Data Structures and Algorithms

Lecturer: Dr. Derek Pao
Office: Room G6514
Email: d.pao@cityu.edu.hk
Tel: 3442-8607

Tentative syllabus

- Review of basic concepts
  - Scalar, Structure, Pointer, and Reference variables
  - Parameter passing in function call
  - 1-D and 2-D Arrays
  - Loop design
  - Relationship between data structure and algorithm
  - Computation complexity
  - C++ class
    - constructor and destructor,
    - operator overloading, friend function
    - static and dynamic binding of function calls,
    - object variable vs object reference (pointer)
- Sorting and Searching algorithms
- Linked lists
- Stacks and Queues
- C++ Standard Template Library
- Recursion
- Binary trees

Intended Learning Outcomes

Students should be able to

- apply the object-oriented programming approach to solve computation problems
- solve computation problems using recursion where appropriate
- demonstrate applications of standard data structures such as linked list, stack, queue and tree
- apply different sorting and searching algorithms

References:

C++ on-line tutorial, http://www.cplusplus.com/doc/tutorial/

D. S. Malik, Data Structures Using C++, 2nd Edition, 2010.

Assessment

Examination: 60% (2 hours, closed book)

Coursework: 40%

- Tutorials and Assignments – 25%
- Tests – 15% (week 8)

In order to pass the course, you need to obtain

(i) at least 30% in examination, and

(ii) at least 30% in coursework.

Copying of assignments is strictly prohibited.
If the submitted program of student *A* is found to be the "same" as that of student *B*, both students *A* and *B* will receive zero mark for that assignment. The scores for Tutorial Attendance & Performance will also be reduced to zero as an additional penalty.

To pass this course:

You need to acquire basic skills in writing programs and solving problems, not just memorize syntax of the programming language or example program codes.

You should not memorize the example programs, but you should pay attention to every detail in the program design. A minor change in a statement can have very different meanings in the overall program.

There are 2 levels of program design

Basic level : coding

- The solution strategy is simple. You can work out the answer using paper and pencil without difficulty.
- You should be able to organize and express the computation steps using the target programming language in a clear and precise manner.
- You should train yourself to read and understand program codes. Also learn from the coding style of the example programs in the notes and tutorials.

More advanced level : algorithm design

- This is the ability to solve more complex problems. You need to find out a suitable solution strategy.
- Typically, you need to observe and make use of suitable properties of the data structures in order to derive an effective solution.
- There is no universal strategy that can solve all problems.
- In this course, I shall introduce to you how to design an algorithm for some classical problems. You should pay attention to the design process rather than memorizing the program codes. Questions in the test/exam will NOT be identical to the examples used in the notes or tutorials.

Warning:

If you skip one or more lectures, you are very likely to fail this course.

If you do not make efforts in solving the tutorial problems yourself, you will likely fail this course.

Proper approach to design computer program

Steps:

1. Study the problem: understand the computation requirements and visualize the solution strategy.

   Work out one or more examples based on your solution strategy using paper and pen.

2. Make a plan for the program organization, function interfaces and control block structures (loop control).
3. Design a draft in the form of pseudo-codes using paper and pen.
4. Write the detailed program when you have the overall picture in your mind.
5. Examine the program carefully, and try to make sure that it is correct.
6. Only when you are convinced that the program is correct, then test the program with properly designed test data.
7. After you get the correct outputs, try to further improve and simplify your program.

Goals:

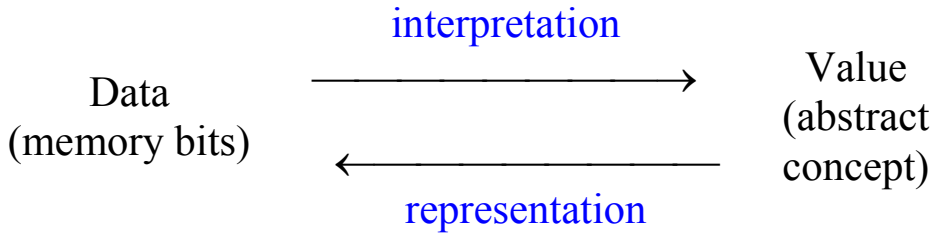
No need to do any debugging for simple problems (e.g. tutorial exercises).

If errors are observed, go back to examine the program codes.

How to locate the source of an error is an important skill that you need to acquire.

Fix the error and learn from the mistake (do not repeat the same mistake in the future).

Data versus value



Examples:

Data	Value	interpretation/representation
0100 0001	char 'A'	8-bit ASCII
	65	signed or unsigned integer
1111 1111	255	unsigned 8-bit integer
	–1	signed 8-bit integer (2's complement)
	–127	signed 8-bit integer (sign-magnitude)

Fundamental (Primitive) data types in C++ (for today's PC/compiler)

- Interpretation is built-in the hardware circuits of the CPU or compiler

data type	size (byte)	interpretation/representation	range of values
bool	1	Boolean (not available in C)	false or true
char	1	signed number (2's complement)	–128 to 127
unsigned char		unsigned number	0 to 255
int	4	signed number (2's complement)	-2^{31} to $2^{31}-1$
unsigned int		unsigned number	0 to $2^{32}-1$
short	2	signed number (2's complement)	-2^{15} to $2^{15}-1$
unsigned short		unsigned number	0 to $2^{16}-1$
long	4	signed number (2's complement)	-2^{31} to $2^{31}-1$
unsigned long		unsigned number	0 to $2^{32}-1$
long long	8	signed number (2's complement)	-2^{63} to $2^{63}-1$
unsigned long long		unsigned number	0 to $2^{64}-1$
float	4	IEEE 32-bit floating point number	$\pm 1.4\times10^{-45}$ to $\pm 3.4\times10^{38}$
double	8	IEEE 64-bit floating point number	$\pm 5\times10^{-324}$ to $\pm 1.798\times10^{308}$
pointer	4	memory address	0 to $2^{32}-1$

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

White space characters include Space, Line feed, Carriage feed, Horizontal tab, Vertical tab, and Form feed.

