Industrial Engineering
Faculty of Engineering
Esa Unggul University

MODUL PRAKTIKUM
PEMROGRAMAN KOMPUTER

Program Studi Teknik Industri
Fakultas Teknik
Universitas Esa Unggul
2017

KATA PENGANTAR

Assalamu'alaikum Wr. Wb

Puji syukur kita panjatkan ke hadirat Tuhan Yang Maha Esa yang telah memberikan nikmat dan karunia-Nya sehingga Modul Praktikum Peemrograman Komputer menggunakan *Software* C++ dapat terselesaikan. Modul yang disusun bertujuan untuk menjadi saran yang membantu praktikan untuk memahami jalannya praktikum yang akan dilakukan. Selain itu juga, modul ini menjadi pegangan dan panduan pelaksanaan praktikum bagi praktikan yang berisi metode pelaksanaan praktikum serta teori-teori yang mendukung dalam materi yang dipraktikkan.

Pepatah ilmuwan sebuah sistem mengatakan bahwa "*tidak ada sistem yang terbaik, selalu ada sistem yang lebih baik*". Dengan salah satu pedoman ini yang membuat penulis menjadi terus menerus melakukan sebuah perbaikan dalam upaya mendekati kesempurnaan dalam penyajian modul bagi mahasiswa praktikan. Sehingga mahasiswa sebagai praktikan mampu mengaplikasikan materi yang diberikan dengan optimal. Penulis juga mengucapkan terimakasih kepada pihak-pihak yang tela membantu penyusunan modul praktikum ini.

Disisi lain, penulis juga menyadari bahwa masih banyak kekurangan yang dalam penulisan serta penyusunan Modul Praktikum ini. Oleh karena itu, saran dan kontribusi yang membangun dalam upaya perbaikan dari *contain* modul ini dari mahasiswa praktikan merupakan bagian penting proses pembelajaran ini serta menjadi masukan yang berharga bagi kami.

Besar harapan kami semoga modul ini dapat bermanfaat bagi yang mempelajarinya, khususnya mahasiswa Teknik Industri, Universitas Esa Unggul.

Jakarta, 21 Agustus 2017

Penulis

Riya Widayanti, S.Kom, MMSI & Asisten

DAFTAR ISI

TATA TERTIB PRAKTIKUM

FAKULTAS TEKNIK

Tata Tertib Praktikum dan Penggunaan Laboratorium bagi mahasiswa Fakultas Teknik sbb :

1. Mahasiswa berhak mengikuti suatu praktikum bila nama mahasiswa yang bersangkutan terdaftar sebagai peserta praktikum. Mahasiswa yang tidak terdaftar sebagai peserta dilarang untuk mengikuti praktikum tersebut;
2. Hadir tepat waktu. Dosen dan asisten praktikum berhak mengeluarkan mahasiswa yang terlambat hadir;
3. Untuk praktikum tertentu yang diwajibkan untuk mengenakan jas praktikum, mahasiswa harus memenuhi ketentuan tersebut selama praktikum. Dosen dan/atau Asisten berhak tidak memperkenankan mahasiswa yang tidak mengenakan jas praktikum untuk masuk ke ruang laboratorium dan mengikuti praktikum;
4. Wajib memberitahu kepada dosen / asisten praktikum secara tertulis bila berhalangan hadir karena sakit atau hal lain yang mendesak (contoh : ada keluarga yang meninggal/sakit keras). Bila tidak ada pemberitahuan, dapat dianggap **TIDAK HADIR;**
5. Dilarang menandatangani daftar hadir atas nama orang lain. Apabila dosen/asisten dapat membuktikan bahwa mahasiswa tidak hadir dan daftar hadir ditanda tangani mahasiswa lain, maka sanksi akan diberikan kepada mahasiswa yang tidak hadir dan mahasiswa yang manandatangani;
6. Tidak diperbolehkan **merokok., membawa** makanan/minuman/senjata tajam selama berada di ruang **laboratorium;**
7. Wajib menjaga ketenanangan dan tidak membuat keributan selama praktikum berlangsung;
8. Wajib **menjaga kebersihan** dan **dilarang mencoret-coret perlengkapan** yang ada di ruang laboratorium (seperti meja gambar, dinding, peralatan lain-lain);
9. Harus duduk sesuai dengan nomor mesin/alat yang ditentukan oleh asisten atau dosen praktikum;

10. Mengikuti prosedur peminjaman alat sesuai dengan kebijakan yang ditetapkan masing-masing laboratorium;
11. Dilarang keras membawa pulang atau mengambil (secara sengaja atau tidak) peralatan dan perlengkapan yang ada di ruang praktikum. Apabila dapat dibuktikan bahwa seorang mahasiswa mengambil peralatan/perlengkapan laboratorium, maka dapat dikenakan sanksi akademik dan harus mengembalikan peralatan/perlengkapan tersebut;
12. Dilarang keras melakukan pengrusakan alat dan perlengkapan yang ada di laboratorium (baik secara sengaja maupun tidak). Apabila dapat dibuktikan bahwa seorang mahasiswa melakukan pengrusakan peralatan/perlengkapan laboratorium, maka mahasiswa Ybs. Wajib **mengganti** peralatan yang rusak tersebut;
13. Mengikuti arahan dan instruksi dari dosen dan asisten peraktikum dan bersikap sopan selama mengikuti praktikum;
14. Berpakaian rapi, sopan dan tidak diperkenankan memakai sandal;
15. Untuk kelas praktikum parallel, mahasiswa harus hadir sesuai dengan jadwal pada kelas yang ditentukan dan dilarang pindah jadual tanpa pemberitahuan dan izin dosen dan/atau aisten praktikum;
16. Meletakkan tas dan barang-barang lain milik mahasiswa yang tidak dibutuhkan selama praktikum pada rak atau tempat yang telah ditentukan;
17. Apabila ada keperluan untuk menggunakan laboratorium untuk keperluan insidential diluar jadual praktikum, harus melapor ke Jurusan/Kepala Laboratorium dan mengisi buku permohonan penggunaan laboratorium;
18. Bagi mahasiswa yang melanggar tata tertib praktikum dan penggunaan laboratorium di atas dapat dikenakan **SANKSI AKDADEMIK**.

