Dipartimento di Fisica e Astronomia Galileo Galilei

Università degli Studi di Padova

Sperimentazioni di Fisica I mod. A – Lezione 7

Variabili e Tipi Fondamentali in C++ (CAP. 2)

Dipartimento di Fisica e Astronomia "G. Galilei",
Università degli Studi di Padova

Variabili e Tipi Fondamentali

Lezione 7:

Parte 1. Tipi Semplici

Introduzione

- I *tipi di dati* sono un elemento fondamentale per ogni linguaggio di programmazione: ci permettono di definire il significato dei dati e quali operazioni su di essi sono consentite

- Il C++ ha un supporto esteso sui tipi:
  - ✓ definisce svariati tipi primitivi
  - ✓ fornisce i meccanismi per estenderli e definirne di nuovi

- Il linguaggio definisce un insieme di tipi fondamentali aritmetici per rappresentare
  - ✓ caratteri, interi, booleani e virgola mobile
  - ✓ un tipo speciale chiamato `void`

I Tipi Aritmetici

Tipo	Descrizione	Dimensione Minima
`bool`	booleano (`true`|`false`)	NA
`char`	carattere	8 bits
`short`	intero	16 bits
`int`	intero	16 bits
`long`	intero	32 bits
`long long`	intero	64 bits
`float`	virgola mobile in singola precisione	$\geq$ 6 cifre significative
`double`	virgola mobile in doppia precisione	$\geq$ 10 cifre significative
`long double`	virgola mobile in precisione estesa	$\geq$ 10 cifre significative

I Tipi Interi

Gli interi possono essere:

- ✓ con segno (`signed`) (rappresentati in base due, notazione Complemento 2)
- ✓ dominio con `n-bit` : $[-2^{n-1}, 2^{n-1} - 1]$

- ✓ senza segno (`unsigned`) (rappresentati in base due, con tutti i bit a disposizione per il valore assoluto del numero)
- ✓ dominio con `n-bit` : $[0, 2^n - 1]$

Tipo	Bit (specs)	Bit	MIN/MAX
`short`	≥ 16	16	-32768 32767
`int`	$\geq$ `short`	32	-214783648 214783647
`long`	≥ 32 | `int`	32	-214783648 214783647
`long long`		64	-9223372036854775808 9223372036854775807

Tipo	Bit (specs)	Bit	MAX
`unsigned short`	≥ 16	16	65535
`unsigned int`	$\geq$ `short`	32	4294967295
`unsigned long`	≥ 32 | `int`	32	4294967295
`unsigned long long`		64	18446744073709551615

bool e char

- Un `bool` può assumere due valori: `true` oppure `false`.

```
bool is_ready = true;
```

- Conversione `bool` → `int`:

```
int ans = true;          //assume valore 1
int promise = false;     // assume valore 0
```

- Conversione `int` → `bool`:

```
bool start = -100;       //assume valore true
                         // come tutti i valori
                         // diversi da zero
bool stop = 0;           // assume valore false
```

- La rappresentazione dei caratteri segue il codice ASCII (8 bit)
- Esistono tre tipi : `char`, `signed char` e `unsigned char`
- La rappresentazione (con o senza segno) dei `char` è lasciata libera dallo standard

Tipo	Bit (specs)	Bit	MIN/MAX
`char`	8	8	0/-127 255/128

Codice ASCII

Byte	Cod.	Char
00000000	0	Null
00000001	1	Start of heading
00000010	2	Start of text
00000011	3	End of text
00000100	4	End of transmit
00000101	5	Enquiry
00000110	6	Acknowledge
00000111	7	Audible bell
00001000	8	Backspace
00001001	9	Horizontal tab
00001010	10	Line feed
00001011	11	Vertical tab
00001100	12	Form Feed
00001101	13	Carriage return
00001110	14	Shift out
00001111	15	Shift in
00010000	16	Data link escape
00010001	17	Device control 1
00010010	18	Device control 2
00010011	19	Device control 3
00010100	20	Device control 4
00010101	21	Neg. acknowledge
00010110	22	Synchronous idle
00010111	23	End trans. block
00011000	24	Cancel
00011001	25	End of medium
00011010	26	Substitution
00011011	27	Escape
00011100	28	File separator
00011101	29	Group separator
00011110	30	Record Separator
00011111	31	Unit separator

Byte	Cod.	Char
00100000	32	Spc
00100001	33	!
00100010	34	"
00100011	35	#
00100100	36	$
00100101	37	%
00100110	38	&
00100111	39	'
00101000	40	(
00101001	41	)
00101010	42	*
00101011	43	+
00101100	44	,
00101101	45	-
00101110	46	.
00101111	47	/
00110000	48	0
00110001	49	1
00110010	50	2
00110011	51	3
00110100	52	4
00110101	53	5
00110110	54	6
00110111	55	7
00111000	56	8
00111001	57	9
00111010	58	:
00111011	59	;
00111100	60	<
00111101	61	=
00111110	62	>
00111111	63	?

