
Professional Embedded C/C++

and

Linux System Programming

An Inhouse-Training for *Paul Scherrer Institute (PSI)*

Brought to you by *Programmable Logic Competence Center (PLC2)*

Instructor Dipl.-Ing. Martin Weitzel, *Techn. Beratung für EDV (TBFE)*

2014-05-05 to 2014-05-09, Villigen, CH

---

1 Part 1: C and C++ - Starting Monday

1.1 Welcome

1.1.1 First let me introduce myself

- My name is Martin Weitzel.
- 40 years ago I started studying Electrical Engineering ...
- ... switching over to Information Technology (Datentechnik) after three years.
- I'm working as a freelancer since 1979.
- Despite having turned to mainly software development soon in my carreer, I hope I still know some of the basics of electronic circuit design and digital logic.
- Therefore I found sometimes to understand the needs of hardware developers better than most of my colleagues with a background in information technology only.

1.1.2 Fields of Expertise

Though I have been exposed to much more in the past 35 years, these are the main areas where I feel competent today:

- C/C++ -- mainly by using these languages in projects and teaching their portable use according to the ISO/ANSI standard;
- Unix and Linux -- having started with a Unix-Clone named UniFLEX in the early 80s and seen much flavours of *ix before Linux took over;
- Tcl/Tk -- a scripting language which is as a nice supplement to compiled languages and also an essential part of a number of Xilinx-tools;
- But let's stop wasting time and start right away ...

1.1.2.1 Teaching in Internet Times ...

- ... all information is at your finger-tips.
- So why bother to take a course (like this)?

Seriously!

- What Do You expect?
- (Shall I leave now, before we actually started?)

NO (I hope :-)

1.1.2.2 An Analogy Used by P.J. Plauger

Let me draw a bit on a picture used many years ago by P.J. Plauger who compared learning and working with C++ with traveling a new, wild country.

- For your first steps it is best to go guided by someone who is familiar with the terrain.
- After this phase, stay close to the pathways that have been shown to you as being safe.
- From there start to take small detours in those areas you need to know better.
- The more you get confident there is no wild beast lurking in some particular dark spot, the more venturous you may go and further explore the (yet) unknwon wilderness.

1.1.2.3 Here Is My Offer

(based on the difficulties I experience when autodidactically acquiring knowledge in fields new to me)

- I mainly show to you what **may be** worth while further exploring yourself.
- Because: "just looking and listening" is OK for a while but doing things yourself is essential to really internalize what you want to learn.
- You may very well explore terrain beside the routes shown to you ...
- ... in fact, I'd really welcome this if you try out everything that looks interresting.

1.1.2.4 While Running "Self-Guided" ...

... I will come to help if:

1. The route you are about to take is a dead-end one or one much too hard to follow to reach the destination you want to get at (what I know because I tried it myself in the past but never succeeded);
2. You feel you have hit some wall that is seemingly hard to break but I know (because I once was at this point too) that it's not of stone but of a soft material and it is worth while to continue your effort.

1.1.2.5 What I Personally Find Challenging

- Not knowing how - in detail - how we will spent the next five days
- Well, there's the agenda I prepared for us, but ...
- ... you may well just take it as preliminary draft.

1.1.2.6 In Detail, PLEASE Take Control About the Core Themes Yourself

- What is, according to your expectations and requirements, sufficiently covered with an overview.
- Where we should stay a bit longer and have a closer examination.
- What might be added spontaneously because it turned out to be of special interest.

1.1.2.7 Dynamically Shaping the Course Content is of Course Limited

By:

- Topics added in passing should match really common interrest or at least correspond to the large majority's wish.
- I'm willing to add topics even if they are close to the borders of my own experience ...

- ... but it makes no sense if I, myself, have to lean out of the window too far.

1.1.2.8 As A Final Note

I don't like to teach facts if I can teach you how to find out!

- Well: yes, some knowledge is simply factual ...
- ... but stop adding facts to your knowledge ...
- ... if you start to recognize a pattern.

Rather explore and finally understand the pattern.

At the basic foundation many things are dead simple

- Try to understand the simple core modell first.
- Then refine where necessary to get the details.

Well, to get a first foot in between the door and its frame you have to over-simplify sometimes ... a little ...

1.2 A Tour of C++

This is the title of small book handed out in addition to the printed training documents.

Its author is Bjarne Stroustrup who originally designed C++ more than 30 years ago and since then until today is involved in improving the language and its standard library.

We will go through the relevant chapters of this book prior to covering the language features mainly by examples.

For brevity references to this book will be named *AToC* followed by the section number.

1.3 C as Foundation of C++

Here is the quick outline of our first lesson:

- Main Function and Subroutines
- Basic Types and Conversions
- Operations for Basic Types
- Arrays, Pointers and Adress Arithmetic
- References vs. Pointers
- Scope and Lifetime

1.3.1 The Very Basics

Maybe you know most of this stuff.

- I will go through rather it quickly.

- Please stop me if I'm moving too fast.

Feel free to interrupt me with your questions at any time!

But be aware ...

- If it gets annoying you will be sent out of the room ...
- ... no, just kidding ...
- ... of course you will be banned completely for the day or even for the rest of the week if you are nerving me with your questions ...
- (Oh no, no, NO! - let's go to the next page quickly.)

To put it absolutely clear:

Questions are welcome!

Any time!!

1.3.2 One - Two - Three

1. Let's firt see what THE GRAND MASTER HILMSELF has to say about his creation in *AToC §1, §2, and §3* (page 1 to 32).
2. Then settle on an idea what's at the core of any micro processor executing a C program based on an Info-Graphic create by my humble self.
3. Finally review some examples that we - or later on you yourself - may want to compile to train your understanding of the topics covered.

1.3.2.1 Each C Programm Needs a Function `main`

This is the minimal C Program:

```
int main() {
    return 0;
}
```

For the complete example see: Introductory/empty.c

In C++ `main` is special in so far that the `return` statement may be omitted:

```
int main() {
}
```

For the complete example see: Introductory/empty.cpp.

1.3.2.2 Sub-Routines

Technically, `main` is a "sub"-routine called from the *run-time start-up module*, a small piece of code, partly written in assembly language, that prepares the execution environment for `main`.

You can roll your own sub-routines:

```
void say_hello() {
    printf("hello, anybody here?\n");
}
```

To call it from `main`:

```
int main() {
    say_hello();
}
```

Some things to note so far:

- Sub-routines are defined with
  - a return type,
  - a name and
  - an argument list.
- The return type `void` says no value is returned.
- The argument list has been empty in all examples so far.
- Also empty argument lists need to be supplied with a call.

For the complete example see: Introductory/say_hello.c

1.3.2.3 Something to try out later (if you have no other ideas)

- Omit the pair of parentheses after `say_hello` when calling the function in the last example.
- Compile the program - will you get an error message?
- Run the program - will you see any output?
- Explain!

1.3.2.4 Function Arguments

Frequently a subroutine is a bit more useful if it does not always the same - which is the reason why there are arguments:

```
void greet(const char message[]) {
    printf("%s\n", message);
}

int main() {
    greet("hello?");
```

```
    printf("... hmm, anybody there ...\n");
    greet("bye, bye!");
    /* and tip-toeing out of the door */
}
```

For the complete example see: Introductory/func_args.c

1.3.2.5 Something to try out later (if you have no other ideas)

- Did you actually compile the program?
- Did you use `gcc` or `g++` for compiling?
- What happens if you enable more warnings (add `-Wall`)?
- Why is this different in C and C++.
- How can you make both compilers happy?
- What happens if you remove the `#include`-line?

1.3.2.6 May I Add a Comment Here?

```
/* this is a comment in C,
   it may extend over several
   lines ...

   ... including empty ones
*/
```

(Stolen from PL/1, ehh?)

```
// and this comment style was
// added by C++ initially and
// was later adopted by C too
```

(Formerly call *C++ Comment Style* but now common-place in C too.)

1.3.2.7 The Argument List in More Detail

Don't worry if you don't completely understand what's going inside the argument list (inside parentheses) ...

- ... we don't understand either - OK, kidding again, of course we know:
- It is a *name* (`message` above)
- and a *type* (array of `char`)
- (Types will be covered later, or at least seen in many examples, so don't worry too much now, you will have plenty of opportunities to ask questions if anything is unclear.)

1.3.2.8 Recursion

Functions may call themselves, what is called recursion.

- Of course this must be limited somehow, otherwise the program would never come to it's end!

- Each recursive invocation works with its own set of arguments.

Here is a famous example, calculating the factorial according to its recursive definition:

```
unsigned long factorial(unsigned long value) {
    if (value == 0) return 1;
    else return n*factorial(n-1);
}
```

And yes, mathematically 0! is 1, see http://en.wikipedia.org/wiki/Factorial.

For the complete example see Introductory/factorial.c.

1.3.2.9 Something to try out later (if you have no other ideas)

- In the complete example determine which calls exactly happen if there are no command line arguments and if there are some.
- Why in the first case the argument is `5uL` and not simply `5`?
- What happens if you change `uL` to `UL`, or `LU`, or `lu` ...?
- What happens if you remove it completely?
- What happens if you do similar changes to the `lu` in the `printf`?
- What does the program calculate for command line arguments that are not numbers?

1.3.2.10 A Famous Recursion Test

The following function is famous for its deep recursion:

```
int ackermann(int m, int n) {
    if (m == 0) return n + 1;
    else if (n == 0) return ackermann(m - 1, 1);
    else return ackermann(m - 1, ackermann(m, n - 1));
}
```

(For the mathematical background see http://en.wikipedia.org/wiki/Ackermann_function.

Looking rather innocent, isn't it?

The following will yield `125` ...

```
int main() {
    printf("%d\n", ackermann(3, 4));
}
```

1.3.2.11 Something to try out later (if you have no other ideas)

Be cautious to try `ackermann` with a first argument of `4` or beyond!

Well, what you might try is:

- m=4 and n=0 (returns quickly) but
- m=4 and n=1 (already requires some patience)

You may also want to try compiling with some optimisation option like `-O2` ...

(And here is an article delving a little deeper in into this topic.)

1.3.2.12 Library Functions

Maybe you have already noticed or guessed:

- `printf` is a function too!
- But none defined by the progam itself.
- Instead it comes from the *C Standard Library*.
- More on this later.

1.3.3 Decide How to Continue

Now its time to decide:

- do you want to have a short pause
- and **in addition** some time to go through the topics once more on your own,
- or shall we rather continue together?

May be after a pause?

Of how long?

1.4 The *Edit Compile Link and Run*-Cycle

Or rather: *Edit Compile Link and* **Test** !!

Here again a quick outline of our next lesson:

- What is a Compilation Unit?
- Why Header Files?
- Object Files and the Linker
- Libraries
- Static vs. Dynamic Linking
- Dependencies and Makefiles

1.4.1 What is a Compilation Unit?

Of course, as the name says:

- A unit of code that can be compiled.
- Or rather: the *smallest* unit of code that can be compiled a single unit.

Technically it consists of:

- Data definitions and
- Function definitions

Both contribute to the initial content of the data and code memory.

1.4.1.1 Compilation Unit Examples

A (rather small) compilation unit might look like this:

```
// This is a compilation unit with a data definition
const char greet[] = "hello, world";
```

Or like this:

```
int ackermann(int m, int n) {
    // you have seen this, we won't repeat ourselfs
}

unsigned long factorial(unsigned long n) {
    auto result = 1uL;
    while (n > 1) result *= n--; // uumpf? sic!
    return result;
}
```

1.4.1.2 Something to try out later (if you have no other ideas)

Is it possible to compile some completely empty file (given it has a suffix the compiler likes)?

- Will it work with both, `gcc` and `g++`?
- What will `nm` tell you about it?
- What `objdump`?

Yes, yes: `objdump` called with just a filename gives you an error message.

And no, no, NO, NO!! We are not going to explain `objdump` here in more detail! Get used to *read the fine manual*.

(If you force someone to answer *RTFM* too often to your questions, he might feel inclined to substitute a different F-word!)

1.4.2 Why Header Files?

Frequently the need arises to put one and the same information in more than one compilation unit.

- Duplicating such code manually is ... Pfui Teufel!
- To *Copy and Paste* it is little better.

Will you really remember all the places where you once inserted some piece of code verbatim whenever this piece needs to be changed?

Really? Consistently?!

The preprocessor with its `#include` mechanism will help you out.

1.4.2.1 Sharing Information with Help of the Preprocessor

The most common thing to share between compilation units are data and function definitions (using `defs.h` as an example below).

```
#include "defs.h"
// ...
```

Because definitions are usually required before any other code, `#include`-statements typically come first - at the *head* of the compilation units and hence the files named here are called header files (or headers in short).

To remind you of their nature `.h` is the suffix commonly used for header files.

1.4.2.2 WARNING: Sloppy Wording Has Just Crossed the Road

Oops - just noticed I use my wording a bit sloppily!

Technically it's more declarations that go into the header file, though, with respect to `struct`-s or `class`-es - both covered later - some call these definitions too :-(

Let's agree on the following: In a header file usually goes everything that doesn't actually cause content of code or data segment being created.

(Again this is not quite true with respect to templates, covered later.)

1.4.2.3 Header File Examples

If the `ackermann` and `factorial` functions were to be called from more than one compilation unit, the following header file content would be appropriate:

```
int ackermann(int m, int n);
unsigned long factorial(unsigned long n);
```

The two lines shown above are called *function prototypes*.