Jakarta, 30 Agustus 2015

Koordinator Lab TI

TATA TERTIB PRAKTIKUM

PEMROGRAMAN KOMPUTER

Tata Tertib Praktikum untuk mata kuliah Pemrograman Komputer yang berlaku bagi mahasiswa praktikan adalah sebagai berikut :

1. Mengikuti seluruh praktikum Pemrograman Komputer dengan menggunakan *software* C++ yang telah ditetapkan.
2. Praktikan yang sudah terdaftar harus menepati waktu yang telah ditetapkan, baik untuk waktu kehadiran maupun waktu pengumpulan tugas serta laporan.
3. Menjaga kebersihan dan keamanan laboratorium komputer.
4. Praktikan bertanggung jawab mengganti perlengkapan atau inventaris yang ada pada laboratorium dalam waktu satu minggu.
5. Praktikan harus memakai pakaian berkerah dan sepatu selama mengikuti praktikum.
6. Praktikan harus mematikan segala bentuk alat komunikasi selama mengikuti praktikum berlangsung.
7. Praktikan tidak diperkenankan mengikuti praktikum bila terlambat lebih dari 45 menit.
8. Tidak diperkenankan untuk berpindah hari dan waktu shift praktikum tanpa seizin asisten.
9. Tidak diperkenankan untuk merokok, makan, minum dan melakukan perbuatan yang dianggap mengganggu jalannya praktikum.
10. Tidak diperkenankan keluar masuk laboratorium tanpa seizin asisten.
11. Tidak diperkenankan untuk berlaku tidak sopan kepada seluruh asisten.
12. Tidak diperkenankan melakukan berbagai bentuk kecurangan selama praktikum berlangsung.

MEKANISME PENILAIAN PRAKTIKUM

PEMROGRAMAN KOMPUTER

Mekanisme Penilaian Praktikum Pemrograman Komputer yang berlaku bagi mahasiswa praktikan adalah sebagai berikut :

1. Kehadiran : 10 %
2. Kuis : 25 %
3. Laporan : 35 %
4. Sidang Presentasi : 30 %

PERTEMUAN I

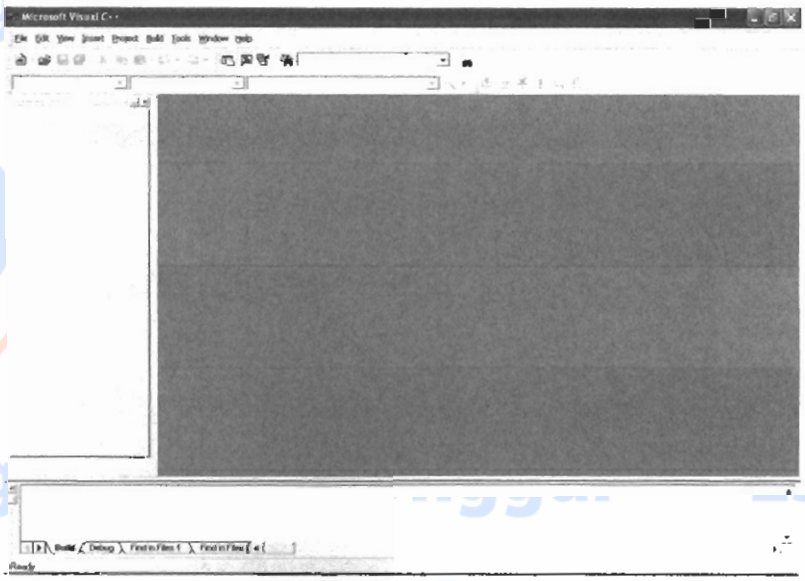
Pengenalan Struktur Program C++

Tujuan:

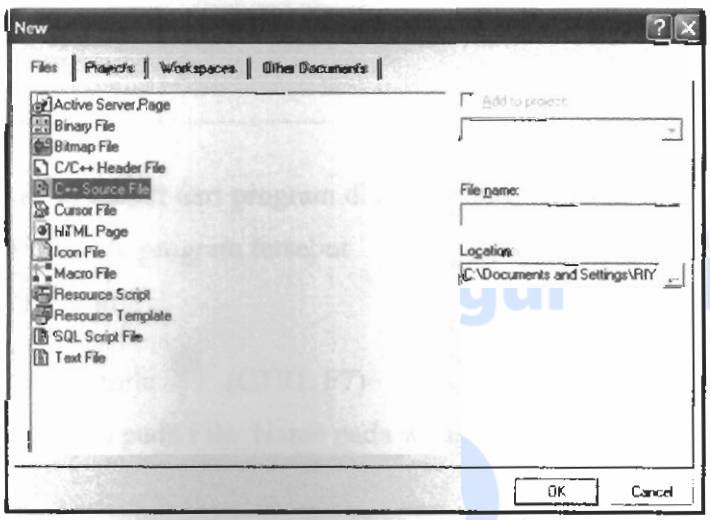
- Memperkenalkan mahasiswa tentang struktur sederhana dari pemrogramana C++

Memulai program C++

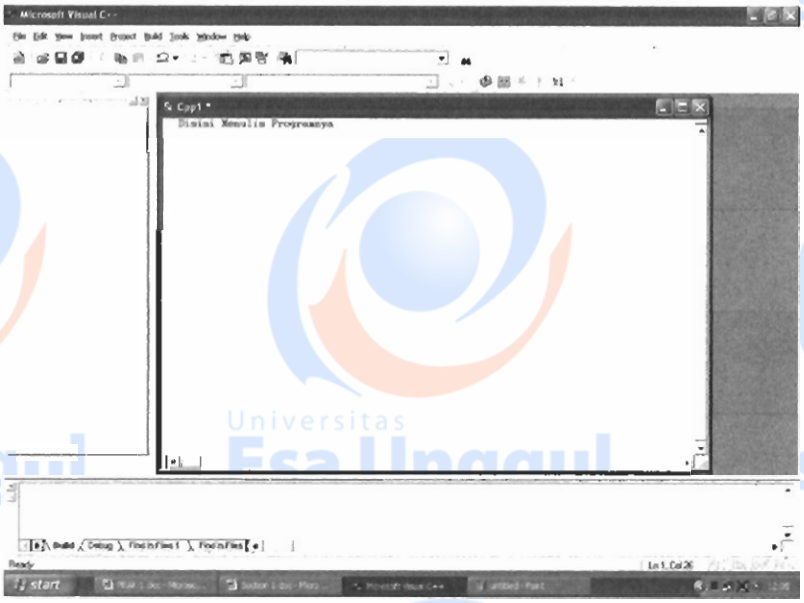
- Pilih Ms. Studio, klik Visual C++;
- Pilih New;



- Klik Files;



- Klik C++ Source Files;
- Mengetik pada lembar Kosong(Window CPP1*)



```
//Program Mencetak Nama

#include <iostream.h>

void main()
{
        cout<<"Riya Widayanti";
}
```

Untuk menghasilkan output dari program di atas berikut ini langkah-langkahnya:

Harus meng-compile program tersebut

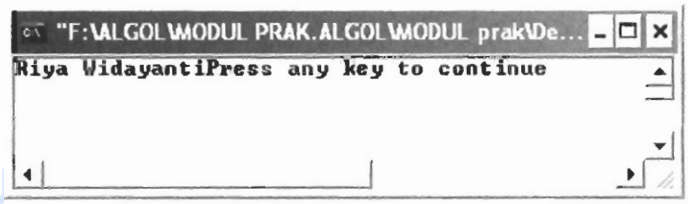
- Klik Build
- Klik Compile (CTRL F7)

  Beri nama pada File_Name pada window Save

Perhatikan window di configuration apakah ada kesalahan pada program Anda

- Klik Rebuild All
- Klik execute. ! (CTRL F5)

Output untuk program di atas:



Penjelasan Program:

//Program Mencetak Nama

- Disebut dengan baris komentar
- Dimulai dengan 2 tanda slash(//)
- Tidak mempengaruhi kerja program
- Dapat digunakan sebagai penjelasan dari statemen program.

