



SAPIENZA  
UNIVERSITÀ DI ROMA

# ***Robot Programming Introduction to C++ (from C)***

Giorgio Grisetti

# Preprocessor

The compilation process starts by calling a text processor (cpp).

The goal of this program is to generate a plain cpp file, eliminating all directives that can be fed to the real compiler (cc1)

Directives of the preprocessor start with a #

The most important are

`#include <filename>` or `#include "filename"`

(expands a file in the output)

`#pragma once`

(avoids multiple expansions if a file is included multiple times)

`#define`

(declares a variable or defines a preprocessor macro)

# Basic Types

To declare a variable of a type use the syntax

`<Type> name; or <Type> name1, name2, ... nameN;`

`char my_char; // 1 byte char or number`

`int my_int; // typically 4 byte integer`

`long int my_long_int; // 8 byte integer`

`float my_float; // 4 byte floating point`

`double my_double; // 8 byte floating point`

# Qualifiers

Qualifiers appear before the type. They alter the behavior/storage of the variable

**const**: once assigned (on construction) can't be changed

**unsigned**: integer types only, value in the positive range

**long**: selects the "large storage" of an int (64 bit, usually)

**short**: selects the "short storage" of an int (16 bit, usually)

**static**: has multiple meanings, depending on the context.

example:

```
const unsigned long int a=10349202999;
```

# Addresses

An address in C++ has a type depending on the stored variable.

To declare an address use the \* operator.

To extract the address of a variable use the &operator;

Example:

```
int a;
```

```
int * a_ptr;
```

```
a_ptr=&a;
```

```
const int* my_const_pointer_to_a= a_ptr;
```

```
float ** f;
```

# Operators and Expressions

Precedence	Operator	Description	Associativity
1	::	Scope resolution	Left-to-right
2	a++ a-- type() type{} a() a[] . ->	Suffix/postfix increment and decrement Functional cast Function call Subscript Member access	
3	++a --a +a -a ! ~ (type) *a &a sizeof co_await new new[] delete delete[]	Prefix increment and decrement Unary plus and minus Logical NOT and bitwise NOT C-style cast Indirection (dereference) Address-of Size-of <sup>[note 1]</sup> await-expression (C++20) Dynamic memory allocation Dynamic memory deallocation	Right-to-left
4	.* ->*	Pointer-to-member	Left-to-right
5	a*b a/b a%b	Multiplication, division, and remainder	
6	a+b a-b	Addition and subtraction	
7	<< >>	Bitwise left shift and right shift	

# Operators and Expressions

<b>7</b>	<< >>	Bitwise left shift and right shift	
<b>8</b>	<==>	Three-way comparison operator (since C++20)	
<b>9</b>	< <= > >=	For relational operators < and ≤ and > and ≥ respectively	
<b>10</b>	== !=	For equality operators = and ≠ respectively	
<b>11</b>	&	Bitwise AND	
<b>12</b>	^	Bitwise XOR (exclusive or)	
<b>13</b>		Bitwise OR (inclusive or)	
<b>14</b>	&&	Logical AND	
<b>15</b>		Logical OR	
<b>16</b>	a?b:c throw co_yield = += -= *= /= %= <<= >>= &= ^=  =	Ternary conditional <sup>[note 2]</sup> throw operator yield-expression (C++20) Direct assignment (provided by default for C++ classes) Compound assignment by sum and difference Compound assignment by product, quotient, and remainder Compound assignment by bitwise left shift and right shift Compound assignment by bitwise AND, XOR, and OR	Right-to-left
<b>17</b>	,	Comma	Left-to-right

