UNIVERSITÀ
DI PARMA

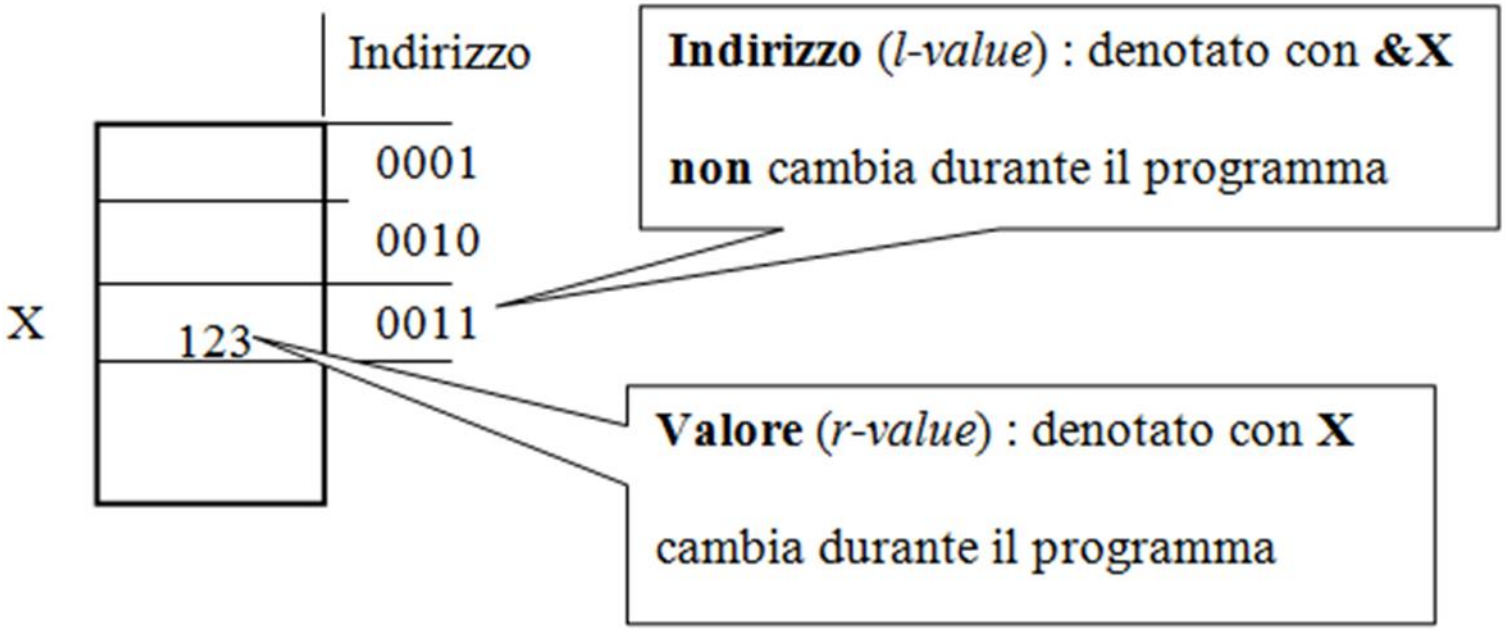
C++ variabili e tipi di dato

variabili

- si definisce variabile uno spazio identificato da un nome in cui è possibile scrivere, recuperare e manipolare dati nel corso del programma
- una variabile è caratterizzata da:
  - il suo valore ***right value*** (*rvalue)*
  - il suo indirizzo, ***left value (****lvalue)*
  - lo spazio occupato
- la ***dichiarazione*** di una variabile associa un identificatore a un tipo e determina l'allocazione di un'area di memoria ***(non assegna un valore alla variabile!)***
- è possibile inizializzare una variabile durante la dichiarazione
- per convenzione l'***identificatore*** (nome variabile) è scritto in minuscolo e dovrebbe essere mnemonico (*non può essere una parola riservata!)*

regole di visibilità

- una variabile può essere utilizzata (cioè è visibile) solo dopo la sua definizione
- le variabili definite in un ambiente (blocco o funzione) sono visibili in tutti gli ambienti in esso contenuti
- all'interno di uno stesso ambiente non posso avere due variabili con lo stesso nome
- in ambiente distinti posso avere variabili con lo stesso nome, anche di tipo diverso
- se in un ambiente sono visibili più variabili con lo stesso nome, il nome si riferisce a quella la cui dichiarazione è «più vicina» al punto di utilizzo


