

3F6 - Software Engineering and Design

Handout 1
Architecture of a Computer
With Markup

Steve Young

These notes are based on material originally developed by Tom Drummond and Paul Smith.

Contents

1. Course Aims
2. Basic Computer Architecture
3. Procedural Design
4. Data Structures
5. Invariants
6. Access Functions
7. Classes

Copies of these notes plus additional materials relating to this course can be found at:
<http://mi.eng.cam.ac.uk/~sjy/teaching.html>.

Course Aims

This 16 lecture module aims to teach some of the key principles of software engineering and design. It is aimed specifically at the construction of engineering systems built by small teams of 5 to 20 people.

We will start with a low-level overview of what computers are doing “close to the metal” and work our way up through object oriented design techniques to system components and software management.

Topics covered are:

Object Oriented Programming:

- **Architecture of a computer** (1 lecture)
How a program executes, the structure of memory and pointers in C/C++.
- **Classes and C++** (2 lectures)
Introduction to object oriented programming in C++; motivation, approach, language facilities.
- **The Unified Modeling Language** (1 lecture)
Using formal UML diagrams to express the architecture and behaviour of software; and the role of these diagrams in software design.

Software Design:

- **Object Oriented Design** (1 lecture)
How to turn a specification into a software design.
- **Design Patterns** (2 lectures)
Commonly recurring design problems and solutions.
- **Refactoring** (1 lecture)
Redesigning existing software to improve its design and to incorporate new capabilities.
- **User interface design** (1 lecture)
Designing for the user, use cases and UI design patterns.

Software Systems:

- **Distributed Systems** (2 lectures)
Client-Server architectures and CORBA.
- **Database Systems** (1 lecture)
Database management systems, transaction processing, concurrency control, check-pointing and recovery.
- **Concurrent Systems** (2 lectures)
Constructing systems with multiple processes/threads.

Management:

- **Software Management** (2 lectures)
How to manage the software construction process for small engineering systems.

Books

C++: Effective Object-Oriented Software Construction,
Kayshav Dattatri, Prentice Hall, 1999

The Object-Oriented Thought Process,
Matt Weisfeld, 2nd Ed, Sams Publishing, 2004

Learning UML 2.0,
Russ Miles and Kim Hamilton, O'Reilly, 2006

Design Patterns: Elements of Reusable Object-Oriented Software,
Erich Gamma et al, Addison-Wesley, 1998

Refactoring: Improving the Design of Existing Code,
Martin Fowler, Booch, Jacobson and Rumbaugh, 1999

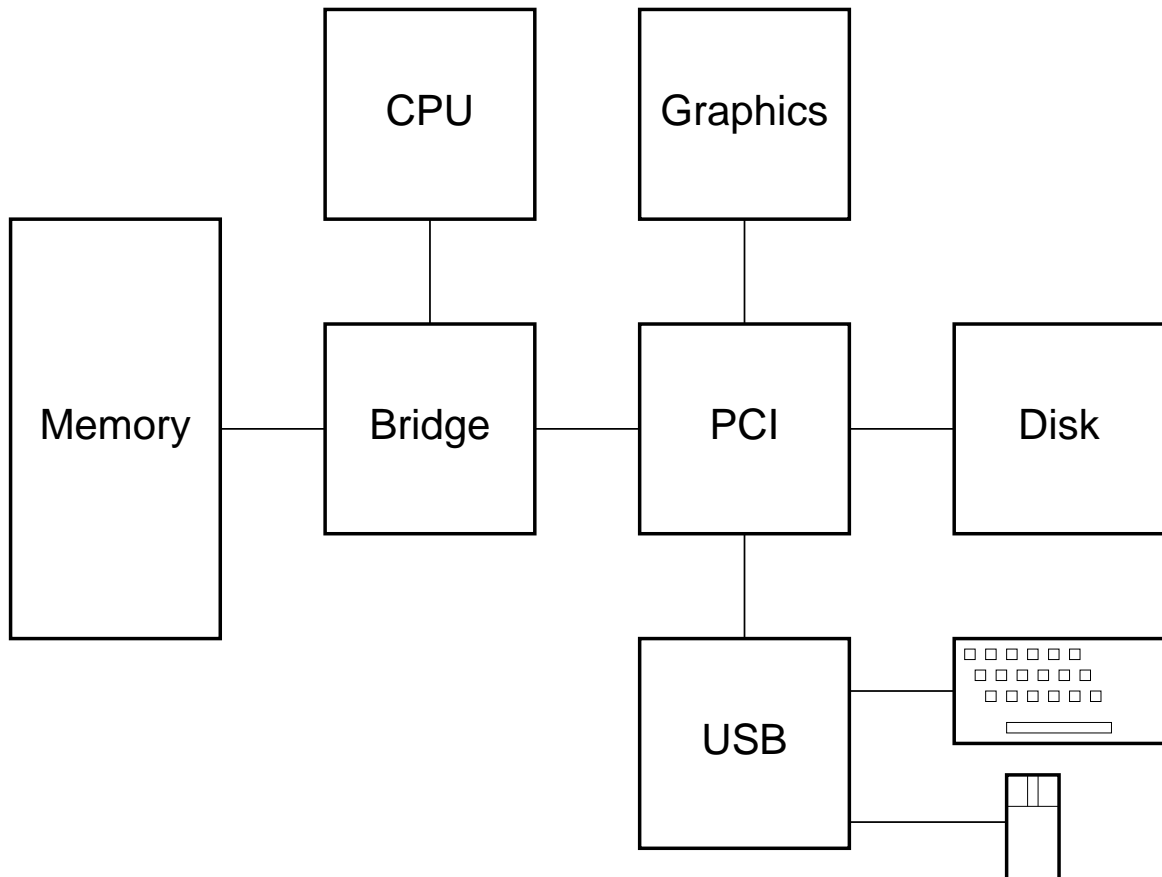
User Interface Design for Programmers, Joel Spolsky, Apress,
2001