Space = `' '` (0x20)
Horizontal tab (HT) = `'\t'` (0x09)
Line feed (LF) is often called newline = `'\n'` (0x0A)
Carriage feed (CR) = `'\r'` (0x0D)

ASCII codes for `'0'` to `'9'` are 0x30 (hexadecimal) to 0x39.
ASCII codes for `'A'` to `'Z'` are 0x41 to 0x5A.
ASCII codes for `'a'` to `'z'` are 0x61 to 0x7A.

ASCII codes for lowercase and uppercase letters differ by 1 bit (5-th bit).

```
'A' = 0100 0001
'a' = 0110 0001
```

```
char ch = 'A' + 32; // ch == 'a'
```

Operators in C++ (not a complete list)

Operator	Symbol	Description
Assignment	=	store a value into a variable
Arithmetic	+, -, *, /, %	
Increment, decrement	++, --	
Unary minus	-	negation
Relational	==, !=, <, <=, >, >=	equal, not equal, less than, etc.
Logical	&&, ||, !	and, or, not
Bitwise	~, &, |, ^, <<, >>	complement, and, or, exclusive-or shift left, shift right
Insertion extraction	ostream << s istream >> i	insertion to an output stream, extraction from an input stream
Member and pointer	x[i]	subscript operator x is an array or a pointer
	*x	indirection, dereference operator x is a pointer
	&x	reference (address-of operator) address of x
	x->y	Structure dereference (arrow operator) x is a pointer to object/struct, member y of object/struct pointed to by x
	x.y	Structure reference (field select operator) x is an object/struct, member y of x

Scalar variables and pointers

Example 1

```
int a, b, c;
int *p;  /* p is an integer pointer
            p stores the address of an integer variable,
            p points to an integer */

a = 1;  /* assign the value 1 to a    */
b = 2;  /* b is assigned the value 2 */
p = &c; /* &c refers to the address of c,
           assign the address of c to p */

*p = a + b;  /* *p refers to the contents at location p
                Effect: value of c is set to 3 */
```

Example 2

```
int a, b, c;
int *p;  // initially p = nullptr or undefined

a = 1;
b = 2;
*p = a + b;  /* Error: null-pointer exception.
                Cannot store a value to an undefined
                memory location */
```

Example 3

```
double d;
int *p;

p = &d;  /* Error: type mismatch.
            An integer pointer cannot be used to point
            to a double precision number.  */
```

Integer arithmetic (Be careful with the use of integer division)

```
int a = 3;
int b = 2;
int c = 4;

cout  << a / b * c << endl;  // output is 4 !!
cout  << a * c / b << endl;  // output is 6
```

The NULL value in C/C++ is used to denote an invalid address.

The symbol NULL is represented by the value 0.

```
int *p = NULL;
double *q = NULL;
// Assign a NULL value to a pointer means that the pointer
// variable is not pointing to a valid data item.


int i = NULL;
double d = NULL;
// Syntactically correct, but the semantics (meaning) of
// the statement is confusing. Zero is not an invalid value.

// correct way to initialize a scalar variable
int i = 0;
double d = 0;
```

Preferred coding syntax is to use `nullptr` instead of `NULL`.

```
int *p = nullptr;  // OK

int i  = nullptr; // syntax error
```

Parameter passing in function call

1. Pass by value

```
void swap1(int x, int y) // x and y are function parameters
{
    int t;  // t is a local (automatic) variable

    t = x;
    x = y;
    y = t;
}

int main()
{
    int i = 1;
    int j = 2;

    swap1(i, j);
    cout << "i = " << i << ", j = " << j << endl;
    // output: i = 1, j = 2
}
```

2. Pass by pointer in C++ (pass by reference in C)

In the C language, reference to a formal parameter is explicitly specified as a pointer.

```
void swap2(int *x, int *y) // x and y are integer pointers
{
    int t;

    t = *x;
    *x = *y;
    *y = t;
}

int main()
{
    int i = 1;
    int j = 2;

    swap2(&i, &j); // pass the address of i and j to swap()
    cout << "i = " << i << ", j = " << j << endl;
    // output: i = 2, j = 1
}
```

3. Pass by reference in C++ (NOT supported in C)

In C++, you can specify a formal parameter in the function signature as a reference parameter.

```
void swap3(int& x, int& y) //x and y are passed by reference
{
    int t;

    t = x;  // x references i
    x = y;  // y references j
    y = t;
}

int main()
{
    int i = 1;
    int j = 2;

    swap3(i, j);
    cout << "i = " << i << ", j = " << j << endl;
    // output: i = 2, j = 1

    // Compiler finds that swap3() receives the arguments
    // by reference, the references (addresses) of i
    // and j are passed to swap3().
    // After the function return, i = 2, j = 1.
}
```

Reference datatype

```
int i = 2;

int& r = i;    //r is a reference to an integer
               //an initial value must be provided in the
               //declaration of r,
               //exception: member variable of a class

int *p = &i;   // &i means address of i

cout << "value of i = " << i << endl;
cout << "value of r = " << r << endl;
cout << "value of p = " << p << endl;
cout << "value of *p = " << *p << endl << endl;
```

Outputs produced:

```
value of i = 2
value of r = 2
value of p = 001AF9C0
value of *p = 2
```

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

r = 4;    //r references i, value of i is changed to 4
int *q;
q = &r;   //q points to i !!

cout << "Execute r = 4" << endl;
cout << "value of i = " << i << endl;
cout << "value of *q = " << *q << endl;
```

Outputs produced:

```
Execute r = 4
value of i = 4
value of *q = 4
```

C++ references differ from pointers in several essential ways:

- It is not possible to refer directly to a reference object after it is defined; any occurrence of its name refers directly to the object it references.
- Once a reference is created, it cannot be later made to reference another object; it cannot be *reseated*. This is often done with pointers.
- References cannot be *null*, whereas pointers can; every reference refers to some object, although it may or may not be valid. Note that for this reason, containers of references are not allowed.