Indirizzo
0001
0010
0011
X
123
Indirizzo (*l-value*) : denotato con **&X**
non cambia durante il programma
Valore (*r-value*) : denotato con **X**
cambia durante il programma

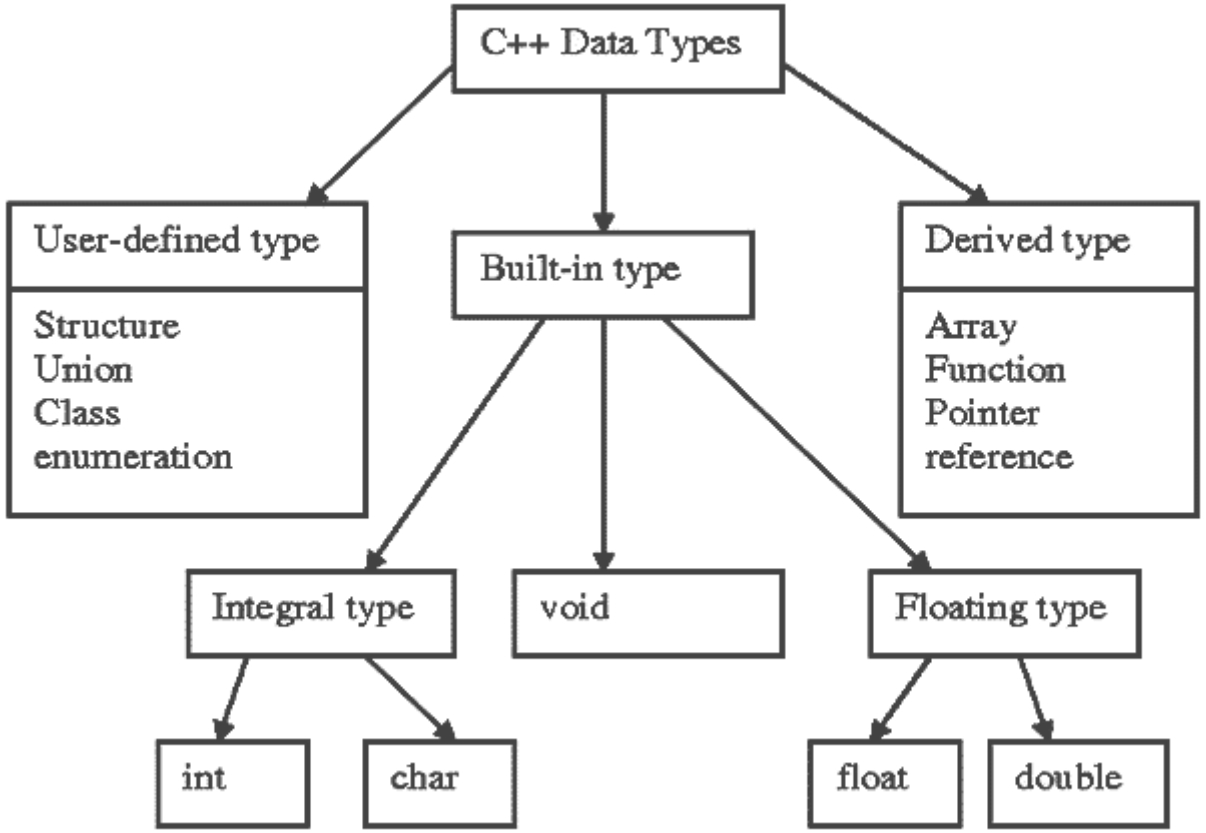
tipo di dato

- è necessario che associare un tipo
  - ad ogni variabile
  - ad ogni parametro di funzione
  - ad ogni valore restituito da una funzione
- i tipi di dato si dividono in 3 categorie
  - tipi di dato ***scalari*** (int, char, float, double, boolean …)
  - tipi di dato ***strutturati*** definiti nelle librerie (string …) o dall'utente (utilizzando struct o class)
  - ***puntatori***
- il tipo specifica
  - la quantità di memoria che verrà allocata per la variabile (o risultato dell'espressione)
  - l'insieme dei valori che è possibile memorizzare in una variabile
  - la modalità di interpretazione di questi valori (schemi di bit - rappresentazione)
  - le operazioni che è possibile eseguire sui valori

