Sejarah C++

Tahun 1978, Brian W. Kerninghan & Dennis M. Ritchie dari AT & T Laboratories mengembangkan bahasa B menjadi bahasa C. Bahasa B yang diciptakan oleh Ken Thompson sebenarnya merupakan pengembangan dari bahasa BCPL (Basic Combined Programming Language) yang diciptakan oleh Martin Richard.

Sejak tahun 1980, bahasa C banyak digunakan pemrogram di Eropa yang sebelumnya menggunakan bahasa B dan BCPL. Dalam perkembangannya, bahasa C menjadi bahasa paling populer diantara bahasa lainnya, seperti PASCAL, BASIC, FORTRAN.

Tahun 1989, dunia pemrograman C mengalami peristiwa penting dengan dikeluarkannya standar bahasa C oleh *American National Standards Institute* (ANSI). Bahasa C yang diciptakan Kerninghan & Ritchie kemudian dikenal dengan nama ANSI C.

Mulai awal tahun 1980, Bjarne Stroustrup dari AT & T Bell Laboratories mulai mengembangkan bahasa C. Pada tahun 1985, lahirlah secara resmi bahasa baru hasil pengembangan C yang dikenal dengan nama C++. Sebenarnya bahasa C++ mengalami dua tahap evolusi. C++ yang pertama, dirilis oleh AT&T Laboratories, dinamakan **cfront**. C++ versi kuno ini hanya berupa kompiler yang menterjemahkan C++ menjadi bahasa C.

Pada evolusi selanjutnya, Borland International Inc. mengembangkan kompiler C++ menjadi sebuah kompiler yang mampu mengubah C++ langsung menjadi bahasa mesin (assembly). Sejak evolusi ini, mulai tahun 1990 C++ menjadi bahasa berorientasi obyek yang digunakan oleh sebagian besar pemrogram professional.

Struktur Bahasa C++

Contoh 1 :

```
// my first program in C++
#include <iostream.h>
int main ()
{
  cout << "Hello World!";
  return 0;
}
```

Hasil :

```
Hello World!
```

Sisi kiri merupakan *source code*, yang dapat diberi nama *hiworld*.cpp dan sisi kanan adalah hasilnya setelah di-kompile dan di-eksekusi.

Program diatas merupakan salah satu program paling sederhana dalam C++, tetapi dalam program tersebut mengandung komponen dasar yang selalu ada pada setiap pemrograman C++. Jika dilihat satu persatu :

// my first program in C++

Baris ini adalah komentar. semua baris yang diawali dengan dua garis miring (**//**) akan dianggap sebagai komentar dan tidak akan berpengaruh terhadap program. Dapat digunakan oleh programmer untuk menyertakan penjelasan singkat atau observasi yang terkait dengan program tersebut.

#include <iostream.h>

Kalimat yang diawali dengan tanda (#) adalah are *preprocessor directive*. Bukan merupakan baris kode yang dieksekusi, tetapi indikasi untuk kompiler. Dalam kasus ini kalimat **#include <iostream.h>** memberitahukan preprocessor kompiler untuk menyertakan header file standard **iostream**. File spesifik ini juga termasuk library deklarasi standard I/O pada C++ dan file ini disertakan karena fungsi-fungsinya akan digunakan nanti dalam program.

int main ()

Baris ini mencocokan pada awal dari deklarasi fungsi **main**. fungsi **main** merupakan titik awal dimana seluruh program C++ akan mulai dieksekusi. Diletakan diawal, ditengah atau diakhir program, isi dari fungsi main akan selalu dieksekusi pertama kali. Pada dasarnya, seluruh program C++ memiliki fungsi **main**.

main diikuti oleh sepasang tanda kurung **()** karena merupakan fungsi. pada C++, semua fungsi diikuti oleh sepasang tanda kurung **()** dimana, dapat berisi argumen didalamnya. Isi dari fungsi **main** selanjutnya akan mengikuti,berupa deklarasi formal dan dituliskan diantara kurung kurawal (**{}**), seperti dalam contoh.

cout << "Hello World";

Intruksi ini merupakan hal yang paling penting dalam program contoh. **cout** merupakan standard output stream dalam C++ (biasanya monitor). **cout** dideklarasikan dalam header file iostream.h, sehingga agar dapat digunakan maka file ini harus disertakan.

Perhatikan setiap kalimat diakhiri dengan tanda semicolon (;). Karakter ini menandakan akhir dari instruksi dan harus disertakan pada setiap akhir instruksi pada program C++ manapun.

return 0;

Intruksi **return** menyebabkan fungsi **main()** berakhir dan mengembalikan kode yang mengikuti instruksi tersebut, dalam kasus ini **0**. Ini merupakan cara yang paling sering digunakan untuk mengakhiri program.

Tidak semua baris pada program ini melakukan aksi. Ada baris yang hanya berisi komentar (diawali //), baris yang berisi instruksi untuk preprocessor kompiler (Yang diawali **#**),kemudian baris yang merupakan inisialisasi sebuah fungsi (dalam kasus ini, fungsi **main**) dan baris yang berisi instruksi (seperti, **cout <<**), baris yang terakhir ini disertakan dalam blok yang dibatasi oleh kurung kurawal (**{}**) dari fungsi **main**.

Struktur program dapat dituliskan dalam bentuk yang lain agar lebih mudah dibaca, contoh :

```
int main ()
{
  cout << " Hello World ";
  return 0;
}
```

Atau dapat juga dituliskan :

```
int main () { cout << " Hello World "; return 0; }
```

Dalam satu baris dan memiliki arti yang sama dengan program-program sebelumnya. pada C++ pembatas antar instruksi ditandai dengan semicolon (;) pada setiap akhir instruksi.

Contoh 2 :

```
// my second program in C++

#include <iostream.h>

int main ()
{
  cout << "Hello World! ";
  cout << "I'm a C++ program";
  return 0;
}
```

Hasil :

Hello World! I'm a C++ program

Komentar

Komentar adalah bagian dari program yang diabaikan oleh kompiler. Tidak melaksanakan aksi apapun. Mereka berguna untuk memungkinkan para programmer untuk memasukan catatan atau deskripsi tambahan mengenai program tersebut. C++ memiliki dua cara untuk menuliskan komentar :

```
//      Komentar baris
/*      Komentar Blok  */
```

Komentar baris, akan mengabaikan apapun mulai dari tanda (//) sampai akhir dari baris yang sama. Komentar Blok, akan mengabaikan apapun yang berada diantara tanda /* dan */.

Variabel, tipe data, konstanta

Untuk dapat menulis program yang dapat membantu menjalankan tugas-tugas kita, kita harus mengenal konsep dari **variabel**. Sebagai ilustrasi, ingat 2 buah angka, angka pertama adalah 5 dan angka kedua adalah 2. Selanjutnya tambahkan 1 pada angka pertama kemudian hasilnya dikurangi angka kedua (dimana hasil akhirnya adalah 4).

Seluruh proses ini dapat diekspresikan dalam C++ dengan serangkaian instruksi sbb :

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

Jelas ini merupakan satu contoh yang sangat sederhana karena kita hanya menggunakan 2 nilai integer yang kecil, tetapi komputer dapat menyimpan jutaan angka dalam waktu yang bersamaan dan dapat melakukan operasi matematika yang rumit.

Karena itu, kita dapat mendefinisikan variable sebagai bagian dari memory untuk menyimpan nilai yang telah ditentukan. Setiap variable memerlukan **identifier** yang dapat membedakannya dari variable yang lain, sebagai contoh dari kode diatas identifier variabelnya adalah **a**, **b** dan **result**, tetapi kita dapat membuat nama untuk variabel selama masih merupakan identifier yang benar.

Identifiers

Identifier adalah untaian satu atau lebih huruf, angka, atau garis bawah (_). Panjang dari identifier, tidak terbatas, walaupun untuk beberapa kompiler hanya 32 karakter pertama saja yang dibaca sebagai identifier (sisanya diabaikan). Identifier harus selalu diawali dengan huruf atau garis bawah (_).

Ketentuan lainnya yang harus diperhatikan dalam menentukan identifier adalah tidak boleh menggunakan **key word** dari bahasa C++. Diawah ini adalah **key** word dalam C++ :

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

Sebagai tambahan, represetasi alternatif dari operator, tidak dapat digunakan sebagai identifier. Contoh :

```
and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq,
xor, xor_eq
```

catatan: Bahasa C++ adalah bahasa yang "case sensitive", ini berarti identifier yang dituliskan dengan huruf kapital akan dianggap berbeda dengan identifier yang sama tetapi dituliskan dengan huruf kecil, sabagai contoh : variabel **RESULT** tidak sama dengan variable **result** ataupun variabel **Result**.

Tipe Data

Tipe data yang ada pada C++, berikut nilai kisaran yang dapat direpresentasikan :

DATA TYPES

Name	Bytes*	Description	Range*
char	1	character or integer 8 bits length.	**signed:** -128 to 127 **unsigned:** 0 to 255
short	2	integer 16 bits length.	**signed:** -32768 to 32767 **unsigned:** 0 to 65535
long	4	integer 32 bits length.	**signed:**-2147483648 to 2147483647 **unsigned:** 0 to 4294967295
int	*	Integer. Its length traditionally depends on the length of the system's Word **type**, thus in MSDOS it is 16 bits long, whereas in 32 bit systems (like Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes).	See **short**, **long**
float	4	floating point number.	3.4e + / - 38 (7 digits)
double	8	double precision floating point number.	1.7e + / - 308 (15 digits)
long double	10	long double precision floating point number.	1.2e + / - 4932 (19 digits)
bool	1	Boolean value. It can take one of two values: true or false NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it. Consult section bool type for compatibility information.	**true** or **false**
wchar_t	2	Wide character. It is designed as	wide characters

		a type to store international characters of a two-byte character set. NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it.	

Deklarasi variabel

Untuk menggunakan variabel pada C++, kita harus mendeklarasikan tipe data yang akan digunakan. Sintaks penulisan deklarasi variabel adalah dengan menuliskan tipe data yang akan digunakan diikuti dengan identifier yang benar, contoh :

```
int a;
float mynumber;
```

Jika akan menggunakan tipe data yang sama untuk beberapa identifier maka dapata dituliskan dengan menggunakan tanda koma, contoh :

```
int a, b, c;
```

Tipe data integer (**char**, **short**, **long** dan **int**) dapat berupa signed atau unsigned tergantung dari kisaran nilai yang akan direpresentasikan. Dilakukan dengan menyertakan keyword **signed** atau **unsigned** sebelum tipe data, contoh :

```
unsigned short NumberOfSons;
signed int MyAccountBalance;
```

Jika tidak dituliskan, maka akan dianggap sebagai **signed**.

Contoh 3 :

Hasil :

```
// operating with variables
#include <iostream.h>

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;
```

4

Inisialisasi Variabel

Ketika mendeklarasikan variabel local, kita dapat memberikan nilai tertentu. Sintaks penulisan sbb :

type identifier **=** *initial_value* ;

Misalkan kita akan mendeklarasikan variabel **int** dengan nama **a** yang bernilai **0**, maka dapat dituliskan :

```
int a = 0;
```

Atau dengan cara lainnya, yaitu menyertakan nilai yang akan diberikan dalam tanda **()**:

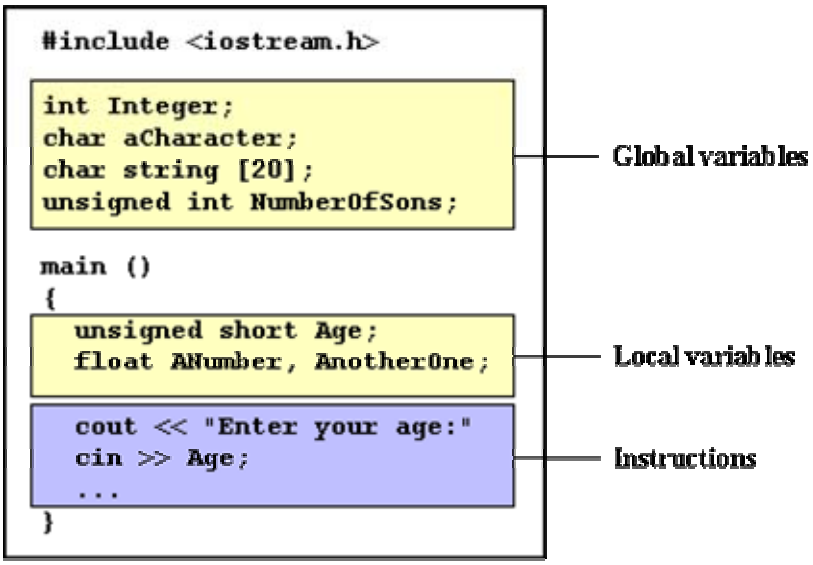
```
type identifier (initial_value) ;
```

Contoh :

```
int a (0);
```

Lingkup Variabel

Pada C++, kita dapat mendeklarasikan variable dibagian mana saja dari program, bahkan diantara 2 kalimat perintah.



variabel Global dapat digunakan untuk setiap bagian dari program, maupun fungsi, walaupun dideklarasikan diakhir program.