Byte	Cod.	Char
01000000	64	@
01000001	65	A
01000010	66	B
01000011	67	C
01000100	68	D
01000101	69	E
01000110	70	F
01000111	71	G
01001000	72	H
01001001	73	I
01001010	74	J
01001011	75	K
01001100	76	L
01001101	77	M
01001110	78	N
01001111	79	O
01010000	80	P
01010001	81	Q
01010010	82	R
01010011	83	S
01010100	84	T
01010101	85	U
01010110	86	V
01010111	87	W
01011000	88	X
01011001	89	Y
01011010	90	Z
01011011	91	[
01011100	92	\
01011101	93	]
01011110	94	^
01011111	95	_

Byte	Cod.	Char
01100000	96	`
01100001	97	a
01100010	98	b
01100011	99	c
01100100	100	d
01100101	101	e
01100110	102	f
01100111	103	g
01101000	104	h
01101001	105	i
01101010	106	j
01101011	107	k
01101100	108	l
01101101	109	m
01101110	110	n
01101111	111	o
01110000	112	p
01110001	113	q
01110010	114	r
01110011	115	s
01110100	116	t
01110101	117	u
01110110	118	v
01110111	119	w
01111000	120	x
01111001	121	y
01111010	122	z
01111011	123	{
01111100	124	|
01111101	125	}
01111110	126	~
01111111	127	Del

Codice ASCII Esteso

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`	128	80	Ç	160	A0	á	192	C0	└	224	E0	α
1	01	Start of heading	33	21	!	65	41	A	97	61	a	129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
2	02	Start of text	34	22	"	66	42	B	98	62	b	130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
3	03	End of text	35	23	#	67	43	C	99	63	c	131	83	â	163	A3	ú	195	C3	├	227	E3	π
4	04	End of transmit	36	24	$	68	44	D	100	64	d	132	84	ä	164	A4	ñ	196	C4	─	228	E4	∑
5	05	Enquiry	37	25	%	69	45	E	101	65	e	133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
6	06	Acknowledge	38	26	&	70	46	F	102	66	f	134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
7	07	Audible bell	39	27	'	71	47	G	103	67	g	135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
8	08	Backspace	40	28	(	72	48	H	104	68	h	136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i	137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j	138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k	139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l	140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m	141	8D	ì	173	AD	¡	205	CD	═	237	ED	ø
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n	142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o	143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
16	10	Data link escape	48	30	0	80	50	P	112	70	p	144	90	É	176	B0	░	208	D0	╨	240	F0	≡
17	11	Device control 1	49	31	1	81	51	Q	113	71	q	145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
18	12	Device control 2	50	32	2	82	52	R	114	72	r	146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
19	13	Device control 3	51	33	3	83	53	S	115	73	s	147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
20	14	Device control 4	52	34	4	84	54	T	116	74	t	148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u	149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v	150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
23	17	End trans. block	55	37	7	87	57	W	119	77	w	151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
24	18	Cancel	56	38	8	88	58	X	120	78	x	152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
25	19	End of medium	57	39	9	89	59	Y	121	79	y	153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z	154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{	155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|	156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}	157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~	158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	⌂	159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

American **S**tandard **C**ode for **I**nformation **I**nterchange

I Tipi a Virgola Mobile

- I numeri reali sono di tre tipi:

`float;`	→	Singola precisione
`double;`	→	Doppia precisione
`long double;`	→	Nuovo nello standard

- Notazione esponenziale (es. `+5.37E+16`)
- La dimensione non è specificata, tipicamente

`float` ≤ `double` ≤ `long double`

Tipo	Cifre significative	Bit mantissa	MAX esponente/ MIN esponente
Float	6	24	+38 -37
	FLT_DIG	FLT_MANT_DIG	FLT_MAX_10_EXP FLT_MIN_10_EXP
Double	15	53	+308 -307
	DBL_DIG	DBL_MANT_DIG	DBL_MAX_10_EXP DBL_MIN_10_EXP
Long Double	18	64	+4932 -4931
	LDBL_DIG	LDBL_MANT_DIG	LDBL_MAX_10_EXP LDBL_MIN_10_EXP

