Software Languages Lab

Structuur van Computerprogramma's 2

dr. Dirk Deridder
Dirk.Deridder@vub.ac.be
http://soft.vub.ac.be/

Vrije Universiteit Brussel - Faculty of Science and Bio-Engineering Sciences - Computer Science Department - Pleinlaan 2 - 1050 Brussels - Belgium

Software Languages.Lab

Chapter 2 - Types and Functions

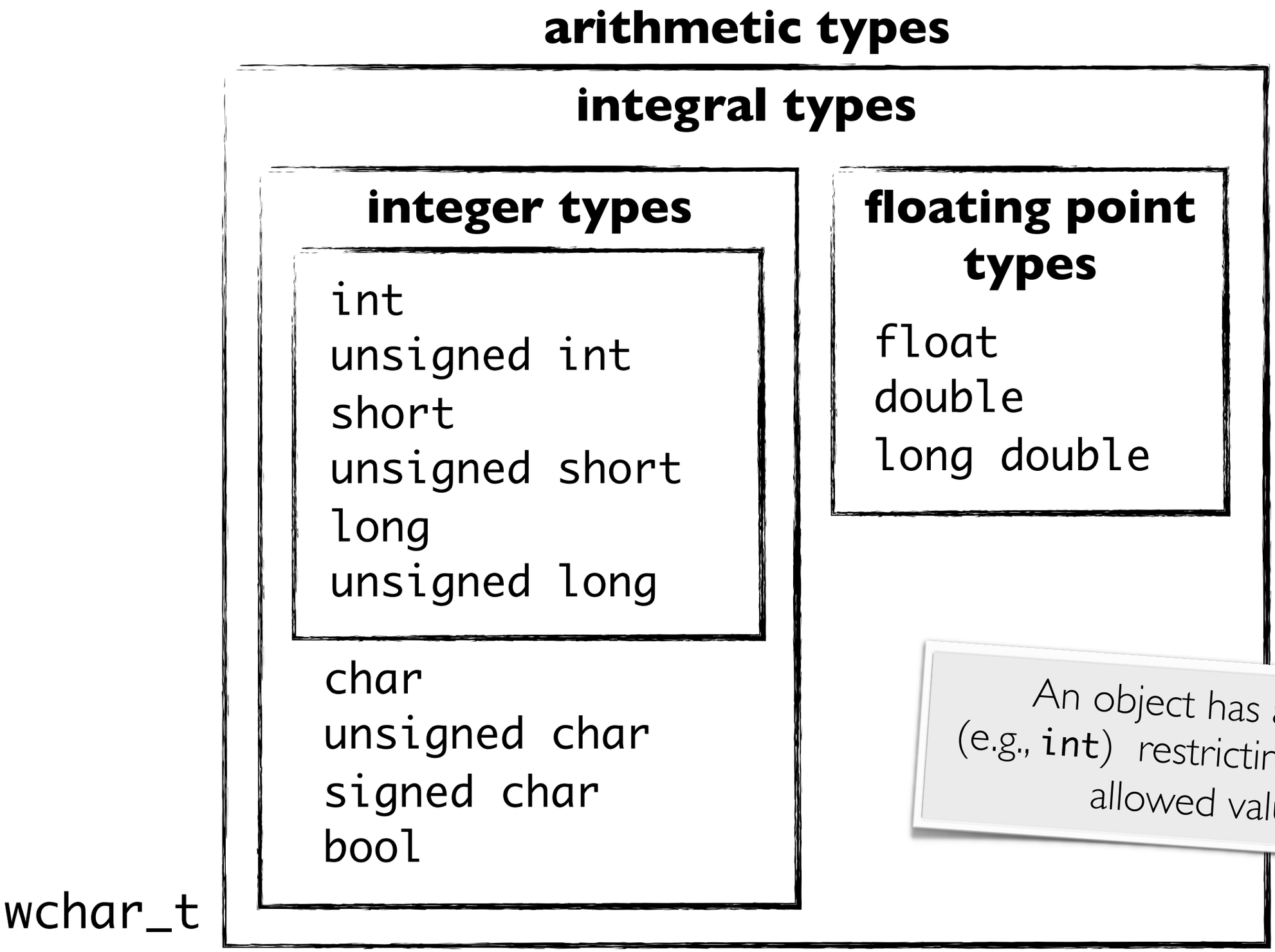
Structuur van Computerprogramma's 2

Built-in Types and Operations

Overview of Concepts

- Primitive types + sizes: arithmetic types, integral types, integer types, floating point types
- Lazy evaluation
- Variable initialization
- Escape characters
- Type conversion
- Character encoding schemes
- Operators, operator precedence
- String literals

Primitive Types



An object has a **type** (e.g., `int`) restricting the set of allowed values

Primitive Type Sizes

```
...

std::cout << "char -> " << sizeof(char) << std::endl;
std::cout << " -> " << sizeof(char) << std::endl;
```

```
char               ->  1 byte(s)  -> 256
signed char        ->  1 byte(s)  -> 256
short              ->  2 byte(s)  -> 65536
int                ->  4 byte(s)  -> 4.29497e+09
long               ->  8 byte(s)  -> 1.84467e+19
unsigned long      ->  8 byte(s)  -> 1.84467e+19
bool               ->  1 byte(s)  -> 256
wchar_t            ->  4 byte(s)  -> 4.29497e+09
float              ->  4 byte(s)  -> 4.29497e+09
double             ->  8 byte(s)  -> 1.84467e+19
long double        -> 16 byte(s)  -> 3.40282e+38

...
```

A type implies a **size** used to reserve the amount of memory

`sizeof(`object`)`
=
`sizeof(`type`)`
=
# of bytes to store the object

Primitive Types and Initialization

```
char c('\n');                  // newline character

int i(0777);                   // prefix zero -> octal

unsigned short s(0xffff); // prefix 0x   -> hexadecimal

long double pi(4.3E12);    // 4.3 * 1 000 000 000 000

bool stupid(true);             // boolean false and true
                               // C++ considers 0 as false and
                               // every non-zero value as true
```

The backslash indicates an **escape character**

'\a'	alert	(7)
'\t'	tab	(41)
'\b'	backspace	(8)
'\\'	backslash	(92)
'\f'	form feed	(12)
'\''	single quote	(39)
'\n'	newline	(10)
'\"'	double quote	(34)
'\r'	return	(13)
'\v'	vertical tab	(11)
'\0'	null character	
...		

