An Introduction to C++

Part 2

More basic C and C++:

Getting the new (updated) sources by CVS

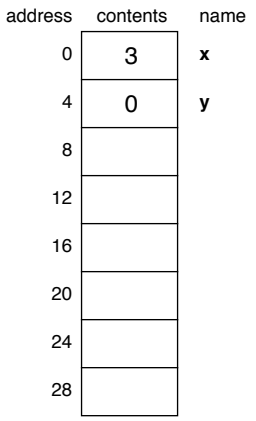
- If you are on your own machine, set CVS access by ssh using either (better add them to your .tcshrc or .bashrc files):
  - bash: `export CVS_RSH=ssh`
  - tcsh: `setenv CVS_RSH ssh`
- Go to your source directory, e.g.
  - `cd Vorlesung/PT`
- Update your sources
  - `cvs update -d`
- The -d option told CVS to look for new directories (here week2). Now go there:
  - `cd week2`

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



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
```

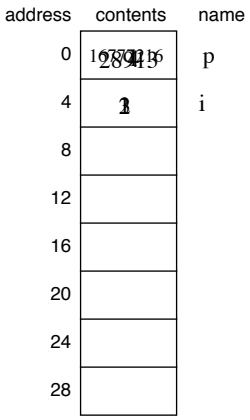
  - Are dereferenced with the * operator

```
*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!



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

Pointer arithmetic

- for any pointer `T *p;` the following holds:
  - `p[n]` is the same as `*(p+n);`
- Adding and integer *n* to a pointer increments it by the *n* times the size of the type – and not by *n* bytes
- Increment ++ and decrement -- increase/decrease by one element
- Be sure to only use valid pointers
  - initialize them
  - do not use them after the object has been deleted!
  - catastrophic errors otherwise

Arrays and pointers

- are very similar, but subtly different!

```
int array[5];

for (int i=0;i < 5; ++i)
  array[i]=i;

int* p = array; // same as &array[0]
for (int i=0;i < 5; ++i)
  cout << *p++;


delete[] p; // will crash
array=0; // will not compile
p=0; // is OK
```

- see these examples!

```
int* pointer=new int[5];

for (int i=0;i < 5; ++i)
  pointer[i]=i;

int* p = pointer;
for (int i=0;i < 5; ++i)
  cout << *p++;
```

-

```
p=pointer;
delete[] p; // is OK
delete[] pointer; // crash
delete[] p; // will crash
p=0; // is OK
pointer=0; // is OK
```

A look at memory: array example

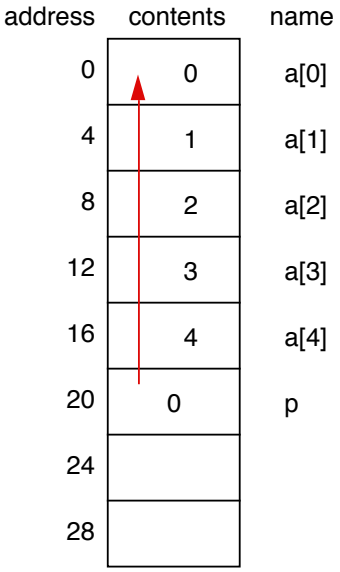
- Array example

```
int array[5];

for (int i=0;i < 5; ++i)
  array[i]=i;

int* p = array; // same as &array[0]
for (int i=0;i < 5; ++i)
  cout << *p++;


delete[] p; // will crash
array=0; // will not compile
p=0; // is OK
```



A look at memory: pointer example

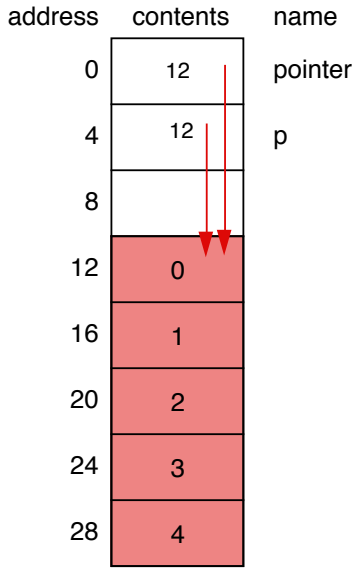
- Array example

```
int* pointer=new int[5];

for (int i=0;i < 5; ++i)
  pointer[i]=i;

int* p = pointer;
for (int i=0;i < 5; ++i)
  cout << *p++;

delete[] pointer; // is OK
 delete[] pointer; // crash
delete[] p; // will crash
p=0; // is OK
pointer=0; // is OK
```