Precisione in Virgola Mobile

```
#include <iostream>

int main( )
{
   using namespace std;
   cout.setf(ios_base::fixed, ios_base::floatfield);
   float tub = 10.0/3.0;      // preciso a 6 cifre
   double mint = 10.0/3.0;    // preciso a 15 cifre

   const float MEGA = 1.E6;

   cout << "tub = " << tub << endl;
   cout << "MEGA tub = " << tub*MEGA << endl;
   cout << "10MEGA tub = " << tub*MEGA*10 << endl;

   cout << "mint = " << mint << endl;
   cout << "MEGA mint = " << mint*MEGA « endl;
   cout << "10MEGA mint = " << mint*10*MEGA « endl;

   return 0;
}
```

```
tub = 3.333333
MEGA tub = 3333333.250000
10MEGA tub = 33333332.500000
mint = 3.333333
MEGA mint = 3333333.333333
10MEGA mint =33333333.333333
```

Precisione Finita

```
// fltadd.cpp - esempio sulla precisione finita dei float
#include <iostream>

int main( )
{
  float a = 2.34E22f;
  float b = a + 1.0f;

  std::cout << "a = " << a << std::endl;
  std::cout << "b - a = " << b - a << std::endl;

  return 0;
}
```

Il suffisso f indica una costante di tipo float

Output del programma

```
a = 2.34e+22
b - a = 0
```

Quale Tipo di Dato Utilizzare?

- Il C++ non definisce strettamente la dimensione dei tipi di dati in modo da essere compatibile con diverso hardware
- Alcune regole pratiche:
  - ✓ usare un tipo `unsigned` se sicuri che i valori non possono essere negativi
  - ✓ usare `int` per tutta l'aritmetica tra interi
    `short` è solitamente troppo piccolo e `long` ha spesso la stessa dimensione di un `int`. Usare `long long` se il numero supera il dominio garantito dagli `int`
  - ✓ non usare `char` oppure `bool` in aspressioni aritmetiche
    L'utilizzo di `char` può essere problematico perchè su alcune architetture è di tipo `signed` e in altre `unsigned`
  - ✓ usare `double` per l'aritmetica a virgola mobile
    Sulle architetture odierne (processori a 64-bit) le operazioni a doppia precisione sono più veloci di quelle in singola. La precisione offerta dai tipi `long double` a volte non è necessaria (quindi il `double` è un risparmio computazionale

Conversioni tra Tipi di Dato

I La conversione tra tipi avviene automaticamente quando dato un oggetto, il programma si aspetta un oggetto di un tipo diverso

```
bool b = 42;                    // b => true
int i = b;                      // i => 1
i = 3.14;                       // i => 3
double pi = i;                  // pi => 3.0
unsigned char c1 = -1;          // c1 => 255 con 8-bit chars
signed char c2 = 255;           // c2 undefined
```

II Intero → `bool` ⇒ `false` se il valore è `0` e `true` altrimenti

III `bool` → intero ⇒ 1 se il `bool` è `true` e 0 se `false`

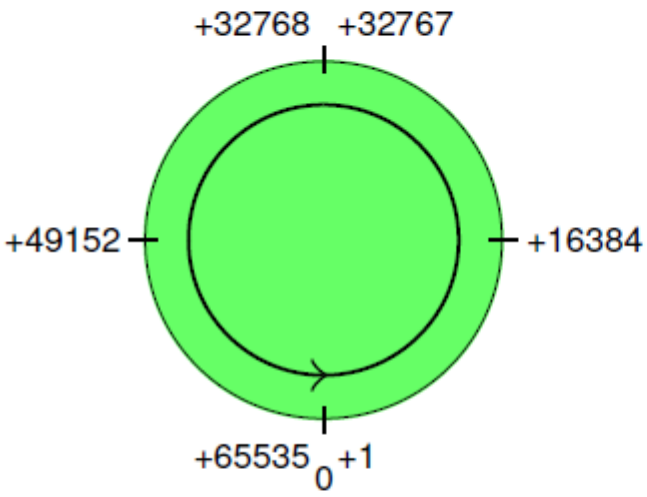
IV Floating-point → intero ⇒ valore è troncato, parte decimale persa

V Intero → Floating-point ⇒ parte decimale è zero. Perdita di precisione possibile