Primitive Types and Type Conversion

```
long l('a'); // char fits into l; l = 97 (see ASCII)

int i(3.14); // compiler will warn; i = 3

double d(0.5);

float f(0.6);

int i(0);


i = d + f; // i will be 1; f will be promoted to double
```

- Compiler **performs the operation in the "widest" type**
  - Implicit type promotion (e.g., float to double, short to int, ...)
- Compiler tries to do a **reasonable conversion for the assignment**
  - Warns if the target is too small
    (but only if the value to be assigned/passed is known at compile time)

ASCII Character-Encoding Scheme (FYI)

Binary	Oct	Dec	Hex	Glyph	Binary	Oct	Dec	Hex	Glyph	Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	space	100 0000	100	64	40	@	110 0000	140	96	60	`
010 0001	041	33	21	!	100 0001	101	65	41	A	110 0001	141	97	61	a
010 0010	042	34	22	"	100 0010	102	66	42	B	110 0010	142	98	62	b
010 0011	043	35	23	#	100 0011	103	67	43	C	110 0011	143	99	63	c
010 0100	044	36	24	$	100 0100	104	68	44	D	110 0100	144	100	64	d
010 0101	045	37	25	%	100 0101	105	69	45	E	110 0101	145	101	65	e
010 0110	046	38	26	&	100 0110	106	70	46	F	110 0110	146	102	66	f
010 0111	047	39	27	'	100 0111	107	71	47	G	110 0111	147	103	67	g
010 1000	050	40	28	(	100 1000	110	72	48	H	110 1000	150	104	68	h
010 1001	051	41	29	)	100 1001	111	73	49	I	110 1001	151	105	69	i
010 1010	052	42	2A	*	100 1010	112	74	4A	J	110 1010	152	106	6A	j
010 1011	053	43	2B	+	100 1011	113	75	4B	K	110 1011	153	107	6B	k
010 1100	054	44	2C	,	100 1100	114	76	4C	L	110 1100	154	108	6C	l
010 1101	055	45	2D	-	100 1101	115	77	4D	M	110 1101	155	109	6D	m
010 1110	056	46	2E	.	100 1110	116	78	4E	N	110 1110	156	110	6E	n
010 1111	057	47	2F	/	100 1111	117	79	4F	O	110 1111	157	111	6F	o
011 0000	060	48	30	0	101 0000	120	80	50	P	111 0000	160	112	70	p
011 0001	061	49	31	1	101 0001	121	81	51	Q	111 0001	161	113	71	q
011 0010	062	50	32	2	101 0010	122	82	52	R	111 0010	162	114	72	r
011 0011	063	51	33	3	101 0011	123	83	53	S	111 0011	163	115	73	s
011 0100	064	52	34	4	101 0100	124	84	54	T	111 0100	164	116	74	t
011 0101	065	53	35	5	101 0101	125	85	55	U	111 0101	165	117	75	u
011 0110	066	54	36	6	101 0110	126	86	56	V	111 0110	166	118	76	v
011 0111	067	55	37	7	101 0111	127	87	57	W	111 0111	167	119	77	w
011 1000	070	56	38	8	101 1000	130	88	58	X	111 1000	170	120	78	x
011 1001	071	57	39	9	101 1001	131	89	59	Y	111 1001	171	121	79	y
011 1010	072	58	3A	:	101 1010	132	90	5A	Z	111 1010	172	122	7A	z
011 1011	073	59	3B	;	101 1011	133	91	5B	[	111 1011	173	123	7B	{
011 1100	074	60	3C	<	101 1100	134	92	5C	\	111 1100	174	124	7C	|
011 1101	075	61	3D	=	101 1101	135	93	5D	]	111 1101	175	125	7D	}
011 1110	076	62	3E	>	101 1110	136	94	5E	^	111 1110	176	126	7E	~
011 1111	077	63	3F	?	101 1111	137	95	5F	_					

```
long l('a');
// l = 97
```

The basic C++ source character set consists of the following 97 characters:

- space
- horizontal/vertical tab
- form feed
- new-line
- the following graphical characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

Universal characters can also be named with:

\U followed by 8 hexadecimal digits

Note: there are other encoding schemes out there (e.g., EBCDIC, UTF-16)

(ASCII = American Standards Code for Information Interchange)

Operations on Arithmetic Types

```
a + b - c * d / e;      // standard arithmetical operations
-a;                     // unary minus
i % j;                  // remainder of dividing i by j (modulo)
x += y;                 // x = x + y, also -= and *= and ...

++y; --y;               // add/subtract 1, then return value of y
x++; x--;               // return value of x, then add/subtract 1

x < y > z <= u >= v;    // arithmetical comparison
!i;                     // logical complement !true == false
i && j || k;            // logical AND, logical OR (Lazy Evaluation?)

~i;                     // bitwise complement ~10000001 == 01111110
i & j | k ^ l;          // bitwise comparison AND, OR, XOR
x << 4;                 // shift value of x, n bits to the left
x >> 4;                 // shift value of x, n bits to the right
```

bit shifting, unless **x** is a stream!

Operations on Arithmetic Types and Type Sizes

```
short acceleration(0);
const short FAST(32746);
...
cout << "    Rocket acceleration = " << acceleration << endl;
cout << "### Starting countdown: ... 3 ... 2 ... 1" << endl;
cout << "### Setting acceleration to FAST" << endl;
acceleration = FAST;
...
cout << "### Let's go a bit faster" << endl;
acceleration = acceleration + 10;
...
acceleration = acceleration + 10;
...
acceleration = acceleration + 10;
...
```

What is the output?