Lingkup dari **variable local** terbatas. Hanya berlaku dimana variable tersebut dideklarasikan. Jika dideklarasikan diawal fungsi (seperti dalam **main**) maka lingkup dari variable tersebut adalah untuk seluruh fungsi **main**. Seperti contoh diatas, jika terdapat fungsi lain yang ditambahkan pada main(), maka variable local yang dideklarasikan dalam **main** tidak dapat digunakan pada fungsi lainnya dan sebaliknya.

Pada C++, lingkup variable local ditandai dengan blok dimana variable tersebut dideklarasikan (blok tersebut adalah sekumpulan instruksi dalam kurung kurawal {}). Jika dideklarasikan dalam fungsi tersebut, maka akan berlaku sebagai variable dalam fungsi tersebut, jika dideklarasikan dalam sebuah perulangan, maka hanya berlaku dalam perulangan tersebut, dan seterusnya.

Konstanta : Literals.

Konstanta adalah ekspresi dengan nilai yang tetap. Terbagi dalam Nilai Integer, Nilai Floating-Point, Karakter and String.

Nilai Integer

Merupakan nilai konstanta numerik yang meng-identifikasikan nilai integer decimal. Karena merupakan nilai numeric, maka tidak memerlukan tanda kutip (") maupun karakter khusus lainnya. Contoh :

```
1776
707
-273
```

C++ memungkinkan kita untuk mempergunakan nilai oktal (base 8) dan heksadesimal (base 16). Jika menggunakan octal maka harus diawali dengan karakter **0** (karakter nol), dan untuk heksadesimal diawali dengan karakter **0x** (nol, x). Contoh :

```
75          // decimal
0113        // octal
0x4b        // hexadecimal
```

Dari contoh diatas, seluruhnya merepresentasikan nilai yang sama : 75.

Nilai Floating Point

Merepresentasikan nilai desimal dan/atau eksponen, termasuk titik desimal dan karakter **e** (Yang merepresentasikan "dikali 10 pangkat n" , dimana n merupakan nilai integer) atau keduanya. Contoh :

```
3.14159     // 3.14159
6.02e23     // 6.02 x 10^23
1.6e-19     // 1.6 x 10^-19
3.0       // 3.0
```

Karakter dan String

Merupakan konstanta non-numerik, Contoh :

```
'z'
'p'
"Hello world"
"How do you do?"
```

Untuk karakter tunggal dituliskan diantara kutip tunggal (') dan untuk untaian beberapa karakter, dituliskan diantara kutip ganda (").

Konstanta karakter dan string memiliki beberapa hal khusus, seperti **escape codes**.

\n	newline
\r	carriage return
\t	tabulation
\v	vertical tabulation
\b	backspace
\f	page feed

\a	alert (beep)
\'	single quotes (')
\"	double quotes (")
\?	question (?)
\\	inverted slash (\)

Contoh :

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Sebagai tambahan, kita dapat menuliskan karakter apapun dengan menuliskan yang diikuti dengan kode ASCII, mengekspresikan sebagai octal (contoh, **\23** atau **\40**) maupun heksadesimal (contoh, **\x20** atau **\x4A**).

Konstanta Define (*#define*)

Kita dapat mendefinisikan sendiri nama untuk konstanta yang akan kita pergunakan, dengan menggunakan preprocessor directive **#define**. Dengan format :

```
#define identifier value
```

Contoh :

```
#define PI 3.14159265
#define NEWLINE '\n'
#define WIDTH 100
```

Setelah didefinisikan seperti diatas, maka kita dapat menggunakannya pada seluruh program yang kita buat, contoh :

```
circle = 2 * PI * r;
cout << NEWLINE;
```

Pada dasarnya, yang dilakukan oleh kompiler ketika membaca **#define** adalah menggantikan literal yang ada (dalam contoh, **PI**, **NEWLINE** atau **WIDTH**) dengan nilai yang telah ditetapkan (**3.14159265**, **'\n'** dan **100**). #define bukan merupakan instruksi, oleh sebab itu tidak diakhiri dengan tanda semicolon (;).

Deklarasi Konstanta (const)

Dengan prefix **const** kita dapat mendeklarasikan konstanta dengan tipe yang spesifik seperti yang kita inginkan. contoh :

```
const int width = 100;
const char tab = '\t';
const zip = 12440;
```

Jika tipe data tidak disebutkan, maka kompiler akan meng-asumsikan sebagai **int**.

Operator

Operator-operator yang disediakan C++ berupa *keyword* atau karakter khusus. Operator-operator ini cukup penting untuk diketahui karena merupakan salah satu dasar bahasa C++.

Assignation (=).

Operator *assignation* digunakan untuk memberikan nilai ke suatu variable.

```
a = 5;
```

Memberikan nilai integer **5** ke variabel **a**. Sisi kiri dari operator disebut *lvalue* (left value) dan sisi kanan disebut *rvalue* (right value). *lvalue* harus selalu berupa variabeldan sisi kanan dapat berupa konstanta, variabel, hasil dari suatu operasi atau kombinasi dari semuanya.

Contoh :

```
int a, b;     // a:?  b:?
a = 10;       // a:10 b:?
b = 4;        // a:10 b:4
a = b;        // a:4 b:4
b = 7;        // a:4 b:7
```

Hasil dari contoh diatas, **a** bernilai **4** dan **b** bernilai **7**.

Contoh :

```
a = 2 + (b = 5);
```

equivalen dengan :

```
b = 5;
a = 2 + b;
```

Arithmetic operators (+, -, *, /, %)

+ addition
- subtraction
* multiplication
/ division
% module

Compound assignation operators
(+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=)

contoh :

value += increase; equivalen dengan **value = value + increase;**
a -= 5; equivalen dengan **a = a - 5;**
a /= b; equivalen dengan **a = a / b;**
price *= units + 1; equivalen dengan **price = price * (units + 1);**

Increase (++) and decrease (--).

Contoh :

```
a++;
a+=1;
a=a+1;
```

Contoh diatas adalah equivalen secara fungsional. Nilai a dikurangi 1.

Operator Increase dan Decrease dapat digunakan sebagai *prefix* atau *suffix*. Dengan kata lain dapat dituliskan sebelum identifier variabel (**++a**) atau sesudahnya (**a++**). operator increase yang digunakan sebagai *prefix* (**++a**), Perbedaannya terlihat pada tabel dibawah ini :

Example 1

```
B=3;
A=++B;
// A is 4, B is 4
```

Example 2

```
B=3;
A=B++;
// A is 3, B is 4
```

Pada contoh 1, **B** ditambahkan sebelum nilainya diberikan ke **A**. Sedangkan contoh 2, Nilai **B** diberikan terlebih dahulu ke **A** dan **B** ditambahkan kemudian.

Relational operators (==, !=, >, <, >=, <=)

Untuk mengevaluasi antara 2 ekspresi, dapat digunakan operator Relasional. Hasil dari operator ini adalah nilai **bool** yaitu hanya berupa **true** atau **false**, atau dapat juga dalam nilai **int**, **0** untuk mereprensentasikan **"false"** dan **1** untuk merepresentasikan **"true"**. Operator-operator relasional pada C++ :

== Equal
!= Different
> Greater than
< Less than
>= Greater or equal than
<= Less or equal than

Contoh :

`(7 == 5)` would return **false**.
`(5 > 4)` would return **true**.
`(3 != 2)` would return **true**.
`(6 >= 6)` would return **true**.
`(5 < 5)` would return **false**.

Contoh, misalkan **a=2**, **b=3** dan **c=6** :

`(a == 5)` would return **false**.
`(a*b >= c)` would return **true** since `(2*3 >= 6)` is it.
`(b+4 > a*c)` would return **false** since `(3+4 > 2*6)` is it.
`((b=2) == a)` would return **true**.

Logic operators (!, &&, ||).

Operator ! equivalen dengan operasi boolean NOT, hanya mempunyai 1 operand, berguna untuk membalikkan nilai dari operand yang bersangkutan. Contoh :

`!(5 == 5)` returns **false** because the expression at its right `(5 == 5)` would be **true**.
`!(6 <= 4)` returns **true** because `(6 <= 4)` would be **false**.
`!true` returns **false**.
`!false` returns **true**.

operator Logika **&&** dan **||** digunakan untuk mengevaluasi 2 ekspresi dan menghasilkan 1 nilai akhir. mempunyai arti yang sama dengan operator logika Boolean *AND* dan *OR*. Contoh :

First Operand **a**	Second Operand **b**	result **a && b**	result **a || b**
true	true	**true**	**true**
true	false	**false**	**true**
false	true	**false**	**true**
false	false	**false**	**false**

Contoh :

`( (5 == 5) && (3 > 6) )` returns **false** (*true && false*).
`( (5 == 5) || (3 > 6))` returns **true** (*true || false*).

Conditional operator (?).

operator kondisional mengevaluasi ekspresi dan memberikan hasil tergantung dari hasil evaluasi (*true* atau *false*). Sintaks :

```
condition ? result1 : result2
```

Jika kondisi **true** maka akan menghasilkan *result1*, jika tidak akan menghasilkan *result2*.

`7==5 ? 4 : 3`	returns **3** since **7** is not equal to **5**.
`7==5+2 ? 4 : 3`	returns **4** since **7** is equal to **5+2**.
`5>3 ? a : b`	returns **a**, since **5** is greater than **3**.
`a>b ? a : b`	returns the greater one, **a** or **b**.

Bitwise Operators (&, |, ^, ~, <<, >>).

Operator Bitwise memodifikasi variabel menurut bit yang merepresentasikan nilai yang disimpan, atau dengan kata lain dalam representasi binary.

op	asm	Description
&	**AND**	Logical AND
|	**OR**	Logical OR
^	**XOR**	Logical exclusive OR
~	**NOT**	Complement to one (bit inversion)
<<	**SHL**	Shift Left
>>	**SHR**	Shift Right

Explicit type casting operators

Type casting operators memungkinkan untuk mengkonversikan tipe data yang sudah diberikan ke tipe data yang lain. Ada beberapa cara yang dapat dilakukan dalam C++, yang paling popular yaitu tipe baru dituliskan dalam tanda kurung **()** contoh:

```
int i;
float f = 3.14;
i = (int) f;
```

Contoh diatas, mengkonversikan nilai **3.14** menjadi nilai integer (**3**). Type casting operator yang digunakan **(int)**. Cara lainnya :

```
i = int ( f );
```

sizeof()

Operator ini menerma 1 parameter, dapat berupa type variabel atau variabel itu sendiri dan mengembalikan ukurannya type atau object tersebut dalam bytes :

```
a = sizeof (char);
```

Contoh diatas akan memberikan nilai 1ke **a** karena **char** adalah tipe data dengan panjang 1 byte. Nilai yang diberikan oleh **sizeof** bersifat konstsn constant.

Prioritas pada operator

Contoh :

```
a = 5 + 7 % 2
```

Jawaban dari contoh diatas adalah 6. Dibawah ini adalah prioritas operator dari tinggi ke rendah :

Priority	Operator	Description	Associativity
1	::	scope	Left
2	() [ ] -> . sizeof		Left
3	++ --	increment/decrement	Right
	~	Complement to one (bitwise)	
	!	unary NOT	
	& *	Reference and Dereference (pointers)	
	(*type*)	Type casting	
	+ -	Unary less sign	
4	* / %	arithmetical operations	Left
5	+ -	arithmetical operations	Left
6	<< >>	bit shifting (bitwise)	Left
7	< <= > >=	Relational operators	Left
8	== !=	Relational operators	Left
9	& ^ |	Bitwise operators	Left
10	&& ||	Logic operators	Left
11	?:	Conditional	Right
12	= += -= *= /= %= >>= <<= &= ^= |=	Assignation	Right
13	,	Comma, Separator	Left

Komunikasi melalui *console*

Console merupakan interface dasar pada computers, biasanya berupa keyboard dan monitor. Keyboard merupakan alat *input* standar dan monitor adalah alat *output* standar. Dalam library *iostream* C++ , standard operasi *input* dan *output* untuk pemrograman didukung oleh 2 data streams: **cin** untuk input dan **cout** untuk output. Juga, **cerr** dan **clog** sebagai tambahan untuk output streams yang di desain khusus untuk menampilkan *error messages*. Dapat diarahkan langsung ke standard output maupun ke log file.

Biasanya **cout** (standard output stream) ditujukan untuk monitor dan **cin** (standard input stream) ditujukan untuk keyboard. Dengan menggunakan dua streams ini, maka kita dapat berinteraksi dengan user dengan menampilkan messages pada monitor dan menerima input dari keyboard.