Advanced CORBA Programming with C++,
Michi Henning and Steve Vinoski, Addison-Wesley, 1999

Code Complete,
Steve McConnell, Microsoft Press, 1993

The C++ Programming Language,
Bjarne Stroustrup, Addison-Wesley, 1997

Architecture of a Computer



Emphasise that primary flow of information is between CPU and memory. It fetches instructions, loads data, modifies it and writes it back.

Memory is used for two things:

- Data
- Program Instructions

Each memory location can only store a byte (8 bits) which is only enough to represent a single character of a string, so memory locations are often grouped together to store larger quantities.

A group of 4 consecutive bytes is known as a Word (ie 32 bits) and is enough to store:

- an integer $-2^{31}..2^{31} - 1$ (int)
- a single precision floating point number (float)
- a pointer (ie a memory address) (type *)
- a program instruction (on most architectures)

A group of 8 consecutive bytes is known as a double word or “DWord” (ie 64 bits) and is enough to store a long int or a double precision float.

The compiler *usually* makes sure that all 4 byte (or larger) quantities start at an address that is divisible by 4.

Pointers

In C++ (or C) it is possible to create variables that are pointers to memory locations.

When the programmer writes `int x=10;`

The compiler sets aside 4 bytes to store the value of x and writes the value 10 there