```
int& r[10]; // array of reference type is NOT allowed
```

- References cannot be uninitialized. Because it is impossible to reinitialize a reference, they must be initialized as soon as they are created. In particular, local and global variables must be initialized where they are defined, and references which are data members of class instances must be initialized in the initializer list of the class's constructor.

Arrays

All elements of an array have the same fixed, predetermined size.

For example:

```
void myFunction()
{
   int b[10]; // static allocation (compile-time)

   int n = 100;
   int *a = new int[n]; // dynamic allocation (run-time)
                        // of the array

   // other statements in the function
}
```

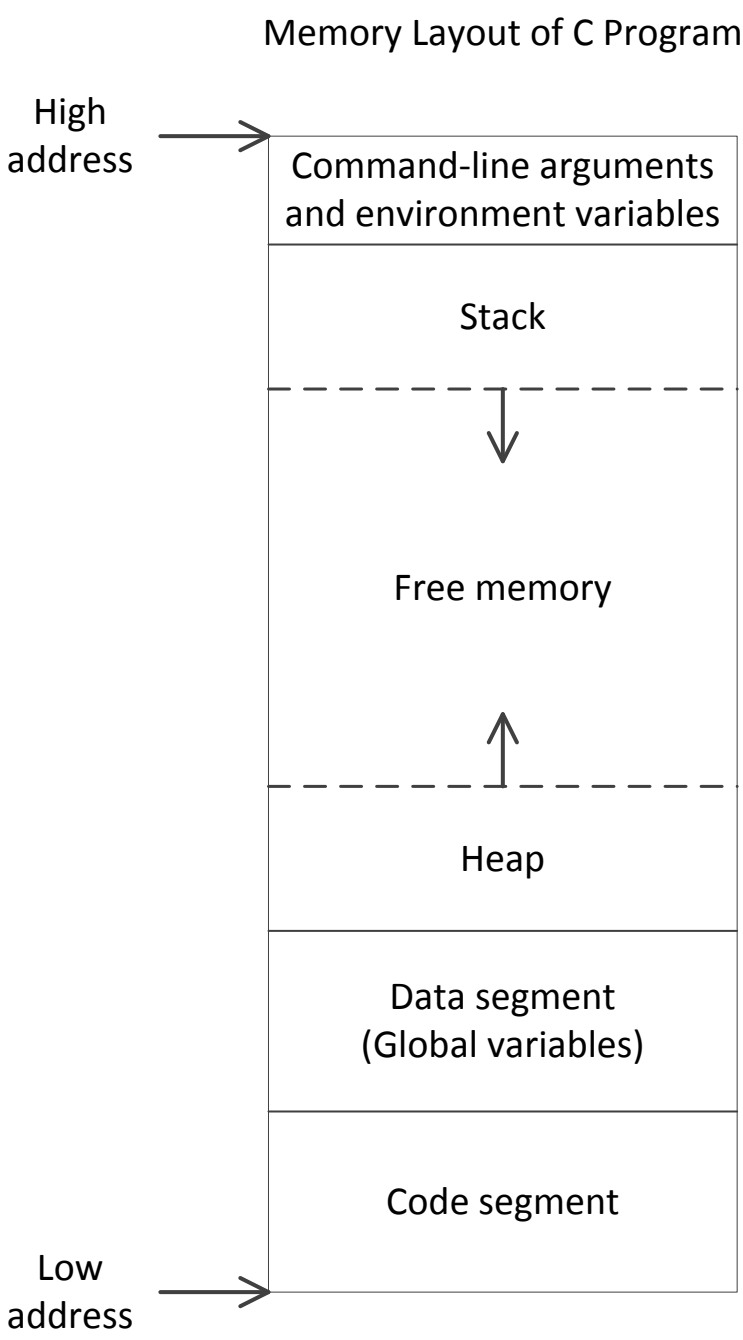
Variables `a, b,` and `n` are called automatic variables. They are mapped to memory locations in the system stack. They cease to exist (destroyed) after the function returns.

The array created by calling the **`new`** operator is allocated in the heap area (a dynamic memory pool managed by the operating system). The memory block is still in use after the function returns.

Memory allocation can also be done using the system calls `malloc()` and `calloc()`.

Remark:

```
int *p = new int[0]; // create an array of size 0

if (p == nullptr)
   cout << "p is nullptr\n";
else
   cout << "p is not nullptr\n";

// p is nullptr or not ??
```



The stack segment is used to support function calls.
The heap segment is used to support dynamic memory allocation.

Stack and heap segments grow in opposite direction.

When the stack pointer meets the heap pointer, free memory is exhausted and the program will be terminated.

The default amount of free memory is typically a few hundred MBytes in today's computer system.

Array mapping function:

Address of the first element in an array is called the *base address* of the array. Today's computers are byte-addressable (address value refers to a specific byte).

Let *esize* be the size of the array element (in term of bytes).

address of $b[i]$ = $base(b) + i * esize$

The size of `int` = 4 bytes.
Index value of an array always starts at 0 (in C/C++).

In the C/C++ language an array variable in a function call is interpreted as a pointer variable. The 2 implementations of the function `sum` are equivalent.

Version 1:

```
int sum_1(int a[], int n)  // n = no. of elements in a[]
{
    int t = 0;
    for (int i = 0; i < n; i++)
        t += a[i]; // add the value stored at location
                   // base(a) + i*esize to variable t
    return t;
}
```

Version 2:

```
int sum_2(int *a, int n)
{
    int t = 0;
    for (int i = 0; i < n; i++)
        t += a[i];

        //same as t += *(a+i)
        //a+i is translated to a+i*esize
        //esize is implied by the data type

    return t;
}
```

Common errors made by students in array declaration:

```
int n=100; // initialization of n is done during run-time

int A[n];  // Error: dynamic array size not allowed in
           // compile-time

void myFunc(int n)
{
   int A[n]; // Error: value of n is not known during
             // compilation time.

