

Programming abstractly

A decorative graphic at the bottom of the slide. It consists of a red triangle on the left side, pointing downwards, and a black trapezoid on the right side, pointing upwards. The two shapes meet at a horizontal line, creating a continuous shape that spans the width of the slide.

Abstraction is selective ignorance

- When you drive a car, thinking about how the engine works is a distraction
- When you repair a car, thinking about how the engine works is essential
- Abstraction is deciding which aspects of a problem to consider and which ones to ignore

Abstraction and chunking

- The main part of programming is creating suitable low-level abstractions and chunks
- The main part of design is creating suitable high-level abstractions

An example of abstraction (in Awk)

```
{
    for (i = 1; i <= NF; ++i)
        ++words[$i]
}
END {
    for (s in words) {
        print words[s], s
    }
}
```

Why was this program easy?

- automatic input loop
- input broken into fields
- variable-length strings
- associative arrays
 - elements created automatically
 - easy iteration
- automatic memory management

A similar program in C++

```
#include <iostream>
#include <string>
#include <map>
int main() {
    std::map<std::string, int> words;
    std::string s;
    while (std::cin >> s) ++words[s];
    std::map<std::string, int>::iterator i;
    for (i = words.begin();
         i != words.end(); ++i)
        std::cout << i->second << "\t"
                  << i->first << std::endl;
}
```

Why was this program easy?

- easy to read a word at a time
- variable-length strings
- associative arrays
 - elements created automatically
 - easy iteration
- automatic memory management

Comparing C++ with Awk

- The Awk program is shorter
- An informal test shows that the Awk program is about twice as fast
- So why bother with the C++ version at all?

The real difference

- Awk was designed to solve this kind of problem
- C++ was designed to make it easy to implement libraries for a wide variety of problems
- The standard library doesn't have particular applications in mind

What about performance?

- Awk has a built-in operation to read a line and break it into fields
- This operation is carefully optimized, because it is so common
- The C++ version spends most of its time reading input
- Writing a high-performance “Awk input” abstraction might pay off

Possible conclusions

- If you have a language that is intended to solve your specific problem, use it
- Otherwise, a language that supports a range of abstractions is useful—if you use it that way

Chunking

- We read text in words and phrases, not letters
- Similarly, we group together visual patterns that recur in programs
(Example: `*p++ = *q++`)
- Finding useful chunks makes programs easier to understand, even when all the details are still right out in the open

An example of chunking



*Paris
in the
the Spring*

What did the text say?

The point of chunking

- We see what we expect to see
- Therefore, if we wish to be clear, we should write what people expect to see
 - which means we need to know (or influence) what people will expect
 - which probably requires a community

Another example of chunking

```
char* strcpy(char* p, const char* q) {
    while (true) {
        *p = *q;
        if (*q == '\0')
            return;
        ++p; ++q;
    }
}

char* strcpy(char* p, const char* q) {
    while (*p++ = *q++) ;
}
```

Design

- Design is mostly breaking large programs into smaller parts
- Crucial decisions include
 - where to draw the boundaries between the parts
 - how the parts should communicate
 - the interface(s) between the parts

What makes a good design

- A clear understanding of the problem
- Each component is well defined
- Each component has sensible, useful properties
- The components accurately model the problem

Design strategy

- Design is constructing a model
- A good model behaves similarly to what it models
- Therefore:
 - How our models behave is the most important thing about them
 - We should think about behavior before anything else

Design tactics

- What are the important pieces of our design?
- How do they behave? What operations do they support?
- Once we have decided on behavior, we can often begin writing code immediately

C++ class definitions as a design aid

- When you write down the public parts of a class definition, you are already part way toward your design
- You can compile the class before you fill in the details

Object-oriented programming

- An object has a *type*, a *state*, and a *behavior* (or behaviors)
- Sometimes we care about these properties, sometimes not
- An object-oriented language will make it easy to support objects to different degrees of abstraction

Why is OOP useful?

- Programming objects are useful abstractions of physical objects
- Even programs that do not deal with physical objects often want to offer behavior that models physical objects
- It is no surprise that OOP started out as a tool for writing simulation programs

OOP is not the world

- Pure FP (functional programming) is the opposite of pure OOP
 - In OOP, everything is data, even programs
 - In FP, everything is program, even data
- The resulting style is dramatically different

Generic programming

- A generic program is one that uses as little knowledge as possible about its surroundings
- Different languages express generic programs differently
 - Smalltalk uses generic typing
 - C++ uses templates
 - FP languages often support generic types

C++ templates

- Types that are dynamic during compilation and static during execution
- Often used to express containers and iterators
- Can be used as a way of connecting parts of a system

Memory management

- Some languages handle it automatically
- If you are using a language that doesn't, you must make it part of your abstractions (handles, iterators, etc)
- C++ often makes it easy to do so
- Memory is not the only resource that programs must manage

Advice about programming

- Understanding the problem clearly is the hardest part of programming
- Making your design fit the your understanding is second hardest
- If you got both those parts right, implementation is usually easy
- So if your implementation goes to pieces, take another look at your design

Advice about languages

- Language and design usually depend on each other, at least a little
- Choice of language should depend on the whole context
 - what is available
 - local culture
 - what problems you want to solve
- Learn several languages—thoroughly

Meta-advice

Programming is a human activity;
forget that and all is lost.