Know your types!
(and conversions)

?

```
    Rocket acceleration = 0
### Starting countdown: ... 3 ... 2 ... 1
### Setting acceleration to FAST
    Rocket acceleration = 32746
### Let's go a bit faster
    Rocket acceleration + 10 = 32756
### Let's go a bit faster
    Rocket acceleration + 10 = 32766
### Let's go a bit faster
    Rocket acceleration + 10 = -32760
### What is happening? -- FASTER, FASTER !!!
    Rocket acceleration + 1000 = -31760
### Houston, we have a problem ...
```

Operations on Arithmetic Types: Precedence (1)

OPERATOR	FUNCTION	EXAMPLE	REMARKS
type() ++ --	value construction post increment post decrement	double('c') la++ la--	 lvalue only lvalue only
sizeof() ++ -- ~ ! - +	size of object/type pre increment pre decrement bitwise complement logical not unary minus unary plus	sizeof(a) ++la --la ~i !a -a +a	or sizeof(double) lvalue only lvalue only integral type only
* / %	multiplication division modulo	a * a a / a i % i	 integral type only
+ -	addition subtraction	a + a a - a	
<< << >> >>	shift left output to stream shift right input from stream	i << j cout << a i >> j cin >> la	integral type only integral type only to lvalue only
< <= > >=	less than less than or equal greater than greater than or equal	a < a a <= a a > a a >= a	

Operators higher in the list take precedence over lower ones. Operators in the same box have equal precedence (use parentheses).

Operations on Arithmetic Types: Precedence (2)

OPERATOR	FUNCTION	EXAMPLE	REMARKS
== !=	equal to not equal to	a == a a != a	
&	bitwise AND	i & j	integral type only
^	bitwise XOR	i ^ j	integral type only
|	bitwise OR	i | j	integral type only
&&	logical AND	i && j	
||	logical OR	i || j	
= *= /= %= += -= <<= >>= &= |= ^=	assign multiply and assign divide and assign modulo and assign add and assign subtract and assign shift left and assign shift right and assign bitwise AND and assign bitwise OR and assign bitwise XOR and assign	la = a la *= a la /= a la %= a la += a la -= a li <<= i li >>= i li &= i li |= i li ^= i	left operand lvalue integral type only integral type only integral type only integral type only integral type only

Operations on Arithmetic Types: Precedence (3)

fac.cpp

```
#include <iostream>
using namespace std;

int fac(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fac(n - 1);
    }
}

int main() {
    cout << fac(5);
    return 0;
}
```

```
g++ -Wall -o fac fac.cpp
./fac
>>>120
```

fac2.cpp

```
#include <iostream>
using namespace std;

int fac(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fac(--n);
    }
}

int main() {
    cout << fac(5);
    return 0;
}
```

```
g++ -Wall -o fac fac2.cpp
./fac
>>>0
```

fac3.cpp

```
#include <iostream>
using namespace std;

int fac(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * fac(n--);
    }
}

int main() {
    cout << fac(5);
    return 0;
}
```

```
g++ -Wall -o fac fac3.cpp
./fac
>>>Segmentation fault
```

Are these all the same?

Operations on Primitive Types (1)

`operator` keyword

2 `bool`'s as formal parameters

```
bool operator&&(bool e1, bool e2);
```

```
(5 < 4) && (1 != 0)
```

`bool` as return type

operator name

formal parameter named `value` which holds a value of type `T`

```
T& operator=(T& lvalue, T value);
```

```
x = 5 //int x;
```

return type is a reference to a type `T`

formal parameter named `lvalue` which holds a reference to a type `T`

- Operations are (infix) functions
- You can redefine operators for user-defined types (see later)

Operations on Primitive Types (2)

- Assignments are operators and can be compounded

```
T& operator=(T& lvalue, T value);
```

```
x = y = z = 0; // x = (y = (z = 0));

x += 3;        // x = x + 3;
```

- I/O operators

a declaration, so parameter names not required

```
std::ostream& operator<<(std::ostream&, int);
std::istream& operator>>(std::istream&, int&);
```

insertion operator

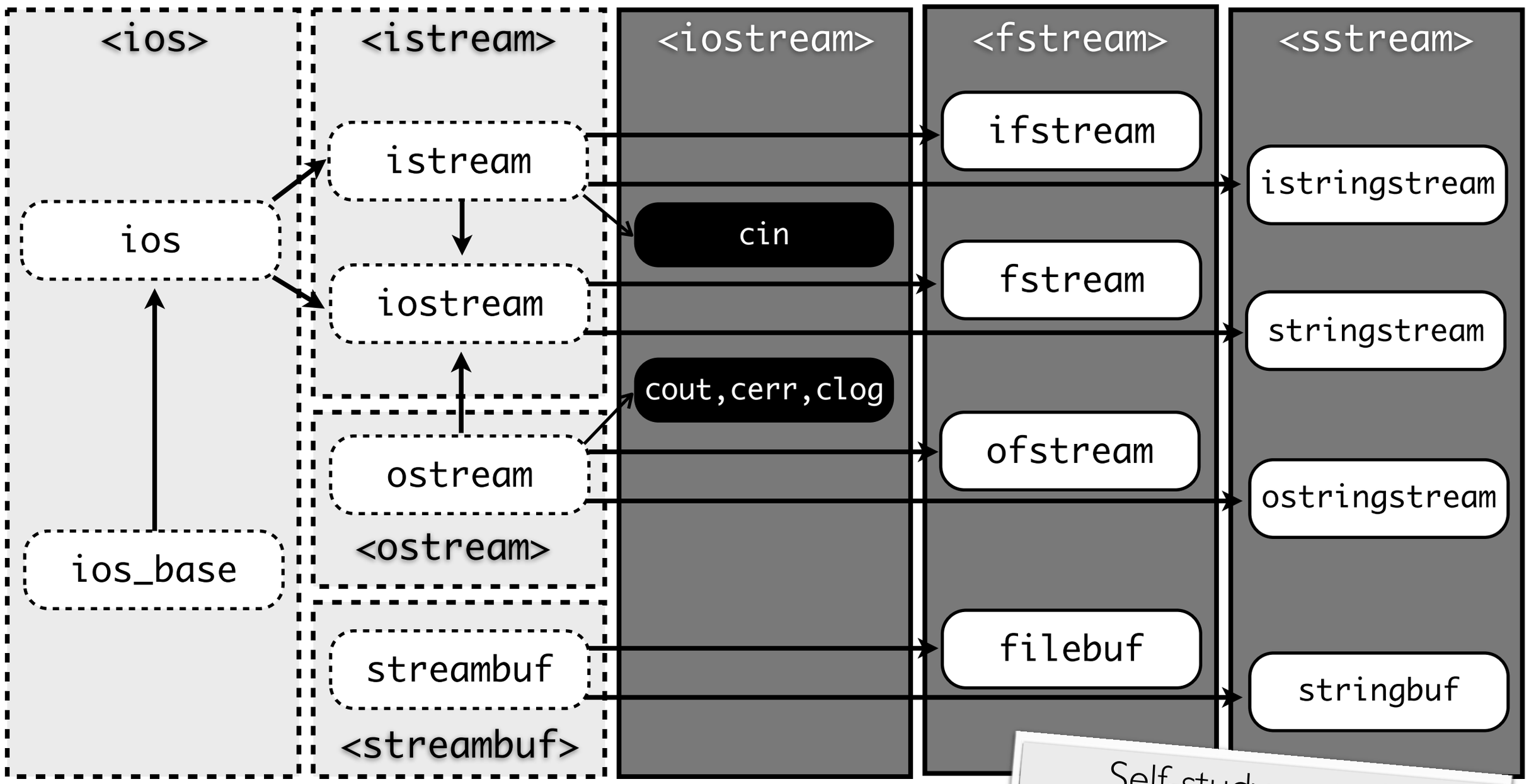
extraction operator

```
std::cin >> x >> y; // read x from cin and pass it to y
std::cout << x << y;// write x on cout and then y
```