   // function body

}
```

Addition and subtraction operations on pointers.

```
double d[10];
double *p, *q, *r;

p = d;     // p points to d[0]
q = p + 8; // q points to d[8]

int k  = q - p; // p and q must be pointer of same data type
// k = 8, number of elements (double) between q and p

p++; // after increment, p points to d[1]
q--; // after decrement, q points to d[7]

r = p + q; // syntax error
// the 2nd operand of the + operator must be int or unsigned
```

Simple examples on loop design for the processing of arrays.

for-loop, while-loop, and do-loop

For-loop:

```
for (initialization; loop_test; loop_counting)
{
    // loop-body
}
```

Typical uses: compute the sum of an array with *n* elements

```
int sum = 0;  //initialization
for (int i = 0; i < n; i++)
   sum += A[i];
```

A for-loop can be replaced by a while-loop

```
initialization_statement;
while (loop_test)
{
    // loop-body

    loop_counting;
}
```

Using a while-loop to compute the sum of an array:

```
int sum = 0;
int i = 0;
while (i < n)
{
   sum += A[i];
   i++;
}
```

do-loop

```
int sum = 0;
int i = 0;
do
{
    sum += A[i];
    i++;
} while (i < n);
```

The above do-loop may produce error if the length of the array is equal to zero, i.e. `n == 0;`

Use a do-loop only when you are 100% sure that the loop-body would be executed at least once for all possible input data.

DO NOT use `!= (Not equal)` to test the end of a range

```
for (i = 1; i != n; i++)
{
   // loop body
}
```

The test `i != n` is a poor idea.

How does the loop behave if `n` happens to be zero or negative ?
Debugging of the program for this type of error can be very difficult.

The remedy is simple. Use `<` rather than `!=` in the test condition:

```
for (i = 1; i < n; i++)
{
   // loop body
}
```

Another bad programming style is to modify the value of the loop-counter inside the body of a for-loop.

```
for (i = 1; i < n; i++)
{
   // main body of the loop

   if (someCondition)
      i = i + displacement;

   // i++ is executed before going back to top of the loop
}
```

A cstring is an array of char terminated by the delimiter '\0'.

```
char t[] = "This is a cstring";
```

The contents of the array:

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t[]	T	h	i	s	*sp*	i	s	*sp*	a	*sp*	c	s	t	r	i	n	g	\0
(hex)	54	68	69	73	20	69	73	20	61	20	63	73	74	72	69	6E	67	00

sp denotes the space char ' ' (0x20).
'\0' is the end of string delimiter.

Example cstring functions (the actual implementations in the C-library may be different from the codes shown below)

```
unsigned strlen(const char *str)
{
   // return the length of the string (exclude '\0')
   unsigned i = 0;
   while (str[i] != '\0')
      i++;

   return i;
}

int strcmp(const char *str1, const char *str2)
{
   // compare str1 with str2
   int i = 0;
   int c = str1[0] - str2[0];
   while (c == 0 && str1[i] != '\0' && str2[i] != '\0')
   {
      i++;
      c = str1[i] - str2[i];
   }
   return c;

   // if str1 > str2,  return a positive value
   // if str1 == str2, return 0
   // if str1 < str2,  return a negative value

   // Remark: implementation of the C-library function
   //         returns 1, 0, or -1
}
```

```
char* strchr(const char *str, int character)
{
   //return a pointer to the first occurrence of chararacter
   //in the cstring str

   //Note that the parameter character is a 4-byte int.

   char *p = str;
   while (*p != character)
   {
      if (*p != '\0') // '\0' is considered part of
         p++;         // the cstring in this function
      else
         return nullptr;
   }
   return p;
}
```

Example usages of `strchr()`

```
int main()
{
   char t[] = "This is a cstring";

   char *p = strchr(t, 's');

   int index = -1;

   if (p != nullptr)
      index = p - t;  // p and t are byte addresses

   cout << "Index of char 's' in t[] = " << index << endl;
   // Output: Index of char 's' in t[] = 3

   int count = 0;
   char *s = t;
   while (p = strchr(s, 'i'))  // p != nullptr
   {
      count++;
      s = p + 1;
   }
   cout << "No. of times 'i' appears in t[] = " << count
        << endl;
   // Output: No. of times 'i' appears in t[] = 3
}
```

```
int atoi(const char *a)  // ASCII to integer
{
   // Convert a string of digits (sign-magnitude format)
   // to an integer (2's complement format).

   int i = 0;
   while (isspace(a[i]) // skip the leading white space char
      i++;

   int sign = 0;  // non-negative

   if (a[i] == '-')
   {
      sign = 1;  // negative
      i++;
   }
   else if (a[i] == '+')
      i++;

   int value = 0;
   while (isdigit(a[i])) // terminate loop if a[i] == '\0'
   {
      value = value * 10 + (a[i] - '0');
      i++;
   }

   if (sign)
      value *= -1;

   return value;
}

// Remark: isalpha(c) returns true if c is a letter
//         'a' to 'z' or 'A' to 'Z'
```

Examples on loop-design

Find the maximum value in an array of integers

Common mistake of students:

```
int findMax(int A[], int n)  // n = no. of elements in A[]
{
   int max; //variable to store the result

   max = 0;

   for (int i = 0; i < n; i++)
      if (A[i] > max)
         max = A[i];

   return max; //wrong result if numbers in A[] are -ve
}
```

In the following examples, I would like to draw your attention to the uses of

preconditions – properties of the input data

postconditions – results to be produced by the loop, and

assertions – properties that are true at specific program locations.

Also, the examples serve to illustrate the relations between "data structures" and "algorithms".

Find the minimum value in an array of integers

Version 1: unordered array

```
//  precondition: n > 0 and A[] is unordered
int findMin_1(int A[], int n)  //n = no. of elements in A[]
{
   int min; //var to store the result

   min = A[0];
   for (int i = 1; i < n; i++)
      //assert: min = smallest value in A[0..i-1]
      if (A[i] < min)
         min = A[i];

   //postcondition: when the loop terminates,
   //               min = smallest value in A[0..n-1]

   return min;
}
```

Version 2: ordered array

```
//  precondition: n > 0 and A[] is in ascending order
int findMin_2(int A[], int n)  //n = no. of elements in A[]
{
   //assert: given A[] is sorted in ascending order
   //        A[0] is the smallest number in A[0..n-1]

   return A[0];
}
```

Find the location of the minimum value in an array of integers

```
//  precondition: n > 0 and A[] is unordered
int findMinLoc_1(int A[], int n)
{
   int min;

   min = 0;
   for (int i = 1; i < n; i++)
      //assert: ???
      if (A[i] < A[min])
         min = i;

   return min;
}
```

```
//  precondition: n > 0 and A[] is in ascending ordered
int findMinLoc_2(int A[], int n)
{
   return 0;
}
```

Remove duplicated values in an array of integers

Unordered array

```
A[] = {3, 6, 4, 7, 3, 3, 5, 6, 4, 3, 2, 8}
n = 12

After removing duplicated values
A[] = {3, 6, 4, 7, 8, 2, 5}
n = 7
```

Note that the order of the numbers in the resultant array is not important, because the array is unordered.

```
// Precondition: n > 0 and A[] is unordered
void removeDup(int A[], int& n) // n is passed by reference
{

   for (int i = 0; i < n-1; i++)
   {
      // assert: A[0..i] are distinct
      int j = i+1;
      while (j < n)  // should not use for-loop, why ??
      {
         if (A[j] == A[i])
         {
            // move the last element to location j
            A[j] = A[n-1];
            n -= 1;  // no. of element in A[] is reduced
         }
         else
            j++;  // increment j is conditional
      }
   }
   /* Postcondition: elements in A[0..n-1] are distinct, and
      A[] is unordered. */
}
```

Contents of the array after removal of duplicated values

index	0	1	2	3	4	5	6	7	8	9	10	11
A[]	3	6	4	7	8	2	5	4	4	3	2	8
	← valid data values in array → n = 7							← logically removed → (garbage values)				

Ordered array

```
A[] = {2, 3, 3, 3, 3, 4, 4, 5, 6, 6, 7, 8}
n = 12

After removing duplicated values
A[] = {2, 3, 4, 5, 6, 7, 8}
n = 7
```

Property of the array should be preserved.

Numbers in A[] should be maintained in ascending order.

```
// Precondition: n > 0 and A[] is in ascending ordered
void removeDup_2(int A[], int& n)
{
   int i = 0;
   while (i < n)
   {
      //assert: A[0..i] are distinct and ordered
      int j;
      for (j = i+1; j < n && A[j] == A[i]; j++)
         ;

      //assert: A[j] != A[i] and A[i..j-1] have equal value
      if (j > i+1) //(j-i) > 1 copies of the same number
      {
         //shift A[j..n-1] to the left by j-i-1 slots
         for (int k = j, t = i+1; k < n; k++, t++)
            A[t] = A[k];

         n -= (j-i-1);  //j-i-1 elements are removed
      }
      i++;
   }
}
/* Postcondition: elements in A[0..n-1] are distinct and
   arranged in ascending order. */
```

Remarks:

- Each variable (e.g. i, j, k, n) in the program has a meaning.
- You need to resolve the relationships among the variables in your program statements.

```
// Optimized implementation with linear time complexity

// Precondition: n > 0 and A[] is in ascending ordered
void removeDup_3(int A[], int& n)
{
   int k = 0;
   // k points to the location to store the next distinct
   // elmement.

   int i = 0;
   while (i < n)
   {
      int j;
      for (j = i+1; j < n && A[j] == A[i]; j++)
         ;

      //assert: A[j] != A[i] and A[i..j-1] have equal value

      A[k++] = A[i];
      i = j;
   }
   n = k; // k = no. of distinct elements retained in A[]
}
```

Merge two sorted arrays of integer

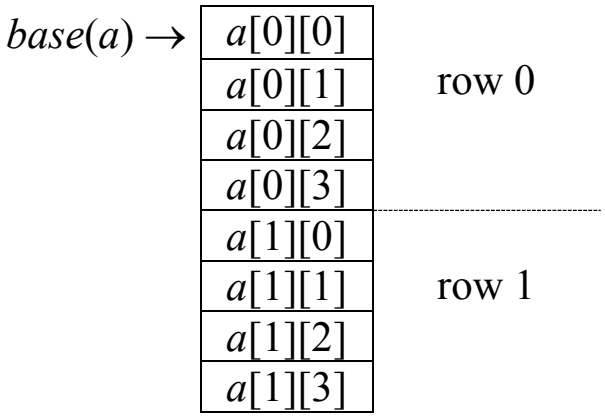
```
//Precondition: A[] and B[] are sorted in ascending order

void merge(const int A[], int na, const int B[], int nb,
           int C[], int& nc) // nc is passed by reference
{
   // array C[] has been created by the calling function

   int i, j, k;

   i = j = k = 0;
   while (i < na && j < nb)
   {
      //assert: C[0..k-1] is sorted in ascending order
      if (A[i] <= B[j])
      {
          C[k] = A[i];  // the 3 statements can be replaced
          k++;          // by C[k++] = A[i++];
          i++;
      }
      else
         C[k++] = B[j++];
   }