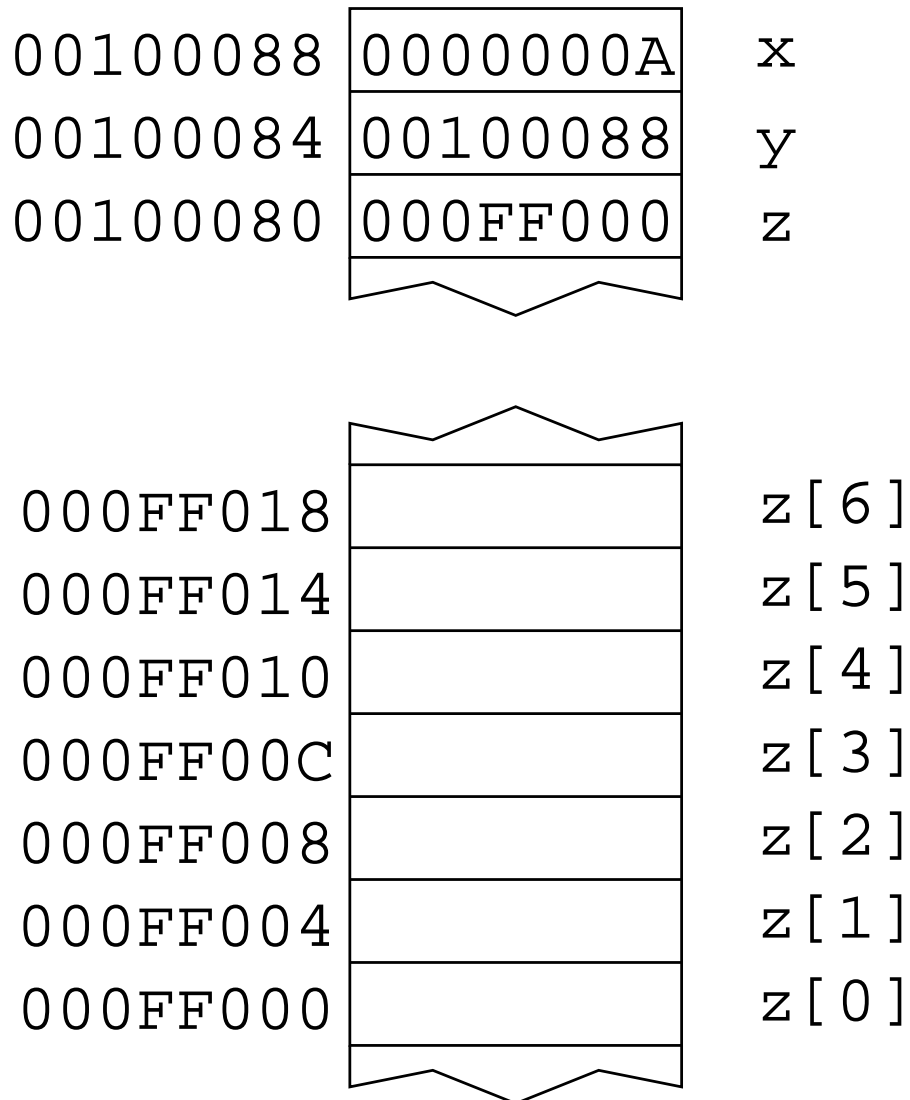
When the programmer writes `int *y = &x;`

The compiler sets aside 4 bytes to store the value of y which is the *address* of an integer and writes the address of x there.

Arrays are also represented as pointers.

When the programmer writes `int z[40];`

The compiler sets aside $4*40 = 160$ bytes to store 40 integers and another 4 bytes to store z which is the address of the first integer in the array.



Draw arrows to show that y points to x, and z points to start of array.

Program in Memory

When the compiler sees

```
int x = 10;
int *y = &x;
```

it compiles the C code into machine assembly instructions:

```
movl    $10, -8(%ebp)
leal    -8(%ebp), %eax
movl    %eax, -4(%ebp)
```

These machine assembly instructions then have an exact representation as numeric codes:

Address Program Instructions

```
0000050 2904 c7c4
0000054 f845 000a
0000058 0000 458d
000005C 89f8 fc45
0000060 c3c9 0000
0000064 4700 4343
0000068 203a 4728
000006C 554e 2029
```

NB This is x86 assembly code. \$10 appears as bytes 8 thru 12.

Because this program is in memory just like data, it is possible to have pointers which point to executable program segments. Hence indirect jumps via pointers.

Procedural Design

(or C++ as it's been taught to you so far)

Procedural programming is concerned with data structures and algorithms. The thought process used to design software in this approach is something like:

1. What has to be done in what order to achieve the task?
2. What data structures will this require?
3. What are the algorithms that will act on this data?
4. How can these algorithms be expressed as a hierarchy of functions?

Function-oriented design has been practiced since programming began, and there is a large body of design methods based on functional decomposition (aka *stepwise refinement* or *top-down design*).

However, while the functions can hide the details of the algorithms, the data structures are left exposed. Functions have free access to the data and this can cause problems.

Especially in large systems built by teams.

Why is procedural design still used?

Object-oriented approaches now dominate large system design. But procedural design is still important:

- In systems with minimal state (e.g. an ATM), or which can be implemented by parameter passing, object-oriented approaches offer no significant advantages, and may even be less efficient.
- Many organisations have standards and methods based on functional decomposition.
- There are an enormous number of legacy systems which have been developed using a procedural approach, and need to be maintained (e.g. The HTK Toolkit).
- In many cases, a procedural design yields smaller faster code and is chosen for that reason (e.g. Linux kernel).
- The C standard is *much* simpler than the C++ standard so there are many more standards-compliant compilers for C.
- Complex algorithms within a large system need a functional decomposition embedded within an object-oriented system.

Emphasise the last point - procedural design is appropriate for small systems and complex components within large systems.

