C++

a short introduction by some examples

Topics covered:

- Build process and the preprocessor
- Data types
- Type casting and arrays
- Pointers and references
- Stack and Heap: Memory management

The source code in the examples relates to the GNU C++ Compiler (g++) but should work with almost every other C++ compiler.

Contact:

Fabian Miczek
fabian@mytum.de
ICQ: 150499133

Build process

How the source code is translated into an executable program

	source code	
1. Preprocessor	↓	text substitutions
	preprocessed source code	
2. Compiler	↓	machine code generation
	object file	
3. Linker	↓	bundles object files and links to system libraries
	executable (or shared library)	

Important: Distiguish between *compile-time* and *run-time* behaviour/errors etc !!

Command line calls of the g++ compiler:

`g++ -E test.cpp`	stops after preprecessing (console output)
`g++ -c test.cpp`	stops after compiling (object file test.o)
`g++ test.cpp`	stops after linking (executable file a.out or a.exe)

Preprocessor

an example

dummy.h

```
void anothertestfunction()
{
    cout << "some dummy text";
}

#error Preprocessing stopped!
```

myprogram.cpp

```
#define PVAR 2
#include "someheader.h"

#if PVAR == 1
#include "dummy.h"
void againanothertestfunction()
{
    // do nothing
    return;
}
#endif

int main()
{
     #ifdef PVAR_SEEN_IN_SOMEHEADER
     testfunction();
     #endif

     COUTPUT("PVAR was ");
     cout << PVAR;
     return 0;
}
```

someheader.h

```
#ifdef PVAR
#define PVAR_SEEN_IN_SOMEHEADER
#endif

void testfunction()
{
    cout << "Hello World";
}

#define COUTPUT(x) cout << x
```

Preprocessor output

`g++ -E myprogram.cpp`

```
void testfunction()
{
    cout << "Hello World";
}

int main()
{
    testfunction();
    cout << "PVAR was ";
    cout << 2;
    return 0;
}
```

Preprocessor

some remarks

Useful for

- *simple* text substitutions (when it's just messy to type so much)
- general build control
- switchable debugging output

```
#ifdef DEBUG
cout << "x = " x << endl;
#endif
```

```
#ifdef DEBUG
#define DEBUGOUT(msg) cout << msg << endl
#else
#define DEBUGOUT(msg)
#endif
DEBUGOUT("x = " << x);
```

- platform specific adjustments (some variables are automatically set by compiler)

```
#ifdef __CINT__
#endif
```

```
#ifdef WIN32
#endif
```

```
#ifdef LINUX
#endif
```

Do **not** use

- macros as function replacement

```
#define M2(x) if (x>0) { x=0; }
M2(5);          // will produce a compiler error
#define M1(x) if  ((x>1) && (x<10)) { cout << x; }
M1(x++);        // if x was 5 before, it outputs 7, x is 8 afterwards
```

  → use so called *inline functions* for time critical things instead!

- preprocessor variables for contant values

```
#define PI 3.1415          // will work, but very bad style
const double pi=3.1415;  // the C++ way to do that
```

Preprocessor / Standard C++ skeleton

- Preprocessor variables can be set via the command line compiler call i.e.
  `g++ -DPVAR=2 -DDEBUG test.cpp`
- Preprocessor things *should* be written uppercase
- #include "header.h" searches in the current directory and then in standard directories
- #include <header.h> searches the file directly in the standard directories

---

Standard C++ skeleton

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

int main(int argc, char *argv[])        // alternatively int main() when no args are needed
{
     // your code, for example:
     cout << "Number of parameters: " << argc << endl;
     for (int i=0; i<argc; i++)
          cout << "argv[" << i << "] = " << argv[i] << endl;

     // main should always return an integer error code (0 if the program was successfull)
     return 0;
}
```

Source files should always end with a newline without any characters in it!

Basic data types in C++

No type: void

Logical type: bool (1 byte) can be true or false

```
bool b1 = true;     int a=5; bool b2 = (a > 1);
```

Integer types:

	size (usually)	range unsigned	range signed
char	1 byte	0..255	-128..127
short (int)	2 bytes	0..65535	-32768..32767
int	4 bytes	0..4294967295	-2147483648..2147483647
long (int)	4 bytes (32bit) 8 bytes (64bit)	0..4294967295 0..18446744073709551615	-2147483648..2147483647 -9223372036854775808 .. 9223372036854775807

Without sign declaration char is unsigned and short,int,long are signed

