

# C++

Additions not related to objects

# Overview

- Additions and changes not related to objects
  - type `bool`
  - pass-by-reference & the Copy-Constructor
  - user-defined overloading
  - function template and class template
  - exception handling
  - ...

# Type bool

- Represents boolean-values
- Conversion rules with the `int` type


```
bool b1, b2, b3;  
int j, k;  
b1 = 3*5; // b1 = true  
b2 = 0; // b2 = false  
j = b1; // j = 1  
j = b1 || b2; // j = 1  
j = b1 && b2; // j = 0  
b1 = j == 0; // b1 = true
```

# Reference variables (1)

- `int a, *ptr_a;`
- `int &ref_a = a;`  
*// ref\_a is an address, a reference variable*
- `ref_a = 5;        // or a = 5`
- `ptr_a = &ref_a; // or ptr_a = &a;`

# Reference variables (2)

A reference variable is similar to a **const pointer**

- `int a, *ptr_a;`
  - `int &ref_a = a;`
  - `ref_a = 5;`
  - `ptr_a = &ref_a;`
- 
- `int a, *ptr_a;`
  - `int * const ptr_a = &a;`
  - `*ptr_a = 5;`
  - `ptr_a = ptr_a;`

This implies:

- a reference variable must be initialized when defined
- must refer always to the same variable, reassignment is not allowed

# Call-by-reference (1)

```
▪ int f(int& t_in) {  
    t_in = 99;  
    ...  
}
```

```
▪ int f(const int& t_in) {  
    t_in = 99; // ERROR  
    ...  
}
```

A good style

# Call-by-reference (2)

- Two possible realizations in C++
  - `void doSomething(Data * data);`
    - pointer-based
    - Advantages and drawbacks of pointer approach
  - `void doSomething(Data & data);`
    - Reference based
    - Non null-checking necessary

# Return-by reference

```
▪ int & g(int &x){  
    return x  
}
```

```
▪ const int & h(int x);
```

```
...  
h(j) = 99; // ERROR
```



# Summary

- variables hold values

- `int v = 5;`

- pointers hold addresses of variables

- `int p = new(int);`

- `*p = 5;`

- references refer to contents of another variable

- `int& r = p;`

# The Copy Constructor

```
ChewingGum(const ChewingGum& rhs);
```

- default copy constructor
  - automatically generated if not present
  - produces a complete **shallow copy** of the passed object
- user-defined copy constructor
  - can take arbitrary measures to provide a copy of the `rhs` object (e.g. deep copies)

# The Assignment Operator

- `ChewingGum& operator  
=(const ChewingGum& rhs);`
- Sets an object to be a copy of a passed object
- Default behavior: shallow copies

# The Copy Constructor and the Assignment Operator

- copy constructors and assignment operators
  - automatically generated in each class
  - no inheritance

```
class Employee {  
    //...  
    Employee(const Employee&);  
    Employee& operator=(const Employee& );  
};
```

```
Manager m("Homer", 3);  
Employee e = m;          // Slicing!!
```

# The Copy Constructor and dynamic memory

- **always** declare a user-defined **copy constructor** for classes with dynamically allocated memory
- the default implementation leads to
  - undefined behaviour (probably an access violation)

# The Assignment Operator and dynamic memory

- **always** declare a user-defined **assignment operator** for classes with dynamically allocated memory
- the default implementation leads to
  - a memory leak
  - undefined behaviour (probably an access violation)

# An example (1)

```
#include <iostream.h>
#include <string.h>
class Message {
    char *subject;
    char *message;
    //A function to initialize data members
    init_message(const char *,const char *);
public:
    //A constructor
    Message(const char *, const char * = "");
    //The copy-constructor
    Message(const Message & m);
    //Overloading of the assignment operator =
    const Message& operator=(const Message &);
    //The destructor
    ~Message()
};
```

## An example (2)

*//A function to initialize data members*

```
Message::init_message(const char *s, const char *m){  
    subject = new char [strlen(s)+1];  
    strcpy(subject,s);  
    message = new char [strlen(m)+1];  
    strcpy(message,m);  
}
```

*//A constructor*

```
Message(const char *s, const char * m){  
    init_message(s,m);  
}
```

*//The destructor*

```
~Message(){  
    delete subject;  
    delete message;  
}
```



## An example (3)

```
//The copy constructor
```

```
Message::Message(const Message & m){  
    init_message(m.subject,m.message);  
}
```

```
//Overloading of the assignment operator =
```

```
Const Message& Message::operator=(const Message & m){  
    // always check for self-assignment  
    if (this == &m) return *this;  
    delete subject;  
    delete message;  
    init_message(m.subject,m.message);  
    return *this; //the left element is returned  
}
```

# Assignment sequences

- C++ allows for

```
int a, b, c, d;  
a = b = c = d = 5;
```

- Objects should allow this as well
- assignment operator needs to return a reference to `(*this)`

# References (1)

- C++ as a language and programming guidelines
  - Stroustrup, B. (1999). *The C++ Programming Language*, Addison-Wesley.
  - Meyers, S. (1998). *Effective C++*, Addison-Wesley
  - Meyers, S. (1995). *More Effective C++*, Addison-Wesley
  - Meyers, S. (2000). *Effective STL*, Addison-Wesley
  - Alexandrescu, A. (2002). *Modern C++ Design*, Addison-Wesley

# References (2)

- Process memory layout, etc.
  - Intel Cooperation (2002), *IA-32 Intel(R) Architecture Software Developer's Manual Volume 1: Basic Architecture*, Chapters 3 and 6
  - Intel Cooperation (2002), *IA-32 Intel(R) Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, Chapter 2
  - Santa Cruz Operation, Inc. (1997), *System V Application Binary Interface Intel386(tm) Architecture Processor Supplement*, Fourth Edition

## References (3)

- **C++ Applications** by the creator of C++  
Bjarne Stroustrup
  - <http://www.research.att.com/~bs/applications.html>