- They contain the *signature* but not the implementation.
- Argment names are optional.

```
int ackermann(int, int);
unsigned long factorial(unsigned long);
```

1.4.2.4 Header File Include Guards

As header files often depend on other header files, their content must be protected against being processed twice.

This is the standard technique:

```
#ifndef UTILITY_FUNCTIONS_H
#define UTILITY_FUNCTIONS_H

    // actual content of utility function header file
    // ...
```

```
#endif
```

1.4.2.5 Something to try out later (if you have no other ideas)

What should never go into a header file is:

- actual data definitions and/or
- function definitions together with their body (= implementation)
- **except** for functions that are marked `inline` (discussed later).

Don't believe? Just try it out by including the header containing

```
int foo() {
    return 42;
}
```

in two compilation units and try to link the resulting object files in the same executable!

1.4.3 Object Files and the Linker

Creating an executable program usually involves two main steps:

- Compiling and
- Linking

Both can be run independantly with the same command that has already been used for full compilation and linking: `gcc` or `g++`

Put differently:

The above commands are just call wrappers and there are command line options for a more detailed control which parts of the tool-chain will actually be executed.

RTFM!

1.4.3.1 Compiling Source Files to Object Modules

The command line switch `-c` (lower case) makes the call wrapper stop after creating the object file.

- `gcc -c somefile.c`
  - assumes C source code in `somefile.c`
  - creates object module in `somefile.o`
- `g++ -c somefile.cpp`
  - assumes C++ source code in `somefile.cpp`
  - creates object module in `somefile.o`

The object file has the name of the source file with the suffic changed to `.o` (for operating systems in the Unix/Linux tradition).

1.4.3.2 Something to try out later (if you have no other ideas)

You can also stop the call wrapper after other steps:

- Using the option `-E` does only the preprocessing step:
  - Output is sent to standard output i.e. (the terminal screen)
  - You may want to pipe it through a pager (`more` or `less`)

Have an `#include` directive in the compiled source code and note how the included file content appears in the output.

If you are at it, you may want to explore two other preprocessor mechanisms:

- Macro Replacement (`#define XYZ whatever`)
- Conditional compilation (`#if` - `#else` - `#endif`)

1.4.3.3 Something else to try out later (if you have no other ideas)

- Using the option `-S` stops before assembling:
  - Output is in a file with the suffix `.s`
  - It's a text file - you can view it in an editor
  - But you probably need some basic of assembler programming to really understand it

Are you are interrested in what assembler code results from which source and also want to try different compilers?

Look here: `http://gcc.godbolt.org`!

1.4.3.4 Linking Object Files to Executables

This starts with (one or more) object file(s):

- `gcc somefile.o`
  - assumes object module in somefile.o
  - links against the C standard library and
  - uses the C runtime start up
- `g++ somefile.o`
  - assumes object module in somefile.o
  - links against the C++ standard library and
  - uses the C++ runtime startup

So, what makes the difference here, if the file has the same name in both examples (well, just testing if you nodded off).

1.4.3.5 Other Tools Applying to Object Modules

While it may seem the only use of object modules is to name them to the linker, there are a few other commands of interrest on some occasions.

- `size` show memory space requirements
- `nm` show defined and expected references

Both commands can also be used on executable programs, for which they show total size requirements and how the linker resolved and finally assigned unique addresses to named symbols.

Want more details?

RTFM!

1.4.3.6 Something to try out later (if you have no other ideas)

Though not strictly C/C++ programming in the narrow sense of learning the syntax of a language and maybe some library functions, playing with separate compilation of small object modules and using `nm` to look at the symbols and how they are resolved can be enlightening.

When there is time reiterate this chapter once more by yourself and acquire some practical skills, you will seen have more examples with separate compilation on which you may try this.

Again, it's not strictly C/C++ but it may well give you the certainty to have fully understand the purpose of the Linker tool and how it combines object modules into executable programs.

1.4.3.7 Runtime Startup Module for C

The runtime startup module calls the `main` function in its middle as if it were an ordinary function - there is nothing special a C compiler assumes about it:

1. Command line arguments are taken from the operating system and
2. prepared to be handed over to `main` in the expected structure.
3. Standard `FILE`s (`stdin`, `stdout` and `stderr`) are prepared.
4. The `main` function is actually called (just as an ordinary function)
5. After `main` returned, its return value is saved to be finally handed over to the operating system in the last step.
6. **All still open `FILE`-s** are closed - not only the standard channels - which especially flushes yet unwritten buffers.
7. The process is terminated, handing over the return value of `main`.

1.4.3.8 Something to try out later (for the real adventurous souls)

Try to find the code for the runtime startup module linked to C program.

Disassemble!

For Unix/Linux it is probably not really complex.

But you might e.g. try to find out how all still open files are finally closed.

1.4.3.9 Runtime Startup Module for C++

The additional steps for C++ surround the call to `main`:

- before `main` starts constructors of global objects are called

- after `main` returned destructors of global objects are called

Don't get too confused by this constructor/destructor mumbo-jumbo. It will be covered later - for now you don't need to know.

(And you also don't need to know for now that all the d'tors of block local `static` objects are called too and the order of such calls is exactly the reverse of c'tor calls.)

1.4.3.10 Bare Metal Embedded Applications

C applications running embedded without support of an operating system also need a runtime startup module.

It often does less than a runtime startup module for a hosted application in that:

- None of the standard channels are prepared to be used.
- No command line arguments are handed over to `main`.
- In fact, the first function may have a different name and
- typically never to returns, i.e. runs in an endless loop.

So the runtime startup module may be quite simple as shown in this Example for Zynq/Zedboard.

1.4.4 Libraries

Many of what a C/C++ program has to do is repetitive in the sense that a set of (decently parametrized) subroutines would be useful in a large number of programs.

A library function already known because it had been used in example is `printf`.

Such are typically put into a library, which can come in two flavours:

- Static Libraries - more or less a collection of object modules put into a single file.
- Dynamic Libraries - a pre-linked unit of code, similar to an executable but running on behalf of another application.

1.4.4.1 Static Libraries

A static library is a single file, containing any object modules and typically some kind of index to simplify lookup.

For Unix/Linux thefile name suffix of static libraries is usually `.a` and the content is managed with the *Librarian* tool `ar`.

- `ar t` ... list content
- `ar c` ... create a new library
- `ar r` ... replace object module(s)

The above is only meant as example, for more details have a look at the manual page `ar(1)`.

1.4.4.2 Dynamic Libraries

The appropriate view of a *Dynamic Library* could be that of an executable without a `main` function. It will be dynamically loaded if it's references satisfy some external references of another executable.

Dynamic Libraries (Windows: DLLs, Unix/Linux Shared Objectes) are closely interweaved with support from an operating system that will load them on request. i.e. connect them to the address space of the process requiring them.

In a sense, for a dynamic library the operating system will do *just in time* what static linking otherwise does ahead of time.

1.4.5 Linking Means Connecting References

The main objective of linking is to connect addresses between separately compiled modules, i.e.:

- Access to global variables, which ultimately in an executing program all live at their unique address, needs to be adjusted in all referrers.
- Same for called subroutines, only with regard to addresses in code memory.

1.4.5.1 External References (What is Expected)

Functions defined in other modules have to introduced via their *Prototypes*:

```
extern int foo(void);           // no args, return int
extern double sqrt(double);     // double arg, return double
extern long atol(const char *); // char pointer arg, return long
```

The keyword `extern` can be - and often is - omitted for function prototypes.

Referencing data items from other modules should always add `extern`:

```
extern unsigned int call_count;
extern char module_name[];
extern float module_version;
```

1.4.5.2 External References in C

In C the rules when and when not external references must be explicitly named are slightly more complex as in C++. The reason is the age of the language and that any later extensions carefully tried not to break backward compatibility so that legacy code would still compile without any change.

Here are some of the main differences to C++:

- Extern references for subroutines will be automatically assumed for symbols looking like a function call.
- Data references without `extern` and without an initialisation may be treated as external references and hence **not** cause name clashes across modules.

1.4.5.3 External References in C++

In C++ the rules are a bit more regular, as when the language was designed there had been no legacy code that could have been broken.

Therefore: C++ could favour a strict *definition-references modell* for linking (which in C is only optional and most often applied in a relaxed way).

Extern References must be made explicit.

Any unknown symbol used (i.e. undefined and not declared to be `extern`) causes a compilation error.

1.4.5.4 Actual Definitions (What is Offered)

So that the linker can finally construct a valid executable - or that linking an executable to a dynamic library to be finally linked on loading at runtime can succeed - there must be definitions for

- each subroutine called from some object module and
- each global data item referenced from somewhere.

1.4.5.5 Data Definitions

Definitions of data items look alike their counterparts introducing external references but omit the keyword `extern` and may optionally pecify give an initial value.

```
unsinged int call_count;
char module_name[] = "MyModule Version";
float module_version = 0.9;
```

In C++ `extern` can be combined with an initialisation which turns the whole construct into a definition.

In fact, in C++ this combination is even to recommend for global constants which otherwise would not be visible accross the boundaries of a compilation unit.

1.4.5.6 What we - together - could do later (if we have no other ideas)

I can show you many of the subtleties and quirks when linking, where the C rules are different in detail from C++, etc.

But I doubt it pays (now) or rather: if you have any strange problems when linking under Unix/Linux with either

- unresolved externals or
- name clashes

just send me mail, preferrably with a stripped-down (smallest) failing case and I'll try to explain the details based on the concrete problem you ran into.

1.4.5.7 Subroutine Definitions

Subroutine definitions are very easily to recognize as they need to include the implementation of the

function in a block of code following where the declaration (prototype) ends with a colon.

```
int foo() { return 42; }

long atol(const char *s) {
    long result = 0;
    const int minus = (*s == '-');
    if (minus) ++s;
        while ('0' <= *s && *s <= '9')
            result = 10*result - (*s++ - '0');
    return !minus ? -result : result;
}
```

1.4.5.8 Something to look at later (if you have no other ideas)

YOU: Am I expected to understand the example on the last page ... well, maybe I understand that `foo` returns the value `42` each time it's called, but `atol`?

ME: You need not understand it fully right now, but as you already know a bit about C you might review the code and try to analyse what it does.

YOU: Well, maybe I have an idea, but tell: What's this subtracting `'0'` from a character, to which the expression `*s` boils down?

ME: The *printable digits* in the machine character set are usually not represented with a bit pattern that means 0, 1, 2 etc. if used in a numeric context. Instead, e.g. in ASCII or ISO-8859 character set variants, the code points for the printable digits are 48, 49 etc.

YOU: So you would have to subtract `48` ...

ME: Yes and specifying exactly this value as `'0'` makes the program a bit more robust in case you port the code to a machine with a substantially different machine character set like EBCDIC, where the printable digits are at code points 240..249.

YOU: So `'0'` is just a portable way to say `48` if compiled for a machine charater set ISO-8859-1 ...

ME: ... yes, and `240` in an EBCDIC environment, like still in use on some ancient and modern main frames.

YOU: I see, but ... doesn't the code nevertheless depend on the fact that all the printable digits have consecutive code points in the machine character set?

ME: So it is, but this is a requirement which C89, the first ANSI/ISO standard for the C laguage put it into the specifiation and afterwards C89 became part of C++98, the first ISO standard for C++.

YOU: Doesn't this limit the range of character sets that can be supported by C/C++?

ME: Not really - it was found then that even rather exotic character sets that were in use in some large and small machines from on the late 60s had the printable digits at consecutive code points.

YOU: So well, I could also re-code printable letters 'a', 'b', 'c' ... in a `char` variable to the numeric values 0, 1, 2 ... with the expression `ch -'a'`?

ME: No, at least not portably, as the letters of the (western) alphabet happen to use consecutive code points in ASCII and ISO-8859-X, they do not so in EBCDIC.

YOU: And therefore the C and C++ standards did not put such an reqirement into the language specification, as it would have limited the range of machines and operating systems were the standard can be implemented?

ME: Let's say "implemented efficiently" as there would of course be always the possibility that C/C++ maps the native machine character set to some internal coding immediately on input and output.

YOU again [after looking at the code for a long time]: Well, I now think I understand what's going on in the code: it turns a sequence of printable characters to its numerical representation in a variable of type `long`.

ME: So it is.

YOU: But say, isn't the code a bit convoluted, by doing all the calculation as a negative number?

ME: You think it would be more straight forward to do it as shown below?

```
long atol(const char *s) {
    long result = 0;
    const bool minus = (*s == '-');
    if (minus) ++s;
    while ('0' <= *s && *s <= '9'))
        result = 10*result + (*s++ - '0');
    return minus ? -result : result;
}
```

YOU: Actually I do - did you write it the other way to confuse the reader?

ME: No I didn't - it has a purpose.

YOU: Which?

ME: Guess!

YOU: Don't know - give me a clue!

ME: Ever heard about representing integral numbers in two's-complement?

YOU: Yes, but ...

ME: ... enough of a clue - ask me later if you really have no idea!

1.4.6 Static vs. Dynamic Linking

Viewn from the build process, that turns a set of object modules into a finally executable program, the difference boils down to how the libraries are named - under Linux/Unix:

- `libsomelib.a` ... names a static library
- `libsomelib.so` ... names a dynamic library

Often the difference is even a bit more subtle on Unix/Linux: To the linker only `somelib` is named - i.e. **without** the initial `lib` in its name - and it decides based on which files are actually found in the library search path based on other settings and defaults whether to chose the static or dynamic version of a library.

1.4.6.1 Something to find out later (if you are a really curious Unix/Linux guy)

There is a program Introductory/mathfunc.c which makes use of functions from the *Math Library* which are not part of the C standard library under Unix/Linux (for historic reasons).

