unfortunately is not required to use Unicode standard
- ♦ Character literals
 - ♦ 'a', 'b', 'c', '1', '2', ...
 - ♦ '\t' ... tabulator
 - ♦ '\n' ... new line
 - ♦ '\r' ... line feed
 - ♦ '\0' ... byte value 0

Strings

- ♦ **String type**

- ♦ C-style character arrays `char s[100]` should be avoided
- ♦ C++ class `std::string` for single-byte character strings
- ♦ C++ class `std::wstring` for multi-byte character strings

- ♦ **String literals**

- ♦ `"Hello"`
- ♦ Contain a trailing `'\0'`, thus `sizeof("Hello")==6`

Example: `char myword[] = { 'H', 'e', 'l', 'l', 'o', '\0' };`

Boolean (logical) type

- ♦ Type
 - ♦ `Bool`
- ♦ Literal
 - ♦ `true`
 - ♦ `false`

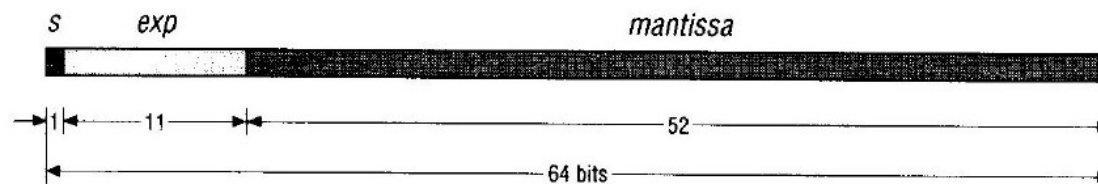
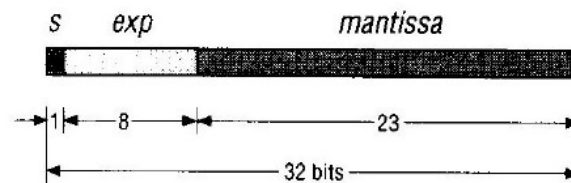
Floating point numbers

- ♦ Floating point types
 - ♦ single precision: `float`
 - ♦ usually 32 bit
 - ♦ double precision: `double`
 - ♦ Usually 64 bit
 - ♦ extended precision: `long double`
 - ♦ Often 64 bit (PowePC), 80 bit (Pentium) or 128 bit (Cray)
- ♦ Literals
 - ♦ single precision: `4.562f, 3.0F`
 - ♦ double precision: `3.1415927, 0.`
 - ♦ extended precision: `6.54498467494849849489L`

Recall: IEEE floating point representation

- The 32 (64) bits are divided into sign, exponent and mantissa

Single Precision



Double Precision

Type	Exponent	Mantissa	Smallest	Largest	Base 10 accuracy
float	8	23	1.2E-38	3.4E+38	6-9
double	11	52	2.2E-308	1.8E+308	15-17

Floating point arithmetic

- Truncation can happen because of finite precision

$$\begin{array}{r} 1.00000 \\ 0.0000123 \\ \hline 1.00001 \end{array}$$

- Machine precision **e** is smallest number such that $1 + \mathbf{e} \neq 1$
- Exercise: calculate **e** for `float`, `double` and `long double` on your machine
- Be very careful about roundoff
- For example: sum numbers starting from smallest to largest

Implementation-specific properties of numeric types

- defined in header `<limits>`
- `numeric_limits<T>::is_specialized` // is true if information available
- most important values for integral types
 - `numeric_limits<T>::min()` // minimum (largest negative)
 - `numeric_limits<T>::max()` // maximum
 - `numeric_limits<T>::digits` // number of bits (digits base 2)
 - `numeric_limits<T>::digits10` // number of decimal digits
 - and more: `is_signed`, `is_integer`, `is_exact`, ...
- most important values for floating point types
 - `numeric_limits<T>::min()` // minimum (smallest nonzero positive)
 - `numeric_limits<T>::max()` // maximum
 - `numeric_limits<T>::digits` // number of bits (digits base 2)
 - `numeric_limits<T>::digits10` // number of decimal digits
 - `numeric_limits<T>::epsilon()` // the floating point precision
 - and more: `min_exponent`, `max_exponent`, `min_exponent10`, `max_exponent10`, `is_integer`, `is_exact`
- first example of templates, use by replacing `T` above by the desired type:
`std::numeric_limits<double>::epsilon()`

A more useful program

See demo_0/in_out.cpp

- a variable named 'x' of type 'double' is declared
- a double value is read and assigned to x
- The square root is printed

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    cout << "Enter a number:\n";
    double x;
    cin >> x;
    cout << "The square root of "
    << x << " is "
    << sqrt(x) << "\n";
    return 0;
}
```

Action required – Exercise 1

Exercise 2: Write a program that reads arbitrary values a, b, c and prints the solution to the quadratic Equation.

Take care of imaginary solutions

$$ax^2 + bx + c = 0$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Variable declarations

- ♦ Var. declarations have the syntax: **type variablelist;**
 - ♦ `double x;`
 - ♦ `int i, j, k; // multiple variables possible`
 - ♦ `bool flag;`
- ♦ can appear **anywhere** in the program

```
int main() {  
    ...  
    double x;  
}
```
- ♦ can have **initializers**, can be **constants**
 - ♦ `int i=0; // C-style initializer`
 - ♦ `double r(2.5); // C++-style constructor`
 - ♦ `const double pi=3.1415927;`

Advanced types

- **Enumerators** are integer which take values only from a certain set
 - `enum trafficlight {red, orange, green};`
 - `enum occupation {empty=0, up=1, down=2, updown=3};`
 - `trafficlight light=green;`
- **Arrays** of size n
 - `int i[10]; double vec[100]; float matrix[10][10];`
 - indices run from 0 ... n-1! (FORTRAN: 1...n)
 - last index changes fastest (opposite to FORTRAN)
 - Should not be used in C++ anymore!!!
- Complex types can be given a new name
 - `typedef double[10] vector10;`
 - `vector10 v={0,1,4,9,16,25,36,49,64,81};`
 - `vector10 mat[10]; // actually a matrix!`

Example – Enumerators

See [demo_0/enum_example.cpp](#)

```
//example program to demonstrate working
// of enum in C
#include <iostream>
using namespace std;

enum year{Jan, Feb, Mar, Apr, May, Jun, Jul,
          Aug, Sep, Oct, Nov, Dec};

int main()
{
    int i;
    for (i=Jan; i<=Dec; i++)
        cout << i << endl;

    return 0;
}
```

More on typedef

See [demo_0/typedef_example.cpp](#)

Every variable has a data type. `typedef` is used to define new data type names to make a program more readable to the programmer.

For example:

```
main()
{
    int money;
    Money = 2;
}
```

```
| main()
| {
|     typedef int Francs;
|     Francs ch_money = 2
| }
```

Expressions and operators

- **Arithmetic**
 - multiplication: $a * b$
 - division: a / b
 - remainder: $a \% b$
 - addition: $a + b$
 - subtraction: $a - b$
 - negation: $-a$
- **Increment and decrement**
 - pre-increment: $++a$
 - post-increment: $a++$
 - pre-decrement: $--a$
 - post-decrement: $a--$
- **Logical (result bool)**
 - logical not: $!a$
 - less than: $a < b$
 - Less than or equal: $a \leq b$
 - greater than: $a > b$
 - greater than or equal: $a \geq b$
 - equality: $a == b$
 - inequality: $a != b$
 - logical and: $a \&\& b$
 - logical or: $a || b$
- **Conditional: $a ? b : c$**
- **Assignment: $a = b$**

Bitwise operations

- **Bit operations**
 - bitwise not: `~a`
 - inverts all bits
 - **left shift**: `a << n`
 - shifts all bits to higher positions, fills with zeros, discards highest
 - **right shift**: `a >> n`
 - shifts to lower positions
 - bitwise and: `a & b`
 - bitwise xor: `a ^ b`
 - bitwise or: `a | b`
- **The `bitset`** class implements more functions. We will use it later in one of the exercises.
- Interested students should refer to the recommended C++ books
- The **shift operators** have been redefined for I/O streams:
 - `cin >> x;`
 - `cout << "Hello\n";`
- The same can be done for all new types: "**operator overloading**"
- Example: **matrix operations**
 - `A+B`
 - `A-B`
 - `A*B`

Compound assignments

- $a *= b$
- $a /= b$
- $a \%= b$
- $a += b$
- $a -= b$
- $a <<= b$
- $a >>= b$
- $a \&= b$
- $a \wedge= b$
- $a |= b$
- $a += b$ equivalent to $a=a+b$
- allow for simpler codes and better optimizations

Operator precedences

- Are listed in detail in all reference books or look at

http://www.cppreference.com/operator_precedence.html

- Arithmetic operators follow usual rules
 - $a+b*c$ is the same as $a+(b*c)$
- Otherwise, **when in doubt use parentheses**

Statements

- ♦ **simple statements**
 - ♦ `;` // null statement
 - ♦ `int x;` // declaration statement
 - ♦ `typedef int index_type;` // type definition
 - ♦ `cout << "Hello world";` // all simple statements end with ;
- ♦ **compound statements**
 - ♦ more than one statement, enclosed in curly braces

```
{  
    int x;  
    cin >> x;  
    cout << x*x;  
}
```

The *if* statement

- Has the form

```
if (condition)
    Statement
```

- Or

```
if (condition)
    statement
else
    statement
```

- can be chained

```
if (condition)
    statement
else
    if(condition)
        statement
    else
        statement
```

- Example:

```
if (light == red)
    cout << "STOP!";
else if (light == orange)
    cout << "Attention";
else {
    cout << "Go!";
}
```

The switch statement

→ can be used instead of deeply nested if statements:

```
switch (light) {
    case red:
        cout << "STOP!";
        break;
    case orange:
        cout << "Attention";
        break;
    case green:
        cout << "Go!";
        go();
        break;
    default:
        cerr << "illegal color";
        abort();
}
```

- **do not forget the break!**
- **always include a default!**
 - the telephone system of the US east coast was once disrupted completely for several hours because of a missing default!
- also multiple labels possible

```
switch(ch) {
    case 'a':
    case 'e':
    case 'i':
    case 'o':
    case 'u':
        cout << "vowel";
        break;

    default:
        cout << "other character";
}
```

The for loop statement

- ♦ has the form

```
for (init-statement ; condition ; expression)  
    statement
```

- ♦ example:

```
for (int i=0; i<10; ++i)  
    cout << i << "\n";
```

- ♦ can contain **more than one statement in for(;;)**, but this is very bad style!

```
double f;  
int k;  
for (k=1, f=1 ; k<50 ; ++k, f*=k)  
    cout << k << "! = " << f << "\n";
```

The while statement

- ♦ is a simpler form of a loop:

```
while (condition)
    statement
```

- ♦ example:

```
while (trafficlight()==red) {
    cout << "Still waiting\n";
    sleep(1);
}
```


The do-while statement

- ♦ Is similar to the while statement

```
do  
    statement  
while (condition);
```

- ♦ Example

```
do {  
    cout << "Working\n";  
    work();  
} while (work_to_do());
```

The break and continue statements

- **break** ends the loop immediately and jumps to the next statement following the loop
- **continue** starts the next iteration immediately
- An example:

```
while (true) {  
    if (light() == red)  
        continue;  
    start_engine();  
    if (light() == orange)  
        continue;  
    drive_off();  
    break;  
}
```

A loop example: what is wrong?

demo_0/loop_example.cpp

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Enter a number: ";
    unsigned int n;
    cin >> n;
    for (int i=1; i<=n; ++i)
        cout << i << "\n";

    int i=0;
    while (i<n)
        cout << ++i << "\n";
```

```
i=1;
do
    cout << i++ << "\n";
while (i<=n);

i=1;
while (true) {
    if(i>n)
        break;
    cout << i++ << "\n";
}
}
```

Pointers, Arrays, and References

- An array of elements of type **char** can be declared like this:

```
char v[6];           // array of 6 characters
```

- Similarly, a pointer can be declared like this:

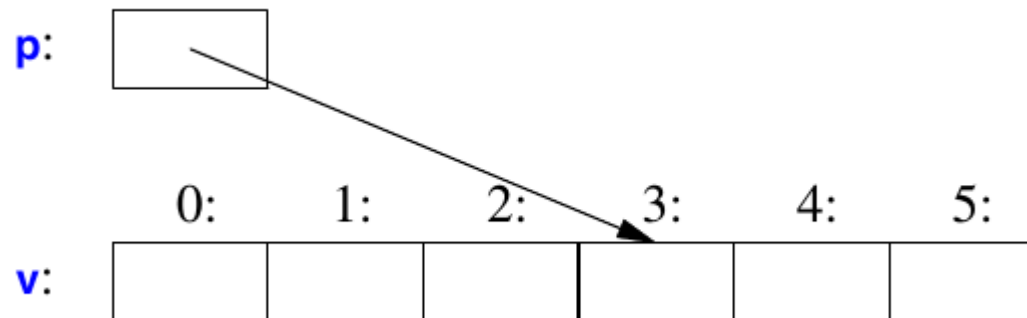
```
char* p;             // pointer to character
```