Output (cout)

Penggunaan **cout** stream dhubungkan dengan operator overloaded << (Sepasang tanda *"less than"*). Contoh :

```
cout << "Output sentence"; // prints Output sentence on
screen
cout << 120;               // prints number 120 on screen
cout << x;                 // prints the content of variable
x on screen
```

Operator << dikenal sebagai *insertion operator,* dimana berfungsi untuk menginput data yang mengikutinya. Jika berupa string, maka harus diapit dengan kutip ganda ("), sehingga membedakannya dari variable. Contoh :

```
cout << "Hello";      // prints Hello on screen
cout << Hello;        // prints the content of Hello
variable on screen
```

Operator *insertion* (<<) dapat digunakan lebih dari 1 kali dalam kalimat yang sama, Contoh :

```
cout << "Hello, " << "I am " << "a C++ sentence";
```

Contoh diatas akan menampilkan **Hello, I am a C++ sentence** pada layar monitor. Manfaat dari pengulangan penggunaan operator insertion (<<) adalah untuk menampilkan kombinasi dari satu variabel dan konstanta atau lebih, contoh :

```
cout << "Hello, I am " << age << " years old and my zipcode
is " << zipcode;
```

Misalkan variable age = 24 dan variable zipcode = 90064 maka output yang dihasilkan :

```
Hello, I am 24 years old and my zipcode is 90064
```

Contoh :

```
cout << "First sentence.\n ";
cout << "Second sentence.\nThird sentence.";
```

Output :

```
First sentence.
Second sentence.
Third sentence.
```

Selain dengan karakter new-line, dapat juga menggunakan manipulator **endl**, contoh :

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

Output :

```
First sentence.
Second sentence.
```

Input (cin).

Menangani standard input pada C++ dengan menambahkan overloaded operator *extraction* (>>) pada **cin** stream. Harus diikuti bengan variable yang akan menyimpan data. Contoh :

```
int age;
cin >> age;
```

Contoh diatas mendeklarasikan variabel age dengan tipe int dan menunggu input dari cin (keyborad) untuk disimpan di variabel age.

cin akan memproses input dari keyboard sekali saja dan tombol ENTER harus ditekan.

Contoh :

```
// i/o example
#include <iostream.h>

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
```

Output :
Please enter an integer value: 702
The value you entered is 702 and its double is 1404.

cin juga dapat digunakan untuk lebih dari satu input :

```
cin >> a >> b;
```

Equivalen dengan :

```
cin >> a;
cin >> b;
```

Dalam hal ini data yang di input harus 2, satu untuk variabel **a** dan lainnya untuk variabel **b** yang penulisannya dipisahkan dengan : spasi, tabular atau *newline*.

Struktur Kontrol

Sebuah program biasanya tidak terbatas hanya pada intruksi yang terurut saja, tetapi juga memungkinkan terjadinya percabangan, perulangan dan pengambilan keputusan. Untuk mengatasi kebutuhan itu C++ menyediakan struktur kontrol yang dapat menangani hal-hal tersebut.

Untuk membahas hal tersebut diatas, akan ditemui istilah ***block of instructions***. Blok instruksi adalah sekumpulan instruksi yang dibatasi dengan tanda semicolon (;) tetapi dikelompokan dalam satu blok yang dibatasi dengan kurung kurawal **{ }**.

Struktur Kondisional : *if* and *else*

Digunakan untuk mengeksekusi sebuah atau satu blok instruksi jika kondisi terpenuhi, sintaks:

```
if (condition) statement
```

condition merupakan ekspresi yang dievaluasi. Jika kondisi bernilai **true**, maka ***statement*** akan dijalankan. Jika **false**, maka ***statement*** akan diabaikan dan program menjalankan instruksi selanjutnya.

Contoh, Akan tercetak **x is 100** jika nilai yang disimpan pada variable **x** adalah 100:

```
if (x == 100)
  cout << "x is 100";
```

Jika ada lebih dari satu instruksi yang akan dijalankan maka harus dibuat dalam blok instruksi dengan menggunakan tanda kurung kurawal { }:

```
if (x == 100)
 {
  cout << "x is ";
  cout << x;
 }
```

Dapat juga menggunakan keyword *else,* jika kondisi tidak terpenuhi. Penulisannya digabungkan dengan **if** :

if (*condition***)** *statement1* **else** *statement2*

Contoh :

```
if (x == 100)
  cout << "x is 100";
else
  cout << "x is not 100";
```

Akan tercetak **x is 100** jika nilai x adalah 100, jika tidak akan tercetak **x is not 100**.

Contoh :

```
if (x > 0)
  cout << "x is positive";
else if (x < 0)
  cout << "x is negative";
else
  cout << "x is 0";
```

Struktur perulangan (loops)

Loops merupakan perulangan *statement* dengan jumlah tertentu jika kondisi terpenuhi.

The *while* loop.

Sintaks:

while (*expression***)** *statement*

Fungsi dari statement diatas adalah mengulang *statement* jika *expression* bernilai *true*.

Contoh :

```
// custom countdown using while
#include <iostream.h>
int main ()
{
  int n;
  cout << "Enter the starting number > ";
  cin >> n;
```

```
  while (n>0) {
    cout << n << ", ";
    --n;
  }
  cout << "FIRE!";
  return 0;
}
```

Output :
Enter the starting number > 8
8, 7, 6, 5, 4, 3, 2, 1, FIRE!

Algoritma program dimulai dari **main** :
1. User meng-input nilai untuk **n**.
2. Instrukti while mengevaluasi apakah **(n>0)**. Ada dua kemungkinan :
true: meng-eksekusi *statement* (step **3,**)
false: melompati *statement*. lanjut ke step **5.**.
3.Mengeksekusi *statement* : cout << n << ", ";
--n;
(Menampilkan **n** di layar dan mengurangi **n** dengan 1).
4. Akhir dari blok. kembali ke step **2.**
5. lanjut menuju program setelah blok. Cetak : **FIRE!** dan program berakhir.

The *do-while* loop.

Format:

do *statement* **while (***condition***);**

Secara fungsional, hampir sama dengan *while* loop, hanya saja *condition* dalam *do-while* dievaluasi setelah eksekusi *statement* , dengan kata lain, sedikitnya satu kali eksekusi *statement* walaupun kondisi tidak terpenuhi. Contoh :

```
// number echoer
#include <iostream.h>
int main ()
{
  unsigned long n;
  do {
    cout << "Enter number (0 to end): ";
    cin >> n;
    cout << "You entered: " << n << "\n";
  } while (n != 0);
  return 0;
}
```

Output :
Enter number (0 to end): 12345
You entered: 12345
Enter number (0 to end): 160277
You entered: 160277

```
Enter number (0 to end): 0
You entered: 0
```

The *for* loop.

Format :

```
for (initialization; condition; increase) statement;
```

Fungsinya akan mengulang *statement* jika *condition* bernilai benar. Sama seperti *while* loop., hanya saja **for** memungkinkan untuk memberikan instruksi *initialization* dan intruksi *increase,* sehingga dapat menampilkan loop dengan counter.

Algoritma perulangan *for* :

1. *initialization,* digunakan untuk memberikan nilai awal untuk variable counter. Dieksekusi hanya sekali.
2. *condition,* Dievaluasi, jika bernilai **true** maka loop berlanjut, sebaliknya loop berhenti dan *statement* diabaikan
3. *statement,* dieksekusi, bisa berupa instruksi tunggal maupun blok instruksi (dalam tanda **{ }**).
4. *increase,* dieksekusi kemudian algoritma kembali ke step 2.

Contoh :

```
// countdown using a for loop
#include <iostream.h>
int main ()
{
  for (int n=10; n>0; n--) {
    cout << n << ", ";
  }
  cout << "FIRE!";
  return 0;
}
```

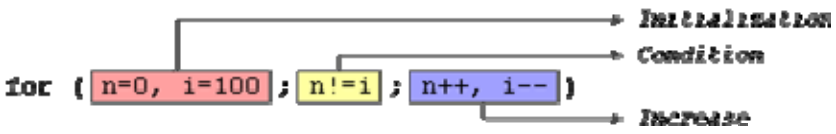
Output :

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

Initialization dan *increase* bersifat optional. Sehingga dapat dituliskan : **for (;n<10;)** untuk for tanpa *initialization* dan *increase*; atau **for (;n<10;n++)** untuk for dengan *increase* tetapi tanpa *initialization*. Dengan operator koma (,) kita dapat mendeklarasikan lebih dari satu instruksi pada bagian manapun termasuk dalam loop **for**, contoh :

```
for ( n=0, i=100 ; n!=i ; n++, i-- )
      {
        // whatever here...
      }
```

Loop diatas akan meng-eksekusi sebanyak 50 kali :



nilai awal **n** = **0** dan **i** = **100**, dengan kondisi **(n!=i)** (yaitu **n** tidak sama dengan **i**). Karena **n** mengalami penambahan 1 dan **i** mengalami pengurangan 1, maka kondisi loop akan salah setelah loop yang ke-50, yaitu ketika **n** dan **i** bernilai 50.

Kontrol Percabangan (Bifurcation) dan Lompatan (jumps)

Instruksi *break*

Dengan menggunakan instruksi *break*, program akan keluar dari loop walaupun kondisi untuk berakhirnya loop belum terpenuhi. Dapat digunakan untuk mengakhiri *infinite loop*, atau untuk menyebabkan loop selesai sebelum saatnya, contoh :

```
// break loop example
#include <iostream.h>
int main ()
{
  int n;
  for (n=10; n>0; n--) {
    cout << n << ", ";
    if (n==3)
    {
      cout << "countdown aborted!";
      break;
    }
  }
  return 0;
}
```

Output :

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

Instruksi *continue*

Instruksi *continue* menyebabkan program akan melewati instruksi selanjutnya hingga akhir blok dalam loop. Atau dengan kata lain langsung melompat ke iterasi selanjutny. Contoh berikut akan melewati angka 5 dalam hitungan mundur :

```
// break loop example
#include <iostream.h>
int main ()
{
  for (int n=10; n>0; n--) {
    if (n==5) continue;
```

```
    cout << n << ", ";
  }
  cout << "FIRE!";
  return 0;
}
```

Output :

10, 9, 8, 7, 6, 4, 3, 2, 1, FIRE!

Instruksi *goto*

Menyebabkan lompatan dalam program. Tujuan dari lompatan diidentifikasikan dengan label, yang berisikan argumen-argumen. penulisan label diikuti dengan tanda colon (:). Contoh :

```
// goto loop example
#include <iostream.h>
int main ()
{
  int n=10;
  loop:
  cout << n << ", ";
  n--;
  if (n>0) goto loop;
  cout << "FIRE!";
  return 0;
}
```

Output :

10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!

Struktur Seleksi : *switch*.

Instruksi *switch* digunakan untuk membandingkan beberapa nilai konstan yang mungkin untuk sebuah ekspresi, hampir sama dengan *if* dan *else if*. Bentuk umumnya :

```
switch (expression) {
  case constant1:
    block of instructions 1
    break;
  case constant2:
    block of instructions 2
    break;
  .
  .
  .
  default:
    default block of instructions
  }
```

switch meng-evaluasi *expression* dan memeriksa apakah equivalen dengan *constant1*, jika ya, maka akan meng-eksekusi *block of instructions 1* sampai terbaca keyword **break**, kemudian program akan lompat ke akhir dari stuktur selektif *switch*.