When compiling this program you get an error unless you add `-lm` to the arguments during the linking step. What the linker will actually use is a static or dynamic library named `libm.a` or `libm.so` in some standard path compiled into the linker itself (to which of course can be added too).

Now, you may trace the system calls a program makes (RTFM: strace!) and that way could "watch" which files are tried to open during linking. If you do everything right - i.e. supply the necessary options to `strace` and maybe `grep` through the output that is generated - you will detect which `libm` is accessed ...

... and for a dynamic library you will also be abled to trace access when you run the executable program generated from `mathfunc.c`.

1.4.6.2 Static Linking in more Detail

Static linking is - simply put - the process of scanning archive files which contain a set of object modules for modules that satisfy external references of other modules.

- Some of the subtleties, like adding modules that introduce new external references, are handled automatically by modern tools, so the developer will not have to care.
- Still in some cases, like for static libraries mutually depending on each other, it will be necessary to delve a bit deeper, understand what's going on beyond the cover, and eventually change order or add some arguments in the linker command line.

1.4.6.3 Dynamic Linking in more Detail

Though it may already have become obvious:

- Linking dynamically has a first step in which dynamic libraries are named to the linker.
- During this step the linker just checks whether external references can be principially(!) satisfied with the set of libraries named.
- In the final executable entries are made which would cause a required library to be loaded when a piece of code or data in it is referenced.
- Actually connecting an executing process to a shared library is done later by the operating system, trapping accesses to unconnected external symbols that need to be satisfied.

1.4.6.4 Something to look at later (if you have no other ideas)

`man ldd` (RTFM)

1.4.6.5 Static or Dynamic Linking - Which One Should be Preferred?

The biggest advantage of dynamic linking comes into play if there is a large number of applications, all of which access the same library:

- The library code needs to be held in memory only once, being mapped by the memory management unit (MMU) to every process accessing it.
- Even if the dynamic library contains a number of subroutines never ever used by any of these applications, the former advantage may still have a large payoff.

1.4.6.6 Problems with Dynamic Linking

One of the main sources of failure are search paths, i.e. the list of directories telling the linker where it can find dynamic libraries:

- It may be named on the command line for linking, but
- at runtime it needs to be available in the process' environment.

For Unix/Linux it is essential to export the (shell) variable `LD_LIBRARY_PATH`, containing a colon-separated list of directories where dynamic libraries are placed (much like `PATH` for executables).

There are other alternatives, like putting the pathname into the executable ... so again: RTFM

1.4.6.7 Problems with Static Linking

One of the biggest advantages of static libraries compared to naming object modules explicitely to the Linker is:

- The Linker automatically pulls (or rather copies) only those object modules actually required because they satisfy some external reference of another module.
- This is done recursively, i.e. new external references introduced by an object module pulled from a library may pull more object additional modules.

It should be understood that decisions made by the Linker which modules to pull are made statically.

Hence, especially for sloppily structured object modules, subroutines (code and related global data reservations) may be added to an executable though actually never used.

1.4.6.8 Linking Programs for Embedded Targets

This is not much different from linking in general.

For *Bare Metal Applications* there is no choice but to link statically because there will be no operating system to connect a shared libray at run time.

Of course, a feature close to dynamic linking *might* be part of a statically linked function, e.g. available for your target in an advanced support library.

1.4.6.9 Historic Note on Static Libraries

You may skip this section without loosing much but if you ever wondered about the many options the Librarian tool `ar` has, or why there are other tools like `ranlib` that seem to be associated with static libraries, or why the Linker has special markers as a provisions to put libraries named on its command line into groups, the paragraphs following may help to sched some light on it.

Now it's time to decide: continue reading or skip ...

Once there was Unix in its childhood days (viewn from todays perspective) and this was at a time were resources were rather scarce, as well with respect to computing power as to storage space. To give you an idea about the latter: Early Unix systems could well live with really small hard disk (say 5 MByte), having 20 or even 40 MByte was not little but plenty.

Due to the nature of the original Unix file system each file occupied a multiple of 512 Bytes on disk, i.e. if only a few bytes had to be stored the reminder would not be usable. So, quite a lot of small files would have been a waste of a precious ressource like disk space.

As the hierarchical file system where a directory on the disk could not only contain files but sub-directories, nesting to infinite depth (as long as disk space allows), was one thing Unix was famous for, it seems the natural choice to store collections of object modules should would have been to simply use (sub-) directories which may be named to and then searched by the Linker. But because many object files, as e.g. the standard functions in the C library, were rather small (some 50..200 bytes as a typical size), this choice would have meant to waste a lot of disk space.

Therefore the *archive*-format was invented, not only for libraries but as a general solution to the *"Collection of Many Small Files"-problem. Hence the name `ar` which stands for *Archiver* (tool). It simply put many small files in a sequence, adding a short header before each that contained its name and size, allowing to do a linear search through the archive, skipping members until the desired could be copied out.

New members were easily put at the end (adding to the end of a file was an operation the Unix file system supported efficiently), while replacing members was done by marking the old version invalid and put the new to the end. (Overwriting a file byte by byte would have been available as efficient operation, but new members of an archive replacing old ones often had a different size and hence would not have fitted into the space available for the member to be replaced.)

Now, using the archive format to store object modules posed a new problem: since the main work of the linker was to look for extern references that needed to be satisfied and select object mdoules based on this, it was not unusual that by selecting one module new external references were introduced.

As already mentioned in the early days of Unix also CPU power was a scarce ressource, hence to make linking efficient it was desirable to place the object modules into the archive in a certain order, so that - in a linear search - modules introducing extern references were visited **before** the modules satisfying such references.

In other words: for best performance object modules had to be put into an archive in [tologically sorted] order with respect to their (extern) references, otherwise the Linker would have to run twice over the library. (Though the Linker didn't do this by itself but rather complained about unsatisfied references when the object modules were not accordingly sorted, it was a common solution to name a library twice on the command line in such situations.)

Since with an increasing number of object modules in a library the proper order was more and more difficult to obtain, the utility `ranlib` was made available, which put an index as first member into the library, helping the linker to do its work without making many passes over the library. So now, since several decades, the problem of having to put object modules in a certain (sequential) order in a static

library has completely vanished - as long as only a single library is considered. But it may still be present if multiple libraries are involved that are either specified to the Linker in the wrong order or mutually reference object modules inside each other.

The wrong order case will manifest itself even today if libraries *requiring* modules in other libraries are named to the Linker *after* such other libraries. The solution is - of course - to change the order in which these libraries are named to the Linker.

The other case, mutually referencing libraries, is rare in practice, because separate libraries are usually intended for separate and independant reuse and hence dependencies between libraries usually go into one direction only, i.e. dependencies between all of the libraries required in a single project usually build a directed acyclic graph (DAG).

Nevertheless there is a solution if there are such depenencies, which is to group mutually dependent libraries for the linker by placing special markers in the sequence of (library) arguments, indicating the start and the end of a group.

If the linker encounters the start of the group it first notes its still unsatisfied references. Then, if from any library inside the group a module is selected (because it satisfies an external reference of another module) and this module introduce new external references, they are added to the list.

When reaching the end-marker for the group and the initial list has received new members, the libraries in the group are scanned once more, eventually selecting more object modules from their contents. This is repeated until the list no longer grows by new entries, meaning that all yet unsatisfied external references must refer to object modules outside the - possibly rescanned - group and the linker advances to the libraries (or groups) named after the just processed group.

1.4.6.10 An Alternative View of the Unix/Linux-Kernel

The Unix/Linux-Kernel may be viewn as a dynamic library available to all (compiled) applications executing as a process.

It has a strictly controlled call interface which typically uses some special machine operations (call gates) as a means of granting additional priveledges when changing to execute in kernel mode, compared to user mode where kernel memory can not be directly accessed.

1.4.7 Dependencies and Makefiles

When compiling and linking with the final goal of producing an executable program, staying in full control of depencies between the intermediate and final products and their respective prerequisites can become hard too manage if a dozen or more files are involved.

The usual solution is to leave dependency management to a build system, with `make`, its archetype once grown under Unix`, being still in use today though typically in enhanced variants like GNU make.

1.4.7.1 Something we can do now (together) or you may do later (on your own)

For some of the programs used in this course a first understanding may be gained by reviewing the following examples:

- RandomNumbers/Makefile and
- Thermostate/Combined_SWIG/Makefile.

1.4.7.2 Something to try out at later (if you have no other ideas)

Assuming position independant code actually enlarges the memory footprint, how could the Makefile of the thermostate example be extended so that position independant code will only be generated for shared libraries?

And also: can you verify (or negative) the above assumption on code size?

1.5 Linking between C and C++ and Other Languages

The outline for this lesson:

- Calling conventions in general
- What is the purpose of `extern "C"`?
- Generating call wrappers with SWIG

1.5.1 Calling Conventions in General

To put it simple, calling conventions comprise the set of rules by which the caller of a subroutine and the subroutine itself

- communicate information between each other so that each side gets what it expects where it expects it;
- use shared resources like CPU registers so that they do not step on each others feet.

1.5.2 What is the purpose of `extern "C"`?

Especially when C++ has to cope with a tool chain that is not really aware of C++ features (i.e. featuress not present in C - like overloading based on argument types), calling conventions might differ between the two languages.

To access some function from a separate module that was compiled with C, the following may be necessary in C++:

```
extern "C" int foo(int, const char *);
```

1.5.2.1 Something to try out later (if you have no other ideas)

Have a look at the minimal code examples in Extern_C and make them to compile an link correctly.

- Manually compile `c-module.c` and `c-main.c` into the respective object module files (`c-module.o` and `c-main.o`).
- Manually link an executable `c-main` from the object modules.
- Do similar for `cpp-module.cpp` and `cpp-main.cpp`.

- How would you include the result `c-module.o` into the linking of the `cpp-main`-executable?

1.5.2.2 Something more challenging to try out at later

- Redesign the example moving the `extern` declarations to header files.
- How could you - in header file `c-module.h` - cope with the fact that for a pure C compiler `extern "C" ...` is rather a syntax error?
- Assume `void bar(double)` were also to be called from `c-main.c` - can this be achieved?
- Could you come up with a `Makefile` to control the build steps of `c-main` and `cpp-main`?

HINT (for the second step): search in some online documentation for C++ for the predefined macro `__CPLUSPLUS__`.

1.5.3 Generating call wrappers with SWIG

A rather interresting thing to know about C and C++ is the ease with which modules implemented in C may be connected with most any of the scripting languages in widespread use today (and even a large number of the more exotic ones).

This is best shown in the example in directory Thermostate/Combined_SWIG.

1.5.3.1 Something to look at later (if you have no other ideas)

If you actually consider to combine C/C++ modules with software written in a high level scripting language, some pretty good literature to start with is the SWIG-Tutorial: `http://www.swig.org/tutorial.html`.

1.6 The C Standard Library (Overview)

Outline of this lesson:

- String Operations
- Input and Output
- Mathematical Functions
- Random Numbers
- Raw Memory Access

1.6.1 String Operations

The C String storage convention is to store individual characters at adjacent memory locations with the last character having the value `'\0'`.

Refering to the string is via the address of its first character.

Copying strings, comparing strings etc. requires the use of helper functions defined in the header file `<string.h>`, like:

- `strcpy`,
- `strcmp`

A very basic use of the latter (in the form of `strncmp`) can be seen in the example program to guess Introductory/fgrep.c.

For more information about the available functions look them up in an Online Reference: `http://en.cppreference.com/w/c/string/byte`.

1.6.2 Input and Output

Input and output is done via `FILE`-s, an opaque data structure and library functions working with it.

These functions are defined in the header file `<stdio.h>`.

The basic scheme can be seen in the program Introductory/fgrep.c.

For a glimpse of formatted output see example Introductory/mathfunc.c.

For more information about the available functions look them up in an Online Reference: `http://en.cppreference.com/w/c/io`.

1.6.3 Mathematical Functions

The header file to include is `<math.h>`.

A basic example that prints a table of trigonometric functions can be found in Introductory/mathfunc.c.

For more information about the available functions look them up in a Online Reference: `http://en.cppreference.com/w/c/numeric/math`.

1.6.3.1 Something to do soon (really recommended)

Register a book mark to your favorite online reference, like

- this one `http://en.cppreference.com/w/`
- or that `http://www.cplusplus.com/reference/`

because sooner ot later you will have to stand on your own feet and cannot rely one somebody else to drop the matching clickable links.

1.6.3.2 Something to try out later (if you have no other ideas)

More closely review the programs in

- Introductory/fgrep.c and
- Introductory/mathfunc.c.

Add comments where things become clear to you (after some closer inspection) and maybe note down

your questions you want to pose when walking through the programs with the instructor later on (probably on the next day).

When actually compiling the program to print the trigonometric functions have a close look to its head comment and try to remember (or find out with help of the fine manual) why there is an `-ml` in the command line and whether this option needs to be placed exactly there and not before the source file name.

1.6.4 Random Numbers

In C there is a standard function `rand` which generates pseudo random numbers.

Its use may be very limited for practical reasons, so in C++ it was replaced with a more flexible facility.

See examples RandomNumbers/dice_rand.c and RandomNumbers/dice_random.cpp.

1.6.5 Raw Memory Access

There are some low level functions that just move around or compare bytes in memory, like:

- `memcopy`
- `memmove`
- `memcmp`

If you are interrested, you may lookup their use in the online documentation.