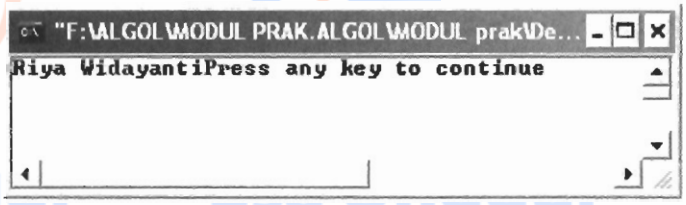
#include <iostream.h

- Kalimat yang dimulai dengan tanda kres (#) disebut perintah untuk preprocessor
- Bukan baris kode yang akan dieksekusi melainkan indikator untuk compiler
- Untuk program di atas, #include <iostream.h> memberitahu preprocessors untuk memasukan standar header iostream file. Spesifikasi file untuk deklarasi dasar input-output library in C++.

Perhatikan window di configuration apakah ada kesalahan pada program Anda

- Klik Rebuild All
- Klik execute. ! (CTRL F5)

Output untuk program di atas:



Penjelasan Program:

//Program Mencetak Nama

- Disebut dengan baris komentar
- Dimulai dengan 2 tanda slash(//)
- Tidak mempengaruhi kerja program
- Dapat digunakan sebagai penjelasan dari statemen program.

#include <iostream.h

- Kalimat yang dimulai dengan tanda kres (#) disebut perintah untuk preprocessor
- Bukan baris kode yang akan dieksekusi melainkan indikator untuk compiler
- Untuk program di atas, #include <iostream.h> memberitahu preprocessors untuk memasukan standar header iostream file. Spesifikasi file untuk deklarasi dasar input-output library in C++.

void main()

- Baris ini menyatakan deklarasi fungsi utama (**main**) dimulai.
- Fungsi utama ini akan dijalankan pertama kali setiap program C++ dieksekusi, dimana peletakannya bisa di akhir, tengah, ataupun di akhir.
- Dalam program C++ minimal memiliki fungsi **main**.
- **main** selalu diikuti dengan tanda kurung buka & kurung tutup/*paranthesis* "( )" karena ini adalah sebuah fungsi. Dalam C++ semua fungsi-fungsi selalu diikuti dengan *paranthesis*, dan bersifat optional untuk argument yang di dalamnya (penjelasan lebih lanjut untuk **FUNCTION**)
- **Isi/statemen** dari fungsi main diawali dengan **"{"** dan ditutup dengan **"}"**

cout<<"Riya Widayanti";

- Instruksi ini merupakan hal yang terpenting dalam program tersebut.
- Cout adalah standar output dalam C++ (tercetak dalam layar) dan kalimat ini ditulis diantara tanda petik.
- Setiap statement program **harus** diakhiri dengan karakter titik koma **";"**

Program di atas dapat juga ditulis sebagai berikut (dalam hal ini hanya satu baris saja):

```
//Program Mencetak Nama

#include <iostream.h> void main(){  cout<<"Riya Widayanti";}
```

Dalam C++ pemisahan instruksi diakhiri dengan titik dua untuk masing-masing baris.

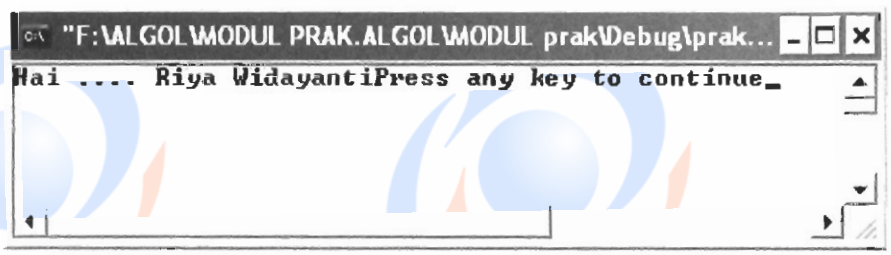
Di bawah ini contoh program dengan instruksi lebih dari satu.

```
//Program Mencetak Nama

#include <iostream.h>

void main()
{
        cout<<"Hai .... ";
        cout<<"Riya Widayanti";
}
```

Output untuk program di atas:



Kita dapat menuliskan instruksi seperti di bawah ini untuk program di atas:

```
//Program Mencetak Nama

#include <iostream.h>

void main()
{
        cout<<
          "Hai .... ";
        cout<<
      "Riya Widayanti";
}
```

Hasil dari program di atas sama dengan dengan program sebelumnya.

KOMENTAR

Komentar adalah bagian code yang tidak mempengaruhi kerja dari program. Tujuan pemberian komentar adalah untuk menyisipkan note atau penjelasan dari statemen program yang kita buat.

C++ mempunyai 2 cara untuk menyisipkan komentar:

//komentar

/* komentar*/

PERTEMUAN II

Variabel, Tipe Data dan Konstanta

Tujuan:

- Mahasiswa dapat membedakan variabel, konstanta dan tipe data serta diimplementasikan ke dalam bahasa C++.

Programming C++ tidak hanya sebatas pencetakan teks ke layar, lebif jauh dalam pertemuan ini kita akan membahas variabel. Untuk mengawali pembicaraan tersebut, perhatikan pernyataan berikut ini:

Saya meminta Anda untuk menyimpan angka 7 ke dalam memori Nada, dan kemudian saya meminta kembali untuk menyimpan angka 4. Anda baru saja menyimpan 2 nilai dalam memori Anda. Setelah itu saya meminta anda untuk menjumlahkan angka pertama dengan angka 2, dan Anda akan menyimpan angka 9 (yaitu 7 + 2) dan 4 di memori Anda. Untuk selanjutnya dilakukan proses pengurangan untuk kedua angka tersebut sebagai hasil.