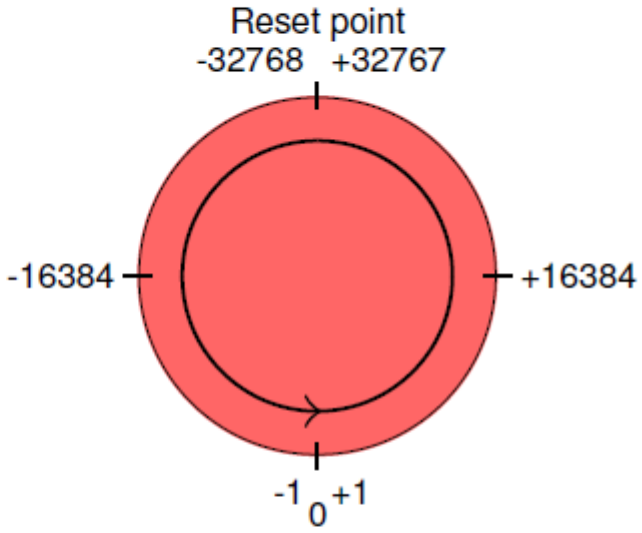
VI Valore out-of-range → `unsigned` ⇒ resto del valore, modulo dominio

VII Valore out-of-range → `signed` ⇒ risultato indefinito

Aritmetica degli Interi



Interi unsigned



Interi con segno

Le Costanti

- Ogni costante ha un tipo determinato dal suo formato e valore
- Costanti intere, espresse in base
  - ✓ `otto` se iniziano con uno zero : (es 020)
  - ✓ `sedici` se iniziano con `0x` oppure `0X` : (es 0x20)
  - ✓ `dieci` (tutti gli altri)
- Costanti a virgola mobile:
  - ✓ hanno un punto (es 3.14) oppure un esponente che indica la notazione scientifica (`3.14E-3`, o `3.14e-3`)
  - ✓ lo zero può essere scritto come `0.`, `0.0`, `0e0`, `0E0`, ...
- Costanti carattere:
  - ✓ sono indicate tra i singoli apici : (es `'a'`)
- Costanti stringa (come array di caratteri) :
  - ✓ sono indicate tra i doppi apici : (es `"Hello World"`)

Sequenze di Caratteri Escape

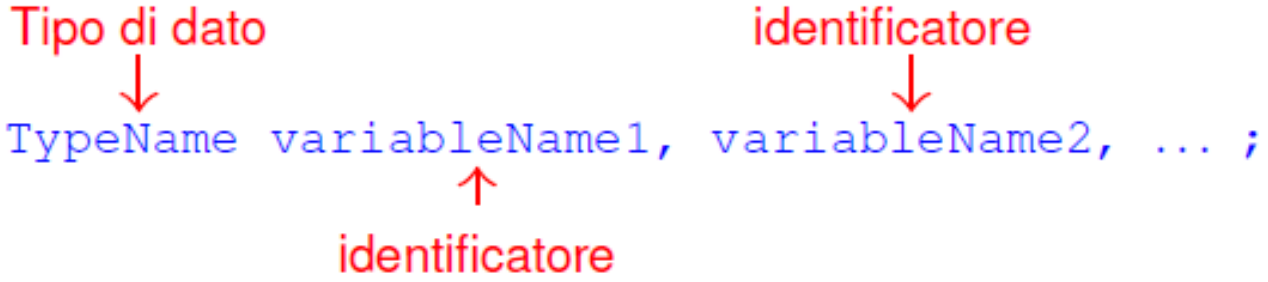
- Alcuni caratteri non hanno un immagine visibile (per esempio il carattere "a capo")
- Per poterli rappresentare viene utilizzata una sequenza di due caratteri : `'\' + 'carattere'`

newline	\n	horizontal tab	\t	alert	\a
vertical tab	\v	backspace	\b	double quote	\"
backslash	\\	question mark	\?	single quote	\'
carriage return	\r	formfeed	\f		

```
cout << '\n';      // prints a newline
cout << "\tHi!\n" // tab, text, plus a newline
```

Le Variabili

- Una variabile fornisce un nome ad uno spazio di memoria che contiene dati che possono essere utilizzati nel programma
- il tipo della variabile determina la dimensione, la struttura della memoria utilizzata, il dominio dei valori immagazzinabili e l'insieme delle operazioni disponibili sulla variabile



- La definizione può fornire un valore iniziale per gli elementi che definisce

```
int sum = 0, value,  // tre variabili di tipo int
    unit_sold = 0;   // solo la I e la III sono iniz. a 0
```

Inizializzazione di Variabili

- Una oggetto inizializzato acquisisce un valore specifico nel momento in cui viene creato

- si può inizializzate una variabile con una costante, un'espressione o con il valore di un'altra variabile

```
double discount_rate = 0.20;
double price = 30.50, discount = price * discount_rate;
```

- inizializzazioni e assegnazioni sono operazioni differenti in C++
- si parla di inizializzazione quando si crea una variabile e si inizializza ad un valore
- con l'assegnazione si sovrascrive una variabile con un nuovo valore

```
double rate = 0.20;     // inizializzazione
double discount_rate;
discount_rate = rate;   // assegnazione
```