Jika *expression* tidak sama dengan *constant1*, maka akan diperiksa apakah *expression* equivalen dengan *constant2*. jika ya, maka akan dieksekusi *block of instructions 2* sampai terbaca **break**. Begitu seterusnya, jika tidak ada satupun konstanta yang sesuai maka akan mengeksekusi **default:**

contoh :

***switch* example**

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
 }
```

***if-else* equivalent**

```
if (x == 1) {
  cout << "x is 1";
  }
else if (x == 2) {
  cout << "x is 2";
  }
else {
  cout << "value of x unknown";
 }
```

Function

Function adalah satu blok instruksi yang dieksekusi ketika dipanggil dari bagian lain dalam suatu program. Format dari *function* :

type name (argument1, argument2, ...) statement

Dimana :

- ***type***, adalah tipe dari data yang akan dikembalikan/dihasilkan oleh *function*.
- ***name***, adalah nama yang memungkinkan kita memanggil function.
- ***arguments*** (dispesifikasikan sesuai kebutuhan). Setiap argumen terdiri dari tipe data diikuti identifier, seperti deklarasi variable (contoh, int x) dan berfungsi dalam function seperti variable lainnya. Juga dapat melakukan *passing parameters* ke function itu ketika dipanggil. Parameter yang berbeda dipisahkan dengan koma.
- ***statement***, merupakan bagian badan suatu *function*. Dapat berupa instruksi tunggal maupun satu blok instruksi yang dituliskan diantara kurung kurawal **{}**.

Contoh *function 1* :

```
// function example
#include <iostream.h>

int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;

}
```

Output :

```
The result is 8
```

Program diatas, ketika dieksekusi akan mulai dari fungsi **main**. **main** function memulai dengan deklarasi variabel **z** dengan tipe **int**. Setelah itu instruksi pemanggilan fungsi **addition**. Jika diperhatikan, ada kesamaan antara sruktur pemanggilan dengan deklarasi fungsi itu sendiri, perhatikan contoh dibawah ini :

```
int addition (int a, int b)
               ↑      ↑
z = addition (  5  ,   3  );
```

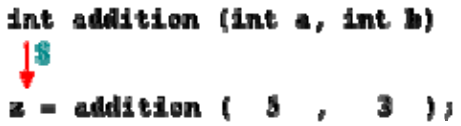
Instruksi pemanggilan dalam fungsi **main** untuk fungsi **addition**, memberikan 2 nilai : **5** dan **3** mengacu ke parameter **int a** dan **int b** yang dideklarasikan untuk fungsi **addition**.

Saat fungsi dipanggil dari **main**, kontrol program beralih dari fungsi **main** ke fungsi **addition**. Nilai dari kedua parameter yang diberikan (**5** dan **3**) di-*copy* ke variable local ; **int a** dan **int b**.

Fungsi **addition** mendeklarasikan variable baru (**int r;**), kemudian ekspresi **r=a+b;**, yang berarti **r** merupakan hasil penjumlahan dari **a** dan **b**, dimana **a** dan **b** bernilai **5** dan **3** sehingga hasil akhirnya **8**. perintah selanjutnya adalah :

```
return (r);
```

Merupakan akhir dari fungsi **addition**, dan mengembalikan kontrol pada fungsi **main**. Statement **return** dikuti dengan variabel **r** (**return (r);**), sehingga nilai dari **r** yaitu 8 akan dikembalikan :



Dengan kata lain pemanggilan fungsi (**addition (5,3)**) adalah menggantikan dengan nilai yang akan dikembalikan (**8**).

Contoh *function 2* :

```
// function example
#include <iostream.h>

int subtraction (int a, int b)
{
  int r;
  r=a-b;
  return (r);
}

int main ()
{
  int x=5, y=3, z;
  z = subtraction (7,2);
  cout << "The first result is " << z << '\n';
  cout << "The second result is " << subtraction (7,2) << '\n';
  cout << "The third result is " << subtraction (x,y) << '\n';
  z= 4 + subtraction (x,y);
  cout << "The fourth result is " << z << '\n';
  return 0;
}
```

Output :

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

Fungsi diatas melakukan pengurangan dan mengembalikan hasilnya. Jika diperhatikan dalam fungsi **main**, dapat dilihat beberapa cara pemanggilan fungsi yang berbeda.

Perhatikan penulisan pemanggilan *function,* format penulisan pada dasarnya sama.
Contoh 1 :

```
z = subtraction (7,2);
cout << "The first result is " << z;
```

Contoh 2 :

```
cout << "The second result is " << subtraction (7,2);
```

Contoh 3 :

```
cout << "The third result is " << subtraction (x,y);
```

Hal lain dari contoh diatas, parameter yang digunakan adalah variable, bukan konstanta. Contoh diatas memberikan nilai dari **x** dan **y**, yaitu **5** dan **3**, hasilnya **2**.

contoh 4 :

```
z = 4 + subtraction (x,y);
```

Atau dapat dituliskan :

```
z = subtraction (x,y) + 4;
```

Akan memberikan hasil akhir yang sama. Perhatikan, pada setiap akhir ekspresi selalu diberi tanda semicolon (;).

Function tanpa tipe (Kegunaan *void*)

Deklarasi fungsi akan selalu diawali dengan tipe dari fungsi, yang menyatakan tipe data apa yang akan dihasilkan dari fungsi tersebut. Jika tidak ada nilai yang akan dikembalikan, maka dapat digunakan tipe **void**, contoh :

```
// void function example
#include <iostream.h>

void dummyfunction (void)
{
  cout << "I'm a function!";
}

int main ()
{
  dummyfunction ();
  return 0;
}
```

Output :

```
I'm a function!
```

Walaupun pada C++ tidak diperlukan men-spesifikasikan **void**, hal itu digunakan untuk mengetahui bahwa fungsi tersebut tidak mempunyai argumen, atau parameter dan lainnya. Maka dari itu pemanggilan terhadap fungsinya dituliskan :

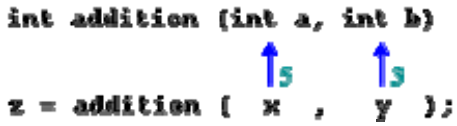
```
dummyfunction ();
```

Argument passed *by value* dan *by reference*.

Parameter yang diberikan ke fungsi masih merupakan *passed by value*. Berarti, ketika memanggil sebuah fungsi, yang diberikan ke fungsi adalah nilainya, tidak pernah men-spesifikasikan variablenya. Sebagai Contoh, pemanggilan fungsi **addition, menggunakan perintah berikut :**

```
int x=5, y=3, z;
z = addition ( x , y );
```

Yang berarti memanggil fungsi **addition** dengan memberikan nilai dari **x** dan **y**, yaitu **5** dan **3**, bukan variabelnya.



Tetapi, dapat juga memanipulasi dari dalam fungsi, nilai dari variable external. Untuk hal itu, digunakan argument *passed by reference*, Contoh :

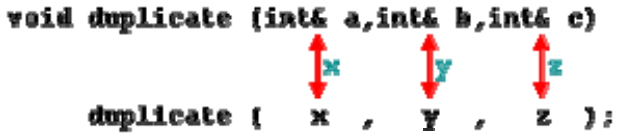
```
// passing parameters by reference
#include <iostream.h>

void duplicate (int& a, int& b, int& c)
{
  a*=2;
  b*=2;
  c*=2;
}

int main ()
{
  int x=1, y=3, z=7;
  duplicate (x, y, z);
  cout << "x=" << x << ", y=" << y << ", z=" << z;
  return 0;
}
```

Output :

x=2, y=6, z=14

Perhatikan deklarasi **duplicate,** tipe pada setiap argumen diakhiri dengan tanda *ampersand* (**&**), yang menandakan bahwa variable tersebut biasanya akan *passed by reference* dari pada *by value*.

Ketika mengirimkan variable *by reference,* yang dikirimkan adalah variabelnya dan perubahan apapun yang dilakukan dalam fungsi akan berpengaruh pada variable diluarnya.



Atau dengan kata lain, parameter yang telah ditetapkan adalah **a**, **b** dan **c** dan parameter yang digunakan saat pemanggilan adalah **x**, **y** dan **z**, maka perubahan pada **a** akan mempengaruhi nilai **x**, begitupun pada **b** akan mempengaruhi **y**, dan **c** mempengaruhi **z**.

Itu sebabnya mengapa hasil output dari program diatas adalah nilai variable dalam main dikalikan 2. jika deklarasi fungsi tidak diakhiri dengan tanda ampersand (&), maka variable tidak akan *passed by reference,* sehingga hasilnya akan tetap nilai dari **x**, **y** dan **z** tanpa mengalami perubahan.

Passing by reference merupakan cara efektif yang memungkinkan sebuah fungsi mengembalikan lebih dari satu nilai. Contoh, fungsi ini akan mengembalikan nilai sebelum dan sesudahnya dari nilai awal parameter :

```
// more than one returning value
#include <iostream.h>

void prevnext (int x, int& prev, int& next)
{
  prev = x-1;
  next = x+1;
}

int main ()
{
  int x=100, y, z;
  prevnext (x, y, z);
  cout << "Previous=" << y << ", Next=" << z;
  return 0;
}
```

Output :
Previous=99, Next=101

Nilai *Default* dalam argument

Ketika mendeklarasikan sebuah fungsi, dapat diberikan nilai *default* untuk setiap parameter. nilai ini akan digunakan ketika parameter pemanggil dikosongkan. Untuk itu cukup dideklarasikan pada saat deklarasi fungsi, Contoh :

```
// default values in functions
#include <iostream.h>

int divide (int a, int b=2)
{
  int r;
  r=a/b;
  return (r);
}

int main ()
{
  cout << divide (12);
  cout << endl;
  cout << divide (20,4);
  return 0;
}
```

Output :

```
6
5
```

Dapat dilihat dalam fungsi **divide**. Instruksi 1:

```
divide (12)
```

Instruksi 2 :

```
divide (20,4)
```

Fungsi Overloaded function

Dua fungsi yang berbeda dapat memiliki nama yang sama jika prototype dari argumen mereka berbeda, baik jumlah argumennya maupun tipe argumennya, Contoh :

```
// overloaded function
#include <iostream.h>

int divide (int a, int b)
{
  return (a/b);
}

float divide (float a, float b)
{
  return (a/b);
}

int main ()
{
  int x=5,y=2;
  float n=5.0,m=2.0;
  cout << divide (x,y);
  cout << "\n";
```

```
  cout << divide (n,m);
  cout << "\n";
  return 0;
}
```

Output :
2
2.5

Contoh diatas mempunyai nama fungsi yang sama, tetapi argumennya berbeda. Yang pertama bertipe **int** dan lainnya bertipe **float**. Kompiler mengetahuinya dengan memperhatikan tipe argumen pada saat pemanggilan fungsi.

inline Function

Directive inline dapat disertakan sebelum deklarasi fungsi, untuk menspesifikasikan bahwa fungsi tersebut harus di-compile sebagai suatu kode saat dipanggil. Sama halnya dengan deklarasi *macro*. Keuntungannya dapat terlihat pada fungsi sederhana yaitu hasil yang diberikan akan lebih cepat. (jika terjadi *stacking of arguments*) dapat dihindari. Format deklarasi :

```
inline type name ( arguments ... ) { instructions ... }
```

Pemanggilannya, sama dengan pemanggilan fungsi pada umumnya. Tidak diperlukan penulisan keyword *inline* pada setiap pemanggilan.

Recursivity Function

Rekursif merupakan kemempuan sebuah fungsi untuk memanggil dirinya sendiri. Sangat berguna untuk pengerjaan sorting atau perhitungan factorial. Contoh, format perhitungan factorial :

$$n! = n * (n-1) * (n-2) * (n-3) \ldots * 1$$

Misalkan, 5! (5 faktorial), akan menjadi :

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Contoh :

```
// factorial calculator
#include <iostream.h>

long factorial (long a)
{
  if (a > 1)
   return (a * factorial (a-1));
  else
   return (1);
}
```

```
int main ()
{
  long l;
  cout << "Type a number: ";
  cin >> l;
  cout << "!" << l << " = " << factorial (l);
  return 0;
}
```

Output :
Type a number: 9
!9 = 362880

Prototyping function.

Format :

type name (argument_type1, argument_type2, ...);

Hampir sama dengan deklarasi fungsi pada umumnya, **kecuali** :

- Tidak ada *statement* fungsi yang biasanya dituliskan dalam kurung kurawal { }.
- Diakhiri dengan tanda semicolon (;).
- Dalam argumen dituliskan tipe argumen, bersifat optional.

Contoh:

```
// prototyping
#include <iostream.h>

void odd (int a);
void even (int a);

int main ()
{
  int i;
  do {
    cout << "Type a number: (0 to exit)";
    cin >> i;
    odd (i);
  } while (i!=0);
  return 0;
}

void odd (int a)
{
  if ((a%2)!=0) cout << "Number is odd.\n";
  else even (a);
}

void even (int a)
{
  if ((a%2)==0) cout << "Number is even.\n";
  else odd (a);
}
```

Output :

```
Type a number (0 to exit): 9
Number is odd.
Type a number (0 to exit): 6
Number is even.
Type a number (0 to exit): 1030
Number is even.
Type a number (0 to exit): 0
Number is even.
```

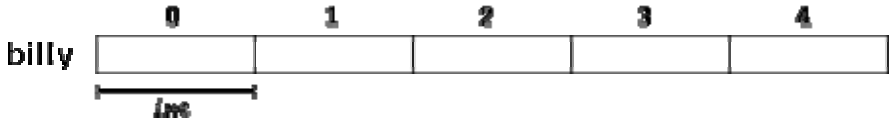
Contoh diatas tidak menjelaskan tentang efektivitas program tetapi bagaimana prototyping dilaksanakan. Perhatikan prototype dari fungsi **odd** dan **even**:

```
void odd (int a);
void even (int a);
```

Memungkinkan fungsi ini dipergunakan sebelum didefinisikan. Hal lainnya mengapa program diatas harus memiliki sedikitnya 1 fungsi prototype, karena fungsi dalam **odd** terdapat pemanggilan fungsi **even** dan dalam **even** terdapat pemanggilan fungsi **odd**. Jika tidak satupun dari fungsi tersebut dideklarasikan sebelumnya, maka akan terjadi error.

Arrays

Array adalah himpunan elemen (variable) dengan tipe yang sama dan disimpan secara berurutan dalam memory yang ditandai dengan memberikan index pada suatu nama variable. Contohnya, kita dapat menyimpan 5 nilai dengan tipe **int** tanpa harus mendeklarasikan 5 identifier variabel yang berbeda. Perhatikan contoh dibawah ini :



Bagian kosong diatas merepresentasikan *elemen* array, dalam kasus ini adalah nilai integer. Angka **0** - **4** merupakan index dan selalu dimulai dari **0**. Seperti penggunaan variable pada umumnya, array harus dideklarasikan terlebih dahulu, dengan format sbb :

