SAPIENZA
UNIVERSITÀ DI ROMA

Robot Programming Introduction to C++ (marathon version)

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--`	Suffix/postfix increment and decrement	
	type`()` *type*`{}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`. ->`	Member access	
3	`++a --a`	Prefix increment and decrement	Right-to-left
	`+a -a`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new new[]`	Dynamic memory allocation	
	`delete delete[]`	Dynamic memory deallocation	
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	

https://en.cppreference.com/w/cpp/language/operator_precedence

Operators and Expressions

7	`<<` `>>`	Bitwise left shift and right shift	
8	`<=>`	Three-way comparison operator (since C++20)	
9	`<` `<=` `>` `>=`	For relational operators < and ≤ and > and ≥ respectively	
10	`==` `!=`	For equality operators = and ≠ respectively	
11	`&`	Bitwise AND	
12	`^`	Bitwise XOR (exclusive or)	
13	`|`	Bitwise OR (inclusive or)	
14	`&&`	Logical AND	
15	`||`	Logical OR	

16	`a?b:c`
	`throw`
	`co_yield`
	`=`
	`+=` `-=`
	`*=` `/=` `%=`
	`<<=` `>>=`
	`&=` `^=` `|=`
		Ternary conditional[note 2]
		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

17	`,`	Comma	Left-to-right

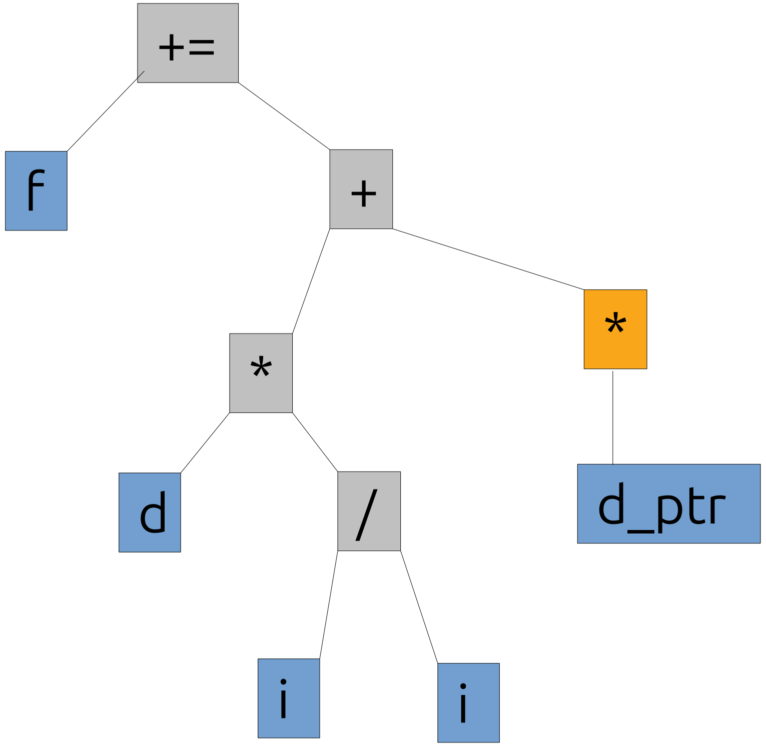
https://en.cppreference.com/w/cpp/language/operator_precedence

Expressions

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

double* d_ptr;


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



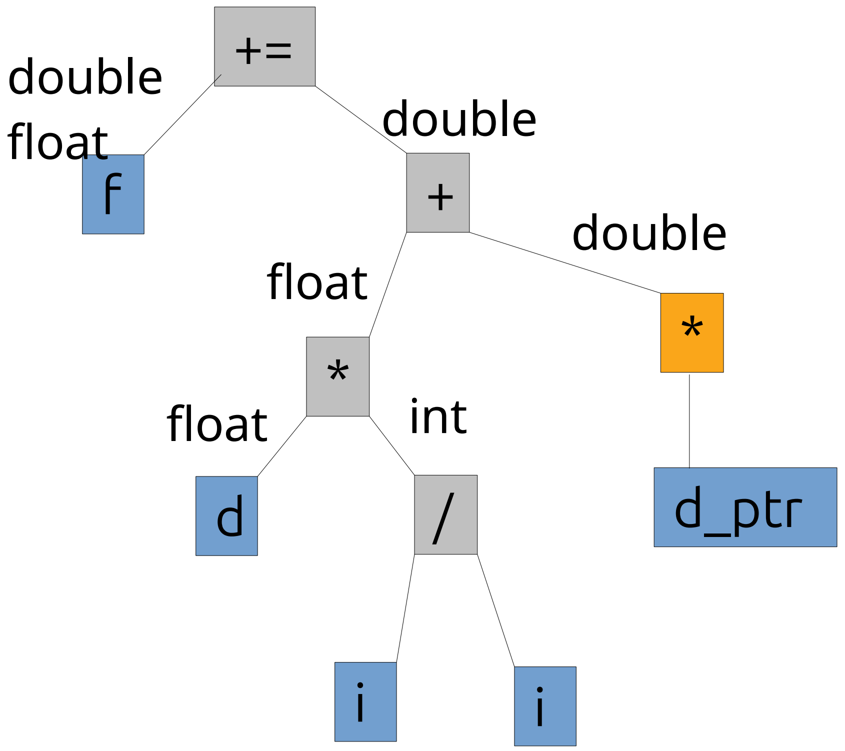
An expression is associate to one (or more) parsing trees.