- In declarations, **[]** means “array of” and ***** means “pointer to.”
- All arrays have 0 as their lower bound, so v has six elements, v[0] to v[5].
- A pointer variable can hold the address of an object of the appropriate type:

```
char* p = &v[3];      // p points to v's fourth element  
char x = *p;          // *p is the object that p points to
```

Pointers, Arrays, and References

- In an expression, prefix unary `*` means “contents of” and
- The prefix unary `&` means “address of.”
- We can represent the result of that initialized definition graphically:



```
char* p = &v[3];  
char x = *p;
```

```
// p points to v's fourth element  
// *p is the object that p points to
```

Static memory allocation

- Declared variables are assigned to memory locations

```
int x=3;  
int y=0;
```

- The variable name is a **symbolic reference** to the contents of some real memory location
- It only exists for the compiler
- No real existence in the computer

address	contents	name
0	3	x
4	0	
8		y
12		
16		
20		
24		
28		

Pointers

- **Pointers store the address of a memory location**
 - are denoted by a ***** in front of the name
`int *p; // pointer to an integer`
 - **Are initialized using the & operator**
`int i=3;`
`p = &i; // & takes the address of a variable`
 - Are dereferenced with the * operator
`*p = 1; // sets i=1`
 - Can be dangerous to use
`p = 1; // sets p=1: danger!`
`*p = 258; // now messes up everything, can crash`
- Take care: `int *p;` does not allocate memory!

address	contents	name
0	258	p
4	3	
8		i
12		
16		
20		
24		
28		

Dynamic allocation

- Automatic allocation
 - `float x[10];` // allocates memory for 10 numbers
- Allocation of flexible size
 - `unsigned int n; cin >> n; float x[n];` // will not work
 - The compiler has to know the number!
- Solution: dynamic allocation
 - `float *x=new float[n];` // allocate some memory for an array
 - `x[0]=...;...` // do some work with the array x
 - `delete[] x;` // delete the memory for the array. x[i], *x now undefined!
- Don't confuse
 - `delete`, used for simple variables
 - `delete[]`, used for arrays

Example – dynamic memory alloc

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

int main ()
{
    int i,n;
    int * p;
    cout << "How many numbers would you like to type? ";
    cin >> i;
    p= new (nothrow) int[i];
    if (p == nullptr)
        cout << "Error: memory could not be allocated";
    else
    {
        for (n=0; n<i; n++)
        {
            cout << "Enter number: ";
            cin >> p[n];
        }
        cout << "You have entered: ";
        for (n=0; n<i; n++)
            cout << p[n] << ", ";
        delete[] p;
    }
    return 0;
}
```

How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,

References

- are aliases for other variables:

```
float very_long_variabe_name_for_number=0;
```

```
float& x=very_long_variabe_name_for_number;  
// x refers to the same memory location
```

```
x=5; // sets  
very_long_variabe_name_for_number to 5;
```

```
float y=2;
```


```
x=y; // sets  
very_long_variabe_name_for_number to 2;  
// does not set x to refer to y!
```

Function calls

demo_0/function_call.cpp

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

```
float square(float x) {
    return x*x;
}
```

- a function “square” is defined
 - return value is **float**
 - parameter x is **float**
- 

```
int main() {
    cout << "Enter a number:\n";
    float x;
    cin >> x;
    cout << x << " " <<
    square(x) << "\n";
    return 0;
}
```

- and used in the program
- 

Function call syntax

- Syntax:

```
returntype functionname  
    (parameters )  
  
    {  
        functionbody  
    }
```

- returntype is **“void”** if there is **no return value**:

```
void error(char[] msg) {  
    cerr << msg << “\n”;  
}
```

- There are several kinds of parameters:
 - pass by value
 - pass by reference
 - pass by const reference
 - pass by pointer
- Advanced topics to be discussed later:
 - inline functions
 - default arguments
 - function overloading
 - template functions

Pass by value

demo_0/pass_by_value.cpp

- The variable in the function is a copy of the variable in the calling program:

```
void f(int x) {  
    x++; // increments x but not the variable of the  
        calling program  
    cout << x;  
}
```

```
int main() {  
    int a=1;  
    f(a);  
    cout << a; // is still 1  
}
```

- Copying of variables time consuming for large objects like matrices

Pass by reference

demo_0/pass_by_reference.cpp

- The function parameter is an alias for the original variable:

```
void increment(int& n) {  
    n++;  
}
```

```
int main() {  
    int x=1; increment(x); // x now 2  
    increment(5); // will not compile since 5 is  
    literal constant!  
}
```

- avoids copying of large objects:
 - `vector eigenvalues(Matrix &A);`
- but allows unwanted modifications!
 - the matrix A might be changed by the call to eigenvalues!

Pass by const reference

- ♦ Problem:
 - ♦ `vector eigenvalues(Matrix& A);` // allows modification of A
 - ♦ `vector eigenvalues(Matrix A);` // involves copying of A
- ♦ how do we **avoid copying and prohibit modification?**
 - ♦ `vector eigenvalues (Matrix const &A);`
 - ♦ now a reference is passed → no copying
 - ♦ The parameter is const → cannot be modified

Pass by pointer

- Another method to pass an object without copying is to pass its address
- Used mostly in C
- `vector eigenvalues(Matrix *m);`
- Disadvantages:
 - The parameter must always be dereferenced: `*m;`
 - In the function call the address has to be taken:

```
Matrix A;  
cout << eigenvalues(&A) ;
```

- **rarely needed in C++**

A swap example

- Five examples for swapping number
 - `void swap1 (int a, int b) { int t=a; a=b; b=t; }`
 - `void swap2 (int& a, int& b) { int t=a; a=b; b=t; }`
 - `void swap3 (int const & a, int const & b){int t=a; a=b; b=t; }`
 - `void swap4 (int *a, int *b) {int *t=a; a=b; b=t; }`
 - `void swap5 (int* a, int* b) {int t=*a; *a=*b; *b=t; }`
- Which will compile?
- What is the effect of:
 - `int a=1; int b=2; swap1(a,b); cout << a << " " << b << "\n";`
 - `int a=1; int b=2; swap2(a,b); cout << a << " " << b << "\n";`
 - `int a=1; int b=2; swap3(a,b); cout << a << " " << b << "\n";`
 - `int a=1; int b=2; swap4(&a,&b); cout << a << " " << b << "\n";`
 - `int a=1; int b=2; swap5(&a,&b); cout << a << " " << b << "\n";`

Type casts: static_cast

- Variables can be converted (cast) from one type to another
- **static_cast** converts one type to another, using the best defined conversion, e.g.
 - `float y=3.f;`
 - `int x = static_cast<int>(y);`
 - replaces the C construct `int x= (int) y;`
- Can also be used, converts one pointer type to another,
- useful for low-level programming, for example to look at representations of floating point numbers or check for endianness
 - `float y=3.f;`
 - `float *fp = &y;`
 - `int *ip = static_cast<int*>(fp)`
 - `std::cout << *ip;`

Namespaces

- What if a `square` function is already defined elsewhere?
- C-style solution: give it a unique name; ugly and hard to type
`float UNIL_square(float);`
- Elegant **C++** solution: **namespaces**
 - Encapsulates all declarations in a modul, called “namespace”, identified by a prefix
 - Example:
`namespace UNIL`
`{`
`float square(float);`
`}`
- Can be accessed from outside as:
 - `UNIL::square(5);`
 - `using UNIL::square;`
`square(5);`
 - `using namespace UNIL;`
`square(5);`
- Standard namespace is `std`
- For backward compatibility the standard headers ending in `.h` import `std` into the global namespace. E.g. the file “`iostream.h`” is:

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

Namespaces can be nested

Default function arguments

demo_0/default_fun_args.cpp

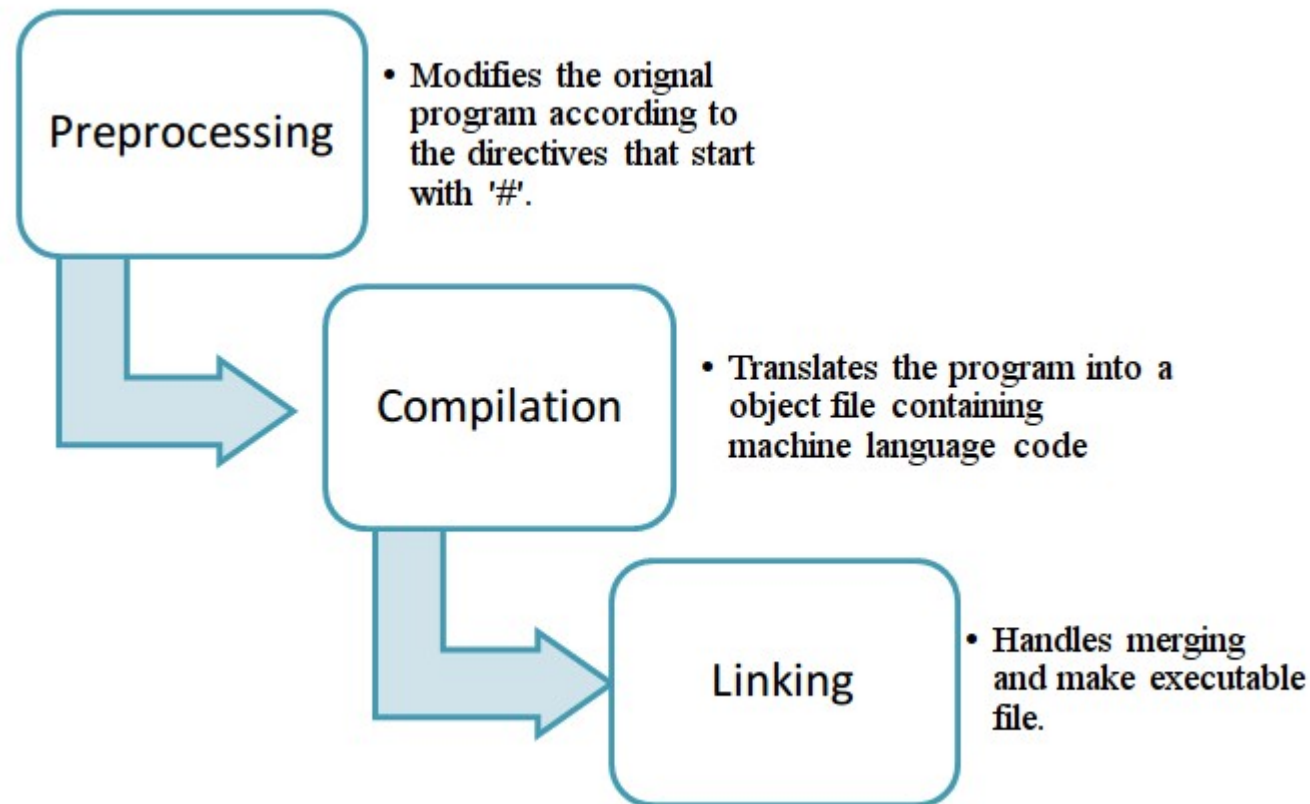
- are sometimes useful

```
float root(float x, unsigned int n=2); // n-th root of x
int main()
{
    root(5,3); // cubic root of 5
    root(3,2); // square root of 3
    root(3); // also square root of 3
}
```

- the default value must be a constant!

```
unsigned int d=2;
float root(float x, unsigned int n=d); // not allowed!
```

2. Preprocessing/Compiling/Linking



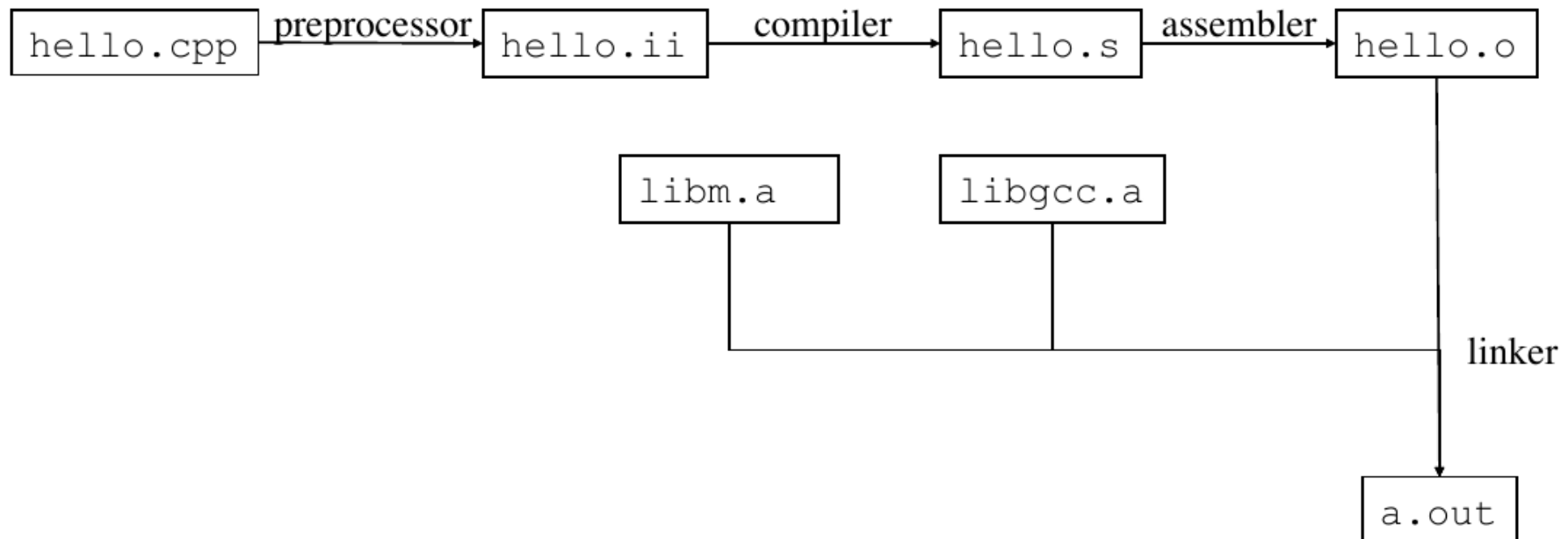
Steps when compiling a program

- What happens when we type the following?