# Expressions

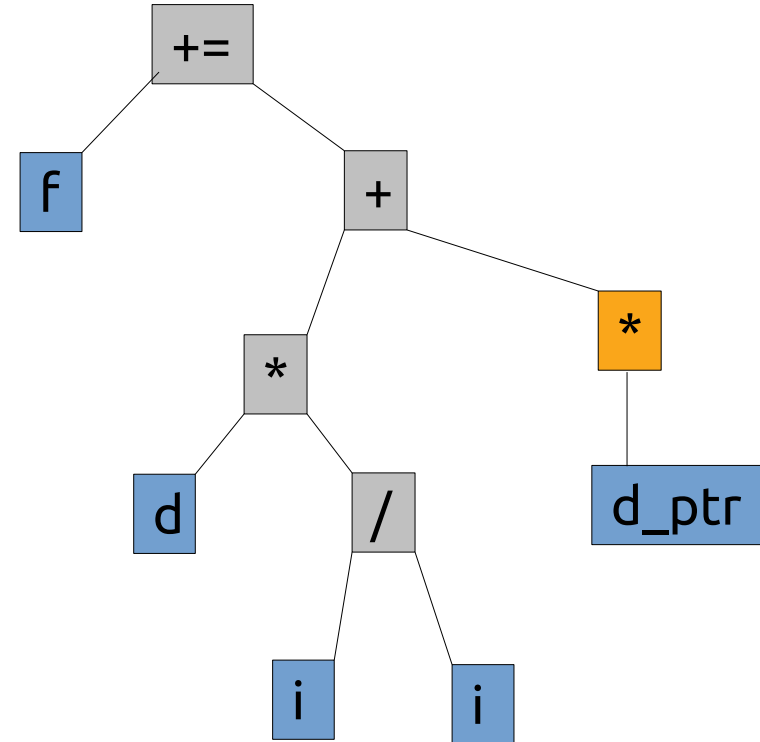
```
float f, int i, double d, bool  
a;
```

```
double* d_ptr;
```

```
f+=d* (i/i) + (*d_ptr)
```

An expression is associate to  
one (or more) parsing trees.

The tree is evaluated according  
to a postorder visit  
(left,right,self)





# Expressions

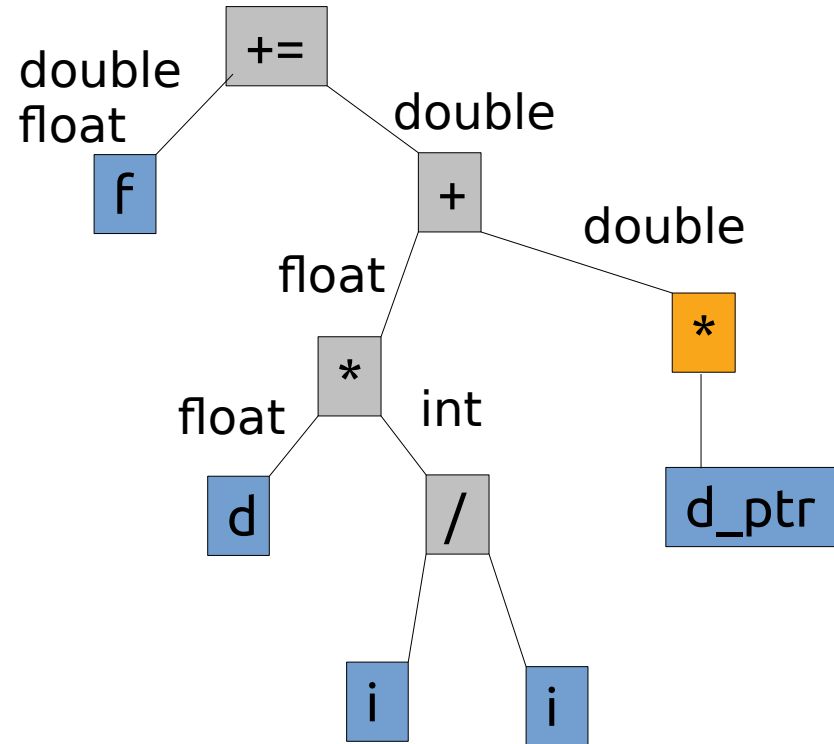
```
float f, int i, double d, bool  
a;
```

```
double* d_ptr;
```

```
f+=d* (i/i) + (*d_ptr)
```