returns a `std::ostream&` reference

Intermezzo: `iostream` Library

- An object-oriented (see later) library providing stream input/output functionality
  - `ios, istream, ostream, streambuf` provide base classes (not included directly)

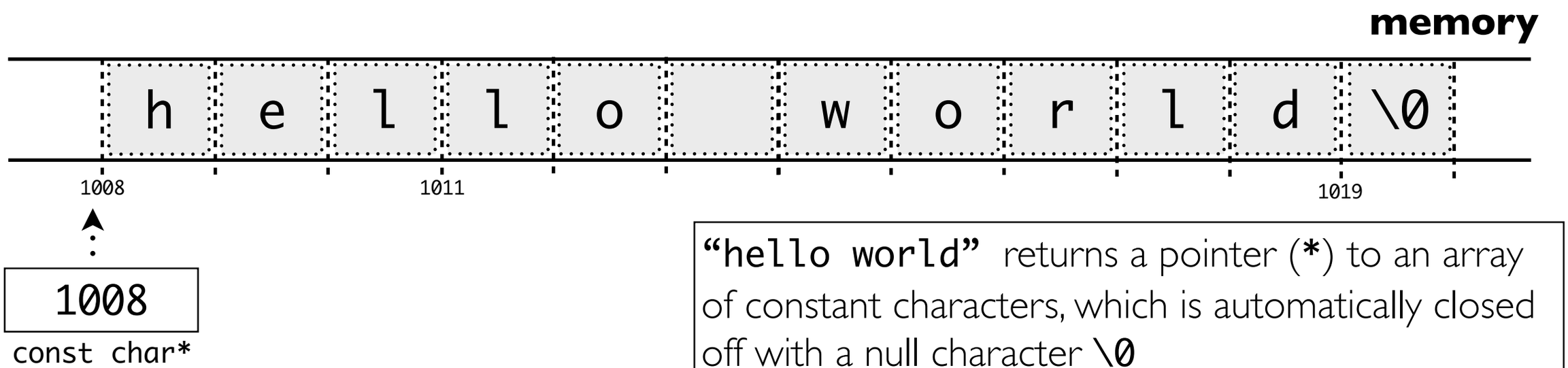


(most important elements)

Self-study material
Look it up !

Primitive Types and C-Style Strings

- String literals can be represented by using **`const char*`**
  - These are **C-style** strings, in C++ one can use the **`string`** container (later)



`"hello world"` returns a pointer (**`*`**) to an array of constant characters, which is automatically closed off with a null character **`\0`**

```
std::ostream& operator<<(std::ostream&, const char*);
const char* hi("hello world");

std::cout << "hello world";                  // hello world
std::cout << "hello" "world";                // helloworld

std::cout << "hello world\n";                // hello world (with newline)
std::cout << "hello world" << std::endl; // hello world (with newline)

std::cout << hi;                             // hello world
```

```
hello worldhelloworldhello world
hello world
hello world
```