```
g++ hello.cpp
```

- Observe the steps by adding some extra flags:

```
g++ --verbose -save-temps hello.cpp
```



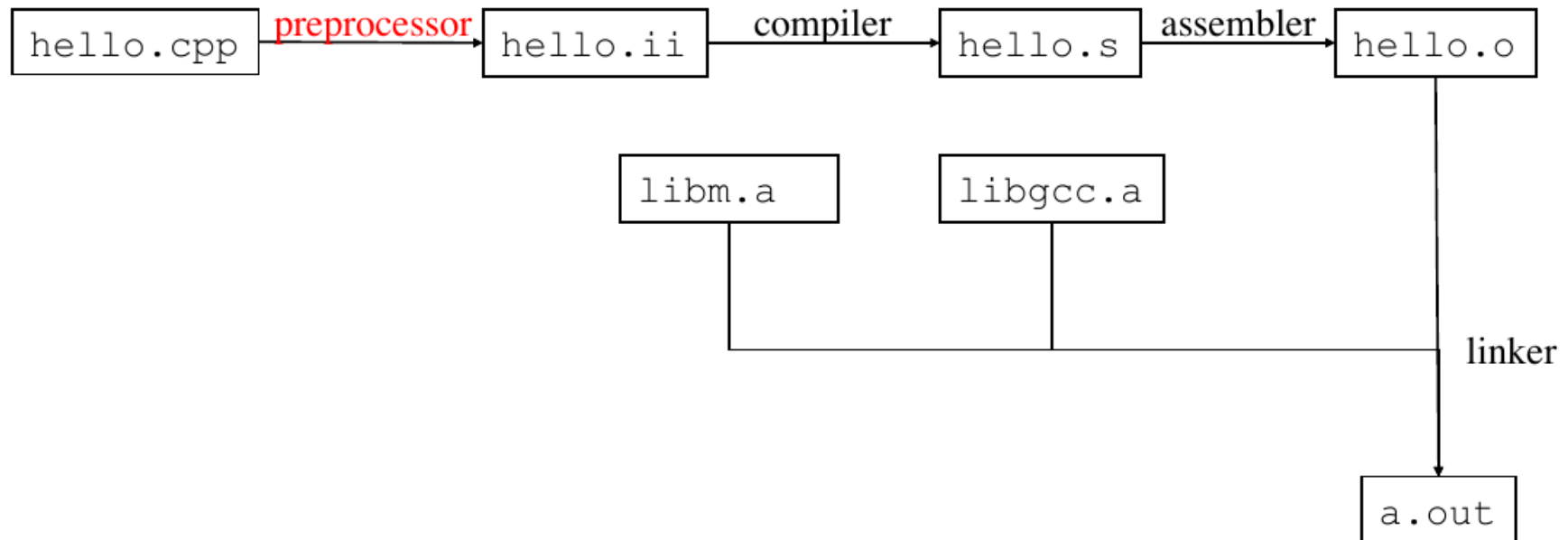
Steps when compiling a program

- What happens when we type the following?

```
g++ hello.cpp
```

- Observe the steps by adding some extra flags:

```
g++ --verbose -save-temps hello.cpp
```



The C++ preprocessor

- ♦ Is a text processor, manipulating the source code
- ♦ Commands start with #
 - ♦ `#define XXX`
 - ♦ `#define YYY 1`
 - ♦ `#define ADD(A,B) A+B`
 - ♦ `#undef ADD`
 - ♦ `#ifdef XXX`
`#else`
`#endif`
 - ♦ `#if defined(XXX) && (YYY==1)`
`#elif defined (ZZZ)`
`#endif`
 - ♦ `#include <iostream>`
 - ♦ `#include "square.h"`

#define

- Defines a **preprocessor macro**
 - `#define XXX "Hello"`
`std::cout << XXX;`
 - Gets converted to
`std::cout << "Hello"`
- Macro arguments are possible
 - `#define SUM(A,B) A+B`
`std::cout << SUM(3,4);`
 - Gets converted to
`std::cout << 3+4;`
- Definitions on the command line possible
 - `g++ -DXXX=3 -DYYY`
 - Is the same as writing in the first line:
`#define XXX 3`
`#define YYY`

#undef

- ♦ Undefines a macro
 - ♦ `#define XXX "Hello"`
`std::cout << XXX;`
`#undef XXX`
`std::cout << "XXX";`
 - ♦ Gets converted to
`std::cout << "Hello"`
`std::cout << "XXX"`
- ♦ Undefines on the command line are also possible
 - ♦ `g++ -UXXX`
 - ♦ Is the same as writing in the first line:
`#undef XXX`

Looking at preprocessor output

- Running only the preprocessor:

```
g++ -E
```

- Running the full compile process but storing the preprocessed files

```
g++ -save-temps
```

- Look at the files pre1.cpp and pre2.cpp, then at the output of
 - `g++ -E demo_1/cprepro/pre1.cpp`
 - `g++ -E demo_1/cprepro/pre2.cpp`
 - `g++ -E -DSCALE=10 demo_1/cprepro/pre2.cpp`

Looking at preprocessor output

```
g++ -E -DSCALE=10 demo_1/cprepro/pre2.cpp
```

```
# 1 "pre2.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "pre2.cpp"
double scale (double x)
{
    return 10 * x;
}
```

#ifdef ... #endif

- Conditional compilation can be done using **#ifdef**

```
#ifdef SYMBOL
    something
#else
    somethingelse
#endif
```

- Becomes, if SYMBOL is defined:

```
something
```

Otherwise it becomes
something else

- Look at the output of

- `g++ -E demo_1/cprepro/pre3.cpp`
- `g++ -DDEBUG -E demo_1/cprepro/pre3.cpp`

#ifdef ... #endif

g++ -E demo_1/cprepro/pre3.cpp

```
double safe_sqrt(double x)
{
#ifdef DEBUG
    if (x<0.)
        std::cerr << "Problem in sqrt\n";
#endif
    return std::sqrt(x);
}
```

```
# 1 "pre3.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "pre3.cpp"
double safe_sqrt(double x)
{
    return std::sqrt(x);
}
```

#ifdef ... #endif

g++ -DDEBUG -E demo_1/cprepro/pre3.cpp

```
double safe_sqrt(double x)
{
#ifdef DEBUG
    if (x<0.)
        std::cerr << "Problem in sqrt\n";
#endif
    return std::sqrt(x);
}
```

```
# 1 "pre3.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "pre3.cpp"
double safe_sqrt(double x)
{

    if (x<0.)

        std::cerr << "Problem in sqrt\n";


    return std::sqrt(x);

}
```

#if ... #elif ... #endif

- Allows more complex instructions, e.g.