1.6.5.1 Something to look at later (if you have no other ideas)

- What is the challenge when copying data between overlapping memory regions?
- Which one of the library functions does it in the right way and why may the other one exist?

Can you come up with a test program that shows the problem and also compares memory footprint and performance of both versions?

1.6.6 Basic Types and Conversions

Conversion between the various arithmetic types are done automatically (i.e. the compiler adds the necessary code for type coercions at runtime) and in addition will typically flag value changing conversions with a warning.

Some practical examples may be found in several programs that have been shown so far and will be shown in future.

1.6.6.1 Something you might want to do later (if you have no other ideas)

Write some small programs that demonstrate compiler warnings - or even errors when the uniform

initialization syntax new in C++11 is used - in case of value changing or potentially value changing operations.

Furthermore, if you are really hot on this topic find out what `numeric_cast` in the Boost Library is good for.

1.6.7 Operations for Basic Types

Covered in an overview style in AToC $1.5 (from page 5).

Practical examples can be found in most any program presented during this training.

For an embedded software developer working close to the hardware operations on single bits may sometimes be of special interrest, so the next examples will highlight these.

1.6.7.1 Bit Operations - A Practical Example

The following is (nearly) an excerpt from the source Smileys/smiley_zprint1.c.

```
if (pic_zdata[(i*width + j)/8] & (1 << (k++ % 8)))
    putchar('*');
else
    putchar(' ');
```

Besides indexing, multiplication, division, modulus, and (post-) increment the following bit operators are used:

- bitwise left shift (<<)
- bitwise and (&)

1.6.7.2 Introducing Temporaries for Readability

To make the code better understandable the large expression should be broken into smaller parts that can be easily explained by comments:

```
const int effidx = (i*width + j)/8;             /* effective index */
const int bitpos = (k++ % 8);                   /* bit position */
const unsigned char mask = (1 << bitpos);       /* bit to mask out */
const int prtmark = pic_zdata[effidx] & mask;   /* print marker ? */
if (prtmark)
    putchar('*');
else
    putchar(' ');
```

It may be a matter of personal test wether you stay with rather short variable names that you explain in a comment or rather use longer names telling the purpose more clearly and go without a comment.

1.6.7.3 The Conditional Expression

Using a C specialty, the conditional expression, the output of either an asterisk or a space can be rewritten as:

```
putchar(prtmark ? '*' : ' ');
```

The semantics of this operation are simple:

- The first part (left to the question mark) is evaluated for its logic value
  - `true` (or anything that is non-zero) or
  - `false` (or anything that is zero)
- Depending on this, the overall result is
  - either the second operand between `?` and `:`
  - or the third operand after `:`

1.6.7.4 Structuring Using Spaces

Putting it all into a single expression again, at least the structure imposed by the conditional expression should be made standout a bit better by adding spaces and line breaks:

```
/* You are not expected to understand this. */
putchar(
    pic_zdata[(i*width + j)/8] & (1 << (k++ % 8))
        ? '*'
        : ' '
);
```

(I took the freedom to add a classic comment above, originating from the source code of Unix Version 6, the precursor of Unix Version 7 which once ran on a DEC PDP/11 and became the first Unix variant used outside Bell Labs in the late 70s.)

1.6.7.5 Something you might try out (if you have no other ideas)

Among software developers there is often some degree of reluctance to introduce temporaries for redability.

- Can you second that - do temporaries cause more or slower code?
- Or rather negate - i.e. such temporaries are optimised away?

1.6.7.6 Something else you might try out (if you have no other ideas)

And what about the conditional expression - what is their influence on the code generated?

To isolate the interresting part you may want to try something like this:

```
extern void bar(char);
void foo(bool b) { if (b) bar('*'); else bar(' '); }
```

And in comparision:

```
void foo(bool b) { bar(b ? '*' : ' '); }
```

Or:

```
void foo(bool b) { const char c = b ? '*' : ' '); bar(c); }
```

1.6.8 Pointers, Arrays and Address Arithmetic

Arrays decays into pointers whenever their name is used in a context other

- determining memory space with `sizeof`
- applying the *address-of* operator `&`

as demonstrated in Introductory/sizeof_subtleties.c

and Introductory/array_subtleties.c.

1.6.8.1 Arrays as Arguments

The key point to understand **when arrays are handed over via an argument list** is that the following has the same meaning:

- `int *arr`
- `int arr[]`

Putting it differently: when handing over an array its size gets lost (and hence would have to be specified via another argument to the callee).

1.6.8.2 Two Dimensional Arrays

Here the key point to understand is that the inner dimension **which is specified more to the right** will actually be important for address calculations, i.e. with

```
T data[15][25];
```

in an expression like `... data[i][j] ...` the offset to calculate from the start address (`&data[0][0]`) is:

- `25*i + j` if calculating in units of the size of a single `T`, or
- `(sizeof (T))*(25*i + j)` in a byte-based low level calculation.

1.6.8.3 Something to try yourself later ...

In example Smileys/smiley_print1.c there is a two dimensional array handed over to the `printer` function.

- Which of the two dimensions in square brackets can be dropped when handing over `pic_data` via the argument `pic`?
- What happens if you both dimensions in the parameter specification of the argument list are dropped?
- Is it possible to hand over the array `pic_data` as if it had only one dimension and leave it up to `printer` to interpret the two dimensions as it likes to?

If you don't want to try this yourself, just continue to the next section.

1.6.8.4 ... or rather to look in the solution?

The internal address calculation that is necessary for double indexing shows explicitely in Smileys/smiley_print2.c like this:

```
void printer(int pic[], int height, int width) {
    // ...
            putchar(pic[i*width + j] ? '*' : ' ');
    // ...
}
```

But due to the change of the parameter type, also the call now has to change:

```
        printer(pic_data[0], 14, 25);
```

Explain!

1.6.9 Introducing Reference Type Variables

At this point I could have inserted about a dozen pages one after the other holding each of the sections from this introductory comparison: Introductory/ptr_vs_ref.cpp

My proposal is we rather walk through the code together and later on you repeat it once more on yourself, maybe applying small modifications at points of special interrest, like enabling code that will not compile to see what error messages you get from the compiler.

But first some facts that apply universally and should help to understand differences and commonalities.

1.6.9.1 Reference vs. Pointer Type Variables

It should be noted that the difference is mainly in the syntax, not in the implementation, because technically both, a pointer and a reference, simply hold an address.

But be aware: if you do an assembly code analyses of a simple program comparing references and pointers, you might not see them at all as the compiler may have taken the chance to optimise them out. The best chance you get is when you compile for debugging (`g++ -O0 ...`).

1.6.9.2 Some Facts About Pointers

A pointer is more explicit as a reference:

- A pointer by itself denotes the address it holds and therefore must be explictely derefeneced with * to get to the content.
- To initialise (or assign to) a pointer an address must be explicitely specified and therefore, if a pointer is set to point to some variable, the address-of operator (&) has to be prefixed to the variable's name.
- Taking the address of the pointer itself gives the memory location where the pointer is stored.

1.6.9.3 Some Facts about Rereferences

A reference is more like an alias:

- If a reference is set to refer to some variable, only the name of the variable is given - though, behind the scenes, its address is taken too.
- Afterwards a reference accesses the content already in its plain use, i.e. dereferencing is always implied.
- Applying the adress-of operator (&) to a reference gives the memory location of the variable for which the reference is an alias.

1.6.9.4 And Now It's Time For the Code Walk

Remember to repeat this once more on your own!

Also note: though in this demonstration code comparing pointers to references plain variables were used for simplicity, the more realisitic and important use case for pointers and references are function arguments.

And now: Introductory/ptr_vs_ref.cpp

1.6.9.5 Swapping Variables Using References

This is the classic example to demonstrate references - a function swapping the content of two variables.

```
void swap(int &rx, int &ry) {
        const int tmp = rx;
        rx = ry;
        ry = tmp;
}
```

At the call site just the variables will have to be named:

```
int x, y;
// ...
swap(x, y);
```

1.6.9.6 Swapping Variables Using Pointers

The following is the C version with pointers:

```
void swap(int *px, int *py) {
        const int tmp = *px;
        *px = *py;
        *py = tmp;
}
```

The call site then will have to apply the adress-of operator (&) to the variables:

```
int x, y;
// ...
swap(x, y);
```

1.6.9.7 Something to try out later (if you have no other ideas)

Understand commonalities and differences between pointers and references by analysing the assembler code of Introductory/swap_ref.cpp and that of Introductory/swap_ptr.c.

1.6.9.8 Something else to try out later (if you have no other ideas)

This is only remotely related, as it has nothing to do with pointers or references but with swapping. Some people use a seemingly clever trick to swap the content of integral values:

```
void swap(int &rx, int &ry) {
        const int tmp = rx;
        rx = ry;
        ry = tmp;
}
```

A full example with a `main` program to compile to executable code can be found in Introductory/swap_xor.c.

- Discuss the pros and cons of the above (maybe based on an assembly analysis and comparison with the functions using a temporary).
- Can you also spot a possible pitfall where the above fails grossly while it goes harmless with the temporary variable variant?

1.6.9.9 Reference or Pointer - Which One to Prefer?

The following may be used as a rule of thumb:

- In code that is or may be compiled with pure C, stay with pointers.
- Otherwise prefer references, *unless*
  - During its lifetime the referrer will refer to different entities, and/or
  - you need a simple way to represent: "there's nothing to reference".

In the latter case, the pointer will have the value `nullptr` - or `0` or `NULL` - see AToC §1.8 (page 12).

Pointers will typically be tested against `nullptr` before the dereferencing operation (*) is applied!

1.6.9.10 Something advanced you might consider later (if you have no other ideas)

What do YOU think - is there also an equivalent to the `nullptr` possible with references, could a reference refer to "nothing" (i.e. "nothing valid").

- How could it happen deliberately?
- How could it happen by accident?
- How to test for this case?

But especially and most important:

- Does it make sense?

1.7 Static Type Checking and Robust Code

Data types were one of the major improvements early high level language added to coding in assembler. (The other one was structured control flow.)

- Viewn at the low level, type safety makes the compiler chose the right operations at assembly level.
- Viewn from the high level, type safety turns meaningless operations that are typically rather errors on the side of the developer into compile time errors.

1.7.1 Implicit Type Conversions

These take place between all basic arithmetic types and extend to pointers in both, C and C++, and Booleans (in C++ only).

- Any arithmetic value which is not zero converts to `true` as a Boolean or more generally, in any context where it is used as a condition.
- Any pointer that does **not** compare equal to `nullptr` (C++11 only) or `0` or `NULL` (traditional notations) is taken to be `true`.

1.7.1.1 Implicit Pointer Conversions in C

The rule is:

- Each kind of typed pointer can be assigned to an untyped pointer (`void *`).
- An untyped pointer (`void *`) can be assigned to each kind of typed pointer.

Though:

- **Differently typed pointers are never compatible with each other.**

(You see where a major hole is punched into the type system of C, giving proponents of other languages the justification to claim: "C is not strictly type-checked"?)

1.7.1.2 Implicit Pointer Conversions in C++

The rule is:

- Each kind of typed pointer can be assigned to an untyped pointer (`void *`).
- **An untyped pointer (`void *`) CAN NOT be assigned to a typed pointer.**

(C++ is closing a big hole in the type checking system by making these conversions a one-way-road.)

And also:

- Two differently typed pointers are not directly compatible with each other,
- except that a derived class is compatible with its base class(es).

More on base and derived classes follow later.

1.7.2 Explicit Type Conversions

Such are also called *Cast Operations* and their syntax is:

- A type in round parenthesis followed by an expression in C (and still C++ too because of downward compatibility).

- One out of four keywords (see next pages)
- followed by a type in angle brackets

- followed by an expression in round parenthesis.

In both forms the value of the expression is converted to the given type.

(For a rather short coverage see AToC §14.2.4 page 161.)

1.7.2.1 Conversion with `static_cast`

There are two main areas where such can be applied:

- Avoid warnings for conversions that would result from conversions the compiler applies automatically.
- Turn down-casts in class hierarchies into *no-ops*.

For the former see Introductory/warnings.c, Introductory/no_warnings.c, and Introductory/no_warnings.cpp.

The latter will be covered later.

1.7.2.2 Something to try out later (if you have no other ideas)

The following is not directly related to conversion warnings but may be of interest too:

Look up what other warnings `gcc` and `g++` are able to issue with

- `gcc --help=warning` or
- `g++ --help=warning`.

Can you get `gcc` and `g++` to warn about unused variables and arguments and missing return values?

1.7.2.3 Something else to try out later (if you have no other ideas)

What happens when you use the new C++-style initialisation for the variables, i.e.

```
short a{y}; // instead of: short a = y;
int b{3.14};    // instead of: int b = 3.14;
```

etc. and compile with `g++ -std=c++0x ...`?

1.7.2.4 Conversion with `const_cast`

Temporarily remove compile time protection from non-modifiable entities.

This may sometimes seem to be required but may have strange effects.

In other words, you might write code like this (for whatever purpose):

```
int main() {
    const int answer_to_the_ultimate_question_of_life = 42;
    // ...
    const_cast<int &>(answer_to_the_ultimate_question_of_life) = 43;
    // ...
}
```

The full demo is in Introductory/const_cast.cpp.

1.7.2.5 What we together could do now (or you can try on your own later)

Compile and run the demo program, find an explanation for its partially strange behavior, and confirm these by trying further modifications.

The general rule is (more or less):

- As long as something originally non-const is handed over unmodifiable, you may succeed casting away const and actually see or not see the modifications made.
- As it is hard to find a precise wording for this "may or may not" and casting away constness may even crash the program, the ISO standard simply turned all `const_cast`-ing into be *undefined behaviour*.