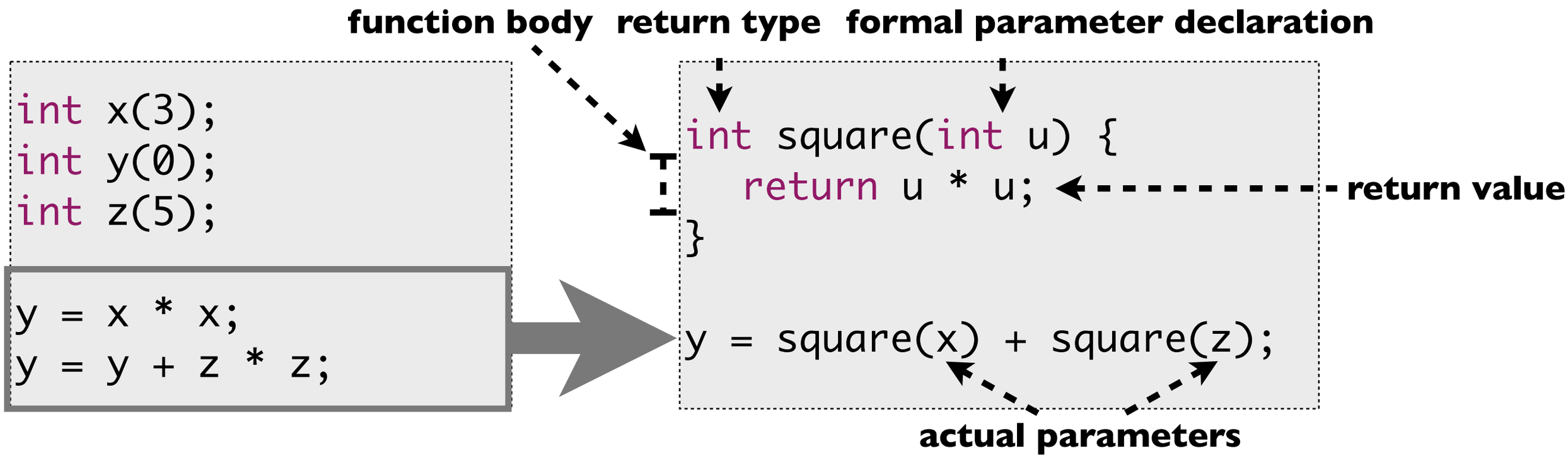
C++ Functions

Overview of Concepts

- Function declaration, function signature, function type
- Function side-effects
- Default parameter values
- Value constructor
- Variadic functions, inline functions, polymorphic functions
- Function overloading, Koenig lookup, argument-dependent lookup
- Expression statement, definition statement, control flow statement, compound statement
- Sequence operator, ? operator, if, while, do while, for, switch, return, break, continue
- Code indentation, code comments
- Automatic local objects, static local objects

Function Definition (Recap from chapter 1)

```
ReturnType
FunctionName(FormalParamDeclarationList){
     StatementList
}
```



Function Declaration : Signature, Type, Return Type

$$T_0\ f(T_1, \ldots, T_n)$$

return type of function `f` → T_0

signature of function `f` → $T_1 \times \ldots \times T_n$

type of function `f` → $T_f = [T_1 \times \ldots \times T_n \rightarrow T_0]$

```
bool smart(int u, long v) {
   return true;
}
```

type of `smart` → $T_{smart} = [int \times long \rightarrow bool]$

signature of `smart` → $int \times long$

return type of `smart` → `bool`

Functions and Side-Effects

- **Side-effect ?**
  - When two identical function calls return different answers
  - May be intentional or accidental
    - The latter is a serious problem !

```
int x(0);       // global variable

int f() {
   return x++; // change and return global
}

int main() {
   f();         // will return 0
   f();         // will return 1
}
```

Mathematical functions don't have side-effects

Default Parameter Values for Formal Parameters

- Formal parameters may be given a default value
  - Starting from right to left
    - If one parameter is omitted then all consequent parameters should be omitted

```
// print i1 separator i2, use base for representing number
void
printline(int i1, int i2, char separator = '\t', int base = 10);

// what is the output for the following function calls?
printline(10,12,',',8);
printline(10,12);
printline(10,12,'|');
printline(10,12,8);
```

```
void
print(const Student&, std::ostream& os = std::cout);

print(fred); // print(fred, std::cout);
print(lisa, std::cerr);
```

Variadic Functions (1)

- A **variadic function** is a function which accepts a variable number of arguments (a.k.a. function of indefinite arity)
  - The function can find out the number of actual parameters at run time

Ellipsis indicates variable number of arguments at this position

```
#include <cstdarg>

void
err_exit(int status, char* format, ...);

err_exit(12, "cannot open file %s: error #%d\n",filename, code);


// analyzing format tells function that there are 2
// further parameters of type const char* (%s) and
// int (%d), respectively
```

`va_start`	Macro to initialize a variable argument list (as a `va_list`)
`va_arg`	Macro to retrieve the next argument (as passed in function call)
`va_end`	Macro to end using a variable argument list (before return)
`va_list`	Type to hold information about the variable arguments

Variadic Functions (2)

```
#include <cstdarg>

using namespace std;

double average(int count, ...) {
   va_list arglist;                    //variable to hold the arguments

   double sum(0);

   va_start(arglist, count);           //initialize arglist

   for (int i = 0; i < count; i++)
      sum += va_arg(arglist, double); //access the next argument
                                      //and interpret it as type double
   va_end(arglist);                    //clean up the arglist

   return sum / count;
}
```

Remember how a variable number of arguments could be passed to the **main** function. Is this also a variadic function?

What do you think about this statement? Compare with Scheme!

```
average(4, 5.0, 10.0, 15.0, 20.0)
```

→

```
12.5
```

Inline Functions

- The keyword **`inline`** indicates to the compiler that all function calls to the function are to be replaced "in line"
  - Avoids a function call

```
inline int maximum(int i1, int i2) {
   // return the largest of two integers
   if (i1 > i2)
      return i1;
   else
      return i2;
}
```

Why not use it all the time?

Where to define it?

```
...
i = maximum(12,x+3);
...
```

Replaced (in-lined) with

```
...
int tmp(x + 3);

if (12>tmp)
   i = 12;
else
   i = tmp;
...
```

Overloading Function Definitions

- Functions with the same name but a different **signature** are different

```
int sum(int i1, int i2, int i3) {
   return i1 + i2 + i3;
}

int sum(int i1, int i2, int i3, int i4) {
   return i1 + i2 + i3 + i4;
}

float sum(float f1, float f2) {
   return f1 + f2;
}

int main() {
   sum(1, 2, 3);       // sum(int,int,int)
   sum(1, 2, 3, 4);    // sum(int,int,int,int)
   sum(5.2, 4.8);      // sum(float,float)
}
```

Polymorphic functions

Overloading Function Definitions

`<math.h>` declares mathematical functions such as `rint(double x)` (see next slide)

```
#include <math.h>

int round(double a) {
   return int(rint(a));
}

int round(int i) {
   return i;
}

int
main() {
   round(1.1); // will call round(double)
   round('a'); // error?
}
```