```
type name [elements];
```

Maka contoh array diatas dideklarasikan sbb :

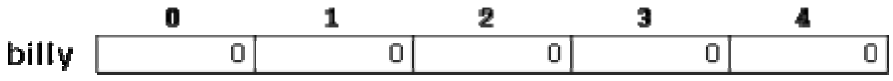
```
int billy [5];
```

Inisialisasi array

Ketika mendeklarasikan array lokal (didalam fungsi), jika tidak diberikan nilai maka isi dari array tidak akan ditentukan (*undetermined*) sampai nilai diberikan. Jika mendeklarasikan array global array (diluar semua fungsi) maka isi dari array akan di-iniisialisasikan sebagai 0 :

```
int billy [5];
```

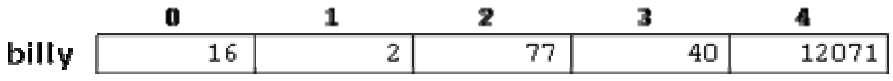
maka setiap elemen array *billy* akan di-inisialisasikan sebagai **0** :



Atau dideklarasikan dengan memberikan nilai array yang dituliskan dalam kurung kurawal :

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

Maka elemen array *billy* akan berisi :

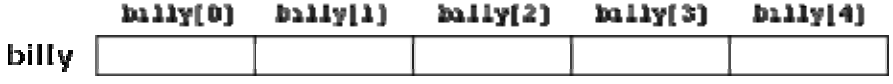


Access to the values of an Array.

Nilai array dapat diakses secara individual, dengan format :

```
name[index]
```

Maka dari contoh sebelumnya nama yang digunakan untuk mengakses masing-masing elemen:



Misalkan akan disimpan nilai 75 pada elemen ketiga, maka intruksinya :

```
billy[2] = 75;
```

Dan jika nilai elemen ketiga tadi akan diberikan ke variable **a**, maka dapat dituliskan:

```
a = billy[2];
```

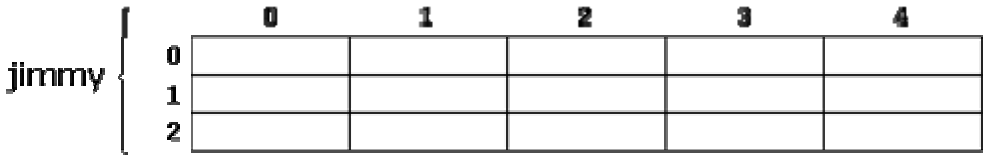
Contoh : Output :

```
// arrays example
#include <iostream.h>
int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
    result += billy[n];
  }
  cout << result;
  return 0;
}
```

12206

Array Multidimensi

Array Multidimensi dapat dikatakan sebagai array dari array. Contoh dibawah ini adalah array berdimensi 2 :



Maka pendeklarasiannya :

```
int jimmy [3][5];
```

Contoh :

```
// multidimensional array
#include <iostream.h>
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;
int main ()
{
  for (n=0;n<HEIGHT;n++)
    for (m=0;m<WIDTH;m++)
    {
      jimmy[n][m]=(n+1)*(m+1);
    }
  return 0;
}
```

```
// pseudo-multidimensional array
#include <iostream.h>
#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;
int main ()
{
  for (n=0;n<HEIGHT;n++)
    for (m=0;m<WIDTH;m++)
    {
      jimmy[n * WIDTH +
m]=(n+1)*(m+1);
    }
  return 0;
}
```

Program diatas tidak akan menghasilkan tampilan, tetapi akan menyimpan nilai dalam memory seperti dibawah ini :

jimmy

	0	1	2	3	4
0	1	2	3	4	5
1	2	4	6	8	10
2	3	6	9	12	15

Penggunaan konstanta *defined* (**#define**) untuk mempermudah jika akan melakukan perubahan.

Array sebagai parameter

Adakalanya array diberikan kedalam fungsi sebagai parameter. Dalam C++ tidak memungkinkan untuk *pass by value* satu blok memory sebagai parameter kedalam suatu fungsi. Untuk menggunakan array sebagai parameter maka yang harus dilakukan saat pendeklarasian fungsi adalah spesifikasi tipe array pada argumen, Contoh :

```
void procedure (int arg[])
```

Contoh :

```
// arrays as parameters

#include <iostream.h>
void printarray (int arg[], int length)
{
  for (int n=0; n<length; n++)
    cout << arg[n] << " ";
  cout << "\n";
}

int main ()
{
  int firstarray[] = {5, 10, 15};
  int secondarray[] = {2, 4, 6, 8, 10};
  printarray (firstarray,3);
  printarray (secondarray,5);
  return 0;

}
```

Output :

```
5 10 15
2 4 6 8 10
```

Dari contoh diatas, instruksi (**int arg[]**) menjelaskan bahwa semua array bertipe **int**, berapapun panjangnya. oleh sebab itu dideklarasikan parameter kedua dengan sifat yang sama seperti parameter pertama.

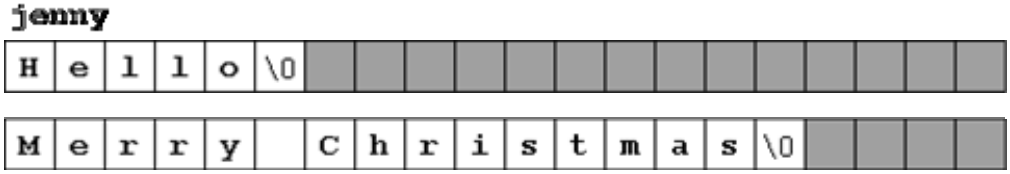
String & Character

Pada C++ tidak ada tipe variable elemen yang spesific untuk menyimpan string. Untuk keperluan ini dapat digunakan array dengan tipe **char**, dimana berisi elemen dengan tipe **char**. Perlu di ingat bahwa tipe **char** digunakan untuk menyimpan 1 karakter, karena itu array dari char digunakan untuk menyimpan string. Contoh :

```
char jenny [20];
```

Dapat menyimpan sampai dengan 20 karakter :

Penyimpanan karakter-nta dapat direpresentasikan seperti dibawah ini :



Perhatikan, karakter NULL (**'\0'**) selalu disertakan diakhir string untuk indikasi akhir dari string.

Inisialisasi string

Sama halnya seperti array-array sebelumnya, inisialisasi pada string sbb :

```
char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

Contoh diatas, merupakan inisialisasi 6 buah elemen bertipe **char,** yaitu **Hello** dan karakter null **'\0'**. Untuk menentukan nilai konstan, pada string digunakan tanda kutip ganda ("), sedangkan untuk karakter kutip tunggal ('). String yang diapit oleh kutip ganda sudah mengandung karakter Null pada akhir string, contoh :

```
char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' };
char mystring [] = "Hello";
```

contoh diatas merupakan deklarasi array **mystring** yang berisi 6 elemen.

Pemberian nilai pada string

Sama halnya seperti pemberian nilai pada array-array sebelumnya, untuk array dengan tipe char dapat dituliskan :

```
mystring[0] = 'H';
mystring[1] = 'e';
mystring[2] = 'l';
mystring[3] = 'l';
mystring[4] = 'o';
mystring[5] = '\0';
```

Cara diatas sangat tidak praktis. Umumnya untuk pemberian nilai pada array bertipe char digunakan fungsi **strcpy**. **strcpy** (**str**ing **copy**) mendefinisikan **cstring** (string.h) library dan dapat dipanggil dengan cara :

strcpy (*string1***,** *string2***);**

instruksi diatas menyebabkan isi dari*string2* di-*copy* ke *string1*. *string2* dapat berupa array, pointer, atau konstanta string.

Contoh : **Output :**

```
// setting value to string
#include <iostream.h>
#include <string.h>

int main ()
{
  char szMyName [20];
  strcpy (szMyName,"J. Soulie");
  cout << szMyName;
  return 0;
}
```

J. Soulie

Perhatikan, header**<string.h>** harus disertakan agar bisa menggunakan fungsi **strcpy**.
Bisa juga menggunakan fungsi sederhana seperti **setstring**, dengan operasi yang sama seperti **strcpy**.

Contoh : **Output :**

```
// setting value to string
#include <iostream.h>

void setstring (char szOut [], char szIn [])
{
  int n=0;
  do {
    szOut[n] = szIn[n];
  } while (szIn[n++] != '\0');
}

int main ()
{
  char szMyName [20];
  setstring (szMyName,"J. Soulie");
  cout << szMyName;
  return 0;
}
```

J. Soulie

Metode lain yang dapat digunakan untuk inisialisasi nilai yaitu input stream (**cin**). Dalam kasus ini, nilai string ditentukan oleh user saat eksekusi program. Ketika menggunakan **cin,** biasanya digunakan metode **getline**, Pemanggilannya sbb :

cin.getline (char *buffer***[], int** *length***, char** *delimiter* **= '\n');**

dimana, ***buffer*** adalah alamat untuk menyimpan input, ***length*** adalah maksimum panjang buffer,dan ***delimiter*** adalah karakter yang digunakan untuk menentukan input akhir, dengan default – atau dengan ('**\n**').

Contoh :

```
// cin with strings
#include <iostream.h>

int main ()
{
  char mybuffer [100];
  cout << "What's your name? ";
  cin.getline (mybuffer,100);
  cout << "Hello " << mybuffer << ".\n";
  cout << "Which is your favourite team? ";
  cin.getline (mybuffer,100);
  cout << "I like " << mybuffer << " too.\n";
  return 0;
}
```

Output :

```
What's your name? Juan
Hello Juan.
Which is your favourite team? Inter Milan
I like Inter Milan too.
```

Perhatikan kedua pemanggilan **cin.getline**, menggunakan identifier yang sama (**mybuffer**). Sama halnya seperti penggunaan operator extraction, sehingga dapat dituliskan :

```
cin >> mybuffer;
```

Instruksi diatas dapat berjalan, hanya saja mempunyai keterbatasan bila dibandingkan dengan **cin.getline**, diantaranya :

- Dapat menerima 1 kata saja (bukan kalimat lengkap).
- Tidak diperkenankan untuk memberikan ukuran buffer. Akan menyebabkan program tidak stabil jika user meng-input lebih besar dari kapasitas array yang ada.

Konversi string ke tipe lainnya

String dapat berisi data dengan tipe lain seperti angka. Contoh **"1977"**. **cstdlib** (**stdlib.h**) library menyediakan 3 fungsi yang dapat menangani hal tersebut :

- **atoi:** converts string to **`int`** type.
- **atol:** converts string to **`long`** type.
- **atof:** converts string to **`float`** type.

Fungsi-fungsi ini menerima 1 parameter dan mengembalikan nilainya kedalam tipe yang diminta (int, long or float). Fungsi ini dikombinasikan dengan metode **getline** pada **cin**.

Contoh :

```
// cin and ato* functions
#include <iostream.h>
#include <stdlib.h>

int main ()
{
  char mybuffer [100];
  float price;
  int quantity;
  cout << "Enter price: ";
  cin.getline (mybuffer,100);
  price = atof (mybuffer);
  cout << "Enter quantity: ";
  cin.getline (mybuffer,100);
  quantity = atoi (mybuffer);
  cout << "Total price: " << price*quantity;
  return 0;
}
```

Output :

```
Enter price: 2.75
Enter quantity: 21
Total price: 57.75
```

Fungsi untuk manipulasi string

cstring library (string.h) mendefinisikan banyak fungsi untuk operasi manipulasi, diantaranya:

strcat: `char* strcat (char* dest, const char* src);`
Appends *src* string at the end of *dest* string. Returns *dest*.

strcmp: `int strcmp (const char* string1, const char* string2);`
Compares strings *string1* and *string2*. Returns `0` is both strings are equal.

strcpy: `char* strcpy (char* dest, const char* src);`
Copies the content of *src* to *dest*. Returns *dest*.

strlen: `size_t strlen (const char* string);`
Returns the length of *string*.

Cttn : **char*** sama dengan **char[]**

Pointer

Variabel merupakan suatu nilai yang disimpan dalam memory yang dapat diakses dengan identifier. Variabel ini sesunggunhnya disimpan pada suatu alamat didalam memory. Dimana setiap alamat memory akan berbeda dengan yang lainnya (unik).

