C++ Programming :Session 2

Anupama Chandrasekhar (Aug 7 2018)

Session Agenda

- Machine Setup
- Datatypes
- Types: Sizes, Alignment, Portability
- Definitions and Declarations
- Scope and Lifetime
- Initialization

Datatypes

- One important point to note about datatypes is that their sizes are mostly implementation dependent, so it’s good to be cautious and not make assumptions about them. To ensure portability be sure about the implementation defined constraints

```
unsigned char c1 = 30; // Well defined because char is guaranteed to be
                       // atleast 8 bits

unsigned char c2 = 260; // Implementation dependent, truncation might occur
                        // if char is 8 bits
```

- One way to be explicit about implementation dependencies and maximize portability is by using numerical limits and static asserts

Types

- Every identifier in a C++ program should have a type associated with it. The type defines what operations can be performed on it and what how that operation should be interpreted.
- Fundamental Type are:
  - Integral Types : **bool, char, int**
  - Arithmetic types: **int, float, double**
  - No information about type: **void**
  - Pointer, Reference and Array types: **int*, double&, char[]**
- User defined Types:
  - Classes and Enumerations : These are defined by users

Types : Bool

- **bool** : can have **true** or **false**, typically **true** is converted to **int** 1 and **false** to **int** 0. **bool** is the type of the result of a function that tests some condition. Note that integers can be converted to bool with nonzero converting to true and zero converts to false.
- In arithmetic and bitwise operations, **bool**s are converted to **int**s, the operation is performed on an int and the result is converted back to **bool** if needed.
- A pointer can be implicitly converted to **bool**, a **nullptr** is **false** and a non-null ptr is **true**.

Types: Char

- **char**: It is the default character type. Typically 8 bits.
- A single character within single quotes is a character literal. Ex. 'a'. The use of character literals than their equivalent **int** values makes programs more portable
- Note that there are other possible encodings like **signed** and **unsigned char**, **wchar_t** (for larger character sets like Unicode)
- Safe assumptions
  - Implementation character set includes decimal digits, 26 alphanumeric, basic punctuation
- Not safe to assume
  - more than 127 characters
  - Contiguous spacing

ASCII Character Mapping

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Types: Char (Conversions)

- Pointers of the different types cannot be freely assigned, they are different types but values can be assigned. Make sure that the values assigned are within the type's limits to avoid surprises.

```
void test_char_conversions(char c, signed char sc, unsigned char uc)
```

```
char* pc = &uc;                 // error: no pointer conversion
signed char* psc = pc;          // error: no pointer conversion
unsigned char* puc = pc;        // error: no pointer conversion
psc = puc;                      // error: no pointer conversion

signed char sc = -120;
unsigned char uc = sc;     // uc == 136 (because 256-120==136)
sc = ++uc;                 // sc is -119 (because 136+1==137 and 256-137==119)
char count[256];           // assume 8-bit chars (uninitialized)
char c1 = count[sc];       // likely disaster: out-of-range access
char c2 = count[uc];       // OK
```

Types : Integers

- Integers come in a few flavors:
  - **int, signed int, unsigned int** (u or U)
  - **short, long** (l or L), **long long**
- Use **unsigned int** to treat the storage as a bit array
- plain **int** is a signed **int**
- **<cstdint>** exposes more variants like int64_t.
- Integer Literals: (compiler warnings are only guaranteed with {})

```
7  1234  976  12345678901234567890
```

Decimal	Octal	Hexadecimal
	0	0x0
2	02	0x2
63	077	0x3f
83	0123	0x63

Types: Integers

- **int** literal conversions can be subtle and implementation specific, so best to be specific.
- For example, **100000** is of type **int** on a machine with 32-bit **int**s but of type **long int** on a machine with 16-bit **int**s and 32-bit **long**s. Similarly, **0XA000** is of type **int** on a machine with 32-bit **int**s but of type **unsigned int** on a machine with 16-bit **int**s. These implementation dependencies can be avoided by using suffixes: **100000L** is of type **long int** on all machines and **0XA000U** is of type **unsigned int** on all machines.