```
signed char v1; unsigned int v2;  unsigned short int v3;  signed int v4;     int v5;
```

Floating point types: float (4 bytes) double (8 bytes) *see lecture notes for details*

With `sizeof(long)` you can determine the size of long in bytes at runtime!

Concerning ROOT: Use platform independent types Int_t, UInt_t, Float_t, etc. !
Look at %ROOTSYS%/include/Rtypes.h to see how that works (by preprocessor)

Arrays / Type casting

by examples

Arrays: always have a **fixed length**

```
int test[100]; test[0]=10; test[99]=5;       for (int i=0; i<100; i++) test[i]=i;

short myshorts[5] = { 5, 8, 34, 9, 101 };
long mylongs[] = { 4358445, 46436 };

char mytext[200]="Hello! This is a string with a maximum length of 200 characters!";
char mytext[]="Hello world";
```

For normal arrays (on stack) the length must be determinable at **compile-time**!!!

The following example works with some compilers, but is not really standard C++

```
int len; cout << "Enter array length: "; cin >> len;
long mylongs[len];
```

Use arrays on heap if the length is determinted **run-time**!!! *(see below how that works)*

Array-like structures with a **variable length** are provided by the STL libraries (not covered here)

Type casting: automatically done by the compiler where possible; sometimes explicitly needed

```
double dv = 5.3;
int a = dv;                     // implicit cast, a is 5   (produces a compiler warning)
int b = (int) dv;               // explicit cast, b is 5   (no warning)
double c = (double)(int) dv;    // cast twice, c is 5.0
double d = a;                   // implicit cast (no warning)
```

Pointers

Pointers are unsigned long integers. The value is an adress in the main memory.
The data type behind this adress is the type of the pointer!

Define with a * after the type name i.e. double* a; double *b; (same behaviour)

Operators on/for pointers:

- &a gets a pointer to the variable a (also called reference)
- *a dereferences a pointer -> gets the value behind a
- a[x] dereferences a pointer at increased position x; a[x] is equivalant to *(a + x)
- a->v equivalant to (*a).v

```
// define pointer to double
double* a;
double b = 5.3;
double c = 2.4;

a = &b; *a = 3.1;
a = &c; *a = 9.9;
// b is now 3.1 and c is now 9.9
```

```
// define pointer with no type
void* ptr;
int b = 5;
double c = 2.4;

ptr = &b;  *(int*)ptr = 3;
ptr = &c;  *(double*)ptr = 9.9;
// b is now 3 and c is now 9.9
```

An array variable without brackets is treated like a pointer to its first element:

```
double arr[5] = {5.1, 7.2, 9.1, 12.3, 15.4};
double* ap;
ap = arr; ap = &arr[0];                 // these two statements are equivalent

cout << *ap << endl;                    // output: 5.1
cout << (unsigned long) ap << endl;     // output: 2293552 (memory adress of arr[0])
ap = ap + 3;
cout << (unsigned long) ap << endl;     // output: 2293576 (first adress + 3*sizeof(double))
cout << *ap << endl;                    // output: 12.3
```

Pointers / References

```
void func1(int a) {
     a = 5;
}
void func2(int* a) {
     *a = 7;
}
int main() {
     int x = 1;
     func1(x);           // leaves x unchanged
     func2(&x);          // the & is important here
     cout << x << endl;  // output: 7
     return 0;
}
```

References

are pointers that are automatically dereferenced. Define with *type*& *name*;
They have to be initialized at the definition and the reference (where it points to) cannot be changed.

```
void func2(int& a) {
     a = 7;
}
int main() {
     int x = 1;
     int& b = x;         // define a reference to x
     b = 3;
     cout << x << endl;  // output: 3
     func2(x);
     cout << x << endl;  // output: 7
     return 0;
}
```

Pointers / References

Warning: Pointers are very useful but often dangerous!
Use references where possible and be careful when using pointers!
The compiler can't even warn you if you write totally senseless code with pointers!

Senseless examples:

```
int *a;
a = 5;
int b = *a;    // crash
*a = 10;       // crash
```

```
char a;
char* c = &a;
c = c + 1;
*c = 10;                 // can crash, very dangerous
```

```
int* a; *a = 5;          // can crash, very dangerous
```

-> always initialize every pointer - at least with 0 or NULL