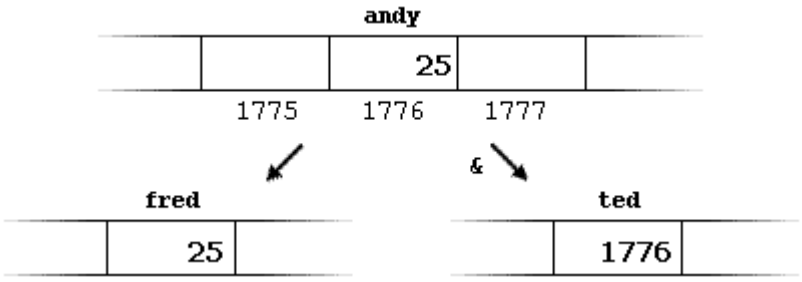
Operator Alamat (Address operator (&))
Pada saat pendeklarasian variable, user tidak diharuskan menentukan lokasi sesungguhnya pada memory, hal ini akan dilakukan secara otomatis oleh kompilerdan operating sysem pada saat run-time. Jika ingin mengetahui dimana suatu variable akan disimpan, dapat dilakukan dengan memberikan tanda *ampersand* (&) didepan variable , yang berarti ***"address of"***.
Contoh :

```
ted = &andy;
```

Akan memberikan variable **ted** alamat dari variable **andy**, karena variable **andy** diberi awalan karakter *ampersand* (&), maka yang menjadi pokok disini adalah alamat dalam memory, bukan isi variable. Misalkan **andy** diletakkan pada alamat **1776** kemudian dituliskan instruksi sbb :

```
andy = 25;
fred = andy;
ted = &andy;
```

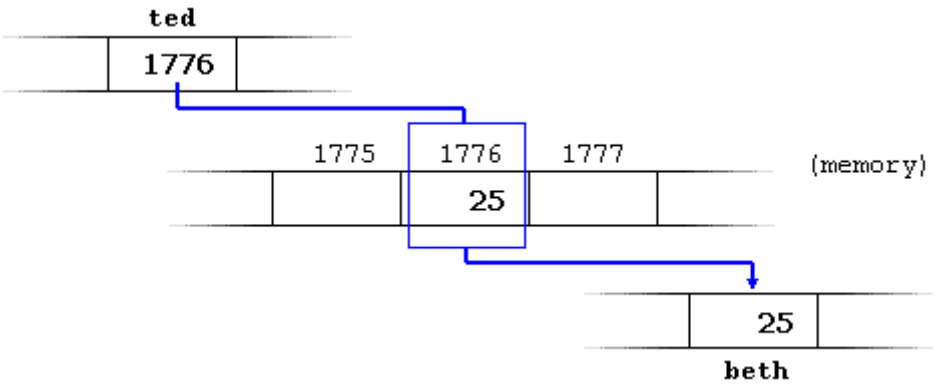
Maka hasilnya :



Operator Reference (*)
Dengan menggunakan pointer, kita dapat mengakses nilai yang tersimpan secara langsung dengan memberikan awalan operator *asterisk* (*) pada identifier pointer, yang berarti ***"value pointed by"***. Contoh :

```
beth = *ted;
```

(dapat dikatakan:"beth sama dengan nilai yang ditunjuk oleh ted") **beth** = **25**, karena **ted** dialamat **1776**, dan nilai yang berada pada alamat **1776** adalah **25**.



Ekspresi dibawah ini semuanya benar, perhatikan :

```
andy == 25
&andy == 1776
ted == 1776
*ted == 25
```

Ekspresi pertama merupakan *assignation* bahwa **andy=25;**. Kedua, menggunakan operator alamat (address/derefence operator (&)), sehingga akan mengembalikan alamat dari variabel **andy**. Ketiga bernilai benar karena *assignation* untuk **ted** adalah **ted = &andy;**. Keempat menggunakan reference operator (*) yang berarti nilai yang ada pada alamat yang ditunjuk oleh **ted**, yaitu **25**. Maka ekspresi dibawah ini pun akan bernilai benar :

```
*ted == andy
```

Deklarasi variable bertipe pointer

Format deklarasi pointer :

```
type * pointer_name;
```

Dimana ***type*** merupakan tipe dari data yang ditunjuk, bukan tipe dari pointer-nya. Contoh :

```
int * number;
char * character;
float * greatnumber;
```

Contoh :

```
// my first pointer
#include <iostream.h>

int main ()
{ int value1 = 5, value2 = 15;
  int * mypointer;

  mypointer = &value1;
  *mypointer = 10;
  mypointer = &value2;
  *mypointer = 20;
  cout << "value1==" << value1 << "/ value2==" << value2;
  return 0;
}
```

Output :

```
value1==10 / value2==20
```

Perhatikan bagaimana nilai dari **value1** dan **value2** diubah secara tidak langsung. Pertama **mypointer** diberikan alamat **value1** dengan menggunakan tanda ampersand (&). Kemudian memberikan nilai **10** ke nilai yang ditunjuk oleh **mypointer**, yaitu alamat dari **value1**, maka secara tidak langsung **value1** telah dimodifikasi. Begitu pula untuk **value2**.

Contoh :

```
// more pointers
#include <iostream.h>

int main ()
{
  int value1 = 5, value2 = 15;
  int *p1, *p2;

  p1 = &value1;     // p1 = address of value1
  p2 = &value2;     // p2 = address of value2
  *p1 = 10;         // value pointed by p1 = 10
  *p2 = *p1;        // value pointed by p2 = value pointed by p1
  p1 = p2;          // p1 = p2 (value of pointer copied)
  *p1 = 20;         // value pointed by p1 = 20

  cout << "value1==" << value1 << "/ value2==" << value2;
  return 0;
}
```

Output :

```
value1==10 / value2==20
```

Array dan Pointer

Identifier suatu array equivalen dengan alamat dari elemen pertama, pointer equivalen dengan alamat elemen pertama yang ditunjuk. Perhatikan deklarasi berikut :

```
int numbers [20];
int * p;
```

maka deklarasi dibawah ini juga benar :

```
p = numbers;
```

p dan **numbers** equivalen, dan memiliki sifat (*properties*) yang sama. Perbedaannya, user dapat menentukan nilai lain untuk pointer **p** dimana **numbers** akan selalu menunjuk nilai yang sama seperti yang telah didefinisikan. **p**, merupakan *variable pointer*, **numbers** adalah *constant pointer*. Karena itu walaupun instruksi diatas benar, tetapi tidak untuk instruksi dibawah ini :

```
numbers = p;
```

karena **numbers** adalah array (constant pointer), dan tidak ada nilai yang dapat diberikan untuk identifier konstant (*constant identifiers*).

Contoh : **Output :**

```
// more pointers                          10, 20, 30, 40, 50,
#include <iostream.h>

int main ()
{
  int numbers[5];
  int * p;
  p = numbers;  *p = 10;
  p++;  *p = 20;
  p = &numbers[2];  *p = 30;
  p = numbers + 3;  *p = 40;
  p = numbers;  *(p+4) = 50;
  for (int n=0; n<5; n++)
    cout << numbers[n] << ", ";
  return 0;
}
```

Inisialisasi Pointer

Contoh :

```
int number;
int *tommy = &number;
```

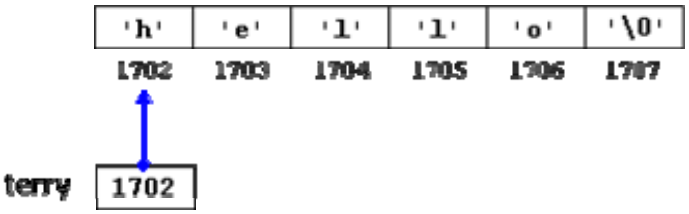
Equivalen dengan :

```
int number;
int *tommy;
tommy = &number;
```

Seperti pada array, inisialisasi isi dari pointer dapat dilakukan dengan deklarasi seperti contoh berikut :

```
char * terry = "hello";
```

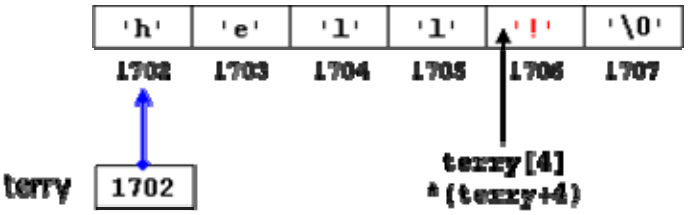
Misalkan "hello" disimpan pada alamat 1702 dan seterusnya, maka deklarasi tadi dapat digambarkan sbb :



terry berisi nilai **1702** dan bukan **'h'** atau **"hello"**, walaupun **1702** menunjuk pada karakter tersebut. Sehingga jika akan dilakukan perubahan pada karakter **'o'** diganti dengan tanda **'!'** maka ekspresi yang digunakan ada 2 macam :

```
terry[4] = '!';
*(terry+4) = '!';
```

Penulisan **terry[4]** dan ***(terry+4)**, mempunyai arti yang sama. Jika digambarkan :



Pointer Arithmatika

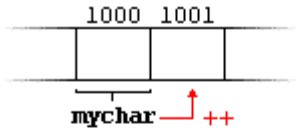
Contoh, *char* memerlukan 1 byte, *short* memerlukan 2 bytes dan *long* memerlukan 4. Terdapat 3 buah pointer :

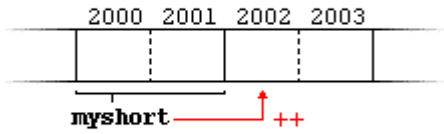
```
char *mychar;
short *myshort;
long *mylong;
```

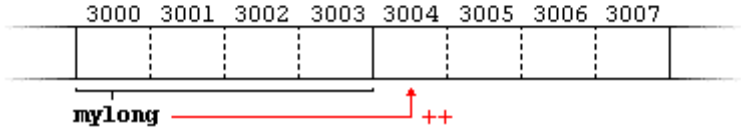
ekspresi diatas akan menunjuk pada lokasi dimemory masing-masing **1000**, **2000** and **3000**, sehingga jika dituliskan :

```
mychar++;
myshort++;
mylong++;
```

mychar, akan bernilai 1001, myshort bernilai 2002, dan mylong bernilai 3004. Alasannya adalah ketika terjadi pertambahan maka akan ditambahkan dengan tipe yang sama seperti yang didefinisikan berupa ukuran dalam bytes.







Perhatikan ekspresi dibawah ini :

```
*p++;
*p++ = *q++;
```

Ekspresi pertama equivalen dengan *(p++) dan yang dilakukan adalah menambahkan **p** (yaitu alamat yang ditunjuk, bukan nilai yang dikandungnya).
Ekspresi kedua, yang dilakukan pertama adalah memberikan nilai *q ke *p dan kemudian keduanya ditambahkan 1 atau dengan kata lain :

```
*p = *q;
p++;
q++;
```

void pointer

Tipe pointer *void* merupakan tipe khusus. *void* pointers dapat menunjuk pada tipe data apapun, nilai integer value atau float, maupun string atau karakter. Keterbatasannya adalah tidak dapat menggunakan operator asterisk (*), karena panjang pointer tidak diketahui, sehingga diperlukan operator *type casting* atau *assignations* untuk mengembalikan nilai *void* pointer ketipe data sebenarnya.

Contoh :

```
// integer increaser
#include <iostream.h>

void increase (void* data, int type)
{
  switch (type)
  {
    case sizeof(char) : (*((char*)data))++; break;
    case sizeof(short): (*((short*)data))++; break;
    case sizeof(long) : (*((long*)data))++; break;
  }
}

int main ()
{
  char a = 5;
  short b = 9;
  long c = 12;
  increase (&a,sizeof(a));
  increase (&b,sizeof(b));
  increase (&c,sizeof(c));
  cout << (int) a << ", " << b << ", " << c;
  return 0;
}
```

Output :

```
6, 10, 13
```

Pointer untuk functions

C++ memperbolehkan operasi dengan pointer pada function. Kegunaan yang utama adalah untuk memberikan satu function sebagai parameter untuk function lainnya. Deklarasi pointer untuk function sama seperti prototype function kecuali nama function dituliskan diantara tanda kurung () dan operator asterisk (*) diberikan sebelum nama.

Contoh :

```
// pointer to functions
#include <iostream.h>

int addition (int a, int b)
{ return (a+b); }

int subtraction (int a, int b)
{ return (a-b); }
```

```
int (*minus)(int,int) = subtraction;

int operation (int x, int y, int (*functocall)(int,int))
{
  int g;
  g = (*functocall)(x,y);
  return (g);
}

int main ()
{
  int m,n;
  m = operation (7, 5, addition);
  n = operation (20, m, minus);
  cout <<n;
  return 0;
}
```

Output :

8

Dari contoh diatas, **minus** merupakan pointer global untuk function yang mempunyai 2 parameters bertipe **int**, kemudian diberikan untuk menunjuk function **subtraction**, ditulis dalam satu baris instruksi :

int (* minus)(int,int) = subtraction;

Structures

Data structures

Struktur data merupakan kumpulan berbagai tipe data yang memiliki ukuran yang berbeda di kelompokan dalam satu deklarasi unik, dengan format sbb :

```
struct model_name {
  type1 element1;
  type2 element2;
  type3 element3;
  .
  .
} object_name;
```

dimana *model_name* adalah nama untuk model tipe stukturnya dan parameter optional *object_name* merupakan identifier yang valid untuk objek sturktur. Diantara kurung kurawal { } berupa tipe dan sub-identifier yang mengacu ke elemen pembentuk struktur. Jika pendefinisian stuktur menyertakan parameter *model_name* (optional), maka parameter tersebut akan menjadi nama tipe yang valid ekuivalen dengan struktur. Contoh :

```
struct products {
  char name [30];
  float price;
} ;

products apple;
products orange, melon;
```

Didefinisikan model struktur **products** dengan dua field : **name** dan **price**, dengan tipe yang berbeda. Kemudian tipe struktur tadi (**products**) digunakan untuk mendeklarasikan tiga objek : **apple**, **orange** dan **melon**.