With that, the compiler has any freedom to enforce constness or give leeway.

1.7.2.6 Conversion with `reinterpret_cast`

Such allow to turn anything into anything and will especially be required in embedded programming to set up pointers (or references) to reach device registers for doing memory-mapped I/O.

```
volatile struct uart {
    unsigned char in;
    unsigned char out;
    unsigned short ctl;
} *serial_port = reinterpret_cast<struct uart *>(0xAFFE);
// ...
serial_port->out = 'z'; // send character
while (!serial_port.ctl & (1 << 12))
    ; // wait for send completes
```

(... assuming bit 13 in the control register is some kind of ready indicator ...)

1.7.2.7 Conversion with `dynamic_cast`

The main area where this is applied is with down-casts in class hierarchies that do some checking (and

avoid complete havoc on failure). It will therefore be covered later together with *runtime type identification* (aka. RTTI).

1.7.3 Type-safe I/O in C++

See AToC §8 (from page 85)

1.7.4 Other Areas Improved by C++

Besides the big "add on"-s that will receive chapters of their own there is a number of small areas where C++ has been improved over C, most of which would be nice to have available for C programming too ...

... so they may give some justification to compile "purce C" with `g++`.

- Function overloading and default arguments.
- Range-based syntax for `for`-loops.
- Narrower scope and selectable representation for enumerations.
- Type-safe allocation and better alignment control for heap memory.
- ...

(just to name a few)

1.7.4.1 Default Arguments

This is a quite simple feature, e.g. a function that throws a dice could take defaults arguments for the range 1..6 but allow other values too:

```
int dice(int from = 1, int to = 6) {
    return from + rand() % (to - from + 1);
}
```

If the above were just a prototype declaration, it could be written

```
int dice(int from = 1, int to = 6);
```

as well as:

```
int dice(int = 1, int = 6);
```

1.7.4.2 Something you may want to think about later (or even now)

Why do default values have to go with the prototype declaration and not with the implementation?

Furthermore: Given that with the parametrization shown the following calls are possible

```
dice();     // throws 1..6
dice(3);    // throws 3..6
dice(0, 1); // throws 0..1
```

why would the order of arguments better be changed if the most typical use case were to vary the upper

limit only leaving the lower limit at 1?

1.7.4.3 Function Overloading

While you can specify arguments of completly different types

```
void print(const char *name, int value);
void print(const char *name, double value);
```

or different length argument lists

```
int dice(int from, int to);  // see above
int dice(int to) { return dice(1, to); }
```

the more interresting part is how constness and references attribute to the mechanism.

1.7.4.4 Something to try out later (if you have no other ideas)

Go through the example Introductory/overload_details.cpp to understand which of several overloaded functions is selected in each case.

Test your understanding by slightly modifying the example to leave out some of the overloads or add other possibilities.

1.8 Major Steps in the Evolution of C++

The main purpose of this section is to introduce the terminolgy used to denote some major versions and extensions of the C++ language and to understand how these parts relate to each other.

See AToC $14.1.1 (from page 154).

1.8.1 From a "de facto Standard" to C++98

See AToC $14.1.2 (from page 155).

1.8.2 Informal Extension by TR1

Besides official ISO standards there have sometimes been inofficial extensions like TR1 or documents dealing with special areas like the Performance of C++, the latter having a special impact on embedded programming.

1.8.3 The Role of the Boost Platform

In the finalizing phase of the first ISO standard for C++ a number of comitee members as well as interrested persons from outside founded the *Boost Group* which since then has stimulated the development of many extensions libraries which are publicly available at: http://www.boost.org

Those with a very common applicability have often been a model for a similar or even identical feature finally standardised with C++11.

1.8.4 Adopting the C++11 Standard

Now, nearly three years after the new C++ standard (formerly known as C++0x) has been finished, and about five years after this new version of C++ has been feature complete, still not all compilers offer all of the language and library.

An overview can be found here: `https://wiki.apache.org/stdcxx/C++0xCompilerSupport`

1.8.5 Future Directions

Working groups have already been formed to discuss future directions of C++.

- What is currently named C++14 will rather cleanup some minor issues, not again add major features to the language.
- A major release is currently planned under the name C++17.

1.9 First Stations on the Road from C to C++

Again for a first overview and introduction to the topics that follow let's look at AToC §4 (from page 33).

1.9.1 Structuring Data

Purely combining related data in a struct is the first step like shown in Thermostate/Using_a_struct/thermostate.h and Thermostate/Using_a_struct/thermostate.c

Note that this can still be compiled as pure C.

1.9.1.1 Something to complete later (if you want to)

Complete this example with respect to the other parts that have been once available in Thermostate/Using_a_struct.

1.9.2 Encapsulating Data and Operations

Turning the thermostate example into a class will be done as a live demo.

For applying the initial idea see: Thermostate/Using_class_v1

1.9.3 User defined Types and Type Safety

Classes constitue user defined types and as such should make available all the operations they support.

Have the object instance do for you what needs to be done - don't handle its data from the outside!

For applying more C++ features see: Thermostate/Using_class_v2

1.9.3.1 Something to complete later (if you want to)

Complete this example with respect to the other parts that have been once available in

- Thermostate/Using_a_class_v1 or
- Thermostate/Using_a_class_v2.

(Depending on the steps saved doing the live demo there may be more intermediate versions - so if you particular diligent this afternoon you may complete all of them).

1.9.4 Const-based Overloading of Member Functions

In addition to the overloading possibilities for plain functions, also the `const` after the member function name is part of the signature.

So you can do the following:

```
class MyVector {
    // ...
    double &at(int index) { ... }
    const double &at(int index) const { ... }
};
```

(If the meaning of this keyword at that position did escape your attention, see AToC §4.2.1 page 35.)

1.9.5 Member Data and Operations

Data and operations encapsulated inside a class are individual for each object (instance) of this class.

- The code generated to access some data member has to use the object address as base (adding a member specific offset).
- Any member function receives the object (instance) address as additional argument.

Of course, the usual opitmizations may take place - especially such involved with inlining.

(If you have forgotten about inlining see AToC §4.2.1 page 35.)

1.9.5.1 Something to try out later (if you have no other ideas)

Run an assembly analyses over some code generated on behalf of structs or classes and their members (data and functions) and observe the effects of inlining.

1.9.6 Class Data and Operations

C++ provides an easy means to associate data not with objects (instances) but with their class by adding the keyword `static`.

- In effect this introduces just scoped global data and
- scoped global functions (with special access rights).

No, you didn't miss anything so far, this is NOT covered in AToC.

As a substitute here is a link to that feature in IBM's compiler documentation

---

1.9.6.1 Short Insertion About the IBM Documentation

The documentation linked on the last page is very well and good structured if your C/C++ skills are already beyond the beginner's level. I would have set links to this site more often but as the link policiy - when links are allowed and when forbidden - is not quite obvious to me and especially as I do not want to get into an argument with IBM's lawyers I will leave it with this one link.

Of course you are free to use the IBM online documentation as often as you like and I even recommend it if you think that a specific language feature might become clearer by looking it up there.

1.9.7 Information Hiding / Access Control

To really enforce member data access through member functions only, access can be controlled with labels inserted between the members:

- `private:` - only accessible for mebers and friends
- `protected:` - in addition accessible for derived classes
- `public:` - in addition accessible from everywhere

1.10 Summarizing So Far and More about Classes

Here is another Info-Graphic that may be useful to summarize what has been covered so far and prepare for more.

1.10.1 Base-Classes and Overriding

The different meaning of *overriding* and *overloading* should be noted:

- Overloading is provding several global or member functions with identical names, that can be distinguished by their parameterization.
- Overriding is providing a member function in a derived class that has the same name as a member function that already exists in the base class.

A basic example, only demonstracting the feature and some of its possible pitfalls will be given as live demo from the code prepared in: Introductory/overriding.cpp

---

1.11 Automatically Generated Member Functions

C++ will generate default versions for a number of member functions.

Such are basically demonstrated in the following code examples:

- Introductory/default_ctor.cpp
- Introductory/copy_ctor.cpp
- Introductory/move_ctor.cpp
- Introductory/copy_assign.cpp
- Introductory/move_assign.cpp

1.11.1 Default-, Copy-, and Move-Constructor

These are the rules in more detail:

- A default constructor is automatically provided *if no other constructor exists* and will recursively call default constructors for member data of class type and **leaves member data of basic type unintialized**.
- A copy constructor is automatically provided if no copy constructur is supplied and will do a **member-wise copy** (using assembler level moves for basic types).
- A move constructor is automatically provided if no move constructor and no copy constructur is supplied and will do a **member-wise move** (using assembler level moves for basic types).

1.11.2 Copy- and Move-Assignment

These are the rules in more detail:

- A default copy assignment provided if no copy assignment is supplied and will do a **member-wise assignment** (using assembler level moves for basic types).
- A default move assignmemnt is automatically provided if no move assignment and no copy assignment is supplied and will do a **member-wise move** (using assembler level moves for basic types).

1.11.3 Cleanup (Destructors)

These are the rules in more detail:

- A destructor is automatically provided if none is supplied and will recursively call destructors for member data.

- **For base classes the destructor is automatically called when the destructor for the derived class has finished.**

The latter does not depend on the origin of the derived class destructor: it may have been explicitely supplied or automatically generated.

1.12 So far the topics planned for Monday ...

... and for the rest of the day you may want to review the language features covered so far once more for yourself and - maybe - go into more depth with topics of special interrest to you.

If you have no ideas yourself or the suggestions made so far are insufficient or not what you like to do you may also continue with the proposals made from here:

1.12.1 The Smiley Example Series

This series contains a total of seven source files which you may best inspect in the order given following.

To summarize:

- Four of the examples print a small ASCII graphic from a data structure.
- The other three help to prepare the data structure from a text file.

1.12.1.1 The C Version of the Smiley Printer

Start with this: Smileys/smiley_print1.c

- Compile and run and - taking the source file name as a clue - compare the output to your expectations.
- Review the code and understand how it works.
- Could you have guessed the output from the source without actually compiling and running?

(If you train your eyes a bit you might actually see the smiley shining through the `0`s and `1`s used for initialisation.)

And if there are any questions, ask me.

That's the whole purpose I'm still here now!

1.12.1.2 Another C Version of the Smiley Printer

Continue with this: Smileys/smiley_print2.c

- Compile and run this too.
- Review the code and understand how it works.

(And if there are any questions, ask me.)

1.12.1.3 Preparing `pic_data` for C Version

The initialised data array from which the smiley is generated in these examples could have been prepared manually.

But of course you don't think I did it that way ... or am I looking such insane?

For the C versions shown so far it has been prepared with this program: Smileys/smiley_get.c

Review its code now.

(And yes, I'm still here in case you have any questions.)

1.12.1.4 A C++ version of the printer

Continue with this: Smileys/smiley_print.cpp

- Compile and run this too.
- Review the code and understand how it works.
- Could you have guessed the output from the source without actually compiling and running?

(And if there are any questions ...)

1.12.1.5 Preparing `pic_data` for C++ Version

The initialised data for C++ is much similar except the data type used here as base type is `bool`, not `int`.

- There are some more differences if you review the code in Smileys/smiley_get.cpp now and compare it to the former.
- What might be more interesting is whether changing the array type from `int` to `bool` will safe memory.

Any idea how you could find out?

(... yes, I'm still here ...)

1.12.1.6 Storing the Data in a More Compact Form

Now continue with the program Smileys/smiley_zprint.c.

The data array uses a single bit only for each asterisk to print, so there should really be some memory safed.

- Can you find out how much?
- Could you have guessed the output without running the program?

(With respect to the last question: you might of course base your guess on the source file name but if you are really see a happy face "shining through" **from the initialisation data** you are a true hero in my eyes ... or just boldly hamming up.)

1.12.1.7 Preparing the Data in Compact Form

The helper to generate the initialized data is this: Smileys/smiley_getz.c.

Yes, it's structure could be improved by putting some parts of it into subroutines and separate concerns like generating the bytes from bits and generating nicely formatted output.

I wouldn't like to make it too complicate for this first run, but if you want to try your hands on it, feel

free to do it now.

You may also rework some of the other examples, or ...

1.12.2 ... Now for Something Completely Different

If you rather want to try your hands on `make`, this is your chance.

That's the idea:

- The programs to generate the initialised data might well leave their result in a file ...
- ... which the programs requiring such data array simply `#include`.

In this situation it would make sense to automatically re-generate the included files in case Smileys/smiley.txt changes and also trigger a recompilation of the various printers.

Furthermore, as changes in the generators might be important too, the generated products should have the appropriate dependencies.

---

2 Tuesday

2.1 Overloading Operators

Overloading operators was one of the early features C++.

In the 80s it was a popular field, especially for new programming languages, though today it is not seen without criticism and some languages (like Java) openly argue against it.

2.1.1 Overloading with Member Functions

The first possibility is to define a member function inside the class that constitutes the left hand side operand.

- For a unary operator there is no additional explicit argument.
- For a binary operator there is **one** additional argument, constituting the right hand side operand.

For some operators this is the only possibility, like for assigmnent, indexing and function call.

2.1.2 Overloading with Global Functions

The other possibility is to define a global function (outside any class).

- For a unary operator there is **one** explicit argument.
- For a binary operator there are **two** arguments arguments.

