

Programs is understandable.

↳ correctness and reliability
Maintainable
well design.

Development process:

- ① Analysis
- ② Design
- ③ Implementation
- ④ Repeat

Syntax and Semantics

Syntax is structure as semantics is to meaning

Syntax

- ① Token: smallest piece of program language
- ② Expression: group of token, phrase
- ③ Statement: putting phrase together, meaningful command

Semantics

- Syntactically correct \neq correct semantics
- Cause program to crash

Grammar

define the syntax is the program language

<expression>

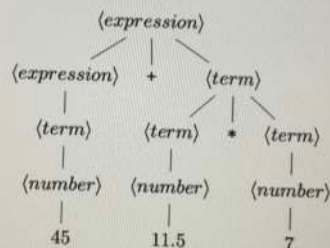
<term>

<number>

(expression)	::=	(term)
		(expression) '+' (term)
		(expression) '-' (term)
(term)	::=	(number)
		(term) '*' (number)
		(term) '/' (number)
(number)	::=	'floating-point literal'

19.

► Parsing the expression 45 + 11.5 * 7 is as easy as:



Hello World.

#include <iostream> ← Not include in core func

int main() { ← Required, Entry Function (Main func)

std::cout << "Hello World" << std::endl;

return 0; ← 0 communicated successful terminated

} of the application. No longer Required.

↖ Braces block

std::cout define in iostream.

<< : insert notation.

'' '' String, ' ' character
std::endl ostream

std Particular entity in the standard namespace

Object

Type	Defines a set of possible values and a set of operations for an object
Object	Our abstraction of a memory cell; holds a value of a given type
Value	The set of bits in memory interpreted according to a type
Variable	A named object
Declaration	A statement that gives a name to an object
Definition	A declaration that sets aside memory for an object

build types

Boolean (bool)
Character (char)
int (int)
Floating (double)

Boolean, character and int are **integral values**

bool: true / false

char: 'a' single character.

Other types: @ pointer, Array, references, data structure and classes

bool

- Convert to integer, true: 1 false: 0
int \rightarrow Bool, 0 \rightarrow false, not zero \rightarrow true

Char

- 26 characters of English
- 0-9
- Basic punctuation character
- Escape character: use backslash

Int

- ① short
- ② int
- ③ long

three forms:

$\left\{ \begin{array}{l} \text{int} \\ \text{signed int} \\ \text{unsigned int} \end{array} \right.$

Integer literals

four forms $\left\{ \begin{array}{l} \text{Decimal} \\ \text{Binary (start with 0b)} \quad \text{int } b = 0b1110010 \\ \text{Octal (start with a 0)} \quad \text{int } i = 042 \\ \text{Hexadecimal (start with 0x)} \quad \text{int } i = 0x42 \end{array} \right.$

Suffix $\left\{ \begin{array}{l} \text{u : unsigned literal} \quad 101U \\ \text{L : long literal} \quad \text{long } i = 101L \end{array} \right.$

Floating-point type

Computer's approximation to the math concept of real #

three form $\left\{ \begin{array}{l} \text{float} \\ \text{double} \\ \text{long double} \end{array} \right.$

Suffix: $\left\{ \begin{array}{l} \text{F} : \text{float} \quad 3.14\text{F} \\ \text{L} : \text{long double} \end{array} \right.$

Variables

Attribute $\left\{ \begin{array}{l} \text{Name} \\ \text{Type} \\ \text{Address} \\ \text{Scope} \\ \text{Value} \\ \text{Lifetime} \end{array} \right.$

Rule $\left\{ \begin{array}{l} \text{The 1st character must be letter.} \\ \text{Case sensitive} \\ \text{Not recommend to start with "-" } \end{array} \right.$

Address

memory address where data is

Type

It determines the:

- ① range of values the variable can store
- ② set of operations that are defined on the value

Value

Value of variable is the content of the memory cell

- Define the same variable twice is an **error**

Lifetime

the time of variable that bound with memory position

Scope

define by { }

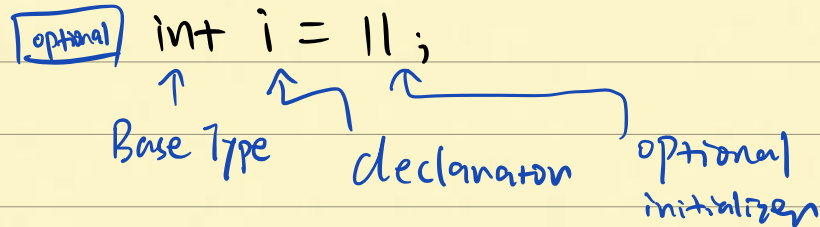
- When a variable with the same name exists in nested scopes, the variable in the inner scope hides the variable in the outer scope.
- Should declare a variable in as local a scope as possible and closest to its first use as practicable