   //copy the remaining elements in A[] or B[] to C[]
   while (i < na)
      C[k++] = A[i++];

   while (j < nb)
      C[k++] = B[j++];

   nc = k;
}
```

2D Arrays

```
#define Rows 2
#define Cols 4
int a[Rows][Cols];  // an array with 2 rows & 4 columns
```

Multi-dimensional arrays are mapped to the linear address space of the computer system.

In C/C++, elements of a multi-dimensional array are arranged in row-major order.



Array mapping function:

i = row index
j = column index
Cols = number of columns in a row
esize = size of an element (no. of bytes)

base(*a*) = address of *a*[0][0]

address of *a*[i][j] = *base*(*a*) + (i × Cols + j) × *esize*

number of elements placed in front of *a*[i][j] = i × Cols + j

Example: Matrix multiplication

```
#define N 100
typedef int Matrix[N][N];  // fixed-size 100x100 matrix

// compute C = A × B
void matrixMul(Matrix A, Matrix B, Matrix C)
{
   for (int i = 0; i < N; i++)
   {
      for (int j = 0; j < N; j++)
      {
         C[i][j] = 0;

         for (int k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
      }
   }
}
```

Remark:
The above function can only operate on matrix of a given size !!

In general, we would like to implement C-functions that can operate on matrixes with different sizes.

To do that, we need another representation of 2D array.

Array of pointers vs 2D-array

```
#define Rows 10
#define Cols 20
int a[Rows][Cols]; // a is a 2D array with 10×20 elements
int *b[Rows];      // b is an array of 10 integer pointers

int i, j, k;

for (i = 0; i < Rows; i++) // create 10 linear arrays
   b[i] = new int[Cols];

k = 1;

//syntactically, you can access elements in b as b[i][j]
for (i = 0; i < Rows; i++)
   for (j = 0; j < Cols; j++)
   {
      b[i][j] = k++;  // b[0][0] = 1;
                      // b[0][1] = 2; and so on

      a[i][j] = b[i][j];
   }
```

Remark: Syntactically a[][] and b[][] appear to be the same, their physical representations are different.

Some programmers may instead define logical 2D array as

```
int **b;     // int *b[]
char **name; // char *name[]
```

This form is more convenient when you need to dynamically create (logical) 2D arrays of variable sizes, or array of variable-length character strings.

Example:

```
char *month[] = {"Illegal month", "January", "February",
                 "March", "April", "May", "June", "July",
                 "August", "September", "October",
                 "November", "December"};
```

Example:
Function to compute 2D matrix multiplication for matrixes of variable size.

```
// Compute C[n][n] = A[n][n] × B[n][n]
// Precondition: memory space for C[][] has been created by
//               the calling function.

void matrixMul_2(int **A, int **B, int **C, int n)
{
   for (int i = 0; i < n; i++)
   {
      for (int j = 0; j < n; j++)
      {
         C[i][j] = 0;

         for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
      }
   }
}
```

Pseudo Code

We need a language to express program development

- English is too verbose and imprecise.
- The target language, e.g. C/C++, requires too much details.

Pseudo code resembles the target language in that

- it is a sequence of steps (each step is precise and unambiguous)
- it has similar control structure of C/C++

Pseudo code

```
x = max{a, b, c}
```

C code

```
x = a;
if (b > x)    x = b;
if (c > x)    x = c;
```

Pseudo code allows the programmer to concentrate on problem solving instead of code generation.

Example: Saddle point in 2-D matrix

An *m*×*n* matrix is said to have a saddle point if some entry A[i][j] is the smallest value on row *i* and the largest value in column *j*.

```
An 6×8 matrix with a saddle point
11 33 55 16 77 99 10 40
29 87 65 20 45 60 90 76
50 53 78 44 60 88 77 81
46 72 71 23 88 26 15 21
65 83 23 36 49 57 32 14
70 22 34 19 54 37 26 93
```

Problem:
Given an *m*×*n* matrix, determine if it contains any saddle points.

Solution expressed in pseudo code:

```
for each row (with row index = i)
{
   j = col index of the smallest element on row i;

   if (A[i][j] is the largest element in column j)
      A[i][j] is a saddle point;

}
```

```
Refined pseudo code

for (i = 0; i < m; i++) //for each row
{
   j = index of smallest element on row i;

   //assert: A[i][j] is the smallest element on row i

   for (k = 0; k < m; k++)  //for each element in column j
      if there does not exist A[k][j] > A[i][j]
         A[i][j] is a saddle point;

}
```

```
/* precondition: elements in a row are distinct.
   m = no. of rows
   n = no. of columns
   A[m][n] is a 2D array of integers
*/
void SaddlePoint(int **A, int m, int n)
{
   for (int i = 0; i < m; i++)
   {
      int j=0, k;
      for (k = 1; k < n; k++)
         if (A[i][k] < A[i][j])
            j = k;

      //assert: a[i][j] is the smallest element on row i.

      int isSP = 1;  // isSP : is saddle point

      for (k = 0; k < m && isSP; k++)
         if (A[i][j] < A[k][j]) //a[i][j] is not largest
            isSP = 0;           //element in col j

      if (isSP)
         cout << "saddle point found at row " << i
              << " , col " << j << endl;