Ketika dideklarasikan, **products** menjadi tnama tipe yang valid seperti tipe dasar *int*, *char* atau *short* dan dapat mendeklarasikan objects (variables) dari tipe tersebut. Optional field yaitu *object_name* dapat dituliskan pada akhir deklarasi struktur untuk secara langsung mendeklarasikan object dari tipe struktur. Contoh :

```
struct products {
  char name [30];
  float price;
} apple, orange, melon;
```

Sangat penting untuk membedakan antara structure **model**, dan structure *object*. *model* adalah *type*, dan *object* adalah *variable*. Kita dapat membuat banyak *objects* (variables) dari satu *model* (type).

Contoh :

```
// example about structures
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
```

```
struct movies_t {
  char title [50];
  int year;
} mine, yours;

void printmovie (movies_t movie);

int main ()
{
  char buffer [50];

  strcpy (mine.title, "2001 A Space Odyssey");
  mine.year = 1968;

  cout << "Enter title: ";
  cin.getline (yours.title,50);
  cout << "Enter year: ";
  cin.getline (buffer,50);
  yours.year = atoi (buffer);

  cout << "My favourite movie is:\n ";
  printmovie (mine);
  cout << "And yours:\n ";
  printmovie (yours);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

***Output* :**

```
Enter title: Alien
Enter year: 1979

My favourite movie is:
 2001 A Space Odyssey (1968)
And yours:
 Alien (1979)
```

Contoh diatas menjelaskan bagaimana menggunakan elemen dari struktur dan struktur itu sendiri sebagai variable normal. Contoh, **yours.year** merupakan variable valid dengan tipe **int**, dan **mine.title** merupakan array valid dari 50 *chars*.

Perhatikan **mine** dan **yours** juga berlaku sebagai valid variable dari tipe **movies_t** ketika di-pass ke-function **printmovie()**. Salah satu keuntungan dari *structures* yaitu kita dapat mengacu pada setiap elemennya atau keseluruhan blok struktur.

Contoh :

```
// array of structures
#include <iostream.h>
#include <stdlib.h>
#define N_MOVIES 5

struct movies_t {
  char title [50];
  int year;
} films [N_MOVIES];

void printmovie (movies_t movie);

int main ()
{
  char buffer [50];
  int n;
  for (n=0; n<N_MOVIES; n++)
  {
    cout << "Enter title: ";
    cin.getline (films[n].title,50);
    cout << "Enter year: ";
    cin.getline (buffer,50);
    films[n].year = atoi (buffer);
  }
  cout << "\nYou have entered these movies:\n";
  for (n=0; n<N_MOVIES; n++)
    printmovie (films[n]);
  return 0;
}

void printmovie (movies_t movie)
{
  cout << movie.title;
  cout << " (" << movie.year << ")\n";
}
```

Output :

```
Enter title: Alien
Enter year: 1979
Enter title: Blade Runner
Enter year: 1982
Enter title: Matrix
Enter year: 1999
Enter title: Rear Window
Enter year: 1954
Enter title: Taxi Driver
Enter year: 1975

You have entered these movies:
Alien (1979)
Blade Runner (1982)
Matrix (1999)
Rear Window (1954)
Taxi Driver (1975)
```

Pointer to structure

Sama seperti pada tipe lainnya, struktur juga dapat ditunjuk oleh pointer. Aturannya sama untuk setiap tipe data. Pointer harus dideklarasikan sebagai pointer untuk struktur :

```
struct movies_t {
  char title [50];
  int year;
};

movies_t amovie;
movies_t * pmovie;
```

amovie merupakan object dari tipe struct **movies_t** dan **pmovie** adalah pointer untuk menunjuk ke objek dari tipe struct **movies_t**. maka, deklarasi dibawah ini juga benar :

```
pmovie = &amovie;
```

Contoh :

```
// pointers to structures
#include <iostream.h>
#include <stdlib.h>

struct movies_t {
  char title [50];
  int year;
};

int main ()
{
  char buffer[50];

  movies_t amovie;
  movies_t * pmovie;
  pmovie = & amovie;

  cout << "Enter title: ";
  cin.getline (pmovie->title,50);
  cout << "Enter year: ";
  cin.getline (buffer,50);
  pmovie->year = atoi (buffer);
  cout << "\nYou have entered:\n";
  cout << pmovie->title;
  cout << " (" << pmovie->year << ")\n";
  return 0;
}
```

Output :

```
Enter title: Matrix
Enter year: 1999

You have entered:
Matrix (1999)
```

Operator **->** merupakan operator penunjuk yang digunakan secara khusus bersama dengan pointer untuk struktur dan pointer untuk class. Memungkinkan kita untuk tidak menggunakan tanda kurung pada setiap anggota struktur yang ditunjuk. Dalam contoh digunakan :

```
pmovie->title
```

Atau dalam penulisan yang lain :

```
(*pmovie).title
```

Kedua ekspresi tersebut diatas : **pmovie->title** dan **(*pmovie).title** benar dan berarti evaluasi elemen **title** dari struktur yang ditunjuk (pointed by) **pmovie**. Harus dibedakan dari :

```
*pmovie.title
```

Yang ekuivalen dengan :

```
*(pmovie.title)
```

Dibawah ini merupaka tabel rangkuman, kombinasi yang mungkin terjadi antara pointer dan struktur :

Expression	Description	Equivalent
pmovie.title	Element **title** of structure **pmovie**	
pmovie->title	Element **title** of structure pointed by **pmovie**	`(*pmovie).title`
***pmovie.title**	Value pointed by element **title** of structure **pmovie**	`*(pmovie.title)`

Nesting structures

Struture juga dapat berbentuk nested (bersarang) sehingga suatu elemen dari suatu struktur dapat menjadi elemen pada struktur yang lain :

```
struct movies_t {
  char title [50];
  int year;
}

struct friends_t {
  char name [50];
  char email [50];
  movies_t favourite_movie;
  } charlie, maria;

friends_t * pfriends = &charlie;
```

Setelah deklarasi diatas, dapat digunakan ekspresi sbb :

```
charlie.name
maria.favourite_movie.title
charlie.favourite_movie.year
pfriends->favourite_movie.year
```

(Dimana 2 ekspresi terakhir ekuivalen)

User defined data types

Definition of own types (typedef).
C++ memungkinkan kita untuk mendefinisikan tipe berdasarkan tipe data yang sudah ada. Untuk itu digunakan keyword **typedef**, dengan format:

typedef *existing_type* *new_type_name* ;

dimana *existing_type* adalah tipe data dasar pada C++ dan *new_type_name* adalah nama dari tipe baru yang didefinisikan. Contoh :

```
typedef char C;
typedef unsigned int WORD;
typedef char * string_t;
typedef char field [50];
```

Contoh diatas telah mendefinisikan empat tipe data baru : **C**, **WORD**, **string_t** dan **field** sebagai **char**, **unsigned int**, **char*** dan **char[50]** yang akan digunakan nanti seperti berikut :

```
C achar, anotherchar, *ptchar1;
WORD myword;
string_t ptchar2;
field name;
```

Union
Union memungkinkan bagian dari memory dapat diakses sebagai tipe data yang berbeda, walaupun pada dasarnya mereka berada pada lokasi yang sama di memory. Pendeklarasian dan penggunaanya hampir sama dengan struktur tetapi secara fungsional berbeda :

```
union model_name {
  type1 element1;
  type2 element2;
  type3 element3;
  .
  .
} object_name;
```

Semua elemen pada deklarasi *union* declaration menempati tempat yang sama dimemory. Ukuran yang digunakan diambil dari tipe yang paling besar. Contoh :

```
union mytypes_t {
  char c;
  int i;
  float f;
  } mytypes;
```

Mendefinisikan tiga elemen :

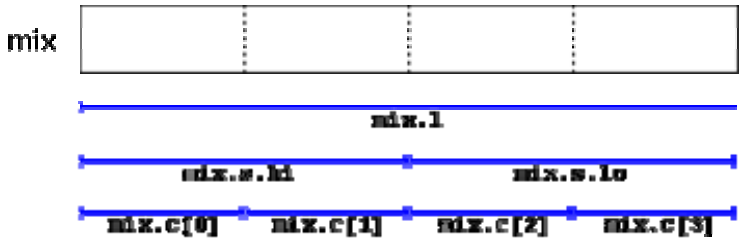
```
mytypes.c
mytypes.i
mytypes.f
```

Tiap data memiliki tipe yang berbeda, karena menempati lokasi yang sama dmemory, maka perubahan terhadap satu elemen akan mempengaruhi elemen yang lain.

Salah satu kegunaan *union,* memungkinkan untuk menggabungkan tipe dasar dengan suatu array atau struktur dari elemen yang lebih kecil. Contoh :

```
union mix_t{
  long l;
  struct {
    short hi;
    short lo;
    } s;
  char c[4];
} mix;
```

Mendefinisikan tiga nama yang memungkinkan kita untuk mengakses grup 4 bytes yang sama : **mix.l**, **mix.s** dan **mix.c** dan dapat digunakan menutut bagaimana kita akan mengaksesnya, sebagai **long**, **short** atau **char**. Tipe data yang sudah digabungkan, arrays dan structures dalam suatu union, maka dibawah ini merupakan cara pengakses-an yang berbeda :



Anonymous unions

Pada C++ terdapat option unions tanpa nama (anonymous union). Jika disertakan union dalam structure tanpa nama objek(yang dituliskan setelah kurung kurawal { }) maka union akan tidak memiliki nama dan kita dapat mengakses elemennya secara langsung dengan namanya. Contoh:

union

```
struct {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yens;
  } price;
} book;
```

anonymous union

```
struct {
  char title[50];
  char author[50];
  union {
    float dollars;
    int yens;
  };
} book;
```

Perbedaan deklarasi diatas adalah program pertama diberi nama pada union (**price**) dan yang kedua tidak. Pada saat mengakses anggota **dollars** dan **yens** dari objek. Pada program pertama :

```
book.price.dollars
book.price.yens
```

Sedangkan untuk program kedua :

```
book.dollars
book.yens
```

Classes

Class adalah metode logical untuk organisasi data dan fungsi dalam struktur yang sama. Class dideklarasikan menggunakan keyword **class**, yang secara fungsional sama dengan keyword **struct**, tetapi dengan kemungkinan penyertaan fungsi sebagai anggota, formatnya sbb :

```
class class_name {
  permission_label_1:
    member1;
  permission_label_2:
    member2;
  ...
  } object_name;
```

Dimana ***class_name*** adalah nama class(user defined *type*) dan field optional ***object_name*** adalah satu atau beberapa identifier objek yang valid. *Body* dari deklarasi berisikan ***members***, yang dapat berupa data ataupun deklarasi fungsi, dan ***permission labels*** (optional), dapat berupa satu dari tiga keyword berikut : **private:**, **public:** atau **protected:**. Digunakan untuk menentukan batasan akses terhadap *members* yang ada :

- **private** , anggota class dapat diakses dari danggota lain pada kelas yang sama atau dari class "*friend*".
- **protected ,** anggota dapat diakses dari anggota class yang sama atau class *friend* , dan juga dari anggota class turunannya(*derived*).
- **public** , anggota dapat diakses dari class manapun.

Default permission label : **private**

Contoh :

```
class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void);
  } rect;
```

Deklarasi class **CRectangle** dan object bernama **rect**. Class ini berisi empat anggota: dua variable bertipe **int** (**x** and **y**) pada bagian **private** (karena private adalah default permission) dan dua fungsi pada bagian **public** : **set_values()** dan **area()**, dimana hanya dideklarasikan propotype_nya.

Perhatikan perbedaan antara nama classdan nama object. Pada contoh sebelumnya **CRectangle** adalah nama class(contoh, user-defined type), dan **rect** adalah object dari tipe **CRectangle**. Sama halnya dengan deklarasi berikut :

```
int a;
```

int adalah nama *class* (type) dan **a** adalah nama *object*(variable).

Contoh :

```
// classes example
#include <iostream.h>

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  CRectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
}
```

Output :

```
area: 12
```

Hal baru dari contoh diatas adalah operator :: dari lingkup yang disertakan dalam pendefinisian **set_values()**. Digunakan untuk mendeklarasikan anggota dari class diluar class tersebut.

Scope operator (::) menspesifikasikan class dimana anggota yang dideklarasikan berada, memberikan scope properties yang sama seperti jika dideklarasikan secara langsung dalam class.

Contoh :

```
// class example
#include <iostream.h>

class CRectangle {
    int x, y;
  public:
    void set_values (int,int);
    int area (void) {return (x*y);}
};

void CRectangle::set_values (int a, int b) {
  x = a;
  y = b;
}
```

```
int main () {
  CRectangle rect, rectb;
  rect.set_values (3,4);
  rectb.set_values (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

Output **:**

```
rect area: 12
rectb area: 30
```

Perhatikan pemanggilan **rect.area()** tidak memberikan hasil yang sama dengan pemanggilan **rectb.area()**. Ini disebabkan karena objek dari class **CRectangle** mempunyai variable sendiri, **x** dan **y**, dan fungsi **set_value()** dan **area()**.