Seluruh proses di atas dapat dibuat dengan sederhana oelh komputer dengan menggunakan variabel. Proses tersebut dapat diekspresikan ke dalam C++ dengan instruksi sebagai berikut:

```
bil1 = 7;
bil2 = 4;
bil1 = bil1 + 2;
hasil = bil1 – bil2;
```

Sudah sewajarnya bila kita menggunakan bilangan yang kecil, namun bila bilangan yang kita gunakan besar maka kita membutuhkan alat bantu untuk menghitungnya. Kita dapat mendefinisikan sebuah variabel dalam memori untuk menyimpan nilai. Setiap variabel membutuhkan sebuah *identifier* sebagai pembeda dari yang lain, contoh di atas adalah bil1, bil2 dan hasil, dalam hal ini kita bebas memberi nama, sepanjang *indentifier*-nya sesuai dengan aturan.

Identifier

- Terdiri dari satu atau lebih karakter,angka, atau simbol garis bawah "_"
- Panjang identifier tidak terbatas, meskipun ada beberapa compiler hanya menampung 32 karakter pertama saja
- Tidak bolch ada spasi
- Tidak diawali dengan angka
- Tidak menggunkan keyword pada C++
- Penting: Bahasa C++ adalah "Case sensitif", artinya identifier yang tertulis huruf kapital tidak sama dengan bila ditulis dengan huruf kecil. Contoh: hasil tidak sama dengan Hasil tidak sama dengan HASIL dan sebagainya.

Tipe Data

Ketika memprogram, kita menyimpan variabel dalam memori komputer, tetapi komputer harus tahu apa yang ingin kita simpan, apakah bilangan sederhana, karakter atau bilangan yang membutuhkan memori yang besar, dengan kata lain membutuhkan space yang berbeda-beda.

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**	**See short, long**

		Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes).	
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 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.	wide characters

Deklarasi Variable

Variabel yang akan digunakan dalam C++ harus di-deklarasikan dengan spesifikasi tipe data-nya.

Sintak pendeklarasian variabel:

type data identifier variabel

contoh:

int a;

float b;

Jika kita ingin mendeklarasikan variabel dengan tipe data yang sama, maka kita bisa mendeklarasikan sebagai berikut:

int a,b,c;

atau

int a;

int b;

int c;

```
//Program Mencari Keliling Persegi Panjang

#include <iostream.h>
void main()
{
        int panjang, lebar, kel; //deklarasi variabel
        panjang = 10 ;
        lebar = 5;
        kel = (panjang + lebar)*2;
        cout<<kel;
}
```

Definisi Variabel

```
//Program Mencari Keliling Persegi Panjang

#include <iostream.h>
void main()
{
        int panjang = 10, lebar = 5; //definisi variabel
        int kel; //deklarasi variabel

        kel = (panjang + lebar)*2;
        cout<<kel;
}
```

Konstanta

Sebuah konstanta adalah setiap ekspresi yang memiliki nilai tetap. Mereka dapat dibagi dalam Bilangan Integer, Floating-Point Bilangan, Karakter dan String.

Integer Numbers

```
1776
707
-273
```

Mereka di atas adalah konstanta numeric dimana merupakan bilangan bulat integer). UNtuk mengekspresikan sebuah konstanta numeric kita tidak perlu menuliskan iantara tanda petik (") tidak seperti pada penulisan karakter.

Floating Point Numbers

Mereka mengekspresikan angka dengan desimal dan / atau eksponen. Mereka apat mencakup titik desimal, karakter e (yang mengungkapkan "oleh sepuluh pada ketinggian Xth", di mana X adalah nilai integer berikut) atau keduanya.

3.14159 *// 3.14159*

6.02e23 *// 6.02×10^{23}*

1.6e-19 *// 1.6×10^{-19}*

3.0 *// 3.0*

Characters and strings

There also exist non-numerical constants, like:

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

Dua ekspresi yang pertama mewakili karakter tunggal, dan dua berikut mewakili string dari beberapa karakter. Perhatikan bahwa untuk mewakili satu karakter menggunakan di antara tanda kutip tunggal (') dan untuk mengekspresikan string lebih dari satu karakter ditulis antara tanda kutip ganda (").

Ketika menulis kedua karakter tunggal dan string karakter dengan cara yang konstan, maka perlu untuk menempatkan tanda kutip untuk membedakan mereka dari pengidentifikasi variabel mungkin atau kata-kata reserved.

Perhatikan ini:

```
x
'x'
```

`x` artinya variable `x`, sedangkan `'x'` artinya konstanta karakter `'x'`.

Karakter konstanta dan konstanta string memiliki kekhasan tertentu, seperti kode berbeda. Ini adalah karakter khusus yang tidak dapat dinyatakan sebaliknya dalam kode sumber dari program, seperti newline (\n) atau tab (\t). Semuanya diawali dengan garis miring terbalik (\). Di sini Anda memiliki daftar kode sendiri seperti:

`\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 (\)

For example:

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

***Defined constants* (#define)**

Anda dapat membuat konstanta sendiri yang akan sering dipakai tanpa harus membuat variabel, hanya dengan menggunakan # define direktif preprocessor. Berikut ini adalah format:

```
#define identifier value
```

For example:

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

They define three new constants. Once they are declared, you are able to use them in the rest of the code as any if they were any other constant, for example:

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

In fact the only thing that the compiler does when it finds **#define** directives is to replace literally any occurrence of the them (in the previous example, **PI**, **NEWLINE** or **WIDTH**) by the code to which they have been defined (**3.14159265**, **'\n'** and **100**, respectively). For this reason, #define constants are considered *macro constants*.

The #define directive is not a code instruction, it is a directive for the preprocessor, therefore it assumes the whole line as the directive and does not require a semicolon (;) at the end of it. If you include a semicolon character (;) at

the end, it will also be added when the preprocessor will substitute any occurence of the defined constant within the body of the program.

Declared Constants (const)

With the **const** prefix you can declare constants with a specific type exactly as you would do with a variable:

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

In case that the type was not specified (as in the last example) the compiler assumes that it is type **int**.

SOAL !

1. **Buatlah program yang mengkonversi Fahrenhiet ke Celcius dan Reamur !**
2. **Buatlah program untuk mencari rata-rata dari 3 bilangan yang diinputkan !**

PERTEMUAN III

Operator

Tujuan:

- Setelah belajar variabel dan konstanta, pada pertemuan kita belajar mengoperasikan variabel dan konstanta tersebut.

Operator dalam C++ adalah:

Assignation (=) tanda "sama dengan"

Operator "sama dengan " digunakan untuk pemberian nilai ke variabel.

```
a = 5;
```

Artinya memberi nilai 5 untuk variabel 10 atau variabel 10 diberi nilai 5.