List Inizialitation

- Il C++ definisce diversi modi per inizializzare una variabile

```
int zero = 0;
int zero = {0};
int zero{0};
int zero(0);
```

C++11

- L'utilizzo di parentesi graffe per inizializzare una variabile è stata introdotta dal C++11 → prende il nome di *list initialization*

```
long double ld = 3.141596536

int a{ld}, b = {ld};     // ERRORE: perdita di informazione
int c(ld), d = ld;       // permesso, init con troncamento
```

C++11 Il compilatore non permetterà l'inizializzazione da liste di una variabile di tipi built-in se questo comporta la perdita di informazione

Default Inizialization

- Quando creiamo una variabile senza specificare un valore iniziale, la variabile è inizializzata al valore di default
- Il valore di default dipende dal tipo della variabile e da dove è stata creata
  - ✓ variabili definite fuori da funzioni sono automaticamente inizializzate a zero
  - ✗ variabili definite nel corpo di funzioni non sono inizializzate

✗ il valore di una variabile non inizializzato non è predicibile

```
std::string global_string;     // initialized
int global_counter;            // init to empty string
int main ()
{
  int local_init;              // undefined
  std::string local_string;    // init to empty string
  ...
}
```

 : Le variabili non inizializzate posso causare problemi RUN-TIME nell'esecuzione dei programmi

Identificatori

- ✓ Un identificatore (i.e. nome di oggetto) può essere composto da:
- ✓ lettere, cifre e dal carattere underscore
- ✓ non ci sono limiti sulla lunghezza del nome
- ✓ gli identificatori sono case sensitive

```
// definiamo 4 variabili distinte:
int somename, someName, SomeName, SOMENAME;
```

- Ci sono un certo numero di convenzioni comunemente accettate per i nomi delle variabili.
- Seguire una delle convenzioni migliora la leggibilità dei programmi
- un identificatore deve riflettere il significato della variabile
- Regole di base:
  - gli identificatori sono principalmente in caratteri minuscoli
  - la libreria standard del C++ standard usa un carattere underscore come separatore di parole. Esempio: `get_background`
  - UpperCamelCase: `GetBackground`
  - lowerCamelCase: `getBackground`

Parole Riservate del C++

Table 2.3. C++ Keywords

alignas	continue	friend	register	true
alignof	decltype	goto	reinterpret_cast	try
asm	default	if	return	typedef
auto	delete	inline	short	typeid
bool	do	int	signed	typename
break	double	long	sizeof	union
case	dynamic_cast	mutable	static	unsigned
catch	else	namespace	static_assert	using
char	enum	new	static_cast	virtual
char16_t	explicit	noexcept	struct	void
char32_t	export	nullptr	switch	volatile
class	extern	operator	template	wchar_t
const	false	private	this	while
constexpr	float	protected	thread_local	
const_cast	for	public	throw	

Table 2.4. C++ Alternative Operator Names

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bitor	not	or	xor	

Variabili e Tipi Fondamentali

Lezione 7:
Parte 2. Tipi Composti

Referenze e Puntatori

- Un tipo composto un tipo di dato che è definito in termini di un altro
- Il C++ possiede svariati tipi composti, affronteremo in questa lezione
  - ✓ referenze
  - ✓ puntatori
- Una referenza definisce un nome alternativo per un oggetto: una referenza non è un oggetto, ma un alias ad un oggetto esistente

```
int ival = 47;
int & ri = ival;
```

- Un puntatore è un oggetto vero e proprio il cui contenuto è l'indirizzo di memoria dove è immagazzinato un altro oggetto

```
int ival = 47;
int * pi = & ival;
```

Referenze

- ✓ Nell'inizializzare una variabile, il valore dell'inizializzatore è copiato nell'oggetto creato

```
ival = 47;
int &ri = ival;
```

- ✓ Nel definire una referenza, colleghiamo la referenza all'inizializzatore
- ✓ Una volta creata, la referenza rimane legata all'oggetto per tutta la sua esistenza.
- ✗ Non è possibile ri-collegare la referenza ad un altro oggetto, run-time
  ⇒ tutte le referenze devono essere inizializzate

```
int i1 = 1024, i2 = 2048;   // i1 and i2 are both ints
int &r1 = i1, r2 = i2;      // r1 is a reference, r2 is an int
int i3 = 1025, &r3 = i3;    // i3 is and int, r3 a reference
                            //                bound to i3
int &r4 = i1, &r5 = i2;     // two int references

int &r6 = 10;    // error: initializer must be an object

double dval = 3.14;
int &r7 = dval; // error: initializer must be an int object
```

La referenza non è un oggetto: è semplicemente un altro nome per un oggetto esistente

Puntatori (I)