Practice Exercise

- Given an input c style string (null terminated string) find the character frequencies.

```
// Write a program that prints character frequency in a lower case string.
#include <iostream>
using namespace std;

void CalcFreq(const char* input)
{

}

int main()
{
    char* input = "donotworrybehappy";
    CalcFreq(input);
}
```

Types: Floats

- Floating point is an approximation of a real number represented in a fixed amount of memory.
- They come in three flavors: **float** (f or F), **double**, **long double**(L or l)
- Floating point literals:

```
1.23  .23  0.23  1.  1.0  1.2e10  1.23e-15
```

- By default, a floating point literal is a of type double. You can force the type float with the suffix f or long double with L. Example: `1.0f`, `3.5e-4L`

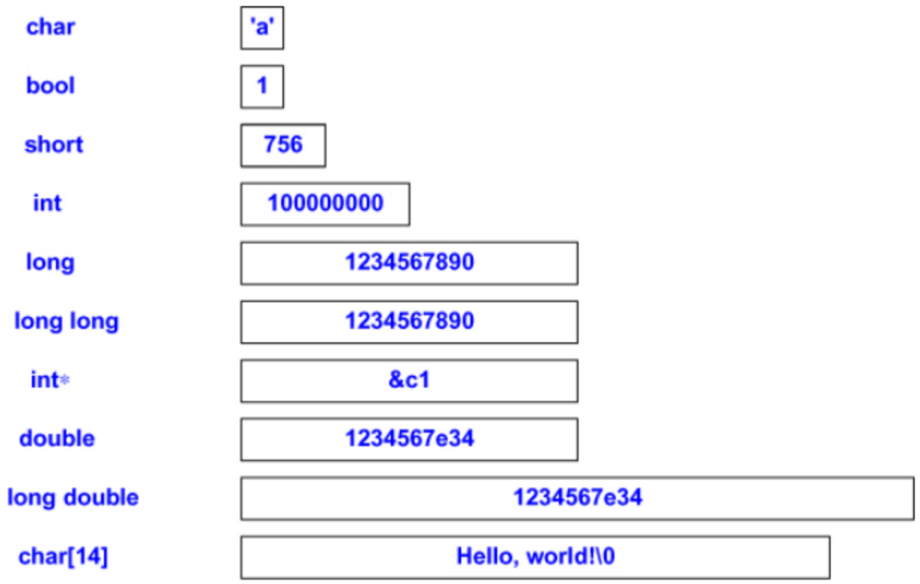
Types: Void

- **void** is syntactically a fundamental type but there are no objects of the type void.
- **void** is used to indicate that a function doesn't return a value or that the base type of a pointer is unknown

```
void x;         // error: there are no void objects
void& r;        // error: there are no references to void
void f();       // function f does not return a value
void* pv;       // pointer to object of unknown type
```

Types: Size, Alignment, Portability

- Do not make assumptions about sizes, always test and be aware of portability issues.
- **Performance note**: One of the reasons for all the different types available is for the developer to make choices based on the specific hardware they are developing for, differences in memory requirements, memory access times and computation speeds



Types: Size, Alignment, Portability

- The size of C++ objects are expressed as multiples of size of a **char**. So **sizeof(char)** == 1 by definition. C++ guarantees:
  - 1 ≡ `sizeof(char)` ≤ `sizeof(short)` ≤ `sizeof(int)` ≤ `sizeof(long)` ≤ `sizeof(long long)`
  - 1 ≤ `sizeof(bool)` ≤ `sizeof(long)`
  - `sizeof(char)` ≤ `sizeof(wchar_t)` ≤ `sizeof(long)`
  - `sizeof(float)` ≤ `sizeof(double)` ≤ `sizeof(long double)`
  - `sizeof(N)` ≡ `sizeof(signed N)` ≡ `sizeof(unsigned N)`
- Where N in the last line is **char**, **short**, **int**, **long** or **long long**
- **char** is at least 8 bits, **short** is at least 16 bits and **long** is at least 32 bits
- Implementation specific details can be found in <limits>

Types: Size, Alignment, Portability

- The standard library header defines an alias **size_t** that can store the size in bytes of every object. So you know that if you have to allocate say a 4GB array its size would fit in **size_t**. Also leaving the decision of which type to pick for **size_t** to the implementation means that the compiler can choose the most performant type for that machine.
- In addition to type, objects (variables) might have alignment restrictions like **int** might need to be aligned on a 4-byte boundary and double on an 8-byte boundary.
- Use the **alignof** operator to check alignment.