Bagian kiri tanda "=" adalah sebuah variabel dimana akan: (*rigth to left rule*)

- menampung suatu konstanta,

```
a = 5;
```

- menampung variabel

```
a = b;
```

- hasil operasi.

```
a = b * 2;
```

Contoh:

```
int a, b;
a = 10;
b = 4;
a = b;
b = 7;
```

C++ merupakan bahasa pemrograman yang dapat mengoperasikan penulisan sebagai berikut:

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

Artinya sama dengan :

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

Operator Aritmatik

+ penambahan

- pengurangan

*** perkalian**

/ pembagian

% module

4 operator pertama Anda sudah mengenal sejak bangku sekolah dasar, mungkin untuk operator ketiga Anda cukup asing. Operator "module" dimana simbolnya adalah percent (%), operator ini merupakan operator sisa hasil bagi.

Contoh:

D = 13 % 3;

Variabel D berisi nilai 1, dimana 1 sisa dari pembagian 13 dengan 3.

Compound Assignation Operator

+=, -=, *=, /=, %=, >>=, <<=, ^=, |=

Contoh:

a - = 8; artinya sama dengan a = a - 8;

a /= b; artinya sama dengan a = a / b;

harga * = unit + 1; artinya sama dengan harga = harga * (unit +1);

Increase(++) dan Decrease(--)OPERATOR

Kenaikan dan penurunan satu pada suatu bariabel;

a++;

a+=1;

a = a + 1;

Statemen di atas mempunyai arti sama yaitu menambah satu pada variabel a. Operator diatas dalam dituliskan didepan (prefix) seperti ++a atau dibelakang seperti a++. A characteristic of this operator is that it can be used both as a *prefix* or as a *suffix*. That means it can be written before the variable identifier (**++a**) or after (**a++**). Although in simple expressions like **a++** or **++a** they have exactly the same meaning, in other operations in which the result of the *increase* or *decrease* operation is evaluated as another expression they may have an important difference

in their meaning: In case that the increase operator is used as a *prefix* (**++a**) the value is increased before the expression is evaluated and therefore the increased value is considered in the expression; in case that it is used as a *suffix* (**a++**) the value stored in **a** is increased after being evaluated and therefore the value stored before the increase operation is evaluated in the expression. Notice the difference:

Example 1	**Example 2**
B=3;	**B=3;**
A=++B;	**A=B++;**
// A is 4, B is 4	*// A is 3, B is 4*

In Example 1, **B** is increased before its value is copied to **A**. While in Example 2, the value of **B** is copied to **A** and **B** is later increased.

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

Relational operators digunakan untuk mengevaluasi sebuah perbandiangn 2 ekspresi . Hasil dari operasi tersebut adalah a **bool** value yang dapat bernilai **true** or **false**.

==	**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**.

Tentunya tidak hanya digunakan untuk konstanta numeric saja, namun dapat juga digunakan pada variable seperti, dibawah ini yang artinya tidak sama dengan **a=2**, **b=3** and **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**.

Be aware. Operator = (one equal sign) is not the same as operator == (two equal signs), the first is an assignation operator (assigns the right side of the expression to the variable in the left) and the other (==) is a relational operator of equality that compares whether both expressions in the two sides of the operator are equal to each other. Thus, in the last expression ((b=2) == a), we first assigned the value **2** to **b** and then we compared it to **a**, that also stores value **2**, so the result of the operation is **true**.

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

Operator **!** is equivalent to boolean operation NOT, it has only **one operand**, located at its right, and the only thing that it does is to invert the value of it, producing **false** if its operand is **true** and **true** if its operand is **false**. It is like saying that it returns the opposite result of evaluating its operand. For example:

!(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**.

Logic operators **&&** and **||** are used when evaluating two expressions to obtain a single result. They correspond with boolean logic operations *AND* and *OR* respectively. The result of them depends on the relation between its **two operands**:

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**

For example:

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

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

Conditional Operator (?).

The conditional operator evaluates an expression and returns a different value according to the evaluated expression, depending on whether it is *true* or *false*. Its format is:

condition ? result1 : result2

if *condition* is **true** the expression will return *result1*, if not it will return *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**.

sizeof()

This operator accepts one parameter, that can be either a variable type or a variable itself and returns the **size** in bytes of that type or object:

a = sizeof (char). This will return **1** to **a** because **char** is a one byte long type. The value returned by **sizeof** is a constant, so it is always determined before program execution.

Other operators

Later in the tutorial we will see a few more operators, like the ones referring to pointers or the specifics for object-oriented programming. Each one is treated in its respective section.

Priority of operators

When making complex expressions with several operands, we may have some doubts about which operand is evaluated first and which later. For example, in this expression:

a = 5 + 7 % 2

we may doubt if it really means:

a = 5 + (7 % 2) with result 6, or

a = (5 + 7) % 2 with result 0

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

The correct answer is the first of the two expressions, with a result of **6**. There is an established order with the priority of each operator, and not only the arithmetic ones (those whose preference we may already know from mathematics) but for all the operators which can appear in C++. From greatest to lowest priority, the priority order is as follows above.

Associativity defines -in the case that there are several operators of the same priority level- which one must be evaluated first, the rightmost one or the leftmost one. All these precedence levels for operators can be manipulated or become more legible using parenthesis signs (and), as in this example:

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

might be written as:

```
a = 5 + (7 % 2);
```

or

```
a = (5 + 7) % 2;
```

according to the operation that we wanted to perform.

So if you want to write a complicated expression and you are not sure of the precedence levels, always include parenthesis. It will probably also be more legible code.

SOAL !

1. **Buatlah program untuk menghasilkan output ukuran dari masing-masing tipe data?**
2. **Buatlah program penginputan 2 nilai, untuk selanjutnya dilakuakn operasi aritmatika.**

PERTEMUAN IV

Communication Through Console

Tujuan:

- Memperkenalkan mahasiswa tentang fungsi pencetakan dan penginputan dalam C++

The *console* is the basic interface of computers, normally it is the set composed of the keyboard and the screen. The keyboard is generally the standard *input* device and the screen the standard *output* device. In the *iostream* C++ library, standard *input* and *output* operations for a program are supported by two data streams: **cin** for input and **cout** for output. Additionally, **cerr** and **clog** have also been implemented - these are two output streams specially designed to show error messages. They can be redirected to the standard output or to a log file.

Therefore **cout** (the standard output stream) is normally directed to the screen and **cin** (the standard input stream) is normally assigned to the keyboard. By handling these two streams you will be able to interact with the user in your programs since you will be able to show messages on the screen and receive his/her input from the keyboard.