   }
}
```

```
// Version 2: do not assume elements in a row are distinct

//determine if A[r][c] is the largest element in column c
int isLargestInCol(int **A, int m, int r, int c)
{
   //precondition: A[r][c] is a valid element
   for (int i = 0; i < m; i++)
      if (A[r][c] < A[i][c])
         return 0;

   return 1;  //A[r][c] is the largest number in col c
}


void SaddlePoint_2(int **A, int m, int n)
{
   for (int i = 0; i < m; i++)
   {
      int j=0;
      for (int k = 1; k < n; k++)
         if (A[i][k] < A[i][j])
            j = k;

      //assert: A[i][j] is the smallest element on row i
      //there may exist A[i][k] == A[i][j] where k > j

      for (int k = j; k < n; k++)
      {
         if (A[i][j] == A[i][k]
             && isLargestInCol(A, m, i, k))
            cout << "saddle point found at row " << i
                 << " , col " << k << endl;
      }
   }
}
```

Some suggestions for good programming style:

- format the source file with proper indentation of statements and align the braces so that the control structures can be read easily
- add a blank character before and after a binary operator

  Example:

```
// poorly formatted statement
if(p->sid>q->sid&&p->program!=q->program)

// properly formatted statement
if (p->sid > q->sid && p->program != q->program)
```

- use informative and meaningful variable and function names
- insert useful comments (i.e. assertions) in the source program
- avoid ambiguous statements

  e.g., `x[i] = i++;`
- minimize direct accesses to global variables, especially you should avoid modifying the values of global variables in a function
- outputs produced by a function should be passed as formal parameters, i.e. avoid side effects
- use single-entry single-exit control blocks, or at most one `break` statement inside a loop. Avoid using the `continue` statement.
- do not use goto statement, especially backward jump
- always make a planning of the program organization and data structures before start writing program codes
- should avoid using the trial-and-error approach without proper understanding of the problem to be solved

Example program with side effects

```
int x;  //global variable

int f(int n)
{
     x += 1;  //side effect: modify the value of x which is
              //not a formal parameter of function f()

     return n + x;
}

int g(int n)
{
     x *= 2; //side effect

     return n * x;
}

int main()
{
     int t;
     …

     t = f(1) + g(2);

     //Logically the same as t = g(2) + f(1);
     //but the results will be totally different.
}
```

Algorithm

An algorithm is a finite set of instructions which, if followed, accomplish a particular task.

Every algorithm must satisfy the following criteria:

(i) **input:** there are zero or more quantities which are externally supplied.

(ii) **output:** at least one quantity is produced.

(iii) **definiteness:** each step (instruction) must be clear and unambiguous.

(iv) **finiteness:** the algorithm will terminate after a finite number of steps.

(v) **effectiveness:** every step must be sufficiently basic that it can be carried out by a person using only pencil and paper. Each step must also be feasible.

(vi) **proof of correctness**

In addition to the above criteria, we also want to make our algorithm as efficient as possible.

General remarks on program (or algorithm) design

- Explicit (meanings are clear and easy to understand) is better than implicit (with hidden assumptions, or meanings not apparent).
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.
- If the implementation is hard to explain, it is a bad idea (even if the program is correct).
- If the implementation is easy to explain, it may be a good idea.
- Complicated program for solving a simple problem is generally not acceptable.

Some common errors of students:

- When you process an array, be careful about index out-of-bound error,

  i.e. `index < 0` or `index >= size_of_array` (logical size or physical size).

  Error for `index >= logical_size` of an array is more difficult to debug, because the error will only be observed at some later steps of the program.

- When using pointers, be careful about the null-pointer exception,

  i.e. you must make sure that the pointer variable has a valid value before using it to access the data item.

Example: How to divide (assume that the computer does not have the division operation).

Problem specification:
Given integers $M \geq 0$ and $N > 0$, find integers Q and R such that

$$M = QN + R \quad \text{and} \quad 0 \leq R < N$$

Q is the quotient and R is the remainder.

Solution strategy (trivial strategy):
Subtract N from M repeatedly until the subtraction would yield a negative result.

Structured algorithm:

```
/* precondition: given M >= 0 and N > 0  */
R = M;
Q = 0;
while (R >= N)
{
   Q = Q + 1;
   R = R - N;
}
/* postcondition: M = QN + R, 0 <= R < N  */
```

- Preconditions specify the domains of the algorithm.
- Postconditions specify the results of the algorithm.
- If the preconditions are satisfied, then the postconditions are guaranteed.

Semi-formal proof of correctness

1. Termination

If N > 0, the loop is guaranteed to terminate within finite time.

2. Correctness

(i) At termination

(a) when the while-loop terminates, R < N

(b) R = M and M >= 0 initially, thus R >= 0 initially

(c) R = R – N is "guarded" by the while condition, thus R cannot become negative.

(ii) M = QN + R is preserved

initially, Q = 0 and R = M

within the loop

Q = Q + 1
R = R – N

let Q' = Q + 1 and R' = R – N

M = QN + R
= (Q' – 1)N + (R' + N)
= Q'N + R'

if M = QN + R is true before the assignments, it will still be true after the assignments.

The equality "M = QN + R" is called the loop invariant.

Loop Design

```
X;
while (B)   /* I */
   Y;
```

`X` is a sequence of steps which initializes the loop.

`Y` is a sequence of steps which constitutes the body of the loop.

`!B` is the termination condition

`I` is the loop invariant.

1. Decide what the loop is supposed to accomplish, i.e. the postcondition. Express the postcondition in the form of `I && !B`.

2. Code `X` which establishes `I`.

3. Design code for `Y` which must maintain `I` and make progress towards the goal, i.e. to make `B` false.

Example: computation of the GCD of two positive integers

Given two positive integers M and N, and assume M >= N.

M = QN + R, where Q is an integer and 0 <= R < N

Suppose g is the greatest common divisor of M and N.

M/g = QN/g + R/g

Since M can be divided by g, and N can be divided by g,
R/g should also be an integer (that is R is also divisible by g).

Hence, if R > 0, the GCD of M and N is equal to the GCD of N and R.