Data structures

In addition to the procedural design, it is also necessary to structure the data on which these algorithms will operate. Procedural programming languages like C provide several tools to assist in this.

Primitive data types: int, char, float, double, etc.

Arrays of data: int[100], "a string", etc.

It is also often convenient to collect several pieces of data together using struct's. These are chunks of data that belong together because they contain different parts of the same conceptual entity.

For example:

```

struct date {
    int    day;
    int    month;
    int    year;
};
date d;
d.day=1;

struct image {
    int    width;
    int    height;
    char   *pixels;
};
image img;
int size;
img.width=100;
img.height=50;
size=img.width*img.height;
img.pixels = malloc(size);
for (i=0; i<size; i++)
    img.pixels[i]=0;

```

Note that struct's are examples of *user defined types*.

Invariants

Both of these data structures pull information together into meaningful clusters. Instances of these data structures can be created and have values assigned to them. However, structures such as date and image typically have additional requirements that must be maintained.

Examples:

- image img

img.pixels must point to an allocated chunk of memory big enough to hold `img.width*img.height` pixels. If the height or width are changed, then the memory must be resized *and* realigned.

- date today

certain constraints must hold, e.g

- `today.month > 0`
- `today.month < 13`
- `today.day > 0`
- `today.day < 32` (... or 29 or 31 or 30 depending on the month and year).

These requirements are called **Invariants**.

What can go wrong?

Every time that a struct is used, the programmer must ensure that the invariants are maintained. In a large system (e.g. millions of lines of code) it is inevitable that *somewhere* in the code, *someone* will make a mistake.

For example, someone might load an image from disk without checking to see if the chunk of memory pointed to by pixels is large enough.

Or they might allocate a new chunk of memory for the image without disposing of the old chunk and so creating a *memory leak*.

Or someone might want to schedule a reminder in a diary application for one month in the future using:

```
reminder.day = today.day;  
reminder.year = today.year;  
reminder.month = today.month+1;
```

```
if(reminder.month == 13) {  
    reminder.month=1;  
    reminder.year++;  
}
```

This looks OK - but what if today is 31/1/2007? We need a way of forcing programmers to obey the invariants. **31/1/2007 + 1 month → 31/2/2007 !!**

Access Functions

One way of doing this (for example in C) is to provide *access functions*. Instead of using the above code fragment, a programmer is expected to only use special functions that guarantee to respect the invariants. For example,

```
add_a_month(today, &reminder);
```

If all programmers use only these access functions (and all the functions are implemented correctly) then date structures will always be valid. However, a large software system will contain a very large number of such rules.

Rule 5967:

date structs must only be modified using functions provided in "date_access.h"

This is not really an adequate solution. We really need to do two things:

1. Provide a correct means of access to the data
2. Make this the *only* means of access to the data

The above approach only achieves the first of these.

Classes

Object-oriented programming enforces correct access to structures by using a programming construct called a class.

Classes are like structs but they contain both data and functions. This means that objects of a class provide services as well as containing data. This approach provides two key benefits:

Encapsulation

The class embodies functions as well as data. The functions give access to the data while maintaining invariants.

Data hiding

The raw data can be made inaccessible from outside the class so the programmer *must* use the access functions.

Classes have:

- 1) data (usually private)
- 2) member functions (usually public) to
 - access the data
 - modify the data

A class definition specifies both the data and member functions, thus defining the class **interface**. In C++ classes look like:

```
class Date {
public:

    int get_day();
    int get_month();
    int get_year();
    void set_date(int d, int m, int y);

private:

    int day;
    int month;
    int year;

};

class Image {
public:

    void load(char * filename);
    void save(char * filename);
    char get_pixel(int x, int y);
    void set_pixel(int x, int y, char value);

private:

    int width;
    int height;
    char * pixels;

};
```

Programming with Classes

Objects are user-defined types. So we can create instances of a particular class (called an object) by simply declaring variables of that type. For example,

```
Date today;
```

This creates an object called `today` of type `Date` just like creating a variable of any type you're used to, eg as in `int i; float x; etc.`

We can use the functions supplied by a class to access the objects we create. Eg.

```
today.set_date(1,11,2007);
```

sets the date of the object `today` to "1st November 2001" and

```
if (today.get_day() == 1) {  
    start_new_month();  
}
```

Note that we can also create classes dynamically and access them via pointers

```
Date *p;  
p = new Date();  
p->set_date(1,1,2008);
```