dichiarazioni di tipo

- Quando si dichiara una *variabile* è necessario ***specificarne*** il tipo in modo esplicito
  - ***int*** x; *// non è specificato il* ***valore*** *che è* ***indefinito***
  - ***double*** h = 3.7;
  - ***string*** s = "hello";
- oppure utilizzare la parola chiave ***auto*** per indicare al compilatore di *dedurre* il tipo dall'inizializzatore. *type inference*
  - ***auto*** y = 5; *// type inference: C++11*
  - ***auto*** k = 2.2;
  - ***auto*** s = string{"hello"}; *// C++14:* **auto** *s = "hello";*
- quando si dichiara una *funzione* è necessario ***specificare*** il tipo di ***ciascun argomento*** e del ***valore restituito*** (*void* se la funzione non restituisce alcun valore)
  - ***double*** media(***int*** a, ***int*** b);

C++ data types


C++ Data Types
User-defined type
Structure
Union
Class
enumeration
Built-in type
Derived type
Array
Function
Pointer
reference
Integral type
void
Floating type
int
char
float
double

occupazione di memoria

```
size of int:     4 bytes
size of double:  8 bytes
size of float:   4 bytes
size of char:    1 bytes
size of bool:    1 bytes
size of int[10]: 40 bytes
size of int*:    4 bytes
size of double*: 4 bytes
```

```
#include <iostream>
int main() {
    int i; double d;
    float f; char c;
    bool b; int v[10];
    int *pi; double *pd;
    std::cout << "size of int:     " << sizeof i  << " bytes " << std::endl;
    std::cout << "size of double:  " << sizeof d  << " bytes " << std::endl;
    std::cout << "size of float:   " << sizeof f  << " bytes " << std::endl;
    std::cout << "size of char:    " << sizeof c  << " bytes " << std::endl;
    std::cout << "size of bool:    " << sizeof b  << " bytes " << std::endl;
    std::cout << "size of int[10]: " << sizeof v  << " bytes " << std::endl;
    std::cout << "size of int*:    " << sizeof pi << " bytes " << std::endl;
    std::cout << "size of double*: " << sizeof pd << " bytes " << std::endl;
    return 0;
}
```

costanti (limite) numeriche

	http://en.cppreference.com/w/cpp/types/numeric_limits		
	`min()`	`lowest()` (C++11)	`max()`
`numeric_limits< char >`	`CHAR_MIN`	`CHAR_MIN`	`CHAR_MAX`
`numeric_limits< short >`	`SHRT_MIN`	`SHRT_MIN`	`SHRT_MAX`
`numeric_limits< signed short >`			
`numeric_limits< int >`	`INT_MIN`	`INT_MIN`	`INT_MAX`
`numeric_limits< signed int >`			
`numeric_limits< long >`	`LONG_MIN`	`LONG_MIN`	`LONG_MAX`
`numeric_limits< signed long >`			
`numeric_limits< long long >`	`LLONG_MIN`	`LLONG_MIN`	`LLONG_MAX`
`numeric_limits< signed long long >`			
`numeric_limits< float >`	`FLT_MIN`	`-FLT_MAX`	`FLT_MAX`
`numeric_limits< double >`	`DBL_MIN`	`-DBL_MAX`	`DBL_MAX`
`numeric_limits< long double >`	`LDBL_MIN`	`-LDBL_MAX`	`LDBL_MAX`

testLimits.cpp

```
#include <limits>
#include <iostream>
using namespace std;
int main()
{
    cout << "type \t lowest \t highest \n";
    cout << "int \t" << numeric_limits<int>::lowest() << '\t'
         << numeric_limits<int>::max() << '\n';
    cout << "int \t" << INT_MIN << '\t' << INT_MAX << '\n';
    cout << "char\t" << static_cast<int>(numeric_limits<char>::lowest()) << '\t'
         << static_cast<int>(numeric_limits<char>::max()) << '\n';
    cout << "char\t" << CHAR_MIN << '\t' << CHAR_MAX << '\n';
    cout << "float\t" << numeric_limits<float>::lowest() << '\t'
         << numeric_limits<float>::max() << '\n';
    cout << "double\t" << numeric_limits<double>::lowest() << '\t'
         << numeric_limits<double>::max() << '\n';
}
```

```
type     lowest          highest
int      -2147483648     2147483647
int      -2147483648     2147483647
char     -128    127
char     -128    127
float    -3.40282e+038   3.40282e+038
double   -1.79769e+308   1.79769e+308
```