Declarations

- Before a name can be used in C++, it needs to be declared => type must be specified to the compiler

```
const double pi {3.1415926535897};

extern int error_number;

const char* name = "Njal";

const char* season[] = { "spring", "summer", "fall", "winter" };

vector<string> people { name, "Skarphedin", "Gunnar" };
```

Declaration: Names

- Contd..

```
template<typename T> T abs(T a) { return a<0 ? -a: a; }

constexpr int fac(int n) { return (n<2)?1:n* fac(n-1); }

constexpr double zz { ii*fac(7) };

using Cmplx = std::complex<double>;

struct User;

enum class Beer { Carlsberg, Tuborg, Thor };

namespace NS { int a; }
```

- Note that many declarations are also definitions. In general if memory is required to represent something then memory is set aside by the definition
- One way to think is of declarations as interfaces and definitions as implementation
- Struct User; if used should be defined elsewhere.

Declarations

```
char ch;                      // set aside memory for a char and initialize it to 0

auto count = 1;               // set aside memory for an int initialized to 1

const char* name = "Anu";     // set aside memory for a pointer to char

                              // set aside memory for a string literal "Anu"

                              // initialize the pointer with the address of that string literal


struct Date { int d, m, y; }; // Date is a struct with three members

int day(Date* p) { return p->d; }   // day is a function that executes the specified code
```

Declarations

- There can be only one definition but there can be multiple declarations, but the types of the declarations need to match.

```
int count;

int count;

extern int error_number;

extern short error_number;
```

```
extern int error_number;

extern int error_number;   // OK: redeclaration
```

Declarations

```cpp
struct Date { int d, m, y; };

using Point = std::complex<short>;

int day(Date* p) { return p->d; }

const double pi {3.1415926535897};
```

```cpp
void f()
{
    int count {1};                    // initialize count to 1

    const char* name {"Anu"};        // name is a variable that points to a constant

    count = 2;                        // assign 2 to count

    name = "Chan";
}
```

- For types, aliases, templates, functions and constants the value is “permanent”. For non-const data, the value can be changed later.

C++ Keywords

C++ Keywords					
alignas	alignof	and	and_eq	asm	auto
bitand	bitor	bool	break	case	catch
char	char16_t	char32_t	class	compl	const
constexpr	const_cast	continue	decltype	default	delete
do	double	dynamic_cast	else	enum	explicit
extern	false	float	for	friend	goto
if	inline	int	long	mutable	namespace
new	noexcept	not	not_eq	nullptr	operator
or	or_eq	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template	this
thread_local	throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual	void
volatile	wchar_t	while	xor	xor_eq	

In addition, the word `export` is reserved for future use.

Declaration: Scope

- Local Scope : In a function, scope { … }
- Class Scope
- Namespace Scope : Point of declaration to end of namespace, maybe accessible by other translation units
- Global Scope
- Statement Scope
- Function Scope

Initialization

- Initialization determines the initial value of the object
- There are four ways to initialize:

```
X a1 {v};
X a2 = {v};
X a3 = v;
X a4(v);
```

- The first way is most recommended and available from C++11. The primary advantage of {} is that it does not allow narrowing.
- Empty initializer {} means use default value.

Initialization

- Empty initialization {} means default value

```
int x4 {};              // x4 becomes 0
double d4 {};           // d4 becomes 0.0
char* p {};             // p becomes nullptr
vector<int> v4{};       // v4 becomes the empty vector
string s4 {};           // s4 becomes ""
```

- Typically integral types is some form of 0, pointers is nullptr and for user defined types it is the default constructed values.
- Leaving out an initializer is possible but often undesirable. Where might it be ok?

Initialization

- If no initializer is specified, global, namespace, local static, static member are initialized to {} of the appropriate type.
- Local variables (stack allocated) and dynamically allocated objects are not initialized by default unless they are user defined types with default constructors.

```
void f()
{
    int x;                        // x does not have a well-defined value
    char buf[1024];               // buf[i] does not have a well-defined value
    int* p {new int};             //*p does not have a well-defined value
    char* q {new char[1024]};     // q[i] does not have a well-defined value
    string s;                     // s=="" because of string's default constructor
    vector<char> v;               // v=={} because of vector's default constructor
    string* ps {new string};      //*ps is "" because of string's default constructor
    //..
}
```