- ✓ Un puntatore è un tipo composto che *punta-a* un altro tipo.
- ✓ Come le referenze, i puntatori sono usati per accedere, indirettamente, ad altri oggetti.
- ✗ a differenza dalle referenze, un puntatore è un oggetto autonomo (può essere re-assegnato)

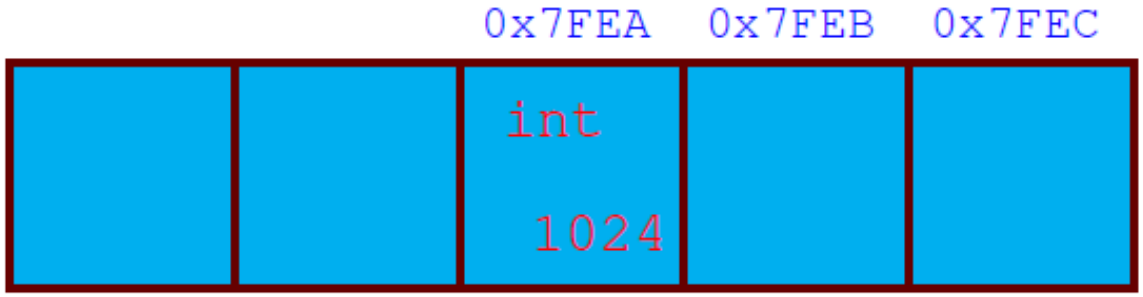
```
int *ip1, *ip2;    // two int pointers
double dp, *dp2;   // dp2: pointer-to-double, dp: double
```

- ✓ Un puntatore trattiene l'indirizzo di memoria di un oggetto
- ✓ Per ottenere l'indirizzo di memoria di un oggetto (i.e. dove è immagazzinato l'oggetto) si utilizza l'operatore `&`

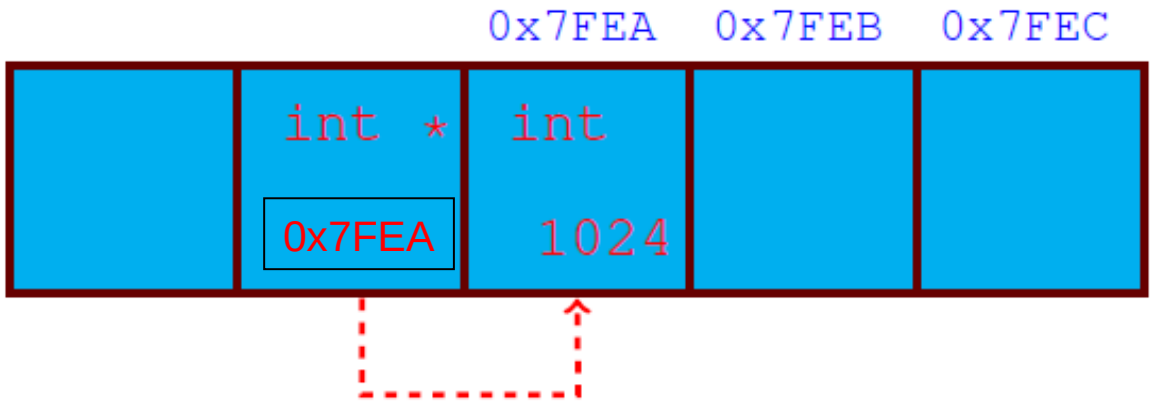
```
int ival = 1024;
int *p = &ival;        //  p holds the address on ival
                       //  p is a pointer to ival

double dval = 3.14;
double *pd = &dval;  // inizializer is a pointer to double
int *pi = pd;      // error: types of pi and pd differ
pi = &dval;        // error: assigning the adress of a double
                   // to a pointer to int
```