operazioni

- operazioni su numeri:
  - +, -, *, /, %,
  - ++, -- *(attenzione sono assegnamenti)*
  - attenzione: la divisione tra interi dà risultato intero (trunc)
  - assegnamento: =, +=, -= ...
- confronti: >, >=, <, <=, !=, ==
  - attenzione: i confronti non si possono concatenare
- operazioni booleane (and, or, not): &&, | |, !
  - cout << (3 < 5) << endl; // output -> 1
  - cout << (3 < 5 < 4) << endl; // output -> 1 (!!!)
  - cout << (3 < 5 && 5 < 4) << endl; // output -> 0

conversione di tipo (cast)

- nel caso di operazioni con operandi misti il risultato dell'espressione viene automaticamente convertito nel tipo con precisione maggiore
  - esempio: ***(15.2 / 2)*** il risultato sarà di tipo ***float***
  - esempio:

```
double d;
int i = 5;
d = i;                    // il valore di i viene convertito in double
```

  - in questo caso si tratta di ***casting implicito***
- in C++ è possibile effettuare una conversione esplicita (***casting esplicito***) mediante l'operatore cast

```
static_cast<tipo>(espressione)
```

casting esplicito

```
static_cast<int>(4.9) = 4
static_cast<int>(4.2) = 4
static_cast<double>(10) / 3 = 3.33333
static_cast<double>(10 / 3) = 3
(double)15 / 2 = 7.5
```

```
#include <iostream>

using namespace std;

int main() {
	cout << "static_cast<int>(4.9) = " << static_cast<int>(4.9) << endl;
	cout << "static_cast<int>(4.2) = " << static_cast<int>(4.2) << endl;
	cout << "static_cast<double>(10) / 3 = " << static_cast<double>(10) / 3 << endl;
	cout << "static_cast<double>(10 / 3) = " << static_cast<double>(10 / 3) << endl;
	cout << "(double)15 / 2 = " << (double)15 / 2 << endl;
	return 0;
}
```

numeri pseudocasuali

```
#include <cstdlib>
#include <iostream>
#include <ctime>

int main() {
    std::srand(std::time(nullptr)); // use current time as seed for random generator
    int random_variable = std::rand();
    std::cout << "Random value on [0 " << RAND_MAX << "]: "  << random_variable << '\n';
    int range;
    std::cout << "Insert range ";
    std::cin >> range;
    std::cout << "Random value on [0 " << range << "]: "  << random_variable % range <<
'\n';
}
```

Dati strutturati

DEFINIZIONE DI NUOVI TIPI DI DATO

typedef

- la dichiarazione ***typedef*** permette di creare un alias per la definizione di un tipo di dato
- esempio tipo semplice

```
typedef unsigned long ulong;   // ulong nuovo tipo
ulong l;                       // l variabile di tipo unsigned long
```

- esempio struttura

```
typedef struct {
    double x;
    double y;
} punto;                       // punto nuovo tipo di dato
punto p1;                      // p1 variabile di tipo punto
p1.y = 4.5;
```

- per dati con significati speciali, è possibile definire insiemi di valori come sequenze di identificatori tramite il costruttore di tipo enum.

```
enum {Lu,Ma,Me,Gi,Ve,Sa,Do} giorno;
giorno = Me;
// oppure
typedef Mese {Gen,Feb,Mar,Apr,Mag,Giu,Lug,Ago,Set,Ott,Nov,Dic};
Mese meseCorrente;
meseCorrente = Mar;
```

- il primo identificatore ha valore 0, il successivo ha valore 1, e così via. È comunque possibile assegnare agli identificatori valori espliciti:

```
enum Color { red, green, blue };
Color r = red;
switch(r) {
   case red  : std::cout << "red\n";   break;
   case green: std::cout << "green\n"; break;
   case blue : std::cout << "blue\n";  break;
}
```