`double rint(double x)` will round its argument to the nearest integer (as a double)

an explicit **typecast**
(`rint` returns a `double`, round returns an `int`)
a.k.a. the **value constructor**
(T(x) constructs a value of type T out of x)

math.h (FYI)

```
/****************************************
*            Math Functions
****************************************/

extern double  acos( double );
extern float  acosf( float );

extern double  asin( double );
extern float  asinf( float );

extern double  atan( double );
extern float  atanf( float );

extern double  atan2( double, double );
extern float  atan2f( float, float );

extern double  cos( double );
extern float  cosf( float );

extern double  sin( double );
extern float  sinf( float );

extern double  tan( double );
extern float  tanf( float );

extern double  acosh( double );
extern float  acoshf( float );

extern double  asinh( double );
extern float  asinhf( float );

extern double  atanh( double );
extern float  atanhf( float );

extern double  cosh( double );
extern float  coshf( float );

extern double  sinh( double );
extern float  sinhf( float );

extern double  tanh( double );
extern float  tanhf( float );

extern double exp ( double );
extern float expf ( float );

extern double exp2 ( double );
extern float exp2f ( float );

extern double expm1 ( double );
extern float expm1f ( float );

extern double log ( double );
extern float logf ( float );

extern double log10 ( double );
extern float log10f ( float );

extern double log2 ( double );
extern float log2f ( float );

extern double log1p ( double );
extern float log1pf ( float );

extern double logb ( double );
extern float logbf ( float );

extern double modf ( double, double * );
extern float modff ( float, float * );

extern double ldexp ( double, int );
extern float ldexpf ( float, int );

extern double frexp ( double, int * );
extern float frexpf ( float, int * );

extern int ilogb ( double );
extern int ilogbf ( float );

extern double scalbn ( double, int );
extern float scalbnf ( float, int );

extern double scalbln ( double, long int );
extern float scalblnf ( float, long int );

extern double  fabs( double );
extern float  fabsf( float );

extern double  cbrt( double );
extern float  cbrtf( float );

extern double hypot ( double, double );
extern float hypotf ( float, float );

extern double pow ( double, double );
extern float powf ( float, float );

extern double  sqrt( double );
extern float  sqrtf( float );

extern double  erf( double );
extern float  erff( float );

extern double  erfc( double );
extern float  erfcf( float );

extern double  lgamma( double );
extern float  lgammaf( float );

extern double  tgamma( double );
extern float  tgammaf( float );

extern double ceil ( double );
extern float ceilf ( float );

extern double floor ( double );
extern float floorf ( float );

extern double nearbyint ( double );
extern float nearbyintf ( float );

extern double rint ( double );
extern float rintf ( float );
```

...

Self-study ! Open the header file and test some of the functions

Also defines e.g.,

```
#define M_PI        3.14159265358979323846264338327950288   /* pi */
```

Function Overloading: Determining which Function is Called

If `f(e1, ..., en)` is unqualified

1. Determine the set F of **candidate functions** that could apply by
   1. finding the **closest encompassing scope** S containing one or more function declarations for `f`
   2. same for namespaces where types of arguments of `f` are declared (Koenig lookup)
2. Find the **best match** in F for the call, i.e. the declaration `f'` whose **signature** best matches the call's signature $(T_{e_1}, \ldots, T_{e_n})$

Koenig lookup?

- Named after Andrew Koenig who invented **argument-dependent lookup** (ADL)
- Check out "Ruminations on C++: a decade of programming insight" and experience, by Andrew Koenig and Barbara E. Moo, Addison-Wesley, 1997

Notes on Lookup

- Definitions in a *closer* scope *hide* definitions in a *wider* scope

```
namespace N {
     void f(int);
     namespace M {
          int f;
          namespace K {
               void g() {
                    f(4);
               }
          }
     }
}
```

This will generate a compile error:

```
../src/Test.cpp:    In function 'void N::M::K::g()':
../src/Test.cpp:22: error: 'N::M::f' cannot be used as a function
```

- Default and unspecified arguments are taken into account when determining **F**

Overloaded Example

```
#include <iostream>
std::ostream& std::operator<<(std::ostream&, char c);
std::ostream& std::operator<<(std::ostream&, unsigned char c);
std::ostream& std::operator<<(std::ostream&, signed char c);
std::ostream& std::operator<<(std::ostream&, const char *s);
std::ostream& std::operator<<(std::ostream&, const unsigned char c);
std::ostream& std::operator<<(std::ostream&, const signed char c);

int f(int i, int j = 0) {
   return 1;
}

int f(int k) {
   return 2;
}

int main() {
   f(3); // What happens?

   std::cout << "hello world";
}
```

The Compiler generates an error for ambiguous function calls :

```
../src/Test.cpp:232: error: call of overloaded 'f(int)' is ambiguous
../src/Test.cpp:140: note: candidates are: int f(int, int)
../src/Test.cpp:144: note:                 int f(int)
```

Illustration of Koenig lookup. The following will get called:

```
std::ostream& std::operator<<(std::ostream&, const char *s);
```

Function Definition : Kinds of Statements

```
ReturnType
FunctionName(FormalParamDeclarationList){
     StatementList
}
```

- **expression statement** (incl. function call, assignment)

```
OptionalExpression ;
```

- **definition statement** (e.g., for local variables)

```
ObjectDefinition ;
```

- **control flow statement** (some have an expression analogue)

```
x = f(y);               // expression statement

double d(3.14);         // definition statement

while (x>0)             // control flow statement
    x = f(f(x));
```

Compound Statements

- **Compound statements** can be used to group statements
- Introduces a new scope, can be nested

```
{
   OptionalStatementList ;
}
```

```
{
   int tmp(x);
   x = y;
   y = tmp;
}
```

Sequence Operator

- The **sequence operator** provides an expression alternative to write compound statements
- The resulting value is the one produced by the last statement in the list

```
Expression1 , Expression2 , ... , Expressionn
```

```
x = (std::cin >> y, 2 * y);
```

≈

```
std::cin >> y;
x= 2*y;
```

`if` Control Flow Statement

```
if (Expression)
  StatementIfTrue
```

```
if (Expression)
  StatementIfTrue
else
  StatementIfFalse
```