2.1.3 Type-Safe Input/Output (Stream Insertion and Extraction)

A live example will be given with the result in Introductory/overload_io.cpp

---

2.1.4 Overloading Indexing (and Alternatives)

A live example will be given with the result in Introductory/overloading.cpp

---

2.1.5 Overloading Function Calls (Functors)

This feature allows to write classes for which the objects may be used followed by round parentheses enclosing a parameter list. i.e. instead of

```
MyClass obj;
obj.do_something(one, 2, 1/3.0);
```

you are enabled to write:

```
obj(one, 2, 1/3.0);
```

It other words: you may consider this feature to give you kind of an unnamed member function in addition to all the named member function a class can have.

Therefore this feature may seem as (too much of) syntactic sugar initially, but it has a long tradition where it is applied in the STL library.

2.1.5.1 Something you might look up in advance if you are impatient

There is an example for overloading the function call operator in AToC §5.5 (from page 64) which will be covered later with the template feature.

2.2 Exceptions

Exceptions have alread been shortly mentioned based on AToC §3.4 (from page 27).

In the following sections they will be revisited shortly.

This Info-Graphic is meant to give a brief overview of the concept.

2.2.1 Escaping from Errors

The main purpose of exceptions is to signal the inability to complete some piece of work without the risk of going unnoticed.

E.g. if reading some input with `scanf`, the return value would have to be tested explicitely, otherwise errors will go unnoticed:

```
int height, width;
```

```
scanf("%d %d", &height, &width);
```

If there is insufficient or non-numeric input the program will fail silently.

In C++ I/O-failures can be turned into execptions:

```
int height, width;
cin >> height >> width;
```

2.2.1.1 Something to try out later (if you have no other ideas)

The last two code fragments are actually excerpts from Introductory/input_demo.c and Introductory/input_demo.cpp.

Compile and run both programs and check their behavior in case of bad input (incomplete, not numeric, partially numeric ...).

Then compare to their counterparts Introductory/input_demo2.c and Introductory/input_demo2.cpp which handle errors.

Small challenge: is error detection in the C version really sufficient or can you still enter input that makes the program (silently) fail ...?

2.2.2 Non-Local Branching

Basically exceptions are kind of a non-local branching feature which always goes back in the hierarchy of (unterminated) function calls.

This is also possible in pure C but would skip the execution of destructors.

There will be an example later in Part 2 (the Linux System Programming section).

2.2.3 Best Practices

A few common pitfals with respect to execptions are summarized in this Info-Graphic.

A rather good in depth treatment with recommendations in check-list style can be found here and here.

(BTW: Personally I think there is more stuff worth reading from that author, Jack Reeves.)

2.2.3.1 Something you should not defer to next week ...

... at least if you are strongly interrested to use exceptions in your future C++ projects:

Plan some time for at least giving the above articles a cursory view so that you can ask me questions as long as I'm available here.

On the other hand: some embedded projects using C++ make an explicit decision not to use exceptions, so you may never need a deeper knowledge (and same of course if you stick with C).

2.2.4 Alternatives

Generally exceptions introduce a cost that gives a C++ program a larger memory footprint - they **DO NOT FIT WELL** to the otherwise used philosophy in the design of C++:

You only pay for what you use.

2.2.4.1 Something to try out later (if you have no other ideas)

Come up with an example demonstrating the cost of exceptions. It should be comprised of at least three separately compiled modules.

The first one should be a function that conditionally may throw an execption. You may control this via a global:

```
extern bool have_problem;
void foo() {
    // ... before
    if (have_problem)
        throw 42;
    // ... after
}
```

This function should then be called from an intermediate level function bar in a separatedly complied module:

```
extern void foo();
void bar() {
    // ... before
    foo();
    // ... after
}
```

Finally there `main` program should call `bar` and allow - e.g. controlled by a command line argument - to make `foo` throw the execption or not.

```
bool have_problem;
int main(int argc, char *argv[]) {
    have_problem = (argc > 1);
    try {
        bar();
        return 0;
    }
    catch (int err) {
        return 1;
    }
}
```

Hint: when you run the program as shown you may make it throw an execption via running it with (any) command line argument (or leave it out not to throw) and view the result code returned from `main` as the content of the shell variable `$?`.

After everything works (as expected) restructure the example (or yet better: create a second one so that you can better compare) which does basically the same but indicates that it has problems via a return value from `foo`:

```
extern bool have_problem;
```

```
bool foo() {
    // ... before
    if (have_problem)
        return false;
    // ... after
}
```

Of course, the intermediary `bar` will now have to forward the problem notification:

```
bool bar() {
    // ... before
    if (!foo())
        return false;
    // ... after
    return true;
}
```

And `main` needs to be modified test the value returned from `bar`:

```
int main(int argc, char *argv) {
    if (bar()) {
        return 0;
    }
    else {
        return 1;
    }
}
```

You may well add some lines of code printing messages at various points (e.g. at before / after) so that you can better trace which parts of your code are executed, but this of course will add to the memory footprint so be careful what you investigate. At least make sure that both versions contain exactly the same (printing) code or you that you can disable the printing code completely for memory footprint comparisons.

2.3 Concrete vs. Abstract Types

See AToC §4 (from page 33).

2.3.1 From Data-Containers to Services

This general route has already been mentioned:

- Avoid the data centric view with separate functions working **on** some pieces of otherwise "dead" data.
- Instead package data and functions with each other and define **services** an object can or needs to offer to its clients.

2.3.1.1 A Very Personal Comment

Well, well, I used this high level "Object Oriented" way of speaking so long for many years that it still comes fluently, but with time my view has shifted:

Any somewhat experienced software developer who follows some natural core principles like

- structuring along logical separations of concerns,
- avoiding repetition, striving for reuse,
- hardening software where it should be robust,
- but make it adaptable where it helps to support different use cases

will soon see when and how encapsulation into objects comes in handy.

Teaching Object Orientation by abstract principles and rules to follow in a guideline style - at least to my experience - has seldom improved software quality in a project.

To me it seems far better to have the experienced developers guide the newcomers so that the latter could adopt good style and best practices from conrete models, not from abstract rules.

2.3.2 Invariants, Pre- and Post-Conditions

- An invarant is a condition that is always true throughout the lifetime of some object.
- A precondition is something that must be fullfilled prior to some operation, like calling a certain member function.
- A postcondition is something a client that uses some obkect (instance) can rely on at some point.

All of the above are more or less theoretical concepts that manifest themselves mostly in comments or reference documentation.

2.3.3 Virtual Member Functions

This Info-Graphic explains the implementation on its left half.

Typical usages will be shown in later examples.

(The right half of this Info-Graphic will not be not be covered right now but it may get a short coverage in a later example, based on your request.)

2.3.4 Parameterizing Interfaces

Interfaces may be compared to a contract which two parties agree on. They can help much to separate concerns, make software more modular and testable and hence have a lot of positive appraisal.

2.3.4.1 Interfaces in Object Oriented Style

In object oriented languages like C++ they are often seen as a special case of abstract classes, which is in fact *one* way an interface may be realized and has in fact lots of advantages.

2.3.4.2 Interfaces Viewn More Generally

From a more general viewpoint interfaces may also be parametrized via template arguments (called policies then and covered later) and viewn from the technical side an object file with a given set of entry points and global data definitions offers an interface which some other object file may use.

2.4 Generalization vs. Composition

This is the part that has so far not been covered in Info-Graphic.

2.4.1 Public Base Classes

With a public base the "is a" relationship between two class is formalized.

```
class Derived : public Base {
    // ...
};
```

- Class `Derived` inherits everything that is public in `Base`
- and can of course add its own members (data as well as functions).

2.4.2 Liskov Substitution Principles

The one thing that makes public inheritance special is the LSP named after Barbara Liskov:

- A derived class should always be usable as replacement for one of its bases.
- This can be implemented in C++ at no cost (for single inheritance).

The LSP limits in important ways to which degree a derived class may replace the implementation of some inherited (public) member function.

2.4.2.1 Limits for Overriding Inherited Member Functions

A Derived Class Must Never Require Stronger Preconditions:

Otherwise, as due to the LSP the compiler allows a derived class object to replace its base, a client might be surprised because some operation fails yet he has done everything right!

A Derived Class Must Never Give Weaker Postconditions:

Otherwise, as due to the LSP the compiler allows a derived class object to replace its base, a client might be surprised because it doesn't get on what it relies.

2.4.3 Private Base Classes

With a private base the "has a" relationship between two class is formalized.

```
class Composite : private Part {
    // ...
};
```

2.4.4 Class as Member

An alternative is explicit containment as member:

```
class Composite {
    // ...
    Part part;
};
```

2.4.5 Components with Adaptable Details (1st Technique)

Public or private base classes may be used to make certain aspects of a class adaptable by a client using this class, be it

- as a public base, or
- as a private base.

Note that this technique does not apply to explicitely contained members!

It is based on virtual member functions called by the base at strategic points where a client might modify the general code.

A design pattern based on this technique is named *Template Methode Pattern* but it has nothing to do with C++ Templates.

2.4.5.1 If Time Allows and You are Interrested ...

... we can go through it now in a life demo modifying the Thermostate Example to give it different strategies to apply the hysteresis when switching on and off.

2.5 Introduction to Programming with Templates

See AToC §5 (from page 59) and also this Info-Graphic for a motivation.

2.5.1 Template Functions (Parametrizing Types)

In case of template function the goal is frequently to get some type independance from a generic implementation. An example might be a sort algorithm that works on an array with any type of element.

This is also schematically illustrated in the right half of this Info-Graphic.

2.5.2 Template Classes (E.g. as Base for Containers)

In case of a template class also types may be parameterized, but other compile time constants may be as well, like buffer sizes.

This is also schematically illustrated in the left half of this Info-Graphic.

2.5.3 Components with Adaptable Details (2nd Technique)

Also templates allow to make certain aspects of a class adaptable. Then this technique is sometimes called policies.

It is based on calling member functions inherited from a base which can be freely set from via an template argument.

2.5.3.1 If Time Allows and You are Interrested ...

... we can go through it now in a life demo modifying the Thermostate Example to give it different strategies to apply the hysteresis when switching on and off.

2.5.4 Comparing Class Adaption Techniques

In the same vein as demonstrated with the two class adaption techniques is the example in this Info-Graphic.

2.6 More Advanced Template Usages

Another view on templates is to consider them as type transformation (functions) to be executed at compile time. This enhances C++ with the power of functional programming at compile time.

For certain real live problems this can yield highly efficient solutions but is probably better left to a few experts (if applied seriously, not for demonstration pruposes only) while the fruits may be very attractive for the mortals.

2.6.1 Something you might want to look at as a motivating example

The Boost.units library extends type checking to physical units (e.g. from the SI system, but this is configurable)

- thus turning it into a compile time error if you assign a speed to a length,
- but still allowing to assign a speed multiplied with a time to a length,
- or a length divided by a squared time to an acceleration,
- (... etc. you get the idea?)
- and also care for the appropriate conversion if a length meassured in inch is assigned to (or otherwise combined with) a lenght meassured im millimetres.

2.6.2 What is Functional Programming?

Functional Programming in its pure form deals with transformations by functions in the mathematical sense, i.e. there is no internal state that may have an influence on the outcome.

Functions are said to be *first class citiziens* and often not only data but also functions are manipulated by functions, for which the term *higher order functional programming* has been coined.

Recently functional programming has gained a little more attention, as when applied in its pure form in a functional programming language, a program lends itself ideally to heavy optimizations, possibly exploiting massive parallelism.

2.6.3 Calculations at Compile-Time

A famous example that calculates the factorial of a number at compile time using templates as a way of functional programming is in Templates/fp_factorial.cpp

---

2.6.4 Type-Mapping at Compile-Time

The the more pragmatic uses of functional programming for type mapping at compile time are demonstrated in the final versions of the series of examples implementing an algorithm in STL style:

- Templates/real_uniq_copy4.cpp
- Templates/real_uniq_copy5.cpp
- Templates/real_uniq_copy6.cpp

2.6.5 Standard Traits (Library)

See AToC §11.6 (from page 128).

---

3 Wednesday

3.1 Essential C++ Standard Library (Overwiew)

3.1.1 The C Library as a Subset

All of the C library is available in C++ too.

- The traditional names of the C header get a `c` added as first character and the `.h` at the end removed.
- All of the contained functions are in the name space `std`.
- Owing to function oveloading some solutions are nicer in C++, like mathematical functions for floating types of various precission.

3.1.2 Strings (Replacing `char`-Arrays)

See AToC §7.2 (from page 75) and this Info-Graphic.

3.1.3 Streams (Replacing `FILE`-s)

For a design overview see this Info-Graphic.

3.1.4 Regular Expressions

AToC §7.3 (from page 78) gives a good coverage.

3.2 Containers and Algorithms

Coverage in AToC starts in §9 (from page 95).

Furthermore the following Info-Graphics deal with aspects of STL containers and algorithms:

- Sequence Containers
- Associative Containers
- Concept of Iterators
- Iterators in More Detail
- Categories of Iterators
- Iterators as Glue to Algorithms

3.2.1 Sequential Containers

There have traditionally been three of them in C++98 with a fourth added in C++11:

- `vector`, `deque`, and `list` (in C++98)
- `forward_list` (introduced with C++11)

A short coverage is given in AToC §9.2 and §9.3 (from page 96).

3.2.2 Associative Containers

Such come in eight flavours, four have been introduces with C++98:

- `set` and `multi_set`
- `map` and `multi_map`

C++11 has added the `unordered_`-variant.

A overview is given in AToC §9.4 (from page 101) and $9.5 (from page 102) which is very brief and cursory.