Output (cout)

The **cout** stream is used in conjunction with the overloaded operator << (a pair of *"less than"* signs).

```
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
```

The << operator is known as *insertion operator* since it inserts the data that follows it into the stream that precedes it. In the examples above it inserted the constant string Output sentence, the numerical constant 120 and the variable x into the output stream **cout**. Notice that the first of the two sentences is enclosed between double quotes (") because it is a string of characters. Whenever we want to use constant strings of characters we must enclose them between double quotes

(") so that they can be clearly distinguished from variables. For example, these two sentences are very different:

cout << "Hello"; ***// prints Hello on screen***

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

The *insertion* operator (<<) may be used more than once in a same sentence: cout << "Hello, " << "I am " << "a C++ sentence". This last sentence would print the message **Hello, I am a C++ sentence** on the screen. The utility of repeating the insertion operator (<<) is demonstrated when we want to print out a combination of variables and constants or more than one variable:

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

If we supose that variable age contains the number 24 and the variable zipcode contains 90064 the output of the previous sentence would be:

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

It is important to notice that **cout** does not add a line break after its output unless we explicitly indicate it, therefore, the following sentences:

cout << "This is a sentence.";

cout << "This is another sentence.";

will be shown followed in screen:

This is a sentence.This is another sentence. Even if we have written them in two different calls to **cout**. So, in order to perform a line break on output we must explicitly order it by inserting a new-line character, that in C++ can be written as **\n**:

cout << "First sentence.\n ";

cout << "Second sentence.\nThird sentence.";

Produces the following output:

First sentence

Second sentence

Third sentence

Additionally, to add a new-line, you may also use the **endl** manipulator. For example:

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

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

Would print out:

First sentence

Second sentence

The **endl** manipulator has a special behavior when it is used with buffered streams: they are flushed. But anyway **cout** is unbuffered by default. You may use either the **\n** escape character or the **endl** manipulator in order to specify a line jump to **cout**. Notice the differences of use shown earlier.

Input (cin)

Handling the standard input in C++ is done by applying the overloaded operator of *extraction* (>>) on the **cin** stream. This must be followed by the variable that will store the data that is going to be read. For example:

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

declares the variable age as an int and then waits for an input from cin (keyborad) in order to store it in this integer variable. **cin** can only process the input from the keyboard once the RETURN key has been pressed. Therefore, even if you request a single character **cin** will not process the input until the user presses RETURN once the character has been introduced. You must always consider the *type* of the variable that you are using as a container with **cin** extraction. If you request an integer you will get an integer, if you request a character you will get a character and if you request a string of characters you will get a string of characters.

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

int main ()
{
 int i;
 cout << "Please enter an integer value: ";
 cin >> i;
```

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

```
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
```

The user of a program may be one of the reasons that provoke errors even in the simplest programs that use **cin** (like the one we have just seen). Since if you request an integer value and the user introduces a name (which is a string of characters), the result may cause your program to misoperate since it is not what we were expecting from the user. So when you use the data input provided by **cin** you will have to trust that the user of your program will be totally cooperative and that he will not introduce his name when an interger value is requested. Farther ahead, when we will see how to use strings of characters we will see possible solutions for the errors that can be caused by this type of user input. You can also use **cin** to request more than one datum input from the user:

cin >> a >> b; is equivalent to: **cin>>a; cin >> b;**

In both cases the user must give two data, one for variable **a** and another for variable **b** that may be separated by any valid blank separator: a space, a tab character or a newline.

SOAL !

1. Menghitung luas permukaan serta isi Balok yang diketahui 3 rusuknya.
   Diketahui rumus:
   L = 2(SB+AC+BC)
   I = ABC
   Ket:
   L = Luas
   A, B, C = Rusuk Balok
   I = Sisi Balok

2. Hitung luas permukaan serta isi bola yang diketahui jari-jarinya.

   Diketahui rumus:

   $L = 4 \Pi R^2$

   $I = (4 \Pi R^3)/ 3$

   Ket:

   L= Luas

   R= Jari-jari

   I= Isi Bola

PERTEMUAN V
Selection Control Structure

Tujuan:

- Memperkenalkan mahasiswa tentang Struktur dasar pemrograman *selection*

Sebuah program tidak selalu terdiri dari instruksi terurut(sequence). Suatu proses dapat juga bercabang ataupun berulang. Konsep baru yang akan diperkenalkan adalah the ***block of instructions***. A block of instructions adalah kelompok instruksi yang dipisahkan oleh tanda titik koma (;) tetapi dibatasi oleh tanda kurung kurawal terbuka dan tertutup: { and }. Tapi bila block of instruction terdiri dari satu statement/instruksi maka boleh tidak menggunakan tanda kurung kurawal terbuka dan tertutup: { and }.

Conditional Structure: *If* And *Else*

It is used to execute an instruction or block of instructions only if a condition is fulfilled. Its form is:

if (*condition*) *statement*

Dimana *condition* merupakan ekspresi yang akan dievaluasi. Jika kondisi bernilai true (memenuhi) makan, ***statement*** akan dieksekusi/dijalankan, namun juka bernilai false (salah), ***statement*** tersebut tidak akan dikerjakan dan program berlanjut ke instruksi/statemen selanjutnya.

Contoh :

For example, the following code fragment prints out **x is 100** only if the value stored in variable **x** is indeed 100:

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

If we want more than a single instruction to be executed in case that *condition* is **true** we can specify a *block of instructions* using curly brackets { }:

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

We can additionally specify what we want that happens if the condition is not fulfilled by using the keyword *else*. Its form used in conjunction with **if** is:

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

For example:

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

prints out on the screen **x is 100** if indeed x is worth 100, but if it is not -and only if not- it prints out **x is not 100**.

The *if* + *else* structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the present value stored in **x** is positive, negative or none of the previous, that is to say, equal to zero.

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

Remember that in case we want more than a single instruction to be executed, we must group them in a *block of instructions* by using curly brackets { }.

The selective Structure: switch.

The syntax of the *switch* instruction is a bit peculiar. Its objective is to check several possible constant values for an expression, something similar to what we did at the beginning of this section with the linking of several *if* and *else if* sentences. Its form is the following:

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

It works in the following way: **switch** evaluates *expression* and checks if it is equivalent to *constant1*, if it is, it executes *block of instructions 1* until it finds the **break** keyword, then the program will jump to the end of the *switch* selective structure.

If *expression* was not equal to *constant1* it will check if *expression* is equivalent to *constant2*. If it is, it will execute *block of instructions 2* until it finds the **break** keyword.

Finally, if the value of *expression* has not matched any of the previously specified constants (you may specify as many **case** sentences as values you want to check), the program will execute the instructions included in the **default:** section, if this one exists, since it is optional.

Both of the following code fragments are equivalent:

***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";
 }
```

I have commented before that the syntax of the ***switch*** instruction is a bit peculiar. Notice the inclusion of the **break** instructions at the end of each block. This is necessary because if, for example, we did not include it after *block of instructions 1* the program would not jump to the end of the switch selective block (}) and it would continue executing the rest of the blocks of instructions until the first appearance of the **break** instruction or the end of the switch selective block. This makes it unnecessary to include curly brackets { } in each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression evaluated. For example:

```
switch (x) {
 case 1:
 case 2:
 case 3:
  cout << "x is 1, 2 or 3";
  break;
 default:
  cout << "x is not 1, 2 nor 3";
}
```

Notice that **switch** can only be used to compare an expression with different constants. Therefore we cannot put variables (**case (n*2):**) or ranges (**case (1..3):**) because they are not valid constants. If you need to check ranges or values that are not constants use a concatenation of **if** and **else if** sentences.