```
#if !defined (__GNUC__)
    std::cout << " A non-GNU compiler";
#elif __GNUC__<=2 && _GNUC_MINOR < 95
    std::cout << "gcc before 2.95";
#elif __GNUC__==2
    std::cout << "gcc after 2.95";
#elif __GNUC__>=3
    std::cout << "gcc version 3 or higher";
#endif
```


#error

- ♦ Allows to issue error messages

```
#if !defined(__GNUC__)  
#error This program requires the GNU compilers  
#else  
...  
#endif
```

- ♦ Try the following

```
g++ -c demo_1/cprepro/pre4.cpp
```

#error

```
g++ -c demo_1/cprepro/pre4.cpp
```

```
#ifndef DEFINEME  
#error Please add -DDEFINEME to the command line  
#endif
```

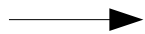
→ g++ -DDEFINEME=bla -E pre4.cpp

#include "file.h" #include <iostream>

- ♦ Includes another source file at the point of invocation
- ♦ Try the following
- ♦ `g++ -E demo_1/cprepro/pre5.cpp`
- ♦ `< >` brackets refer to system files, e.g. `#include <iostream>`
- ♦ `g++ -E demo_1/cprepro/pre6.cpp`
- ♦ With `-I` you tell the compiler where to look for include files. Try:
`g++ -E demo_1/cprepro/pre7.cpp`
`g++ -E -Iinclude pre7.cpp`

g++ -E demo_1/cprepro/pre5.cpp

#include "pre1.cpp"



1 "pre5.cpp"

1 "<built-in>"

1 "<command-line>"

1 "/usr/include/stdc-predef.h" 1 3 4

1 "<command-line>" 2

1 "pre5.cpp"

1 "pre1.cpp" 1

double scale (double x)

{

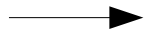
return (10. * x < 0. ? 10. * x : 0.);

}

1 "pre5.cpp" 2

g++ -E demo_1/cprepro/pre6.cpp

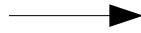
```
#include <iostream>
```



Long list...

g++ -E demo_1/cprepro/pre7.cpp

#include "file.hpp"



1 "pre7.cpp"

1 "<built-in>"

1 "<command-line>"

1 "/usr/include/stdc-predef.h" 1 3 4

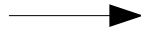
1 "<command-line>" 2

1 "pre7.cpp"

pre7.cpp:1:20: fatal error: file.hpp: No such file or directory
compilation terminated.

g++ -E -Iinclude pre7.cpp

#include "file.hpp"



```
# 1 "pre7.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "pre7.cpp"
# 1 "include/file.hpp" 1
double square(double x)
{
    return x*x;
}
# 1 "pre7.cpp" 2
```

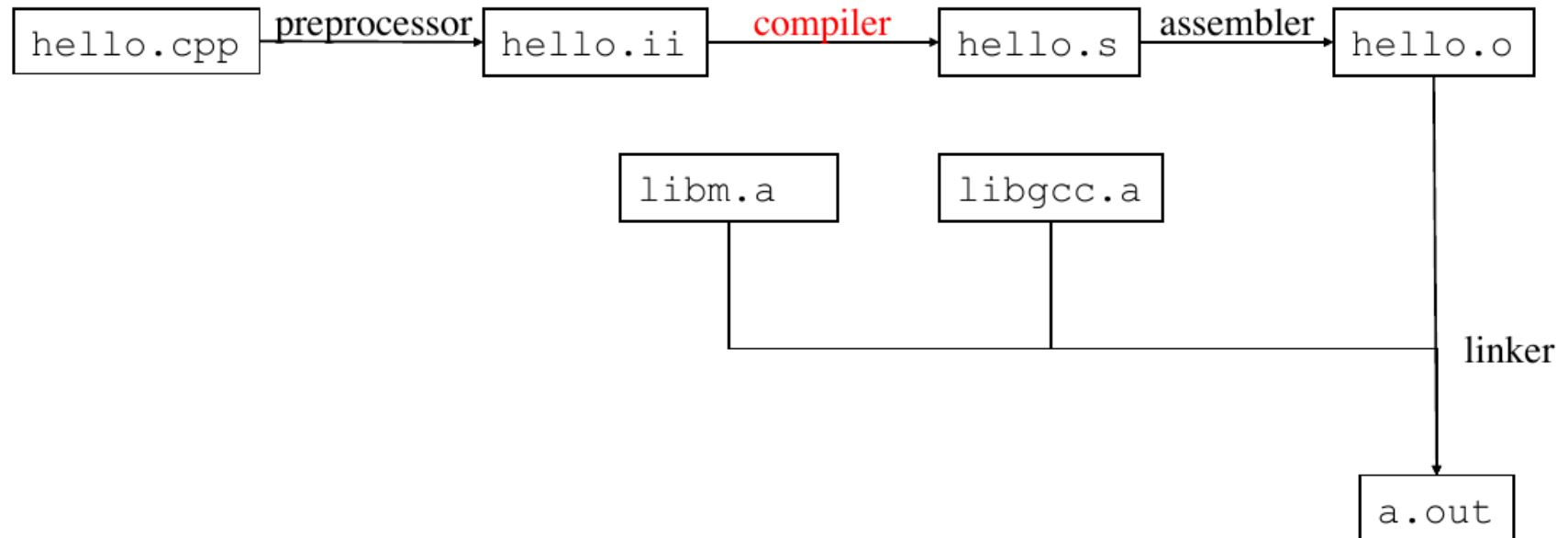
Steps when compiling a program

- What happens when we type the following?

`g++ hello.cpp`

- Observe the steps by adding some extra flags:

`g++ --verbose -save-temps hello.cpp`



Looking at the compilation output

demo_0/functioncall.cpp

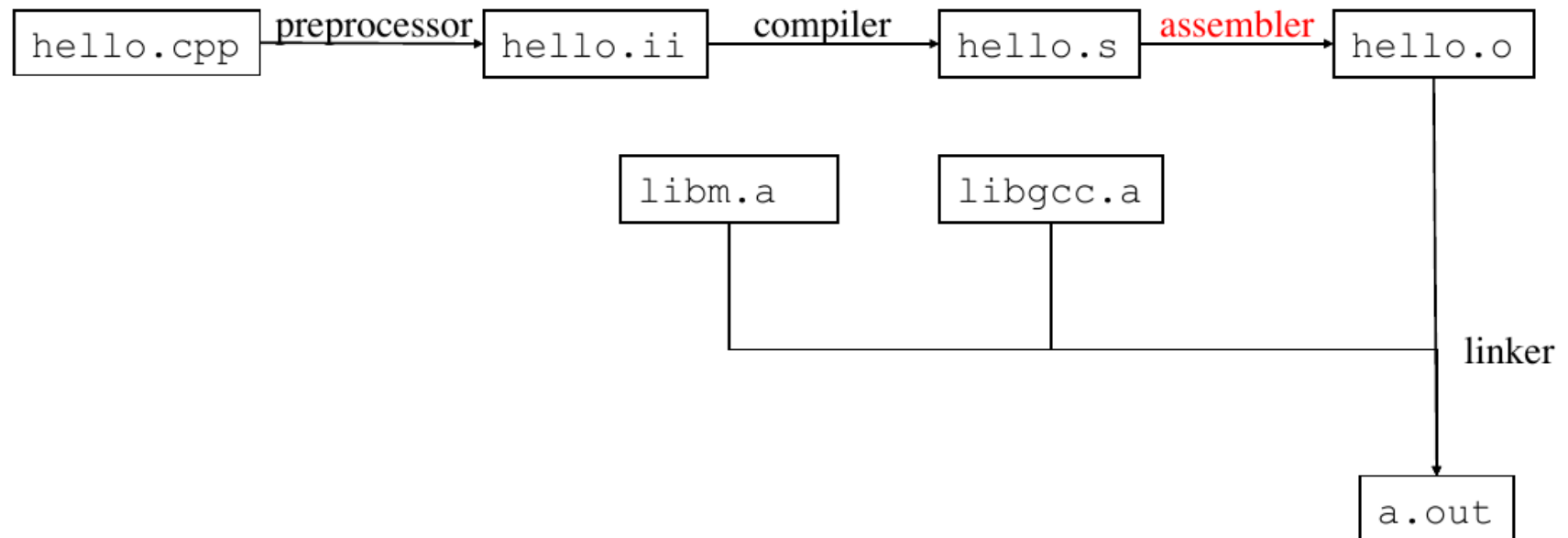
- Let us look at the assembly code of a simple example
 - `g++ -c -save-temps -O0 functioncall.cpp`
 - `g++ -c -save-temps -O3 functioncall.cpp`
 - `g++ -c -save-temps -finline-functions functioncall.cpp`
- Look at functioncall.s - What can you observe?
 - Can you observe automatic “inlining”?