3.2.3 Iterators (Abstracting Traversal)

Iterators as introduced beginning with AToC §10.1 continuing to §10.5 provide the glue between STL containes and STL algorithms.

3.2.4 Algorithms in General

AToC §10.6 gives a very brief overview.

3.2.5 Parameterizing Call-Backs

One of the most important applications of this technique in C++ are STL algorithms requiring a predicate like demonstrated in AToC §10.5 (from page 113).

3.2.5.1 Function Pointers

There are old style function pointers as known from C, which are simply memory cells holding addresses from the code segment and the modern `std::function` class as introduced in AToC $11.5.3 (from page 127) which can be covered in more depth on request with as live demo.

3.2.5.2 Overloading Function Call (Functors)

See AToC §5.5 (from page 64) going already beyond the basic technique as it combines the introduction of this mechanism with an application to template classes.

3.2.5.3 Lambdas (Function Literals)

Also see AToC §5.5 (from page 64) for this feature, which has been originated from LISP and is known from many other programming, but relatively new for C++.

3.3 Ressource-Management

Generally ressource management is a far broader topic than only memory management, though it - and its problems - often expose themselves first as memory manegment problems like:

- Resource Leaks or
- Dangling References

3.3.1 Memory and Other Ressources

See also this Info-Graphic.

3.3.2 Smart Pointers

See AToC §11.2.1 and this Info-Graphic.

3.3.3 Resource-Wrappers (RAII)

RAII is a basic technique once proposed by Bjarne Stroustrup for an idomatic handling of ressources. It

combines operations which complement each other in constructor and destructor of a helper class.

It is very briefly introduced at the end of AToC §11.2 (from page 118) and gets also a very short mention in AToC §4.2.2, but there are numerous external resources explaining it in great detail, like it is here: http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Resource_Acquisition_Is_Initialization

See also at the right hand side (not covered so far) of this Info-Graphic.

3.4 More C++ Standard-Library (Overview)

This is a small selection only which will be extended on request.

3.4.1 Random Numbers

See AToC $12.5 (from page 136).

(Some live demos based on these examples can be done on request.)

3.4.2 Time-Points and Durations

See AToC §11.4 (from page 125).

As the above is extremely scarce, here is some source where the topic receives a deeper coverage:
`http://www.informit.com/articles/article.aspx?p=1881386&seqNum=2`

3.4.3 Simple Concurrency

See AToC §13.7.2 and $13.7.3 *packaged tasks* and the `std::async` helper utility for:

- providing a very easy to use solution for concurrency
- if the data to process can be separated into strictly independant chunks
- so that no synchronisation is necessary,
- except for collecting the results.

---

4 Thursday

4.1 C++11 vs. Posix-Multithreading

C++11 has added basic support for multi threading that typically builds on some threading facility of the platform (i.e. operating system) for which the C++ implementation was done.

As the same API has to be available for a possibly different underlying host functionality, only common abstractions are defined and anything that is special and cannot be commonly expected has been left out.

Mainly missing are priorities, scheduling policies and related advanced features like priority ceiling, so e.g. solutions to practical problems like priority inversion can not build on C++11 mechanism alone.

4.1.1 Creating Threads

See AToC §13.2 (from page 142) and how a client may supply subroutine to be executed concurrently.

4.1.2 Exchanging Data

When threads are involved this is - contrary to processes - never a problem since all execute in the same address space and can reach each others data.

While this is true in principal, data must not only be accessible but have a life time long enough to be valid throughout any thread accesses it. Therefore it typically lives

- either on a fixed address as a global;
- or somewhere on the heap (maybe accessed by `shared_ptr`-s);
- or local to a thread (that must not prematurely be cleaned up).

4.1.3 Race Conditions

Getting a short mention in §13.5 (from page 144) and serving as an example why there is a helper utility to lock a set of mutexes without failing (i.e. get all of them or none at all).

4.2 Synchronisation between Threads

Synchronisation between threads is necessary in two typical scenarios:

- Caused by a member function called in one thread a class invariant may be temporarily invaldidated and it therefore is to avoid that any another thread steps in between.

- In a consumer/producer scenario with some intermediate buffering one side is too far ahead of the other, i.e.:
- there is no data yet to consume because the producer didn't deliver fast enough;

- the intermediate buffer is completely filled because the consumer couldn't keep step with the producer.

4.2.1 Mutexes and Locks

See AToC §13.5 (from page 144) illustrating the problem of some data structure with a temporarily invalidated invariant.

4.2.2 Condition Variables

See AToC §13.6 (from page 146) illustrating the consumer/producer-scenario.

4.2.3 Joining with a Thread

See AToC §13.3 (from page 143)

---

5 Part 2: Linux System Programming - Starting Thursday

Note: From here on the course is to a large part a guided tour through a selection of the many examples from the TLPI-Book.

5.1 Major Concepts in Overview

It is hardly possibly to give in less than two days an overview of moderate depth over the rich system call interface now offered by Linux. Introducing concepts will therefore accompany the tour through the source.

Feel free to ask questions if you feel too little familiar with the concepts the instructor introduces as background.

Again: PLEASE ask your questions to slow down the pace if you think you will fall behind or even get lost otherwise.

5.1.1 General Utilities Used In The Examples

There is a `lib` subdirectory with some common stuff.

5.1.1.1 Header File Used in all Examples

Most important is a header file used in nearly each example: lib/tlpi_hdr.h

It includes some system heades and provides a few `#define`-s.

5.1.1.2 Functions To Print Messages In Case Of Problems

The following header has the prototypes for such functions: lib/error_functions

Implementations are found here: lib/error_functions.c

Some of these function also terminate the program, others just show the message.

5.1.1.3 Printable Names of Error Number Definitions

The following header helps to turn `#define`-s of error numbers into something printable: lib/ename.c.inc

---

5.1.1.4 Helper Functions to Parse Numeric Values From Strings

Those functions are declared in the following header: lib/get_num.h

Their implementation is accordingly found here: lib/get_num.c

The main advantage over `atoi` and cousins is more error checking.

5.1.1.5 Other Helper Functions

Some of the remaining functions in the lib subdirectory are provided for use in specific examples and will be covered together with these examples.

5.2 Unified I/O

From the viewpoint of the application there is no difference whatever the data source or data sink is:

- At may be data stored in a classical files or
- or data exchanged with another process on the same
- or a different system in a networked environment

The price to pay is that from the viewpoint of the Linux kernel I/O has no other structure as beeing just a sequence of bytes.

5.2.1 Basics of the I/O Modell

Basic Unix I/O involves the use of four system calls, as the following example shows: fileio/copy.c

Such are:

- `open` - open a file specified by name, open mode and other details
- `read` - to transfer a given number of bytes **into** main memory
- `write` - to transfer a given number of bytes **out of** main memory
- `close` - to release the resources

A detailed explanation what each of these system calls does is available in the online manual.

(Putting it differently: RTFM)

5.2.1.1 Something to look at later (if you have no other ideas)

Have a look in the online manual and find out about the relation between `open` and `creat`.

Replace the second `open` with `creat` and try to find out what happens if the file already exists.

Use `creat` *not* including write permission for the newly create file and determine whether the program fails to return a valid file descriptor or later, when it actually writes to the file.

5.2.1.2 Changing the Current Position

With respect to the file read or written the system maintains the notion of a current position, also called the *seek* position.

It can be queried and changed, as the following example shows: fileio/seek_io.c

---

5.2.2 I/O in more Detail

The following examples are a bit special and may well be skipped without loosing track.

5.2.2.1 Exclusive Creation Of A File

This *cannot* be achieve by first testing and then opening, as there is a time window for a race condition, what this example shows:

- fileio/bad_exclusive_open.c

The solution is to combine the `O_CREAT` and `O_EXCL` flags in the `open`.

5.2.2.2 Appending to the End Of A File

Done in the wrong way this could also fail due to a race condition. The following example shows how it is done correctly: fileio/atomic_append.c

---

5.2.2.3 Scatter/Gather I/O

The relevant system calls ar illustrated in this example:

- fileio/t_readv.c

(A similar facility exists for output but has no example program here.)

5.2.2.4 Something to look at later (if you have no other ideas)

Lookup the system call `writev` in the online manual and modify the last example to demonstrate its basic use.

Also lookup the system calls `preadv` and `pwritev` and try to understand what they add to the versions without the leading `p`.

5.2.2.5 Truncating a File

The system call `truncate` for doing this is quite easy to understand but nevertheless here is an example: fileio/t_truncate.c

---

5.2.2.6 Something you might try out later (if you have no other ideas)

Lookup the system call `writev` in the online manual and modify the last example to demonstrate its basic use.

5.2.2.7 Anotherthing to look at later (if you have no other ideas)

Also lookup the system calls `preadv` and `pwritev` and try to understand what they add to the versions without the leading `p`.

5.2.2.8 Duplicated File Descriptors / Sharing of Seek-Positions

The following example (actually the solution to an exercise for TLPI chapter 5) shows how seek positions may be shared between file descriptors:

- fileio/multi_descriptors.c

5.2.2.9 I/O Bypassing The Buffer Cache

The following example shows how this may the be achieved:

- filebuff/direct_read.c

5.2.2.10 Displaying Information About a Single File

The following example prints information that can be obtained with the system call `stat` and `lstat`:

- filebuff/direct_read.c

5.2.2.11 Scanning Directories

The interface to scan directories to find the names of the contained files and sub-directories, as constituted by `opendir`, `readdir´ and`closedir`, is demonstrated in the following example:

- dir_links/list_files.c

As `readdir` maintains some internal state (i.e. how far it progressed) it is NOT reentrant and hence should never be used with multi-threading.

The safe replacement is `readdir_r` (r is for reentrant).

5.2.2.12 Walking The File Tree Including Sub-Directories

This can be done with `nftw` which is not directly a system call but a utility function in the library and hence documented in section 3 of the online manual:

- dir_links/nftw_dir_tree.c

5.2.2.13 Monitoring File Events

This interface was added with the Linux kernel 2.6.25 and allows for a process to wait on certain state changes of a file without consuming CPU time as shown in the following example:

- inotify/demo_inotify.c

It actually replaced an older mechanism introduced in kernel version 2.4 (dnotify).

5.3 Process Management

The secure and efficient implementation of the Unix process modell requires a memory management unit:

- Each process is a unit of code running in its own memory space.
- Without the help of system calls memory space of other processes is not accessible .
- Available CPUs will be shared among processes on a time slice basis.

5.3.1 Understanding Memory Layout

The following example aims to show some ways of allocating data space in various regions of the main memory:

- proc/mem_segements

5.3.2 Accessing Command Line Arguments

(Only for completeness here as the topic has been handled already in the part about C/C++ programming: proc/necho.c.)

5.3.3 Accessing and Modifying the Process Environment

The technique to access the process environment - holding exported shell variables - is similar to the technique accessing the command line, as the following example shows:

- proc/display_env.c

The process environment can also be modified as the following example shows:

- proc/modify_env.c

5.3.3.1 Something to look at later (if you have no other ideas)

Find out which standard command does much the same as the last example program does.

Modify the example to call this program via `system` while executing.

Then change some value in the `env`-array before the call and watch how the output changes.

But why isn't the change visible in the environment of the original shell, i.e. the one from which the demo program was called?

5.3.3.2 Non-Local Branches

This example is only contained for completeness here - in practice there are rare chances to use the technique shown:

- proc/longjmp.c
- proc/setjmp_vars.c

Never ever use `setjmp` / `longjmp` in C++ programs in combination with objects having constructors and destructors AS THE EFFECTS WILL BE UNDEFINED.

(Memory leaks are the least to expect, but complete havoc is very likely too ... soon afterwards or also at a much later point in time.)

5.3.3.3 Managing Heap Memory at the Lowest Level

An application program usully will and should not have to rely on techniques in this example, shown only for demonstration purposes.

memalloc/free_and_sbrk.c

(You also may want to refer to the lower left side of this Info-Graphic once more.)

5.3.3.4 CPU Time Consumed by a Process

The following example shows how to access this information:

time/process_time.c

5.3.3.5 Retrieving System Limits and Options

The following examples show how to access some of this information:

- syslim/t_sysconf.c

- syslim/t_fpathconf.c
- sysinfo/t_uname.c

5.3.3.6 Creating a Process

Process creation is done with the system call `fork`.

It clones the memory space of the creating process (aka parent process) as initial execution space of the newly create process.

- This seems inefficient at first glance as the next step often is to re-load the code memory from an execuable file and re-initialize the data space of the newly created process accordingly.
- But it can be made rather efficient by using COW (copy on write) techniques.

5.3.3.7 Waiting for a Process

Frequently a process has no useful work to do once it created one or more children. Nevertheless it will have to continue to exist so that it can get or gather the result of its child or children.

- There is a provision to wait for children not consuming any CPU-time.
- This should always be used as the preferred way if the creator has no useful things to do.
- In addition, a process can be asynchroneously notified by a *signal* when one of its children terminates execution

---

5.3.4 Process Management

Processes can be sent *signals* from other processes (not only from their creator).

- In both, the original and the revised form introduced with Unix System V, signals are an asynchroneous means of communication.
- Their main purpose is to terminate the receiver.
- Catching a signal in the receiving process was originally only mainly meant to allow for cleanup.

A revision of the design of signals introduced with Unix System V signals had the goal to allow for a more flexible and reliable communication between processes.