SOAL !

1. Dalam Penilaian pegawai untuk naik pangkat ditetapkan salah satu yang dinilai adalah kemampuan membuat karya ilmiah. Sebagai pengarang buku mendapatkan kum 3, sebagai pengarang diktat mendapatkan kum 2 dan pengarang paper mendapatkan kum 1. Seseorang dapat naik pangkat bila kumnya lebih dari atau sama dengan 10, dalam pertimbangan bila kumya 7 sampai dengan 9, kurang dari 7 tidak naik pangkat. Bila hasil yang diinginkan adalah Nama, Kum yang dicapai dan Keterangan Kum.
2. Suatu perusahaan XYZ memberikan kredit dengan berdasarkan penghasilan pemohon. Cara Penilaiannya adalah: Pendapatan tetap dihitung penuh, pendapatan tambahan dihitung setengah dan pendapatan keluarga (Istri/Suami0 dihitung sepertiga. Apabila jumlah penghasilan lebih atau sama dengan Rp. 3.000.000, - maka mendapatkan Kredit Rumah, kurang dari nilai tersebut tetapi masih lebih besar dari Rp. 1.500.000,- mendapatkan kredit Kendaraan, selain itu tidak mendapatkan Kredit. Bila hasil yang ingin dikeluarkan adalah Nama, Penghasilan Total, Keterangan Kredit.
3. Perusahaan parker "AMAN dan MURAH" menetapkan bahwa setiap parker kendaraan dikenakan tarip sebagai berikut:
   - Pajak Rp. 250, -;
   - Bila Kendaraan MOBIL diberi kode 1 dengan tarif Rp. 750 per jam minimal parkir 2 jam;
   - Bila kendaraan MOTOR diberi kode 2 dengan tarif Rp. 250 per jam minimal parkir 2 jam;
   - Untuk kendaraan MOBIL dan MOTOR yang parkir lebih dari 4 jam mendapat diskon 10% dari tarif normal per jam tambahannya dan bila lama parkir lebih dari 10 jam mendapat diskon 20% dari tarif normal per jam tambahan;
   - Buat diagram alir dan pemrogramannya dalam bahasa C++ dengan output Jenis Kendaraan, Lama Parkir dan Total bayar.

PERTEMUAN VI

Repeation Control Structure

Tujuan:

- Memperkenalkan mahasiswa tentang Struktur dasar pemrograman Repeation

Repetitive structures or loops

Loops have as objective to repeat a *statement* a certain number of times or while a condition is fulfilled.

The *while* loop.

Its format is: **while (***expression***)** *statement* and its function is simply to repeat *statement* while *expression* is true. For example, we are going to make a program to count down using a *while* loop:

```
// 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;
}
```

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

When the program starts the user is prompted to insert a starting number for the countdown. Then the *while* loop begins, if the value entered by the user fulfills the condition **n>0** (that **n** be greater than **0**), the block of instructions that follows will execute an indefinite number of times while the condition **(n>0)** remains true.

All the process in the program above can be interpreted according to the following script: beginning in **main**:

1. User assigns a value to **n**;
2. The while instruction checks if **(n>0)**. At this point there are two possibilities:
   a. **true:** execute *statement* (step **3**,);
   b. **false:** jump *statement*. The program follows in step **5**;
3. Execute *statement*:
   cout<<n<<",";
   --n;
   (prints out **n** on screen and decreases **n** by 1);
4. End of block. Return Automatically to step 2;
5. Continue the program after the block: print out **FIRE!** and end of program.

We must consider that the loop has to end at some point, therefore, within the block of instructions (loop's *statement*) we must provide some method that forces *condition* to become false at some moment, otherwise the loop will continue looping forever. In this case we have included **--n;** that causes the *condition* to become **false** after some loop repetitions: when **n** becomes **0**, that is where our countdown ends.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without practical delay between numbers.

The *do-while* loop

Format:

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

Its functionality is exactly the same as the *while* loop except that *condition* in the *do-while* is evaluated after the execution of *statement* instead of before, granting at least one execution of *statement* even if *condition* is never fulfilled. For example, the following program echoes any number you enter until you enter 0.

```
// 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;
}
```

```
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 *do-while* loop is usually used when the condition that has to determine its end is determined within the loop statement, like in the previous case, where the user input within the block of intructions is what determines the end of the loop. If you never enter the **0** value in the previous example the loop will never end.

The *for* loop

Its format is: **for** (*initialization*; *condition*; *increase*) *statement*; and its main function is to repeat *statement* while *condition* remains true, like the *while* loop. But in addition, **for** provides places to specify an *initialization* instruction and an *increase* instruction. So this loop is specially designed to perform a repetitive action with a counter.

It works the following way:

1. *initialization* is executed. Generally it is an initial value setting for a counter varible. This is executed only once;
2. *condition* is checked, if it is **true** the loop continues, otherwise the loop finishes and *statement* is skipped;
3. *statement* is executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets { };
4. finally, whatever is specified in the *increase* field is executed and the loop gets back to step 2.

Here is an example of countdown using a *for* loop.

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

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

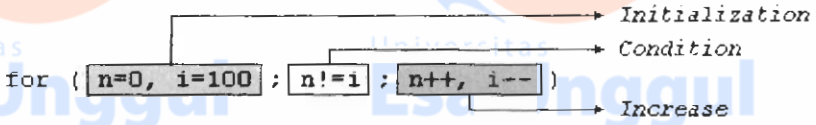
The *initialization* and *increase* fields are optional. They can be avoided but not the semicolon signs among them. For example we could write: **for (;n<10;)** if we want to specify no *initialization* and no *increase*; or **for (;n<10;n++)** if we want to include an *increase* field but not an *initialization*.

Optionally, using the comma operator (,) we can specify more than one instruction in any of the fields included in a **for** loop, like in *initialization*, for example. The comma operator (,) is an instruction separator, it serves to separate more than one instruction where only one instruction is generally expected.

For example, suppose that we wanted to intialize more than one variable in our loop:

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

This loop will execute 50 times if neither **n** nor **i** are modified within the loop:



n starts with **0** and **i** with **100**, the condition is **(n!=i)** (that **n** be not equal to **i**). Beacuse **n** is increased by one and **i** decreased by one, the loop's condition will become false after the 50th loop, when both **n** and **i** will be equal to 50.

Bifurcation of control and jumps.

The break instruction

Using *break* we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before it naturally finishes (an engine failure maybe):

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

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

```
      break;
    }
  }
  return 0;
}
```

The continue instruction

The *continue* instruction causes the program to skip the rest of the loop in the present iteration as if the end of the *statement* block would have been reached, causing it to jump to the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// 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;
}
```

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

The *go to* instruction.