Implicit type conversion can  
occur during the evaluation

Types are “upcasted” to the  
more powerful type



# Expressions

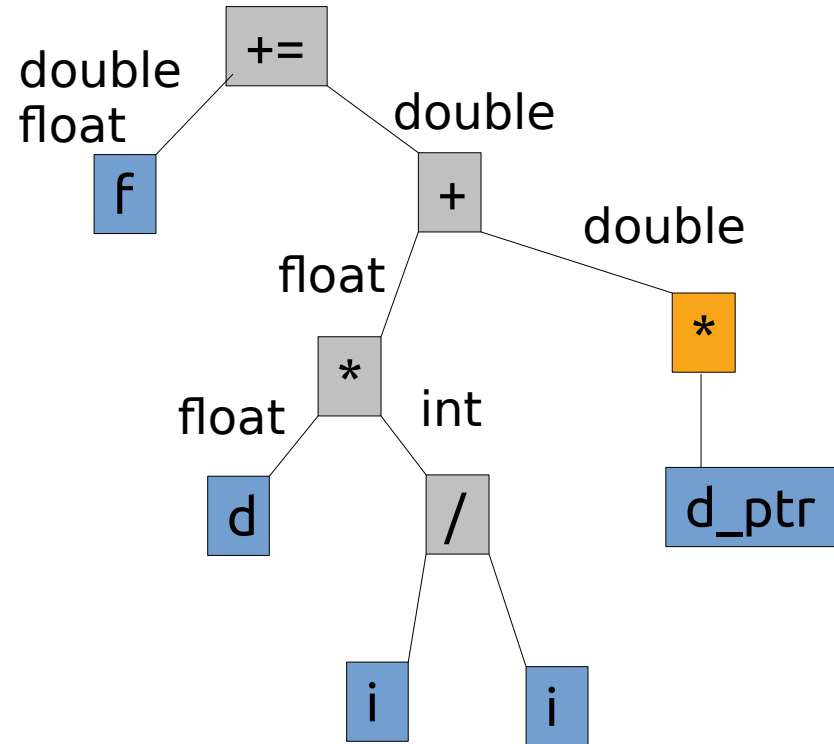
```
float f, int i, double d, bool  
a;
```

```
double* d_ptr;
```

```
f+=d*(i/i)+(*d_ptr)
```

Assignment executes a forced  
cast (and conversion) to the  
type of the left hand side

```
i=f; //f is truncated to  
int
```



# Structures

In C/C++ you can define new types **and** how operators on new types behave

Declaring a new datatype consisting of an aggregation of fields:

```
struct <name> { <field1>; ... ; <field N>;
```

Field can be either a variable or a function (method)

Example:

```
struct MyStruct {  
    int i;  
    float f;  
    void clear() {i=0; f=0};  
};
```

# Structures

Declaring a struct variable, done as if it was a “base type”

```
MyStruct s1, s2; // declares a structs s1, and s2;  
s1.i=5;          // assigns 5 to the field i of s1  
s1.f=3.5;        // assigns 5 to the field i of s1  
  
s2=s1;           // assigns to s2 all values of s1
```

# Functions

A function is declared as

```
<return_type> <function_name> (<arg_list>)  
{ <function body> }
```

Example:

```
int addOne(float f) {  
    return f+1;  
    // the return instruction terminates the  
    // function and returns to the caller the  
    // value of the expression at its right  
}
```

# Functions

A function is declared as

```
<return_type> <function_name> (<arg_list>)  
{ <function body> }
```

Example:

```
MyStruct addOne(MyStruct s) {  
    s.i+=1;  
    s.f+=1;  
    return s;  
}
```

# Functions

Functions are called by the () operator, passing the argument list

```
MyStruct s1, s2;  
int a=addOne(0.5);  
s2=addOne(s1)
```

# Flow Control

- If / else
- For
- Do
- While
- Blocks
- Continue
- Break
- Switch

```
if (<boolean expr>)  
    <statement>
```

```
if (<booleanexpr>)  
    <statement_true>  
else  
    <statement_false>
```



# Flow Control

- If / else
- For
- Do
- While
- Blocks
- Continue
- Break
- Switch

```
if (a==5)  
    a+=5;
```

```
if (!(a%2))  
    ++a;  
else  
    --a;
```

# Flow Control

- If / else
- **For**
- Do
- While
- Blocks
- Continue
- Break
- Switch

```
for (<init>;<cond>;<inc>)  
    statement;
```

Example

```
for (int a=0;a<3;++a)  
    b+=a;
```

# Flow Control

- If / else
- **For**
- Do
- While
- Blocks
- Continue
- Break
- Switch

```
for (int a=0; a<3; ++a)  
    b+=a;
```