```
int add (int x, int y)
{
    return x+y;
}

int f(int x)
{
    return add(x,3);
}
```

Steps when compiling a program

- What happens when we type the following?
`g++ hello.cpp`
- Observe the steps by adding some extra flags:
`g++ --verbose -save-temps hello.cpp`



Looking at the assembler output

"hello.o" may be a binary file. See it anyway?

```
<CF><FA><ED><FE>^G^@^@A^C^@^@^@H^@^@^@I^@^@^@P^B^@^@^@ ^@^@^@^@Y^@^@^@  
<CF><FA><ED><FE>^G^@^@A^C^@^@^@H^@^@^@I^@^@^@P^B^@^@^@ ^@^@^@^@Y^@^@^@  
<D8>^A^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@>F8>^P^@^@^@^@^@^@^@^@^@_text^@^@^@^@  
^@^@^@^@^@^@<F8>^P^@^@^@^@^@^@^@^@^@G^@^@^@G^@^@^@E^@^@^@^@^@^@__TEXT^@^@^@^@  
^@^@^@^@__TEXT^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@+^M^@^@^@^@^@^@^@^@^@ID^@^@^@  
h^S^@^@^@+^@^@^@^@ID<80>^@^@^@^@^@^@^@^@^@^@^@^@__gcc_except_tab__TEXT^@^@^@^@  
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@9C>^O^@^@^@B^@^@^@^@^@^@^@^@^@^@^@^@^@  
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@__cstring^@^@^@^@^@^@^@^@^@__TEXT^@^@^@^@N  
^@^@^@^@^@^@G^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@CC>^P^@^@^@^@^@^@^@^@^@^@^@^@B^@^@^@^@^@^@^@  
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@__compact_unwind__LD^@^@^@^@^@^@^@^@^@^@^@^@N^@^@^@^@^@^@H^@^@  
^@^@^@^@^@^@ID>^P^@^@^@C^@^@^@CO>T^@^@^@N^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@e  
frame^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@0^@^@^@^@^@^@^@^@90>^A^@^@^@^@^@^@  
<D8>^Q^@^@^@C^@^@^@U^@^@^@A^@^@^@K^@^@^@h^@^@^@^@^@^@^@^@^@^@^@^@$^@^@^@P^@^@^@  
^@^@K  
^@^@^@^@^@^@B^@^@^@X^@^@^@3^U^@^@^@ ^@^@^@3W^@^@^@E4>D^@^@^@K^@^@^@^@^@^@^@^@^@^@  
C^@^@^@C^@^@^@ ^@^@^@L^@^@^@T^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@  
^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@UH<89><E5>H<83>  
<EC> H<8B>=^@^@^@^@H<8D>5FN^@^@^@E8>^@^@^@^@H<8D>5^@^@^@^@H<89>EF8>H<89>JFO>H  
<8B>JF8><FF>JFO>IC9>H<89>EE8><89><C8>H<83><C4> J<C3><90>UH<89><E5>H<83><EC>  
H<89>JF8>H<89>JFO>H<8B>JF8>H<8B>JFO>H<8B>EF0>H<89>JE8>H<89><C7>H<89>JE0>  
<E8>^@^@^@^@H<8B>JE8>H<8B>JE0>H<89><C2><E8>^@^@^@^@H<83><C4> J<C3>ffffff.^0_  
<84>^@^@^@^@UH<89><E5>H<81><EC><90>^@^@^@H<89>JB8>H<89><F8>H<8B>DH<8B>IE8>
```

Segmenting programs

See demo_1/square

- **Programs can be**
 - split into several files
 - Compiled separately
 - and finally linked together
- However **functions defined in another file have to be declared before use!**
- The function declaration is similar to the definition
 - but has no body!
 - parameters need not be given names
- **Easiest solution are header files.**
- **Help maintain consistency.**

- file **“square.hpp”**