5.3.4.1 Handling Signals

The following example shows how a process can arrange for catching a signal sent by some other process:

- signals/ouch.c
- signals/intquit.c

5.3.4.2 Checking if a Process Exists

Sending a signal can be used to check if a process (known by its process id or PID) exists, as shown in

this example:

- signals/t_kill.c

Well, this can be considered as well as a way to "use of signals" but others might say this is at the border of misusing a system call for the wrong purpose.

5.3.4.3 Handling Groups of Signals

System V introduced a new interface on signals based on signal gropus and attempting to add a bit of reliability, so that signals could neither get lost nor inadvertently terminate a process which had arranged to catch a certain (set of) signals.

An example of this problem is here:

- signals/signal_functions.c

5.3.4.4 Signal Pitfal: No Queueing

One common pitfal with signals is that they don't get queued as shown with the combination of the following two examples:

- signals/sig_sender.c
- signals/sig_receiver.c

5.3.4.5 Signal Pitfal: No Reentrancy Control

The actions allowed in a signal handler are rather limited, so failure to obey them will often no show in rare cases. The following example tries to illustrate:

- signals/nonreentrant.c

Though the Unix System V and Posix specification name a lot of functions as reentrant, they may still affect the global `errno` which hence is not reliable in signal handlers.

The conservative approach would be to just set a variable of type `sig_atomic_t` and test it where convenient after returning from the handler.

5.3.4.6 Non-Local Branching From a Signal Handler

As long as it is clear that C++ (or at least objects with constructors and destructores and hence the largest part of the C++ standard library is ruled out then), also a non local branch out of a signal handler is possible, as the following example shows:

- signals/sigmask_longjmp.c

5.3.4.7 The Alternate Signal Stack

Signals thrown when memory is exhausted cause a special problem: they cannot allocate an ordinary stack frame and hence must use an alternate signal stack, as is shown here:

- signals/t_sigaltstack.c

5.3.4.8 Realtime Signals: Queueable And With Additional Information

Another addition to the original signalling mechanism are realtime signals illustrated in this example:

- signals/t_sigqueue.c
- signals/catch_rtsigs.c

On important use on Linux with timers follows in a later example.

5.3.4.9 Suspending Signals in Time-Critical Sections

As illustrated by this example, signals my temporarily suspended, what means they will not get lost but queued until enabled again:

- signals/t_sigsuspend.c

5.3.4.10 Accepting Signals Synchroneously

As an alternative to signal handlers, some applications communicating via signals might get simplified by synchroneous signal delivery, as shown in the following example:

- signals/t_sigwaitinfo.c

5.3.4.11 Fetching Signals from a Special File Descriptor

This mechanism was introduce with the Linux kernel 2.6.22 and allows to process delivered signals as if they were read from a file, like the following example shows:

- signals/signalfd_sigval.c

5.3.4.12 Realtime Timers

A program centered around signals may have resulted from an event-driven design and as such will frequently require to sent itself a delayed signal.

This is done in the following example:

- timers/real_timer.c

Other possible applications are timeout for (synchroneous) `reads` and sleeping (very) short amounts of time:

- timers/timed_read.c
- timers/t_nanosleep.c

5.3.4.13 POSIX Clocks and Interval Timers

With the help of a special library also the API *Clocks* and *Interval Timnrs* as defined by POSIX is available. Its use is illustrated in the following examples:

- timers/ptmr_sigev_signal.c
- timers/itimerspec_from_str.c
- timers/ptmr_sigev_thread.c

5.3.4.14 Receiving Timer Notifications via File Descriptor

Shortly after introducing the facility to receive signal notifications via a special file descriptor, with Linux kernel 2.6.25 this was extended to receiving timer notifications, as shown in the following example:

- timers/demo_timerfd.c

5.3.4.15 Classic Process Communication

The classic and only way processes had to communicate with each other were classic files and pipelines.

- Due to transparent buffering in main memory, communicating via a file were by no means a slow mechanism.
- The main advantage of pipelines was the synchronization inherent in the mechanism.
- The limitation with classic pipes is that a common ancestor needs to create the pipeline.

Later on this limitation was lifted by introducing *Named Pipes* in Unix System V.

Another abstraction introduced with the BSD variants of Unix were *Sockets* allowing to send to messages or data streams from one process to another one.

5.4 Processes Creation and Control

Processes and threads both allow for concurrency. Their main difference on Linux is the degree to which they share resources, which ranges

- from *nearly none* for two processes
- to *nearly everything* for two threads.

While the classic interface to process and thread creation is very different, the system call `clone` brings both close together.

Especially noteworthy for newcomers to Unix and Linux is that even process creation is not a heavy weight operation as is often assumed by the uninformed.

5.4.1 Creating Process With `fork`

A sample program illustrating the basic use of `fork` for process creation is here:

- procexec/t_fork.c

How it is really lightweight can be further studied here:

- procexec/footprint.c

5.4.1.1 Parent/Child File Sharing

Though nearly every resource of the parent is decoupled from that of the child, this is not true for open files, for which the same read/write position is advanced:

- procexec/fork_file_sharing.c

5.4.1.2 Even Lighter as `fork` : `vfork`

There is an alternative to fork that avoids copying altogether, though it is not very useful unless it is combined with `exec` (covered later).

- procexec/t_vfork.c

5.4.1.3 Race Conditions with Regular `fork`

It should be noted that `vfork` is only useful because if gives a guarantee that the child runs first, which is not the case for the classic variant:

- procexec/fork_whos_on_first.c

This could be avoided by using signals for explicit synchronisation:

- procexec/fork_sig_sync.c

5.4.1.4 Terminating a Process

Basically this belongs into the category of the most simple system calls as can be clearly seen from the online manual. The difference between the two alternatives `exit` and `_exit` can be nicely shown with an example like this one:

- procexec/fork_stdio_buf.c

(A program showing the registration of exit handlers is shown here

- procexec/exit_handlers.c

is for completeness only, as this is rather a library function standardized in C but a system call, easily to recognized from the documentation of `atexit` in section 3 not 2 of the online manual.)

5.4.1.5 Waiting For a Child Process to Terminate

As a parent may have several children for which it is not clear which one will end first, the `wait` system call takes care of this:

- procexec/multi_wait.c

An alternative is `waitpid` which also allows to find out if a child has been temporarily stopped for debugging.

5.4.1.6 Accessing the Child's Exit Status

The value set when a process terminates will be made avaibalbe to the parent:

- procexec/child_status.c

5.4.1.7 Zombie Processes

If the parent process terminates before the child, the latter is said to become a zombie. The following example demonstrates this:

- procexec/make_zombie.c

As the responsibility for cleaning up the final process state moves to the initializer - the grand-grand-grand...-father of all processes in the system, it needs to contain a code sequence like in the following example:

- procexec/multi_SIDCHLD.c

5.4.1.8 Chaining To a New Executable

Usually the following happens in a child process immediately after it has been `fork`-ed by its parent:

- procexec/t_execve.c
- procexec/envargs.c
- procexec/t_execlp.c
- procexec/t_execle.c
- procexec/t_execl.c

5.4.1.9 File Descriptors Outliving `exec` (and Cousins)

The default behaviour for open files is that the child can access such files via the file descriptor even after `exec` (in all its variants). If this is not desired it needs to be explicitely disabled as shown here:

- procexec/closeonexec.c

(The default behaviour for signal handling is that it is inherited by the child too, but as the handler is no

more valid (its code has been replaced) such signals will be reset to default handling, which is program termination in most cases.)

5.4.1.10 Simplified Process Execution with Synchroneous Waiting

Behind the following, convenient to use library function,

- procexec/t_system.c

all the `fork` - `exec` - `wait` - `exit` machinery is hidden, so a simplified version might look as follows

- procexec/simple_system.c

and here comes the full-blown one:

- procexec/system.c

5.4.1.11 Process Accounting

Accounting for resources used by a process can be enabled by a priveledged process (presumably before starting one of its children) and later on retrieved:

- procexec/acct_on.c
- procexec/acct_view.c

5.4.2 Bringing Processes and Threads Close Together

This is exactly what the system call `clone` does, which was originally introduced with *Plan 9*, a successor of Unix used mostly internally at *Bell Labs*:

- procexec/t_clone.c

5.5 Multi-Threading

The collection of examples is:

- threads/simple_thread.c
- threads/detached_attrib.c
- threads/thread_incr.c
- threads/thread_incr_mutex.c
- threads/thread_multijoin.c
- threads/strerror.c
- threads/strerror_test.c
- threads/strerror_tsd.c
- threads/strerror_tls.c
- threads/thread_cancel.c
- threads/thread_cleanup.c

5.6 Process Groups (Sessions)

The collection of examples is:

- pgsjc/t_setsid.c
- pgsjc/catch_SIGHUP.c
- pgsjc/disc_SIGHUP.c
- pgsjc/job_mon.c
- pgsjc/handling_SIGTSTP.c
- pgsjc/orphaned_pgrp_SIGHUP.c

5.7 Process Priority and Scheduling

- procpri/t_setpriority.c
- procpri/sched_set.c
- procpri/sched_view.c

5.8 Process Ressources

- procres/print_rlimit.c
- procres/rlimit_nproc.c
- procres/t_setpriority.c
- procres/t_setpriority.c

The collection of examples is:

- Overview of Mechanisms
- Overhead involved with Data Exchange
- Sender/Receiver Synchronisation
- Asynchroneous I/O (poll)

5.9 Pipelines

A basic example looks as follows:

- pipes/simple_pipe.c

A nice application of pipes is to use them for process synchronization:

- pipes/pipe_sync.c

And this is - more or less - what the shell does if two Unix commands are run together, one filtering the output of the other:

- pipes/pipe_ls_wc.c

5.9.1 Interface for C `FILE`-s

For C programs reading input from a `FILE` or writing output to a `FILE` there is a simplified interface in form of the library function `popen`.

The implementation of this function might look similar to the following code:

- pipes/popen_glob.c

5.9.2 Named pipes

Such relief the restriction that a common ancestor must have prepared the communication path:

- pipes/fifo_seqnum.h
- pipes/fifo_seqnum_server.c
- pipes/fifo_seqnum_client.c

5.10 System V IPC

- svipc/svmsg_demo_server.c

5.10.1 SysV Message Queues

- svmsg/svmsg_create.c
- svmsg/svmsg_send.c
- svmsg/svmsg_receive.c
- svmsg/svmsg_rm.c
- svmsg/svmsg_chqbytes.c

- svmsg/svmsg_ls.c

- svmsg/svmsg_file.h
- svmsg/svmsg_file_server.c

- svmsg/svmsg_file_client.c

5.10.2 SysV Semaphores

- svshm/svsem_demo.c

- svsem/semun.h
- svsem/svsem_mon.c
- svsem/svsem_setall.c
- svsem/svsem_bad_init.c
- svsem/svsem_good_init.c

- svsem/svsem_op.c

- svsem/binary_sems.h
- svsem/binary_sems.c

5.10.3 Shared Memory

- svshm/svshm_xfr_writer.c
- svshm/svshm_xfr_reader.c

5.11 SysV Memory Mapped I/O

- mmap/mmcat.c
- mmap/t_mmap.c
- mmap/anon_mmap.c

5.12 Virtual Memory Operations

- vmem/t_mprotect.c
- vmem/memlock.c

5.12.1 Posix Message Queues

- pmsg/pmsg_unlink.c
- pmsg/pmsg_create.c
- pmsg/pmsg_getattr.c
- pmsg/pmsg_send.c
- pmsg/pmsg_receive.c
- pmsg/mq_notify_sig.c
- pmsg/mq_notify_thread.c

5.12.2 Posix Semaphores

- psem/psem_create.c
- psem/psem_unlink.c
- psem/psem_wait.c
- psem/psem_post.c
- psem/psem_getvalue.c
- psem/thread_incr_sem.c

5.12.3 Posix Shared Memory

- pshm/pshm_create.c
- pshm/pshm_write.c
- pshm/pshm_read.c
- pshm/pshm_unlink.c

5.13 Unix Domain Sockets

- sockets/us_xfr.h
- sockets/us_xfr_sv.c
- sockets/us_xfr_cl.c
- sockets/ud_ucase.h
- sockets/ud_ucase_sv.c
- sockets/ud_ucase_cl.c
- sockets/us_abstract_bind.c

5.14 Internet Sockets

- sockets/read_line.c
- sockets/i6d_ucase.h
- sockets/i6d_ucase_sv.c
- sockets/i6d_ucase_cl.c
- sockets/is_seqnum.h
- sockets/is_seqnum_sv.c
- sockets/is_seqnum_cl.c
- sockets/inet_sockets.h
- sockets/inet_sockets.c
- sockets/t_gethostbyname.c

5.15 Sockets Server Design

- sockets/id_echo.h
- sockets/id_echo_sv.c
- sockets/id_echo_cl.c
- sockets/is_echo.h
- sockets/is_echo_sv.c
- sockets/is_echo_cl.c
- sockets/is_echo_inetd_sv.c
- sockets/rdwrn.c
- sockets/is_echo_cl.c
- sockets/socknames.c

5.16 Alternative I/O Modells

5.16.1 Multiplexing

- altio/t_select.c
- altio/poll_pipes.c

5.16.2 Signal-Driven

- altio/demo_sigio.c

5.16.3 Linux specific `epoll`

- altio/epoll_input.c
- altio/self_pipe.c

5.17 Goodbye

Besides the print-out you have access to this document in form of files:

- ccpp-slides-psi.html -- the presentation used during this training
- ccpp-training-psi.html -- a sequential format for reading in a browser
- ccpp-training-psi.pdf -- as before, for viewing with a PDF reader
- ccpp-training-psi.md -- the source format for all

The source is transformed into the first two versions via pandoc.