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!
```

A more flexible program: function calls

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

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

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

- a function "square" is defined
  - return value is float
  - parameter x is float
- 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

- 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

- 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";`

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

Type casts

- 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;`
- **`reinterpret_cast`** does not care about the meaning of the variable (e.g. a pointer) but just interprets the bit pattern as the new type
  - `int x = *reinterpret_cast<int*>(&y);`
  - Converts a pointer to a float to a pointer to an int
  - x will contain the bit pattern representing 3.f;
  - Compile and run the file `cast.c`

Type casts (continued)

- const_cast can be used to remove const-ness from a variable
  - Example: need to pass a `double*` to a C-style function which does not change the value, but I only have a `const double*`

```
void legacy_c_function (double* d);

void foo(const double* d) {
   // remove the const
   double* nonconst_d = const_cast<double*>(d);
   // now call the function
   legacy_c_function(nonconst_d);
 }
```

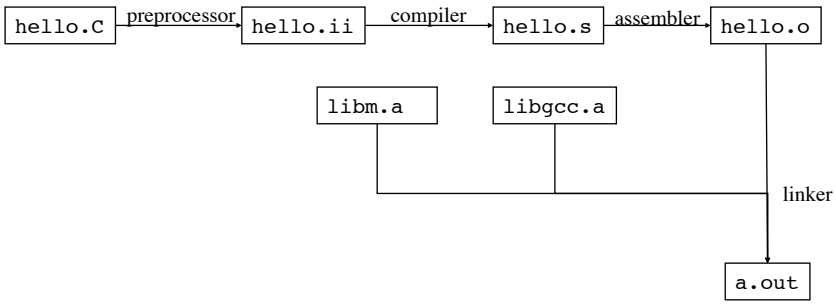
  - Use it very sparingly. Usually the need for `const_cast` is a sign of bad software design
- Other casts to be discussed later:
  - `dynamic_cast`
  - `boost::lexical_cast`
  - `boost::numeric_cast`

Namespaces

- What if a `square` function is already defined elsewhere?
- C-style solution: give it a unique name; ugly and hard to type
  ```
  float ETH_square(float);
  ```
- Elegant C++ solution: namespaces
  - Encapsulates all declarations in a modul, called “namespace”, identified by a prefix
  - Example:
    ```
    namespace ETH
    {
      float square(float);
    }
    ```
- Namespaces can be nested
- Can be accessed from outside as:
  - `ETH::square(5);`
  - `using ETH::square;`
    `square(5);`
  - `using namespace ETH;`
    `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;
  ```

Steps when compiling a program

- What happens when we type the following?
  `g++-3.3 hello.C`
- Observe the steps by adding some extra flags:
  `g++-3.3 --verbose -save-temps hello.C`



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:
  - `g++ -E`
- Running the full compile process but storing the preprocessed files
  - `g++ -save-temps`
- Look at the files pre1.C and pre2.C, then at the output of
  - `g++ -E pre1.C`
  - `g++ -E pre2.C`
  - `g++ -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
  - `g++ -E pre3.C`
  - `g++ -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
  - `g++ -E pre5.C`
- `< >` brackets refer to system files, e.g. `#include <iostream>`
  - `g++ -E pre6.C`
- With -I you tell the compiler where to look for include files. Try:
  - `g++ -E pre7.C`
  - `g++ -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.h"

```
double square(double);
```

- file "square.C"

```
#include "square.h"
double square(double x) {
  return x*x;
}
```

- file "main.C"

```
#include <iostream>
#include "square.h"

int main() {
  std::cout << square(5.);
}
```

Compiling and linking

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

Include guards

- The following fails to compile :
  - ```
    #include "incl.h"
    #include "incl.h"
    ```
- Try it:
  - ```
    g++ -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
  - ```
    g++ 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`
    `g++ main.C -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:
  - `g++ 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
  - Exception guarantee: no-throw

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
  - `g++ -c -save-temps -O0 functioncall.c`
  - `g++ -c -save-temps -O functioncall.c`
  - `g++ -c -save-temps -finline-functions functioncall.c`
- 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!
```