```
double square(double);
```
- file **“square.cpp”**

```
#include “square.hpp”
double square(double x) {
    return x*x;
}
```
- file **“main.cpp”**

```
#include <iostream>
#include “square.hpp”
int main() {
    std::cout << square(5.);
}
```

Compiling and linking

See [demo_1/square](#)

- **Compile** the file `square.cpp`, with the **-c** option (no linking)
`g++ -c square.cpp`
- **Compile** the file `main.cpp`, with the **-c** option (no linking)
`g++ -c main.cpp`
- **Link** the object files
`g++ main.o square.o`
- **Link** the object files and name it `square.exe`
`g++ main.o square.o -o square.exe`

Include guards

- ♦ Consider file “**grandfather.h**”:

```
struct foo{  
    int member;  
};
```

- ♦ and file “**father.h**”:

```
#include “grandfather.h”;
```

- ♦ And finally “**child.cpp**”:

```
#include "grandfather.h"  
#include "father.h"
```

- ♦ What happens here: `g++ -c child.cpp`

Include guards

- Consider file “**grandfather.h**”:

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H
struct foo{
    int member;
};
#endif /* GRANDFATHER_H */
```

- and file “**father.h**”:

```
#include "grandfather.h";
```

- And finally “**child.cpp**”:

```
#include "grandfather.h"
#include "father.h"
```

- Works: `g++ -c child.cpp`

Assert in header <cassert>

- are a way to check preconditions, postconditions and invariants
<cassert> looks something like:

```
#ifdef NDEBUG
#define assert(e)          ((void)0)
#else
#define assert(e) /*implementation defined*/
#endif
```

- If the expression is false the program will abort and print the expression with a notice that this assertion has failed
- Try it
g++ assert.cpp
g++ -DNDEBUG assert.cpp

Assert in header <cassert>

See `demo_1/assert/assert.cpp`

```
#include <iostream>
#include <cmath>
#include <cassert>

double checked_sqrt(double x)
{
    assert(x >= 0.);
    return sqrt(x);
}

int main()
{
    double x;
    std::cout << "Give me a number: x = ";
    std::cin >> x;
    std::cout << "sqrt(x) = " << checked_sqrt(x) << "\n";
}
```

Making a (static) library on Linux/Unix/MacOS X

- Often used *.o files can be packed into a library, e.g.:

```
ar ruc libsquare.a square.o
ranlib libsquare.a
g++ main.cpp -Ilib -Llib -lsquare
```

- `ar` creates an `archive`, more than one object file can be specified
- The name must be `libsomething.a`
- `ranlib` adds a table of contents (not needed on some platforms)
- `-I` specifies the directory where the header file is located
- `-L` specifies the directory where the library is located
- `-lsomething` specifies looking in the library `libsomething.a`

How libraries work

- What is done here:

```
g++ main.cpp -Ilib -Llib -lsquare
```

- After compilation the object files are linked
- If there are undefined functions (e.g. `square`) the libraries are searched for the function, and the needed functions linked with the object files
- Note that the order of libraries is important
- if `liba.a` calls a function in `libb.a`, you need to link in the right order: `-la -lb`

How libraries work

```
cd lib
```

```
g++ -c square.cpp
```

```
ranlib libsquare.a
```

```
ar ruc libsquare.a square.o
```

```
ranlib libsquare.a
```

```
cd ..
```

```
g++ main.cpp -llib -Llib -lsquare
```

Documenting your library

- ♦ After you finish your library, **document it with**
 - ♦ **Synopsis** of all functions, types and variables declared
 - ♦ Semantics
 - ♦ what does the function do?
 - ♦ Preconditions
 - ♦ what must be true before calling the function
 - ♦ Postconditions
 - ♦ what you guarantee to be true after calling the function if the precondition was true
 - ♦ What it depends on
 - ♦ Exception guarantees
 - ♦ References or other additional material

Example documentation

- Header file “square.h” contains the function “square”:
 - **Synopsis:**
`double square(double x);`
 - `square` calculates the square of `x`
 - **Precondition:** the square can be represented in a double
`std::abs(x) <= std::sqrt(std::numeric_limits<double>::max())`
 - **Postcondition:** the square root of the return value agrees with the absolute value of `x` within floating point precision:
`std::sqrt(square(x)) - std::abs(x) <=`
`std::abs(x) * std::numeric_limits<double>::epsilon()`
 - **Dependencies:** none
 - **Exception guarantee:** no-throw

After a while it becomes tedious...

- ♦ Consider

```
g++ -c a.cpp
```

```
g++ -c b.cpp
```

```
g++ -c c.cpp
```

```
u ...
```

```
g++ main.o a.o b.o c.o ... -I... -L... -l...
```

- ♦ Change something

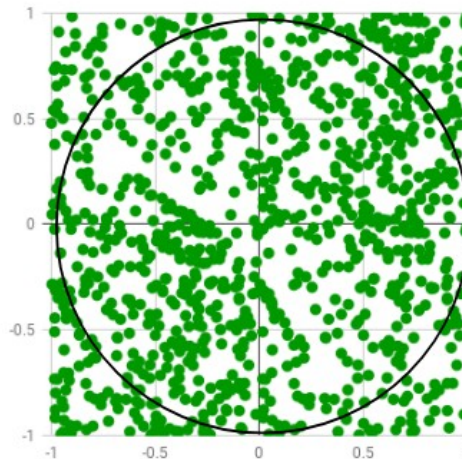
- ♦ Need to keep track of what depends on what

- ♦ Or: Build systems!!!

- ♦ make
- ♦ cmake
- ♦ (SCons, ...)

Action required – Approximate Π

- Monte Carlo estimation are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results.
- One of the basic examples of getting started with the Monte Carlo algorithm is the estimation of Π .
- The idea is to simulate random (x, y) points in a 2-D plane with domain as a square of side 1 unit.
- Imagine a circle inside the same domain with same diameter and inscribed into the square. We then calculate the ratio of number points that lied inside the circle and total number of generated points. Refer to the image below:



Exercise 2: Approximate π

- We know that area of the square is 1 unit sq while that of circle is $\pi * (\frac{1}{2})^2 = \frac{\pi}{4}$
- Now for a very large number of generated points,

$$\frac{\text{area of the circle}}{\text{area of the square}} = \frac{\text{no. of points generated inside the circle}}{\text{total no. of points generated or no. of points generated inside the square}}$$

$$\pi = 4 * \frac{\text{no. of points generated inside the circle}}{\text{no. of points generated inside the square}}$$

Write a program that computes the area of the unit circle by Monte Carlo integration (demo_0/pi_rand.cpp).