- **expression** statement
- **definition** statement
- **control flow** statement
- **compound** statement

```
if (a>b)      // StatementIfTrue is “if(c>d)” control flow statement
  if (c>d)    // If a>b && c>d then StatementIfTrue
    x = 1;    // StatementIfTrue is “x = 1” expression statement
  else        // If a>b && c<=d then StatementIfFalse
    x = 2;    // StatementIfFalse is “x = 2” expression statement

if (x>10) {  // StatementIfTrue is compound statement
  int tmp(y);
  y = x;
  x = tmp;
}
```

Code indentation helps to visually identify what belongs where

Code Indentation and Comments (FYI)

```
if (a>b)          //test whether a bigger than b
  if (c>d)        //choose on c>d criteria
    x = 1;        //jeff, is this the right ID?
  else x = 2;     //used to be 3, but 2 works
if (x>10) {       //only swap if q<10
  int tmp(y);     //store value of y in temp
  y = x;          //... ... ... :-p
  x = tmp;        //voila, swapped !
}
```


```
if (a>b) if (c>d) x=1; else x=2; if (x>10) { int tmp(y); y=x; x=tmp;}
```

```
if (a>b)
  if (c>d)
    x = 1;
  else
    x = 2;


if (x>10) {
  int tmp(y);
  y = x;
  x = tmp;
}
```

```
if (a>b)
  if (c>d) x = 1;
else x = 2;
  if (x>10) {
int tmp(y);
  y = x;
x = tmp;}
```

```
if (a>b)
if (c>d)
x = 1;
else
x = 2;  if (x>10)


{
int tmp(y);
y = x;
x = tmp;
}
```

? Operator

- The **`? operator`** provides an expression alternative for the **`if`** control flow statement
- If **`ExpressionIf`** and **`ExpressionElse`** return lvalues of the same type then the result is also an lvalue

```
Condition ? ExpressionIf : ExpressionElse
```

```
m = (x>=y ? x : y) + 1;
```

≈

```
if (x>=y)
  m = x+1;
else
  m = y+1;
```

```
(x >= y ? x : y) = 0;
```

≈

```
if (x>=y)
  x = 0;
else
  y = 0;
```

`while` Control Flow Statement

```
while (Expression)
   Statement
```

- **expression** statement
- **definition** statement
- **control flow** statement
- **compound** statement

≈

```
if (Expression) {
   Statement;
   while
(Expression)
   Statement;
}
```

```
int
factorial(int n) {
   int result(1);
   while (n > 1) {
      result *= n; // can be done in 1 line
      --n;
   }
   return result;
}
```

`do while` Control Flow Statement

```
do
  Statement
while (Expression)
```

≈

```
Statement
while (Expression)
  Statement
```

```
int sum(0);
do {
   int i(0);
   std::cin >> i;
   sum += i;
}
while (std::cin); // while state of input stream is ok
```

`for` Control Flow Statement (1)

```
for ( ForInitialization ;
      (ForCondition) ;
      ForStep )
  Statement
```

≈

```
ForInitialization;
while (ForCondition) {
   Statement
   ForStep;
}
```

```
int factorial(int n) {
   // make a copy of the input, but 0 becomes 1
   int result(n ? n : 1);
   // if n <=2, the result is n and we are done
   if (result > 2)
      // multiply with every number < n, except 1
      for (int j = 2; (j < n); ++j)
         // use compound assignment for multiplication
         result *= j;
   return result;
}
```

Intermezzo: Performance/Readability/... : Who "Wins" ? (1)

```
#include <iostream>
using namespace std;

int
factorial(int n) {
   int result(n ? n : 1);

   if (result > 2)
      for (int j = 2; (j < n); ++j)
         result *= j;

   return result;
}

int main() {
   cout << fac(10);
   return 0;
}
```

```
*** fac_for(10) = 3628800 ***
started at 12935
ended at 12937
duration 0.003ms
```

```
#include <iostream>
using namespace std;

int
factorial(int n) {
   if (n == 0)
      return 1;
   else
      return n * factorial(n - 1);
}

int main() {
   cout << fac(10);
   return 0;
}
```

```
*** fac_rec(10) = 3628800 ***
started at 12905
ended at 12909
duration 0.005ms
```

Intermezzo: Performance/Readability/... : Who "Wins" ? (2)

```
int fac_for(int n) {
    int result(n ? n : 1);
    cout << "### result is " << result << endl;          // for debugging

    if (result > 2)
        for (int j = 2; (j < n); ++j) {
            result *= j;
            cout << "### result is " << result << endl; // for debugging
        }

    return result;
}
```

```
*** fac_for(10) duration 0.003ms ***
*** fac_rec(10) duration 0.005ms ***
```

```
result is 10
result is 20
result is 60
result is 240
result is 1200
result is 7200
result is 50400
result is 403200
result is 3628800
*** fac_for_logging() result is 3628800 ***
started at 12954
ended at 12976
duration 0.023ms
```

Intermezzo: Performance/Readability/... : Who "Wins" ? (3)

0) The "winner" depends on the objectives

1) Make it work
(just make the code do what it should do, no optimization for speed/size/extensibility)

2) Make it right
(refactor your code, get rid of redundancy, make the implementation modular)

3) Make it fast
(optimize the execution time, use profiling/benchmarking to see the effect)

∞) Continuous testing
(ensure that it works as expected and that you don't break anything while making it right & fast)

`for` Control Flow Statement (2)

What is the output of the following program?

```
void repeater(int n) {
    for (int j(1); (j = n); ++j)
        std::cout << "HI" << std::endl;
}

...
repeater(5);
...
```

This program will loop indefinitely!

