

Some C++ features

Non object-oriented features of C++

June 21, 2009 Dr. Mordo Shalom

C++ is a superset of C

- Just rename the files from .c to .cpp

The iostream library

```
#include <stdio.h> → #include <iostream>
using namespace std;
```

- The 3 FILE s stdin, stdout, stderr are replaced by 3 streams cin, cout, cerr
- An I/O statement to a stream:
 - starts with the name of the stream
 - is followed by a sequence of >> (for input) or << (for output) operators

```
cout << "some string" << 5 << endl;
cin >> a >> b >> c;
```

The iostream library

- Advantage:
 - Simpler
 - Type safe:


```
char *s;
printf("%d\n", s)
```

 is no more possible.
 - Extendable to composite types
- Formatting is done via manipulators
 - endl is such a manipulator.

More about this later..

Dynamic Memory Management

```
ALLOC(type,1) → new type;
ALLOC(type,n) → new type[n];
FREE(p) → delete p;
FREE(p) → delete [] p;
```

- Memory leaks are equally possible as in C
- No need to check for NULL return value:


```
int *p = ALLOC(int, 1);
if (!p) { ...; ERROR; ... }
```
- new throws a bad_alloc exception

More about this later..

Dynamic Memory Management

- In new programs use only the new and delete operators.
- When modifying existing programs, don't mix up, i.e
 - Don't use delete on a pointer obtained by malloc, or vice versa.

Slide 4

MS1 Mordo Shalom, 6/23/2009

Define variables everywhere

- Variables can be defined (almost everywhere)
- Its scope is from the point it is defined, to the end of the block.
- Typical use:
for (int i=0;)

The bool type

- Declaration:
bool b;
- Two literals: true, false
A bool can get one of the above values.

The string "type"

- A string is much like a char *, but
 - Easier to use
 - Saves the memory allocation, de-allocation burden related to strcpy, strcat, etc.. operations.
 - Concatenation with the + operator.
 - Automatic conversion from char * to string, but not vice versa.

The string "type"

```
string a;  
a.assign("Hello");  
for(int i=0; i<a.length();++i) {  
    cout << a[i] << " " << (int)a[i] << endl;  
}  
a+=" Good Bye";  
for(i=0; i<a.length();++i) cout << a[i];  
cout<<endl;
```

The MACRO evil

```
#define PI 3.1415926  
#define sqr(a) ((a)*(a))  
#define ALLOC(type,n) (type *) malloc ...
```

- When do we make use of a MACRO?
 - To define constants.
 - To avoid function call overhead.
 - To achieve type independency

More about
this later..

The MACRO evil

- Problems
 - (non) Type safety.
 - Does not obey scope rules.
 - May clash in a strange way with variable names.
 - Unexpected side-effects (sqr(x++));
 - Very strict and unintuitive syntax.
 - Hard to pinpoint (even) compile time errors.

The MACRO evil

- Solution: Avoid MACRO s usage
- Substitutes
 - enums
 - Constant variables
 - inline functions
 - function templates

More about this later..

constants

- A constant is a variable that can not be modified, or better an "invariable".
- Therefore it must be initialized.
- Syntax:
const <variable declaration> = <const-expression>
- Examples:
const int PI=3.1415926;
const int *p = &x;
int * const p = &x; // What is the difference

Recall interpretation of complex C declarations

Constants (cont'd)

- A constant can not appear at the left side of an assignment, and
- A const * can not be passed to a function that might modify it.
- Therefore we declare functions' formal parameters as const if they are not modified by the function.

inline functions

- Each function call incurs some penalty in terms of running time:
 - PUSH-parameters into stack;
 - CALL-Jump to the start of the function;
 - POP-parameters from stack;
 - RETURN-Jump back to the calling function;
- For small functions this penalty becomes non-negligible.

inline functions (cont'd)

- In C++ a function can be declared as "inline" (by prepending the keyword inline)

```
inline int max(int a, int b) {  
    return a > b ? a : b;  
}
```

 - In this case the code of the function is duplicated for each time the function is called.
 - We avoid the penalty in terms of running time.
 - We get a penalty in terms of space (code size).

inline functions (cont'd)

- The effect is much like a MACRO
- But they are:
 - Type-safe
 - Compiled at the declaration point, as opposed to MACRO's that are compiled after they are substituted.

Sample code in project

ElevatorCPP-1-ADT-IO-DMA

function overloading

- A function is identified not only by its name, but by its "signature" (prototype)
- The signature of `int max(int, int)`
- is different from `double max(double, double)`
- Therefore they are distinct functions
- We are allowed to define and use both of them w/o having a "name-clash".

Recall automatic typecasts..

Default parameters to functions

- Some parameters of a function may be declared as optional.
 - The caller is allowed not to supply values for these parameters.
 - In this case some defaults are assumed.
 - The optional parameters have to be at the end of the list, i.e.
 - No compulsory parameter can appear after an optional one

Default parameters to functions (example)

```
#include<iostream>
using namespace std;
inline unsigned int max(unsigned int a, unsigned int b, unsigned int c=0) {
    int temp= a > b ? a : b;
    return temp > c ? temp : c;
}
main(){
    unsigned int x,y,z;
    cout<<"Enter three numbers: ";
    cin>>x>>y>>z;
    cout<<"The maximum of "<<x<<" , "<<y<<" and "<<z
        <<" is "<<max(x,y,z)<<endl;
    cout<<"Enter two numbers: ";
    cin>>x>>y;
    cout<<"The maximum of "<<x<<" and "
        <<y <<" is "<<max(x,y)<<endl;
    return 0;
}
```

Namespaces

- Consider two files a.h and b.h

```
#ifndef A_H
#define A_H

#define PI=3.1415926
int a;

#endif // A_H
```

```
#ifndef B_H
#define B_H

#define PI=(22/7)
int a;

#endif // B_H
```

- What happens if both are included by the same program ?

Solution → Namespaces

```
#ifndef A_H
#define A_H

namespace a {
const double PI=3.1415926
int a;
}
#endif // A_H
```

```
#ifndef B_H
#define B_H

namespace b {
const double PI=(22/7)
int a;
}
#endif // B_H
```

Where can we use this in our example ?

```
#include<a.h>
#include<b.h>

.....
double area = a
a::a = 5;
.....
```

```
#include<a.h>
#include<b.h>

using a::PI;
.....
double area = a
a::a = 5;
.....
```

```
#include<a.h>
#include<b.h>

using namespace a;
.....
double area = PI * r * r;
a = 5;
.....
```

references

- A reference variable is just an alternative name (i.e. an alias) to an existing variable.
- It has to be initialized
- All subsequent assignments are done to the original variable
- Main usage is passing parameters by reference;

References (example 1)

```
int a=5, b=3;
int &c=a; // a and c represent the same variable
c++;    // increments a too
c=b;    // does not make c a reference to b
```

Call by reference

```
inline void swap(int& a, int& b) {
    int temp=a;
    a=b;
    b=temp;
}
```

```
int x=3;
int y=5;
swap(a,b);
swap(x,3); // illegal
```