```
// given M > 0 and N > 0
m = M;
n = N;

// I ≡ GCD of m and n is equal to GCD of M and N
while ((r = m % n) > 0)
{
   m = n;
   n = r;
}
// postcondition: r = 0 and n is the GCD of M and N
```

Example: exponentiation

Problem specification:
Given two integers M > 0 and N ≥ 0, we want to find $p = M^N$.

Navie implementation:

```
p = 1;
for (i = 1; i <= N; i++)
   p *= M;
```

This method does not work in some application, e.g. cryptography.
Raising a number to its power is useful in many cryptographic applications.
Given positive integers *M*, *N*, and *B*, we want to compute $z = M^N \bmod B$.

M is an 32-bit integer
N is an integer with 30 digits or more
B is an 32-bit integer

A practical solution method to compute exponentiation

Invariant:
Observe that if y is even, $x^y = (x^2)^{y/2}$.

```
/* given M > 0 and N >= 0 */
x = M;
y = N;
p = 1;
/*  I ≡ p·x^y = M^N  */
while (y > 0)
{
     while ((y % 2) == 0)  // y is even
     {
          x = x * x;
          y = y / 2;
     }
     y--;
     p = p * x;
}
/* y = 0 && p = M^N */
```

Space and time complexity

Given a program or an algorithm, we want to estimate how much time and memory space are required to run the program in terms of the size of the problem instance.

Big O-notation

We say that a function $f(n)$ is of the order of $g(n)$, $f(n) = O(g(n))$, then there exists two constants c and n_0 such that $f(n) \le c \times g(n)$ for all $n > n_0$

Examples:

$$2n^3 + 55n^2 - 18n = O(n^3)$$

$$a^n + n^k = O(a^n), \text{ for } a > 1$$

$$c = O(1)$$

If $f1(n) = O(g(n))$ and $f2(n) = O(g(n))$, then $f1(n)$ and $f2(n)$ belongs to the same complexity class. However, it does not imply that $f1(n) = f2(n)$.

Lower bound:

$$f(n) = \Omega(g(n)) \Rightarrow f(n) \ge c \times g(n) \text{ for } n > n_0$$

Exact bound:

$$f(n) = \Theta(g(n)) \Rightarrow c1 \times g(n) \le f(n) \le c2 \times g(n) \text{ for } n > n_0$$

Example: Matrix addition

The problem size is given by `n`, the dimension of the matrix.

```
1    void MatrixAdd(int **A, int **B, int **C, int n)
2    {
3
4       for (int i = 0; i < n; i++)
5          for (int j = 0; j < n; j++)
6             C[i][j] = A[i][j] + B[i][j];
7       return;
8    }
```

line	step count	frequency
1	0	0
2	0	0
3	0	0
4	1	n+1
5	1	n(n+1)
6	1	n^2
7	1	1
8	0	0

total = $2n^2 + 2n + 2$

= $O(n^2)$

Sequential search

```
// A[] is an integer array (no ordering is assumed)
// determine if x is contained in A[0..N-1]

int seqSearch(int A[], int N, int x)
{
  int loc = -1; // location of the element to be searched

  for (int i = 0; i < N && loc < 0; i++)
    if (A[i] == x)
      loc = i;

  return loc;

  // if returned value is < 0, then x is not found
  // otherwise, x == A[loc]
}
```

Time complexity of sequential search is $O(N)$, where N is the length of the array.

Binary search

```
// A[] is an integer array sorted in ascending order,
// determine if x is contained in A[0..N-1]

int binSearch(int A[], int N, int x)
{
  int low = 0;
  int high = N-1;
  int loc = -1;

  // loop invariant:
  // if x is contained in A[], A[low] <= x <= A[high]
  while (low <= high && loc < 0)
  {
     mid = (low + high) / 2;

     if (A[mid] < x)
       low = mid+1;
     else if (A[mid] > x)
       high = mid-1;
     else
       loc = mid; // x == A[mid]
  }

  return loc;
}
// return a negative index : x is not found
// return a non-negative index : x is found at the given
//                               index location
```

Number of loop executions

≤ number of times N can be divided in half with a result ≥ ½

= number of times to double 1 to reach a value > N

= k where $2^{k-1} \le N < 2^k$

= $\lceil \log_2(N+1) \rceil$

= $O(\log_2 N)$

Time complexity of binary search is $O(\log N)$.

Some important complexity classes

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
0	1	0	1	1	2	1
1	2	2	4	8	4	2
2	4	8	16	64	16	24
3	8	24	64	512	256	40320
4	16	64	256	4096	65536	2.09×10^{13}
5	32	160	1024	32768	4294967296	2.6×10^{35}
6	64	384	4096	262144	1.84×10^{19}	1.27×10^{89}

Complexity class		Example applications
constant time	$O(1)$	random number generation, hashing
logarithmic	$O(\log n)$	binary search
linear	$O(n)$	sum of a list, sequential search, vector product
log-linear	$O(n \log n)$	fast sorting algorithms
quadratic	$O(n^2)$	2D matrix addition, insertion sort, bubble sort
cubic	$O(n^3)$	2D matrix multiplication
exponential	$O(a^n)$, $a > 1$	traveling salesman, placement and routing in VLSI, many other optimization problems
factorial	$O(n!)$	enumerate the permutations of n objects

Practical problem sizes that can be handled by algorithms of different complexity classes

Complexity function	Maximum n
1	Unlimited
$\log n$	Effectively unlimited
n	$n < 10^{10}$
$n \log n$	$n < 10^8$
n^2	$n < 10^5$
n^3	$n < 10^3$
2^n	$n < 36$
$n!$	$n < 15$