Puntatori (II)

```
int ival = 1024;
```

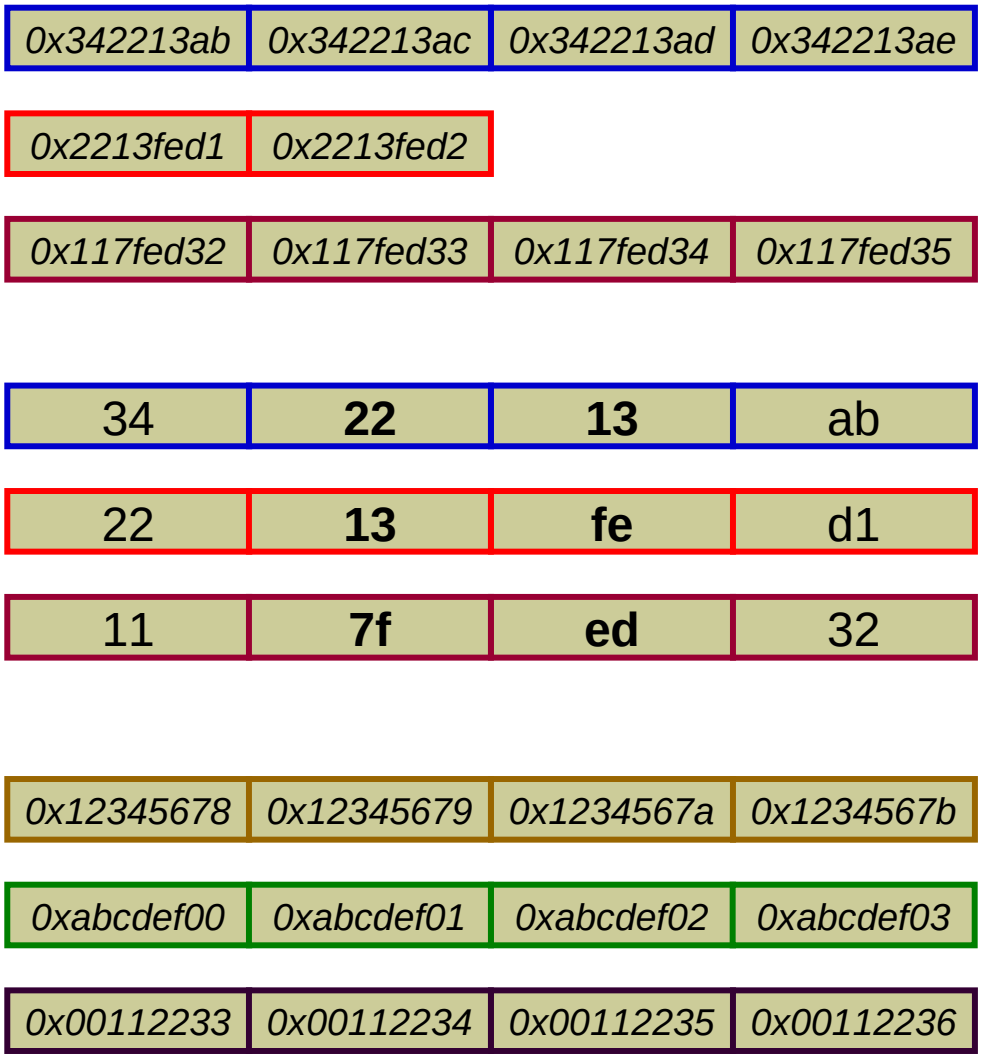



```
int *pi = &ival;
```



Puntatori (III)

```
int a = 2012;
short b = 12;
float c = 7.5;

int *pt1 = &a;
short *pt2 = &b;
float *pt3 = &c;

double *pt3;
long *pt4;
float *pt5;
```



Utilizzo dei Simboli: & e *

- ✓ i simboli & e * vengono utilizzati come operatori sia in espressioni che nelle dichiarazioni
- ✓ il loro significato dipende dal contesto

```
int i = 1024;
```

```
int &r = i;
```

`&` segue un tipo ed è parte di una dichiarazione: `r` è una referenza

```
int *p;
```

`*` segue un tipo ed è parte di una dichiarazione: `p` è un puntatore

```
p = & i;
```

`&` è usato in un'espressione come operatore *address-of*

```
*p = i;
```

`*` è usato in un'espressione come *dereference* operator

```
int &r2 = *p;
```

`&` è parte della dichiarazione; * è un *dereference* operator

Confronto Puntatori - Referenze

- ✓ referenze e puntatori permettono di accedere indirettamente ad altri oggetti
- ✓ Ci sono però differenze importanti :
- ✓ Una referenza non è un oggetto: una volta definita (e inizializzata) non è possibile farla riferire ad un oggetto diverso
- ✓ Un puntatore è un oggetto : se contiene l'indirizzo di un oggetto specifico, è sempre possibile farlo puntare ad un altro oggetto

```
int * p, a = 7;    // puntatore ad un intero e intero
*p = 0;            // modifica il contenuto dell'oggetto
                   // al quale punta
int a2;
p = & a2;        // punta ad un altro oggetto
```

- ✓ il puntatore può essere usato in espressioni relazionali (per es con operatori di uguaglianza == e diseguaglianza !=)

```
if (p)    // ritorna false, se p = 0
          //          true, altrimenti
```

Dichiarazione Multipla

- ✓ La sintassi per la definizione di una variabile è data da un tipo e una lista di dichiaratori:

```
// i e' int, p un pointer a int, r un reference a int
int i = 1024, *p = &i, &r = i;
```

