An Introduction to C++

Part 2

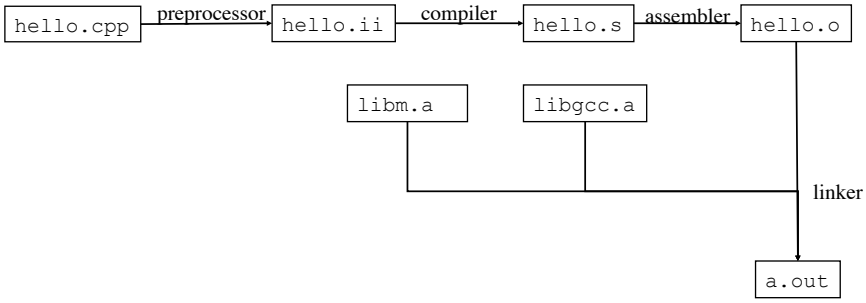
More basic C and C++:

Getting the new (updated) sources by SVN

- Go to your source directory, e.g.
  - `cd pt12`
- Update your sources
  - `svn update`
- Then go there
  - `cd week2`

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"`
    `cout << XXX;`
  - Gets converted to
    `cout << "Hello"`
- Macro arguments are possible
  - `#define SUM(A,B) A+B`
    `cout << SUM(3,4);`
  - Gets converted to
    `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"`
    `cout << XXX;`
    `#undef XXX`
    `cout << "XXX";`
  - Gets converted to
    `cout << "Hello"`
    `cout << "XXX"`
- Definitions 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:
  - `c++ -E`
- Running the full compile process but storing the preprocessed files
  - `c++ -save-temps`
- Look at the files pre1.C and pre2.C, then at the output of
  - `c++ -E pre1.C`
  - `c++ -E pre2.C`
  - `c++ -E -DSCALE=10 pre2.C`

#ifdef … #endif

- Conditional compilation can be done using #ifdef
  -
```
#ifdef SYMBOL
   something
#else
  somethingelse
#endif
```
  - Becomes, if SYMBOL is defined:
```
something
```
  - Otherwise it becomes
```
somethingelse
```
- Look at the output of
  - `c++ -E pre3.C`
  - `c++ -DDEBUG -E pre3.C`

#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 pre4.C`

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

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

Segmenting programs

- 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

- Compile the file square.cpp, with the -c option (no linking)
  - `c++ -c square.cpp`
- Compile the file main.cpp, with the -c option (no linking)
  - `c++ -c main.cpp`
- Link the object files
  - `c++ main.o square.o`

Include guards

- The following fails to compile :
  - ```
    #include “incl.hpp”
    #include “incl.hpp”
    ```
- Try it:
  - `c++ -c guard.C`
- Add include guards to incl.h and try again:
  - ```
    #ifndef SQUARE_H
    #define SQUARE_H

    int x;
    #endif
    ```

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) …
#endif
```

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

Making a library

- Often used *.o files can be packed into a library, e.g.:
  -
```
ar ruc libtest.a square.o
ranlib libtest.a
c++ main.cpp -L. -ltest
```
- `ar` creates an archive, more than one object file can be specified
  - The name must be `lib`*`something`*`.a`
- `ranlib` adds a table of contents (not needed on some platforms)
- `-L` specifes the directory where the library
- `-l`*`something`* specifies looking in the library `lib`*`something`*`.a`

How libraries work

- What is done here:
  - `c++ main.C -L. -ltest`
- 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`

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 (will be discussed later)
  - 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

Contract programming

- For each function define the set of
  - Preconditions
    - Conditions that the caller has to satisfy to get legal and correct behavior.
    - The callee can assert on the conditions, to test the precondition and abort if hey are not satsified. This helps debugging.
  - Postconditions
    - Conditions that the callee guarantees if the caller satisfies the preconditions. Again the callee can assert on the postconditions to help debugging if it is not obvious that the postcondition is satisfied.
  - Invariants
    - Are expressions that stay unchanged when a mutating function is called, if the preconditions are satisfied.
- Document the preconditions, postconditions and invariants and include tests

The cost of a function call

- A function call is expensive:
  - Values in registers might need to be saved in memory
  - Function arguments might need to be stored in memory
  - A jump to the function is done, stopping all pipelines
  - Function arguments might need to be read from memory
  - Only then can the function start to execute
- Let us look at the assembly code of a simple example
  - `c++ -c -save-temps -O0 functioncall.cpp`
  - `c++ -c -save-temps -O functioncall.cpp`
  - `c++ -c -save-temps -finline-functions functioncall.cpp`
- Look at functioncall.s - What can you observe?
  - Can you observe automatic “inlining”?

Inlining

- A function call takes several hundred CPU cycles
- For simple functions that are called often this is a big waste of time:
  - `float square(float);`

```
int main() {
  float sq[10000];
  for (int k=0;k<10000;++k)
    sq[k] = square(k);
}
```

- It is better to inline the function
  - `inline float square(float x) {return x*x;}`
- This leads to the same optimized code as:
  - `sq[k] = float(k)*float(k);`
- Note that for an inline function not only the declaration but the complete function body must be in the header file!

Recursion

- is elegant and allowed

```
unsigned long fac(unsigned short k) {
  return k ? k*fac(k-1) : 1;
}
```

- however these function calls cannot be inlined!
- non-recursive version often faster

```
unsigned long fac(unsigned short k) {
  unsigned long r=1;
  if(k) do { r *=k;} while(--k);
  return r;
}
```

- exception: template codes, as they are evaluated at compile time. We will come back to that later.

Default function arguments

- 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!
```