Declaration

- ① An optional specifier.
- ② A base type

③ A declarator

④ An optional initializer



Declarator

prefix or postfix. common declarator include:

*	int* p	prefix
*const	int*const p	pre
&	int& r	pre
[]	int arr[]	post
()	int fnc()	post

- postfix bind more tightly than prefix
- declarator only apply to a single name only

Constant

Cannot be change after assigned, use it by using non-type specifier const in the object's declaration

const int (k)ReadOnlyVariable = 7



Always start with k

- have to have initializer

Type Conversion

Narrowing Conversion

Convert a value to a type that cannot store even approximation of all the value in original type

eg. double \rightarrow float \because range of double $>$ float

Widening Conversions

Can include at least approximations of all the values of original Type. Always safe

eg. float \rightarrow double

Implicit type Conversion (coerced)

Automatic conversion of values from one type to another

narrow \rightarrow wider

eg. $3.14 + 8 \rightarrow 3.14 + 8.0 \rightarrow 11.14$

\uparrow
int \rightarrow double.

int i = 9.0 / 5.0 = 1

1.8 \rightarrow 1

double d; d = 9 / 5

1 \rightarrow 1.0

Explicit type Conversion (Cast)

Static - cast <type to cast> (value to cast)

eg. Convert int value of 5 → double

Static - cast <double> (5)

Safe / unsafe conversion

From	To
bool	char
bool	int
bool	double
char	int
char	double
int	double

Expression and Statement

Expression

Any variable name, constant, or literal is an expression

One or more expressions combine by an operator also constitutes an express

eg. $x + y$
 $x = 3 + 2$

Token: the smallest piece of a programming language that has meaning.

Operator

Unary operator: -7

Binary operator: $8+7$

Unary + Binary operator: $8-7$

• $x = a ? b : c$

if a is true, $x = b$

else, $x = c$

Grouping operators and operands

- precedence
- associativity
- order of evaluation

precedence

优先级

eg. $3 + 4 * 5 = 3 + 20 = 23$ not $7 * 5 = 35$

Associativity

how operators of the same precedence are grouped.

eg. $int\ i = 1$
 $int\ j = 0$

$i = j = 5 \Rightarrow 0\ j = 5$

↪↪

② $i = j$ while $j = 5$, then $i = 5$

Order of evaluation

precedence specified how the operands are grouped.

es. $\text{int } i = f_1() * f_2()$

- f_1 and f_2 must be called before multiplication can be done

Associativity	Operator	Function	Use
Right	!	logical NOT	! expr
Left	<	less than	expr < expr
Left	<=	less than or equal	expr <= expr
Left	>	greater than	expr > expr
Left	>=	greater than or equal	expr >= expr
Left	==	equality	expr == expr
Left	!=	inequality	expr != expr
Left	&&	logical and	expr && expr
Left		logical or	expr expr

Error in Expression

① Overflow: when calculation produces a result greater in value than which can be stored.

es. $\text{Max } \# + 1$

② floating # imprecise

es. $0.15 + 0.15 = 0.29999\dots$

③ propagation of error

A lot of floating # imprecise cause big error when repeat

Statement

Complete and meaningful command that can be given to a computer

eg. `int x=5;`
`int y=f(x) * 3.5;`

Empty statement

eg: `;`

Compound statement

refer as block, not terminated by semicolon

Branching

< if
 switch

if

```
if (condition)
    what;
```

```
else
    what; { optional
```

```
if (condition) {
    //statement
    ;
```

```
} else if (condition) {
```

```
}
```

Switch

```
switch (op-code) {
    case 0: {
```

```
}
```

```
case 1: // fall through
case 2: {
}
default: {
}
```

- should always have a default case
- if default case should never execute, treat it as error

Iteration.

while
do while
for

do while

Similar to while, except the condition is tested at the end.

for

```
for (int i=1; i<=10; ++i)
{
}
```

procedure: ① $i=1$

↓

if $i \leq 10$? ←

↓ yes

execute → +1

Loop Control

- ① Break
- ② Continue

Function

first line of organization of programming.