# Flow Control

- If / else
- For
- **Do**
- While
- Blocks
- Continue
- Break
- Switch

```
do  
    <statement>  
while (<expr>)
```

```
int i=3;  
do  
    --i;  
while (i);
```

# Flow Control

- If / else
- For
- Do
- **While**
- Blocks
- Continue
- Break
- Switch

```
while (<expr>)  
    <statement>
```

```
int i=10;  
while (! i)  
    --i;
```

# Flow Control

- If / else
- For
- Do
- While
- **Blocks**
- Continue
- Break
- Switch

A statement can be either

- An expression
- A variable declaration
- A control flow instruction (more to come)
- A block

# Flow Control

- If / else
- For
- Do
- While
- **Blocks**
- Continue
- Break
- Switch

A block denotes a sequence of instructions enclosed by {}

```
{  
    <statement1>;  
    <statement2>;  
}
```

All non block statements terminate with a semicolon (;)

# Flow Control

- If / else
- For
- Do
- While
- **Blocks**
- Continue
- Break
- Switch

Example

```
for (int a=0;a<3;++a) {  
    if (a%2)  
        b+=a;  
    f+=1;  
    s1=addOne (s1) ;  
}
```



# Flow Control

- If / else
- For
- Do
- While
- Blocks
- **Continue**
- Break
- Switch

The **continue** statement within a loop causes the sequence of instructions to be interrupted and start from a new iteration of the loop

```
for (int a=0;a<5;++a) {  
    if (a>2)  
        continue;  
    f+=1;  
    // f incremented only 3 times  
    // a counts to 5 anyway;  
}
```

# Flow Control

- If / else
- For
- Do
- While
- Blocks
- Continue
- **Break**
- Switch

The **break** statement within a loop causes the sequence of instructions to be interrupted and exits the loop

```
for (int a=0; a<5; ++a) {  
    if (a>2)  
        break;  
    f+=1;  
    // f incremented only 3 times  
    // a counts to 3;  
}
```

# Flow Control

- If / else
- For
- Do
- While
- Blocks
- Continue
- Break
- **Switch**

Switch handles multiple choices

```
switch (<var>) {  
    case value1: <statements 1>  
    case value2: <statements 2>  
    case value3: <statements 3>  
    . . .  
    [default]: <statement_d>;  
}
```

# Flow Control

- If / else
- For
- Do
- While
- Blocks
- Continue
- Break
- **Switch**

Switch handles multiple choices

```
switch (c) {  
    case '+' : a+=1; //clause 1  
    case '-' : a-=1; //clause 2  
    default : ;  
}
```

```
if (c=='+') all statements  
after clause 1 are matched.  
need to put break;
```

# Flow Control

- If / else
- For
- Do
- While
- Blocks
- Continue
- Break
- **Switch**

Add a break as last statement in each switch case;

```
switch (c) {  
    case '+' :  
        a+=1;  
        break;  
    case '-' :  
        a-=1;  
        break;  
    default : ;  
}
```

# Scopes and Visibility

Variables are visible only in the scope/block where they are declared;

Exiting a scope/block causes the variables declared in them to be destroyed;

Member variables of a struct/class are accessible by the methods of the struct.

# Pointers

A pointer is a memory address of a variable having a certain type

The type of a pointer to variable of type `<T>` is `<T>*`

The address of a variable can be obtained by the operator `&`

Pointers can be assigned

Pointed variables (if not `const`) can be modified, and dereferenced with `*ptr`;

Example

```
int i=0; j=5;  
int* ptr_to_i = &i;  
j=(*ptr_to_i)+1;  
*ptr_to_i=4;
```

```
declare two ints  
assign to ptr_to_i the address  
assign to j the value of i+1;  
what is the value of i?
```

# Arrays

A static array of a certain type can be declared with

`<type> <name> [<integer_constant>]`

Elements of an array can be accessed with `[idx]` after the variable

example:

```
int int_vec[5];
```

```
MyStruct s_vec[10];
```

```
MyStruct s1=s_vec[3]; // element access;
```



# Multi Dimensional Arrays

Arrays can be nested

```
<type> <name> [<c1>] [<c2>] . . . [<cn>]
```

declares a multi dimensional array.