A non-zero value is considered as true in C++, and `j = n` returns an lvalue

`switch` Control Flow Statement (I)

```
switch ( Expression ) {
   case Constant1: Statement1 break;
   ...
   case ConstantN: StatementN break;
   default: Statement
}
```

≈

```
int tmp(Expression);
if (tmp == Constant1)
   Statement1
else if (tmp == Constant2)
      Statement2
   else if ...
         else Statement
```

```
switch (customer_loyalty){
  case 1:
    discount = 1.0;
    break;
  case 2:
    discount = 1.5;
    break;
  case 3:
    discount = 2.5;
    break;
  case 4:
    discount = 15.0;
    break;
  default:
    discount = 0.0;
}
```

Note: **Expression** must be an **int**. Why?

```
if (customer_loyalty == 1)
   discount = 1.0;
else
   if (customer_loyalty == 2)
      discount = 1.5;
   else
      if (customer_loyalty == 3)
         discount = 2.5;
      else
         if (customer_loyalty == 4)
            discount = 15.0;
         else
            discount = 0.0;
```

`switch` Control Flow Statement (2)

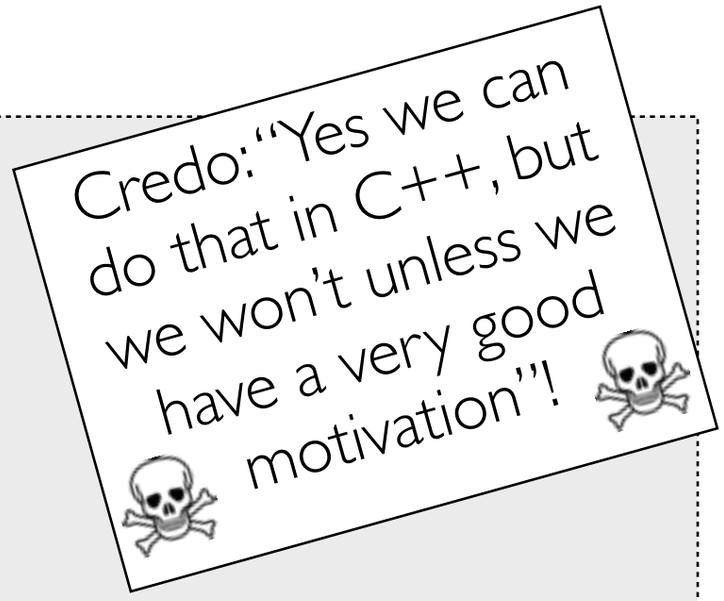
What is the output of the following programs?
Is it easy to add new discount types?
(e.g. loyalty 5=6.0 and loyalty discounts depending on additional promotions)

```
int customer_loyalty(3);
double discount(-1);

switch (customer_loyalty){
  case 1:
    discount = 1.0;
  case 2:
    discount = 1.5;
  case 3:
    discount = 2.5;
  case 4:
    discount = 15.0;
  default:
    discount = 0.0;
}

std::cout << discount;
```

```
int customer_loyalty(3);
double discount(0.0);
bool season_promo(true);
switch (customer_loyalty){
  case 1:
    discount = discount + 1.0;
    if (!season_promo) break;
  case 2:
    discount = discount + 1.5;
    if (!season_promo && customer_loyalty == 1) break;
  case 3:
    discount = discount + 2.5;
    if (!season_promo && customer_loyalty == 2) break;
  case 4:
    discount = discount + 15.0;
    if (season_promo && customer_loyalty == 4)
       discount = discount + 5.0;
    break;
  default:
    discount = 0.0;
}
std::cout << discount;
```



`switch` Example (I)

```
#include <stdlib.h>                    // for the random() functions
#include <time.h>                      // for the time() function
int throwdice(int score) {             // throw dice and return new score
                                       // use current time() value as seed for the
   srandom(time(0));                   // random number generator
   int result(random() % 6 + 1);       // throw dice
   switch (result) {                   // update score, depending on result
   case 1:
      score += 1; break;
   case 2:
      score += 3; break;
   case 3:
      score += 4; break;
   case 4:
      score += 6; break;
   case 5:
      score += 6; break;
   case 6:
      score += 8; break;
   default: exit(1);                   // should not happen, exit(EXIT_FAILURE)
   }
   return score;
}
```

`switch` Example (2)

```
// return true iff c is a vowel letter
bool
isvowel(char c) {
   switch (c) {
      case 'a': case 'e': case 'i':
      case 'o': case 'u':
      case 'A': case 'E': case 'I':
      case 'O': case 'U':
         // no need for break; we exit immediately
         // from the function
         return true;
      default:
         return false;
   }
   exit(EXIT_FAILURE); // should never get here
}
```

`return` and `break` Statements

```
return Expression;
break;
```

The `break` statement breaks out of the **enclosing loop** or **enclosing switch**

```
#include <iostream>

void
break_demo() {
   for (int j = 0; (j < 10); ++j) {
      for (int i = 0; (i < 5); ++i) {
         if (i > 3)
            break;
         std::cout << i << " ";
      }
      std::cout << "\n";
   }
}
```

Output?

```
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
0 1 2 3
```

`continue` Statement

```
continue;
```

The **`continue`** statement interrupts the current iteration and immediately starts the next iteration of the enclosing loop

```
#include <iostream>
void process(int); // declaration
...
int main() {
   int i(0);
   while (std::cin >> i) {
      // an input stream can be converted to a bool;
      // .. the result is true iff input ok
      if (i <= 0) // i not positive, throw away
         continue;
      if (i % 2) // i odd, throw away
         continue;
      process(i); // i ok, process it
   }
}
```

Look this up !

Automatic Local Objects

Automatic local objects for local variables are defined in a function body or compound statement scope and are destroyed when control leaves the scope (frame stack is popped)

```
void f() {
   {                    // Start compound statement,
                        // which has its own nested scope.
      int x(3);         // local object definition
      x *= 2;
   }                    // End compound statement and scope
   std::cout << x; // ???
}
```

```
../src/demo.cpp:166: error: 'x' was not
declared in this scope
```

Static Local Objects

Static local objects are **persistent** across function calls or scope entries (but the same accessibility rules as for other local variables apply)

```
int counter() {
   static int n(0); // Created at first call of counter()
   return ++n;
}
...
std::cout << counter () << " - "
          << counter () << " - "
          << counter () << std::endl;
```

Side-effects?

Output?

3 - 2 - 1

Returning a Reference to a Static Local Object?

Static references can be returned safely

```
int&
silly() {
   static int n(0);
   return n; // ok, n is static
}

...

silly() += 3; std::cout << silly();
```

Output?

3