- ✓ È un errore comune pensare che il modificatore di tipo (`*` o `&`) si applichi a tutte le variabili definite in una sola istruzione
- ✓ Nell'istruzione

```
int* p; // legal but misleading
```

il tipo di base è `int` e non `int*`. Il simbolo `*` modifica il tipo di `p`

- ✓ Infatti,

```
int* p1, p2; // p1 is a pointer-to-int, p2 is a int
```

- ✓ nel caso volessimo due puntatori a int, la sintassi corretta è:

```
int *p1, *p2; // p1 and p2 are pointer-to-int
```

altrimenti, si scriverà:

```
int* p1; // a pointer to int
int* p2; // a pointer to int
```

Non esiste un unico stile per definire i puntatori. È importante la coerenza nel codice

Il Qualificatore `const`

- ✓ A volte è necessario definire delle costanti in un programma: variabili il cui contenuto non può essere modificato
  - ✓ si fa precedere la dichiarazione dalla parola chiave `const`

```
const int buffer_size = 512;
```

- Poichè il contenuto della variabile costante non può essere cambiato, dobbiamo inizializzarla :

```
const int i = 1024;          //initialized at compile-time
const int j = get_size();    //initialized at run-time
const int k;          // error: k is uninitialized const
```

- ✓ una referenza ad una variabile costante deve essere *reference to* `const`

```
const int ci = 1024;
const int &r1 = c1;    //ok: both reference and object const
 int &r2 = c1;          //error: non const reference
                        //       to a const object
```

The `auto` Type Specifier

- spesso si vuole immagazzinare il risultato di un'espressione in una variabile; è quindi necessario dichiarare la variabile e specificarne il tipo
- a volte risulta molto complicato o addirittura impossibile determinare il tipo di un'espressione durante la scrittura del programma
- il nuovo standard, tramite lo specificatore di tipo `auto` lascia questo compito al compilatore
- tutte le variabili dichiarate come `auto` devono essere inzializzate

```
int val1 = 3, val2 = 7;
auto a1 = val1 + val2;
std::cout << "a1 = " << a1 << std::endl;
```

`a1` è dichiarata e inizializzata al risultato `val1 + val2` quindi `int`

```
double val3 = 3.2;
auto a2 = val1 + val3;
std::cout << "a2 = " << a2 << std::endl;
```

`a2` è dichiarata e inizializzata al risultato `val1 + val3` quindi `double`

Strutture

una struttura di dati è il modo per raggruppare insieme di dati diversi definendo le operazioni che si possono fare su di essi

```
struct Sales_data {
   std::string codice_isbn;
   unsigned int pezzi_venduti = 0;
   double fatturato = 0.0;
};
```

C++11

La struttura inizia con la parola chiave `struct`, seguita dal nome della struttura stessa e dal corpo della definizione (tra partentesi graffe).
La definizione della struttura deve terminare con un punto e virgola (;).
E' possibile definire della variabili subito dopo il corpo della struttura.

```
struct Sales_data { ... } libro, * libroPtr;
struct Sales_data { ... };
Sales_data manuale, * manualePtr;
```

Utilizzo delle Strutture

Il corpo della struttura definisce i membri della struttura.
Ogni oggetto creato ha la sua copia indipendente dei membri della struttura.
Modificando il valore di un membro della struttura si cambia solo il valore dell'oggetto coinvolto e non tutti i membri della struttura.
Per accedere ai membri della struttura utilizzeremo l'operatore '.'


```
Sales_data manuale, libro;
std::cin >> libro.codice_isbn;
std::cin >> libro.pezzi_venduti;

Sales_data l1, l2;
if (l1.codice_isbn == l2.codice_isbn) {
  l2.pezzi_venduti += l1.pezzi_venduti;
  ...
}
```

Esempio: Numeri Complessi

Un numero complesso è fatto da una parte reale e una immaginaria :
$z = a + ib$, con $a, b \in \mathbb{R}$

```
#include <iostream>
#include <cmath>
struct complex_data {
  double re = 0;  // real part
  double im = 0;  // immaginary part
};
int main()
{
  std::cout << "Inserire due numeri complessi (re, im): ";
  complex_data z1, z2;
  std::cin >> z1.re >> z1.im;
  std::cin >> z2.re >> z2.im;

  // Calcolo la somma
  complex_data z_sum;
  z_sum.re = z1.re + z2.re;
  z_sum.im = z1.im + z2.im;

  std::cout << "La somma e': ";
  std::cout << z_sum.re << " + i" << z_sum.im << std::endl;
  std::cout << "e ha modulo: ";
  std::cout << sqrt(z_sum.re*z_sum.re + z_sum.im*z_sum.im) << std::endl;

  return 0;
}
```