The elements are stored in row major order.

Example:

```
float f[3][3];  
for (int r=0; r<3;++r)  
    for (int c=0; c<3;++c)  
        f[r][c]=r*c;
```

# Pointers and Arrays

A pointer to an array is the location of its first element.

The pointers to the successive elements can be obtained by applying an offset. Pointers support the [] operator;

Example:

```
float f[5];
```

```
float* f_ptr=f; // legal
```

```
*(f_ptr+1)=9; // equivalent to f[1]=9
```

```
f_ptr[3]=9; // also legal, means f[3]=9;
```

# Pointers and Arrays

Multidimensional arrays can be accessed through pointers using their row\_major order;

Example:

```
float f[5][5];
```

```
float* f_ptr= (float*) f; // pointer cast;
```

```
*(f_ptr+5*3+2)=9; // equivalent to f[3][2]=9
```

# Stack and Heap

Variables in the current scope exist within the scope, and get destroyed after the execution flow exits the scope.

These variables are said to be allocated on the **stack**.

In C/C++ you can declare variables that survive the scope by using the **heap** and dynamic memory allocation.

Doing so you become responsible of freeing the variables you no longer use.

Variables allocated on the heap should be referenced by at least one pointer variable in the current scope otherwise they are definitely lost.

C++ standard library offers some construct to prevent memory leaks.

# Heap Allocation

To allocate a variable on the heap

```
<pointer type> = new <type>; // single var allocation
```

```
<pointer type> = new <type> [<int_size>]; // array  
allocation
```

Example

```
// allocates a single MyStruct object
```

```
MyStruct* s_ptr=new MyStruct;
```

```
// allocates an array of 10 floats
```

```
float* v=new float[10];
```

# Heap Deallocation

Objects on the heap should be destroyed when no longer used

To delete an object

```
delete <ptr>;
```

or

```
delete [] <ptr>; // for arrays
```

**Example**

```
delete s_ptr;
```

```
delete [] v;
```

# Again on Heap

```
float* f2;
{
    float* f=new float[100];
    f[10]=0;
    MyStruct* s=new MyStruct;
    s->i= 10; // equivalent to (*s).i=10, more elegant;
    f2=f;
    delete [] f;
}
f2[10]=5; // error, object has been destroyed;
// error s is lost
```

# Again on Heap

```
MyStruct* s=0; // keep track of s, the pointer
survives
float* f2=0;
{
    float* f=new float[100];
    f[10]=0;
    s=new MyStruct;
    s->i= 10; // equivalent to (*s).i=10, more elegant;
    f2=f;
}
f2[10]=5; // error, object has been destroyed;
```



# Structs with Methods

Functions declared inside a struct become member functions (methods).

A function inside the struct has access to all members of that **instance** of struct.

Methods can be called as if they were regular fields.

The pointer to the instance in the method is accessed by **this**.

Example

```
struct MyStruct {  
    int i;  
    float f;  
    MyStruct addOne() {  
        MyStruct other=*this;  
        other.i+=1;  
        other.f+=1;  
        return other  
    }  
}
```

# Structs with Methods

Functions declared inside a struct become member functions (methods).

A function inside the struct has access to all members of that **instance** of struct.

Methods can be called as if they were regular fields.

The pointer to the instance in the method is accessed by **this**.

Example

```
MyStruct s;  
s.i=0; s.f=1;  
MyStruct s2=s.addOne();
```

# Constructors and Destructors

**Constructors** are special methods called upon allocation/initialization of a variable.

**Destructors** are methods called automatically when the object is destroyed (or the pointer deleted);

```
struct <Type> {  
    <Type> () ; // default constructor  
    <Type> (<args>) ; // ctor with args, overrides default.  
    <Type> (const <Type>& other) ; // copy ctor, invoked on return/copy by value  
    ~<Type> () ; // destructor  
};
```

Default copy ctor copies fields, dtor, does nothing;