Constructor dan destructor

Objek biasanya memerlkan inisialisasi variable atau menentukan memori dinamik selama proses untuk mencapai hasil akhie yang diharapkan dan menghindari pengembalian nilai yang tidak diharapkan.

Untuk mengatasinya dapat digunakan fungsi spesial : *constructor*, yang dapat dideklarasikan dengan pemberian nama fungsi dengan nama yang sama untuk class. Fungsi dari constructor ini akan dipanggil secara otomatis ketika instance baru dari sebuah class dibuat. Contoh :

```
// classes example
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

Output :

```
rect area: 12
rectb area: 30
```

Hasil dari contoh diatas sama seperti contoh sebelumnya. Dalam hal ini, hanya menggantikan fungsi **set_values**, yang sudah tidak ada dengan class *constructor*. Perhatikan cara parameter diberikan ke constructor pada saat instance class dibuat :

```
CRectangle rect (3,4);
CRectangle rectb (5,6);
```

Destructor berfungsi kebalikannya. Secara otomatis akan dipanggil jika objek di keluarkan dari memory, ataupun karena keberadaannya telah selesai (contoh : jika didefinisikan sebuah objek local dalam function dan function tersebut selesai) atau karena merupakan objek yang secara dinamis ditetapkan dan dikeluarkan dengan menggunakan operator **delete**.

Destuctor harus mempunyai nama yang sama dengan class, diberi awalan tile (~) dan tidak mengembalikan nilai. Contoh :

```
// example on constructors and destructors
#include <iostream.h>

class CRectangle {
    int *width, *height;
  public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width * *height);}
};

CRectangle::CRectangle (int a, int b) {
  width = new int;
  height = new int;
  *width = a;
  *height = b;
}

CRectangle::~CRectangle () {
  delete width;
  delete height;
}

int main () {
  CRectangle rect (3,4), rectb (5,6);
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
  return 0;
}
```

Output :

```
rect area: 12
rectb area: 30
```

Overloading Constructors

Sama halnya seperti fungsi, constructor juga dapat mempunyai nama yang sama tetapi mempunyai jumlah dan tipe yang berbeda pada parameternya. Pada saat pemanggilan kompiler akan meng-eksekusi yang sesuai pada saat objek class di deklarasikan.

Pada kenyataannya, ketika dideklarasikan sebuah class dan tidak disebutkan constructornya, maka kompiler secara otomatis akan mengasumsikan dua constructor overloaded ("*default constructor*" dan "*copy constructor*"). Contoh :

```
class CExample {
  public:
    int a,b,c;
    void multiply (int n, int m) { a=n; b=m; c=a*b; };
  };
```

Jika tanpa constructor, Kompiler secara otomatis mengasumsikan anggota-anggota fungsi constructor berikut :

- **Empty constructor**

Merupakan constructor tanpa parameters didefinisikan sebagai ***nop*** (blok instruksi kosong). Tidak melakukan apapun.

```
CExample::CExample () { };
```

- **Copy constructor**

  Merupakan constructor dengan satu parameter dengan tipe yang sama yang ditetapkan untuk setiap anggota variable class nonstatik objek yang disalin dari objek sebelumnya.

```
CExample::CExample (const CExample& rv) {
  a=rv.a;  b=rv.b;  c=rv.c;
  }
```

Penting untuk mengetahui, bahwa kedua constructor default : *empty construction* dan *copy constructor* ada jika tidak ada constructor lain yang dideklarasikan. Jika terdapat constructor dengan sejumlah parameter dideklarasikan, maka tidak satupun dari constructors default ini ada.

Contoh :

```
// overloading class constructors
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    CRectangle ();
    CRectangle (int,int);
    int area (void) {return (width*height);}
};

CRectangle::CRectangle () {
  width = 5;
  height = 5;
}

CRectangle::CRectangle (int a, int b) {
  width = a;
  height = b;
}

int main () {
  CRectangle rect (3,4);
  CRectangle rectb;
  cout << "rect area: " << rect.area() << endl;
  cout << "rectb area: " << rectb.area() << endl;
}
```

Output :

```
rect area: 12
rectb area: 25
```

Contoh diatas **rectb** dideklarasikan tanpa parameter, sehingga diinisialisasikan dengan *constructor* tanpa parameters, yang mendeklarasikan **width** dan **height** dengan nilai **5**.
Perhatikan jika dideklarasikan objek baru dan tidak diberikan parameter maka tidak diperlukan tanda kurung **()**:

```
CRectangle rectb;   // right
CRectangle rectb(); // wrong!
```

Relationships between classes

Friend functions (friend keyword)

Terdapat tiga akses berbeda untuk anggota class : **public**, **protected** dan **private**. Untuk anggota *protected* dan *private*, tidak dapat diakses dari luar fungsi dimana mereka dideklarasikan. Namun, aturan ini dapat di lewati dengan menggunakan keyword *friend* dalam class, sehingga fungsi eksternal dapat mengakses anggota **protected** dan **private** suatu class.

Untuk itu, harus dideklarasikan prototype dari fungsi eksternal yang akan mengakses, Contoh :

```
// friend functions
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle rectparam)
{
  CRectangle rectres;
  rectres.width = rectparam.width*2;
  rectres.height = rectparam.height*2;
  return (rectres);
}

int main () {
  CRectangle rect, rectb;
  rect.set_values (2,3);
  rectb = duplicate (rect);
  cout << rectb.area();
}
```

Output :

24

Dari fungsi **duplicate**, yaitu *friend* dari **CRectangle**, dapat mengakses anggota **width** dan **height** untuk objek yang berbeda dari tipe **CRectangle**. Perhatikan bahwa dalam deklarasi **duplicate()** maupun **main()** tidak dipertimbangkan apakah **duplicate** merupakan anggota dari class **CRectangle**. Secara umum keguanaan fungsi *friend* diluar methodology pemrograman, jadi jika memungkinkan sebaiknya menggunakan anggota dari clas yang sama.

Friend classes (friend)

Selain dapat mendefinisikan fungsi friend, dapat juga didefinisikan class sebagai *friend* dari class lainnya, sehingga memungkinkan class kedua dapat mengakses anggota **protected** dan **private** class pertama. Contoh :

```
// friend class
#include <iostream.h>

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area (void)
      {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rect;
  sqr.set_side(4);
  rect.convert(sqr);
  cout << rect.area();
  return 0;
}
```

Output :

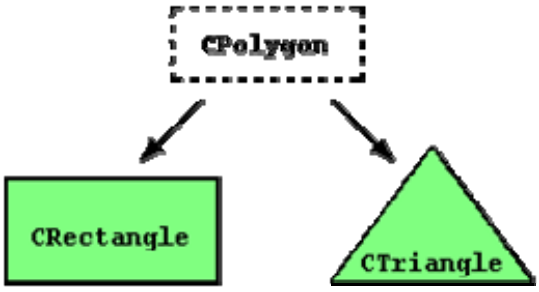
16

Pada contoh diatas, dideklarasikan **CRectangle** sebagai *friend* dari **CSquare**, sehingga **CRectangle** dapat mengakses anggota **protected** dan **private** dari **CSquare**, lebih jelasnya **CSquare::side**, Mendefinisikan lebar dari kotak.

Juga terdapat *empty prototype* dari class **CSquare** diawal program. Merupakan hal yang penting karena dalam deklarasi **CRectangle** mengacu kepada **CSquare** (sebagai parameter dalam **convert()**). Pendefinisian **CSquare** disertakan nanti, jika tidak disertakan pada deklarasi sebelumnya untuk **CSquare**, maka class ini tidak akan terlihat dalam pendefinisian **CRectangle**.

Inheritance between classes

Inheritance memungkinkan kita untuk membuat objek dari objek sebelumnya, sehingga memungkinkan untuk menyertakan beberapa anggota objek sebelumnya ditambah dengan anggota objeknya sendiri. Contoh, membuat class untuk mengetahui apakah segi empat (**CRectangle**), atau (**CTriangle**). Masing-masing mempunyai hal yang sama yaitu, dasar dan tinggi. Dapat direpresentasikan dengan class **CPolygon** kemudian diturunkan menjadi **CRectangle** dan **CTriangle**.



Class **CPolygon** dapat berisi anggota yang dipakai untuk setiap *polygon,* dalam hal ini **width** dan **height**, dan **CRectangle** dan **CTriangle** adalah class turunannya.

Class turunan akan menurunkan seluruh anggota yang dimiliki oleh class dasar(parent)nya. Jadi jika class paret mempunyai anggota **A** dan diturunkan pada class lain dengan anggota **B**,maka clas turunan ini akan memiliki **A** dan **B**.

Untuk menurunkan class, menggunakan operator **:** (colon) pada saat deklarasi, syntax :

```
class derived_class_name: public base_class_name;
```

Dimana *derived_class_name* adalah nama dari *derived* class dan *base_class_name* adalah nama dari class asal. **public** dapat digantikan dengan tipe akses lainnya : **protected** atau **private**, Contoh:

```
// derived classes
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };
```

```
class CRectangle: public CPolygon {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

Output :

20
10

Class **CRectangle** dan **CTriangle** masing-masing mengandung anggota dari **CPolygon**, yaitu : **width**, **height** dan **set_values()**.
Rangkuman tipe akses dan anggota yang bisa mengaksesnya :

Access	public	protected	private
members of the same class	yes	yes	yes
members of derived classes	yes	yes	no
not-members	yes	no	no

Dimana "*not-members*" merepresentasikan referensi dari luar class, seperti dari **main()**, dari class lain atau dari fungsi lain baik global ataupun local.
Pada contoh diatas, anggota yang diturunkan kepada **CRectangle** dan **CTriangle** diikuti dengan hak akses yang sama dengan class asalnya, **CPolygon**:

```
CPolygon::width           // protected access
CRectangle::width         // protected access
CPolygon::set_values()    // public access
CRectangle::set_values()  // public access
```

Ini diakibatkan karena menurunkan class sebagai **public** :

```
class CRectangle: public CPolygon;
```

What is inherited from the base class?
Pada dasarnya setiap anggota dari class dasar diturunkan kepada class turunannya, kecuali :

- **Constructor and destructor**

- **operator=() member**
- **friends**

Jika class dasar tidak mempunyai default constructor atau akan melakukan pemanggilan terhadap constructor overloaded ketika objek turunan yang baru dibuat maka dapat dituliskan dalam setiap definisi constructor dari class turunan:

```
derived_class_name (parameters) : base_class_name
(parameters) {}
```

Contoh :

```
// constructors and derivated classes
#include <iostream.h>

class mother {
  public:
    mother ()
      { cout << "mother: no parameters\n"; }
    mother (int a)
      { cout << "mother: int parameter\n"; }
};

class daughter : public mother {
  public:
    daughter (int a)
      { cout << "daughter: int parameter\n\n"; }
};

class son : public mother {
  public:
    son (int a) : mother (a)
      { cout << "son: int parameter\n\n"; }
};

int main () {
  daughter cynthia (1);
  son daniel(1);

  return 0;
}
```

Output :

```
mother: no parameters
daughter: int parameter

mother: int parameter
son: int parameter
```

Terlihat perbedaan mana yang merupakan constructor dari mother yang dipanggil ketika objek **daughter** dibuat dan ketika objek **son dibuat**. Perbedaannya disebabkan dari deklarasi untuk **daughter** dan **son**:

```
daughter (int a)          // nothing specified: call default
constructor
son (int a) : mother (a)  // constructor specified: call
this one
```

Multiple inheritance

Dalam C++ memungkinkan untuk menurunkan field atau method dari satu atau lebih class dengan menggunakan operator koma dalam deklarasi class turunan. Contoh, akan dibuat class untuk menampilkan dilayar (**COutput**) dan akan diturunkan ke class **CRectangle** and **CTriangle** maka dapat dituliskan :

```
class CRectangle: public CPolygon, public COutput {
class CTriangle: public CPolygon, public COutput {
```

Contoh :

```
// multiple inheritance
#include <iostream.h>

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class COutput {
  public:
    void output (int i);  };

void COutput::output (int i) {
  cout << i << endl;  }

class CRectangle: public CPolygon, public COutput {
  public:
    int area (void)
      { return (width * height); }
  };

class CTriangle: public CPolygon, public COutput {
  public:
    int area (void)
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  rect.output (rect.area());
  trgl.output (trgl.area());
  return 0;
}
```

Output :

```
20
10
```