```
int* a = NULL;           // equivalant to int* a = 0;
*a = 5;                  // dereferencing a NULL pointer will certainly crash
```

Stack and Heap

	Stack	**Heap**
Creation time	fast	a little bit slower
Create (i.e. for an int)	int var1;	int *var1 = new int; "new int" creates an int on the **heap** and returns a reference to it "int *var1" is a pointer that is stored on the **stack**

Delete	only automatically when you leave the definition scope	delete var1;
		delete[] var2; //for arrays
		(or automatically when you quit the whole program, but neat people always clean up before they go, so be neat!)

Maximum size	limited by compiler/operation system, typically 0.1 - 10 MB for the program and *all* stack data	only limited by your physical amount of memory
Accessibility	- direct access in the scope of the definition and direct sub scopes - everywhere by pointer as long as its not deleted var1 = 10; cout << var1;	accessible only via pointers but in any scope until you delete it *var1 = 10; cout << *var1;
Flexibility	size of a variable *should* be determinable at compile time	size of a variable can arbitrary be determined at runtime
Common usage	program variables, counters, temporary variables etc.	large data (structures), classes and strings, compile-time arrays

Stack and Heap

examples

Clean heap solution for arrays with run-time length:

```
int len; cout << "Please enter the size of the array: "; cin >> len;
double *arr = new double[len];
arr[2] = 7.25;
delete[] arr;
```

Scope of stack variables:

```
int a = 5;
for (int i=0; i<10; i++) {
     int b = a + 5;
     a = a + i + b;
}
cout << a;          // no error
cout << b;          // error
cout << i;          // error
```

Scope of heap variables:

```
int* makeBigArray() {
     int* arr = new int[2000];
     arr[296] = 23;
     return arr;
}
int main() {
     int* a;
     a = makeBigArray();
     cout << a[296];
     delete[] a;
}
```

Same on stack - **DONT EVER DO THAT**:

```
int* makeBigArray() {
     int arr[2000];
     arr[296] = 23;
     return arr; // after returning arr is deleted
}
int main() {
     int* a;
     a = makeBigArray();
     cout << a[296]; // perhaps 23, but nobody knows
}
```

Stack and Heap

examples with ROOT objects

Object on stack:

```
void test() {
     const int n = 10;
     double x[n] = {0, 1,  2, 3, 4,  5, 6, 7,  8, 9};
     double y[n] = {1, 3, 10, 1, 3, 10, 1, 3, 10, 6};

     TGraph g(n, x, y);
     g.Draw("APL");
     return;
}
```

works fine, but g is automatically deleted when test returns
-> you won't see the graph

Object on heap:

```
void test() {
     const int n = 10;
     double x[n] = {0, 1,  2, 3, 4,  5, 6, 7,  8, 9};
     double y[n] = {1, 3, 10, 1, 3, 10, 1, 3, 10, 6};

     TGraph *g = new TGraph(n, x, y);
     g->Draw("APL");                    // equivalant to (*g).Draw("APL");
     return;
}
```

works fine, g also "lives" after test has returned, but you should be aware that someone has to make a delete call for the graph at some time.

STL - Standard Template Library

just one example

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

int main() {
    vector<double> x;

    x.push_back(1.695); x.push_back(33.1); x.push_back(9.5);

    cout << "Length: " << x.size() << endl;
    cout << "x[1]:   " << x[1] << endl;

    x.erase(x.begin() + 1);
    x.insert(x.begin(), 23.1);
    x.insert(x.begin(), 9.81);

    cout << "Length: " << x.size() << endl;
    cout << "x[1]:   " << x[1] << endl;

    sort(x.begin(), x.end());

    vector<double>::iterator i;
    for (i = x.begin(); i < x.end(); i++)
        cout << *i << endl;

    double *fx = new double[x.size()];
    uninitialized_copy(x.begin(), x.end(), fx);

    for (int j=0; j<x.size(); j++)
        cout << fx[j] << endl;

    delete[] fx;
    return 0;
}
```

Very important topics that where not covered here:

- Classes and inheritance (essential for C++ programming)
- Strings and character handling
- Streams
- Templates
- Standard C++ libraries and especially the STL library
- Namespaces
- Exceptions
- Multi-file projects and the use of makefiles

Further reading (unordered):

- http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html
- http://www.sgi.com/tech/stl/
- http://www.cplusplus.com/doc/tutorial/
- http://de.wikibooks.org/wiki/C++-Programmierung
- http://en.wikibooks.org/wiki/Category:C++_programming_language
- http://proquest.safaribooksonline.com/browse?category=itbooks.prog.cpp
  (only via LRZ proxy and VPN Client)