# Constructors and Destructors (VecF)

```
struct VecF {
    int size; float* v;
    float get(int i) { return v[i];}
    void set(int i, float f) {v[i]=f;}
    VecF() { size=0; v=nullptr;}
    VecF(int size){
        this->size=size;
        v=new float[size];
    }
    VecF(const VecF& other) {
        size=0; v=0;
        if (! other.size) return;
        size=other.size; v=new float[size];
        for (int i=0; i<size; ++i)
            v[i]=other.v[i];
    }
}
```

```
~VecF() {if (size) delete [] v;}

VecF& operator =(const VecF& other)
{
    if (size) delete[] v; size=0; v=0;
    if (! other.size) return *this;
    size=other.size; v=new float[size];
    for (int i=0; i<size; ++i)
        v[i]=other.v[i];
}
```

**This class appears as a native type. Also supports assignment through = operator!**

# VecF: Example

```
VecF v5(5);  
v5.set(0, 0.1);  
v5.set(1, 0.2);  
...  
VecF v7(v5); // (copy ctor)  
VecF v8=v5;  // (copy ctor)  
v8=v7;       // op=
```

# References

In contrast to C, C++ defines another type of data: the reference.

A reference to an object can be accessed as the object itself, but it does not require copies.

Changes done to a referenced object do side effect on the original copy.

Think to a reference as a dereferenced pointer with a bit more sugar

```
<Type>&  <name> = <variable of Type>;
```

```
int i=1;  
int& i_ref=i;  
i_ref=5;  
i=?;
```

# Inheritance

C++ supports inheritance, with rules similar to Java and Python

A C++ class can derive from multiple base classes

A derived class/struct has all members of the base class plus its own

A pointer to the derived class is also a pointer to the base class but the opposite is not true

When deriving a class, the constructor chain should be preserved (if the default ctor has been deleted in the base, a specific one should be called)

# Inheritance

```
<class|struct> Type [:<public|protected|private>  
<BaseType>] { <body> }
```

Example

```
class A1 {...};
```

```
class A2 {...};
```

```
class B: public A1, public A2 { ... }
```



# Inheritance Example

```
#include <iostream> // discover this later
using namespace std; // discover this later

struct A {
    int my_value;
    A(int v) {
        cerr << "A["<<this<<"]"<<"ctor" << endl;
        my_value=v;
    }
    ~A(){
        cerr << "A["<<this <<"]"<<"dtor" << endl;
    }
    void print() {
        cerr << "A::print() [" << this << "]:
my_value=" << my_value << endl;
    }
};
```

```
struct B: public A {
    int my_other_value;
    // use the initializer list before {
    // it would not be possible to initialize
    // objects whose default ctor has been deleted
    B():
        A(0), // base class initialization
        my_other_value(-1){
        cerr << "B [ " << this << "]" ctor " << endl;
    }

    ~B(){
        cerr << "B [ " << this << "]" << " dtor" << endl;
    }
    void print() {
        // call method of the base class using scoping
        <classname>::<method_or_variable>
        A::print();
        cerr << "B::print() [" << this << "]:
my_other_value=" << my_other_value << endl;
    }

};
```

# Inheritance Example

Simple test main

We try assigning objects to derived and base class variables

Assigning from base to derived does not work, since there might be non initialized fields

The other way, in general works (under the defaults)

```
int main(int argc, char** argv) {
    cerr << "create A " << endl;
    A a(3);
    a.print();

    cerr << "create B " << endl;
    B b;
    b.print();

    // legal assignment a is base class
    // of b, only common member vars are
    // copied
    A a1=b;
    a1.print();
}
```

# Late Binding

In Java all methods support overriding. In C++ only those that are declared virtual. This choice favors efficiency, since pure “non virtual” classes/structs enable for more effective compile time optimizations.

To enable the overriding of a method, you need to put the keyword “**virtual**” before its declaration in the class.

In the derived class you might want (optional) to claim a certain method has been overriding one on the base class. Use the **override** clause after the signature

From that point on, all methods in the derived classes whose signature matches the one of the overridden method will be called.