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

Expressions

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

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



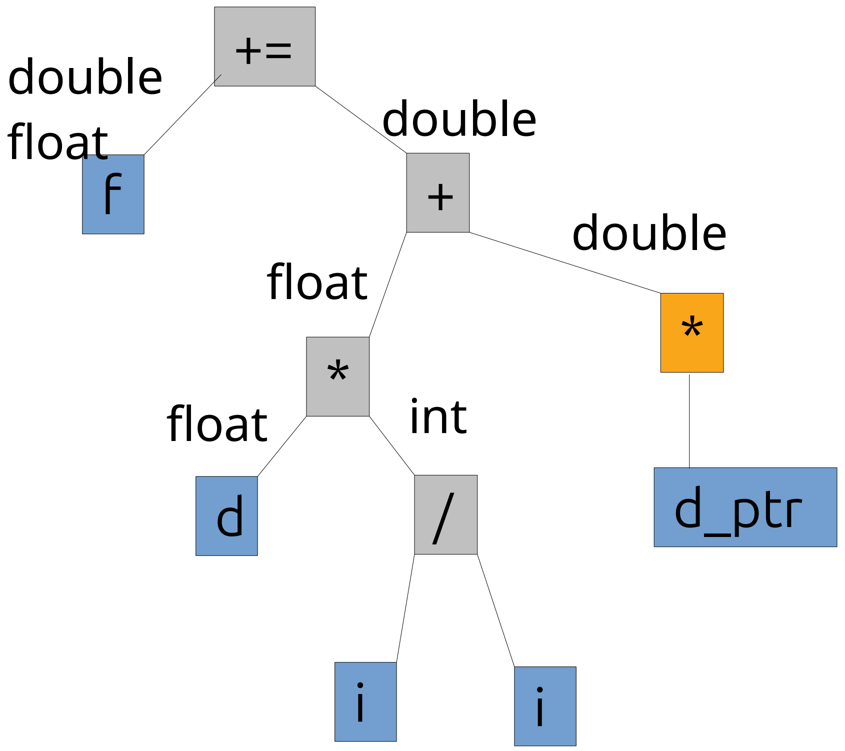
Implicit type conversion can occur during the evaluation

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 on 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 structs s1,  and s2;
s1.i=5;            // assignes 5 to the field i of s1
s1.f=3.5;          // assignes 3.5 to the field f

s2=s1;             // assignes 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;
s1.i=0;
s1.f=0.5;
s2=addOne(s1)

//s2={1, 1.5}
```

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

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

Example:

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

Flow Control

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

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

Example:

```
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 2 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 2 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: <statemens 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;               declare two ints
int* ptr_to_i = &i;         assign to ptr_to_i the address
j=(*ptr_to_i)+1;            assign to j the value of i+1;
*ptr_to_i=4;                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_ptr+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
```

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=??;
```

TAKE A BREAK

NOW

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;
  delete [] 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 copyes 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];
  return *this;
}
```

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

Declaration/Definition

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

struct A {

   A();
    int methodA(int i);

};


struct B {

   B();

   void methodB(A& a);

};
```

a.h

```
#include "a.h"
… other includes
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

a.cpp

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=
```

Operator Overloading

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

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

Sintax, for member operator

```
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

Sintax, of non member operator

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

Operator Overloading

Sintax, for member operator

```
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
}
```

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 sintax 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 matrix class supporting

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

2. (optional) Write a “binary tree” class for integers, that supports sorted insertions and existance checks