- ① Small
- ② Blocks and indenting.
- ③ indent level of function should not be greater than one or two
- ④ should do one thing
- ⑤ Should have one level of abstraction.
 - write code that read like top-down narrative.
- ⑥ descriptive Name.
- ⑦ Small # of argument

Monadic function

- single parameter works as:
 1. Ask question about an object
 2. perform an operation on an object.

Dyadic function

two parameter

Convert into monads when possible

Triads

three parameter.

- Avoid flag parameter. split it instead.
- No side effect: Cannot imply change data or variable.
- Don't repeat: eliminate by pack into function.

Function

① Can avoid Return

② parameter specified in comma-separated list

```
int Square(int x);
```

```
int Square(int x) {  
    return x*x;  
}
```

#include : import fun

Header file:

Contain all the information we need to understand how to call a function

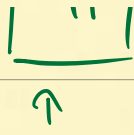
.cc

.hpp

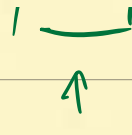
.cc



↑
use
the 3th
.cc



↑
header

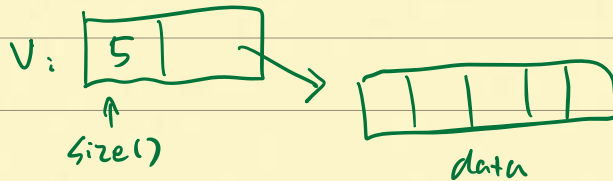


↑
implement
header

Vector

sequence of elements of that you can access the index

- Store both size and element



declare: `std::vector<int> v = {1, 1, 2, 3, 5};`

`std::vector<double> v(4)`

↑
size

- The element are given default value according to type.

Access: `v[2]` `v[2]=10;` `v.at(2)` , `v.at(2)=10`
} tell if exist

size-t: Maximum possible vector length

push-back: append data

Two dimensional Vector

declare: `std::vector (std::vector (int)) vect;`

String

`std::string` complete the string literal

`substr (x, y)` : substring

int Main

`int main (int argc, char* argv[])`

↓ ↑
length input list

`argv[]` initial with length 1, with program output name

Input / Output

need `iostream`

write file

`include <fstream>`

`std::ofstream ofs ("filename. ext");`

`if (!ofs.is_open())`

`// do something`

`}`

`ofs << "Hello world" << std::endl;`

- `bin` to exist file will cause overwrite.

- If want to append, use: `std::ofstream ofs ("filename.txt", std::ofstream::app); // append to end`

Read from input

include `iostream`

```
std::string first-name;
```

```
std::string last-name;
```

```
std::cin >> first-name >> last-name;
```

—————→
data flow

- white space as delimited.

```
std::getline(std::cin, full-name); ← No white space eliminate
```

wrong input

```
int i = 0;
```

```
char c = '0';
```

```
int j = 0;
```

```
std::cin >> i >> c >> j;
```

3	.	1	4	\n
---	---	---	---	----

 input

⇒ `i = 3`

`c = .`

`j = 14`

```
int i = 0
```

```
double d = 0
```

```
std::cin >> i >> d;
```

3	.	1	4	\n
---	---	---	---	----

 input

`i = 3`

`d = 0.14`

Reading from file

```
include <fstream>
```

```
std::ifstream ifs("filename.txt");
```

```
if (!ifs.is_open()) {
```

```
}
```

```
int i = 0
```

```
double j = 0
```

```
std::string str;
```

```
ifs >> i >> j >> str
```

10 3.14 text
filename.txt

⇒ i = 10

↑ j = 3.14

↑ str = text

Struct

Struct is an aggregate of elements of nearly arbitrary types

```
struct Contact {
```

```
    std::string first-name;
```

```
    std::string last-name;
```

```
};
```

Access: Contact person;

```
person.first-name = "Michael";
```

Initialize: Contact person = {"Michael", "Nowak", 217244, "...@example.com"};

- Should follow the order in struct define

- Object of struct can be pass / assign to function.

- Cannot be print or compare

Write struct in C++

- Type name start with capital letter

- Name data members are all lowercase

- Use a struct only for passive object that carry data.

- define each struct in a header file.