You can still use a “specific” method of a base class calling it explicitly with the scoping operator `<Type>::method`

# Late Binding Example

```
class A {
public:
    A(int value_):
        _value(value_){
    }
    void print() {
        cerr << "A:["<<this<<"], value:"<<_value<< endl;
    }
protected:
    int _value;
};

class B: public A {
public:
    B(int value_):
        A(value_) {}
    void print(){
        cerr << "B:["<<this<<"],value:"<<_value<<endl;
    }
};
```

```
class A_lb {
public:
    A_lb(int value_):
        _value(value_){
    }
    virtual void print() {
        cerr<<"A_lb:["<<this<<"],value:"<<_value<<endl;
    }
protected:
    int _value;
};

class B_lb: public A_lb {
public:
    B_lb(int value_):
        A_lb(value_) {}
    void print() override {
        cerr<<"B_lb:["<<this<<"],value:"<<_value<<endl;
    }
};
```

# Late Binding Example

To exploit the power of polymorphism we need to use references or pointers.

The “concrete” object, is still an instance of a specific type in memory.

Structs and Classes in C++ are the same. The first default to public, the second to private.

It is a good custom to use structs for simple data-only items, while using classes for all the rest

```
int main(int argc, char ** argv) {
    cerr << "no late binding" << endl;
    {
        B b(10);
        A& a_ref=b;
        b.print();
        a_ref.print();
    }
    cerr << "late binding" << endl;
    {
        B_lb b(10);
        A_lb& a_ref(b);
        b.print();
        a_ref.print();

        // a is a copy of b
        A_lb copy_of_b_as_a(b);
        copy_of_b_as_a.print();
    }
}
```

# Declaration/Definition

```
#pragma once // put this once at the beginning
```

```
class A {  
    A();  
    int methodA(int i);  
};
```

a.h

```
class B {  
    B();  
    void methodB(A& a);  
};
```

```
#include "a.h"
```

```
... other includes
```

a.cpp

```
A::A() {  
    <ctor definition>  
}  
int A::methodA(int) {  
    definition  
}  
B::B() {  
    <other ctor>  
}  
void B::methodB(A& a) {  
    other method definition  
}
```

For non-template classes it is a good custom to separate

**declaration** of classes (in a .h) and

**definition** (in a .cpp)

The scoping operator in the cpp tells to which class a method belongs

# Operator Overloading

C++ operators (but the .) can be overloaded

This means that you can assign a user defined behavior to a newly defined type.

Syntax, for member function

```
class <name> {  
...  
    <return type> operator <operator_type> (<arguments>) ;  
}
```

the first argument in the list is always the calling object, in class syntax has 1 less argument

Syntax, of non member function

```
<return type> operator <operator_type> (<arguments>) ;
```

# Operator Overloading

C++ operators (but the .) can be overloaded

This means you can assign an operator on a new type a user defined behavior

Syntax, for member function

```
struct VecF {  
...  
    VecF operator+(const VecF& other);  
}
```

the first argument in the list is always the calling object, in class syntax has 1 less argument

Syntax, of non member function

```
VecF operator+(const VecF& first, const VecF& second);
```



# Operator Overloading

Syntax, for member function

```
VecF VecF::operator+(const VecF& other) {  
    Vecf returned(*this);  
    if (size!=other.size) {  
        cerr << "error << endl;  
        return returned;  
    }  
    for(int i=0; i<size; ++i)  
        returned.v[i]+=other.v[i];  
    return returned  
}
```

# Operator Overloading

Syntax, of non member function

```
VecF operator+(const VecF& first, const VecF& second) {  
    Vecf returned(first);  
    if (first.size!=second.size) {  
        cerr << "error << endl;  
        return returned;  
    }  
    for(int i=0; i<size; ++i) returned.v[i]+=second.v[i];  
    return returned;  
};
```

# Using overloading

```
Vecf v1(3), v2(3), v3(3);  
... //populate vectors  
v3=v2+v1;    // yes it does the sum
```

The parsing tree is generated according to the usual precedence of the operators. The evaluation is carried on along the parsing tree.

You can define complex and matrix types as you wish and have a syntax similar to the one used for scalar types.

# Exercises

0. Extend the VecF class so that it supports

- Element access (by reference)
- Addition (done)
- Subtraction
- Multiplication by a scalar ( $v*s$ )
- Dot product ( $v*v$ )

1. Write a “binary tree” class for integers, that supports sorted insertions and existence checks

2. Write a matrix class supporting

- Addition
- Subtraction
- Multiplication by a scalar
- Multiplication by a vector
- Multiplication by another matrix