It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation. The destination point is identified by a label, which is then used as an argument for the goto instruction. A label is made of a valid identifier followed by a colon (:).

This instruction does not have a concrete utility in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using **goto**:

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

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

The exit function

exit is a function defined in **cstdlib** (stdlib.h) library. The purpose of *exit* is to terminate the running program with an specific exit code. Its prototype is: **void exit (int** *exit code***)**. The *exit code* is used by some operating systems and may be used by calling programs. By convention, an *exit code* of 0 means that the program finished normally and any other value means an error happened.

SOAL !

1. Buatlah program untuk menyelesaikan deret di bawah ini beserta jumlahnya

10 + 8 + 6 + 4 + 2 = ?

2. OUTPUT

3 + 2 + 1 = ?

2 + 1 = ?

1 = ?

PERTEMUAN VII

FUNCTION

Using functions we can structure our programs in a more modular way, accessing all the potential that structured programming in C++ can offer us. A function is a block of instructions that is executed when it is called from some other point of the program. The following is its format:

type name **(** ***argument1*****,** ***argument2*****, ...)** ***statement*** where:

- ***type*** is the type of data returned by the function;
- ***name*** is the name by which it will be possible to call the function;
- ***arguments*** (as many as wanted can be specified). Each argument consists of a type of data followed by its identifier, like in a variable declaration (for example, int x) and which acts within the function like any other variable. They allow passing parameters to the function when it is called. The different parameters are separated by commas;
- ***statement*** is the function's body. It can be a single instruction or a block of instructions. In the latter case it must be delimited by curly brackets {}.

Here you have the first function example:

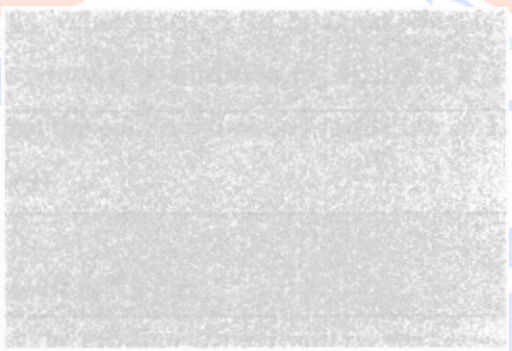
```
// function example
#include <iostream.h>

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

int main ()
```

The result is 8

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



In order to examine this code, first of all remember something said at the beginning of this tutorial: a C++ program always begins its execution with the **main** function. So we will begin there. We can see how the **main** function begins by declaring the variable **z** of type **int**. Right after that we see a call to **addition** function. If we pay attention we will be able to see the similarity between the structure of the call to the function and the declaration of the function itself in the code lines above:

The parameters have a clear correspondence. Within the **main** function we called to **addition** passing two values: **5** and **3** that correspond to the **int a** and **int b** parameters declared for the function **addition**. At the moment at which the function is called from **main**, control is lost by **main** and passed to function **addition**. The value of both parameters passed in the call (**5** and **3**) are copied to the local variables **int a** and **int b** within the function.

Function **addition** declares a new variable (**int r;**), and by means of the expression **r=a+b;**, it assigns to **r** the result of **a** plus **b**. Because the passed parameters for **a** and **b** are **5** and **3** respectively, the result is **8**. The following line of code is **return (r);** finalizes function **addition**, and returns the control back to the function that called it (**main**) following the program from the same point at which it was interrupted by the call to **addition**. But additionally, **return** was called with the content of variable **r** (**return (r);**), which at that moment was 8, so this value is said to be **returned** by the function.

The value returned by a function is the value given to the function when it is evaluated. Therefore, **z** will store the value returned by **addition (5, 3)**, that is **8**. To explain it another way, you can imagine that the call to a function (**addition (5,3)**) is literally replaced by the value it returns (**8**).

The following line of code in **main** is:

cout << "The result is " << z;

that, as you may already suppose, produces the printing of the result on the screen.

Scope of variables [re]

You must consider that the scope of variables declared within a function or any other block of instructions is only their own function or their own block of instructions and cannot be used outside of them. For example, in the previous example it had been impossible to use the variables **a, b** or **r** directly in function **main** since they were local variables to function **addition**. Also, it had been impossible to use the variable **z** directly within function **addition**, since this was a local variable to the function **main**.

Therefore, the scope of local variables is limited to the same nesting level in which they are declared. Nevertheless you can also declare global variables that are visible from any point of the code, inside and outside any function. In order to declare global variables you must do it outside any function or block of instructions, that means, directly in the body of the program.

And here is another example about functions:

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

int subtraction (int a, int b)
{
 int r;
```

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

```
  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;
}
```

In this case we have created the function **subtraction**. The only thing that this function does is to subtract both passed parameters and to return the result. Nevertheless, if we examine the function **main** we will see that we have made several calls to function **subtraction**. We have used some different calling methods so that you see other ways or moments when a function can be called.

In order to understand well these examples you must consider once again that a call to a function could be perfectly replaced by its return value.

For example the first case (that you should already know beacause it is the same pattern that we have used in previous examples):

z = subtraction (7,2);

cout << "The first result is " << z;

If we replace the function call by its result (that is **5**), we would have:

z = 5;

cout << "The first result is " << z;

as well as

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

has the same result as the previous call, but in this case we made the call to subtraction directly as a parameter for cout. Simply imagine that we had written:

cout << "The second result is " << 5;

since **5** is the result of **subtraction (7,2)**.

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

The only new thing that we introduced is that the parameters of **subtraction** are variables instead of constants. That is perfectly valid. In this case the values passed to the function **subtraction** are the values of **x** and **y**, that are **5** and **3** respectively, giving **2** as result.

The fourth case is more of the same. Simply note that instead of:

z = 4 + subtraction (x,y);

we could bave put:

z = subtraction (x,y) + 4;

with exactly the same result. Notice that the semicolon sign (**;**) goes at the end of the whole expression. It does not necessarily have to go right after the function call. The explanation might be once again that you imagine that a function can be replaced by its result:

z = 4 + 2;

z = 2 + 4;

***Functions with no types. The use of* void.**

If you remember the syntax of a function declaration:

type name (argument1, argument2 ...) statement

you will see that it is obligatory that this declaration begins with a ***type***, that is the type of the data that will be returned by the function with the **return** instruction. But what if we want to return no value?

Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value, moreover, we do not need it to receive any parameters. For these cases, the **void** type was devised in the C language. Take a look at:

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

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

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

I'm a function!

Although in C++ it is not necessary to specify **void**, its use is considered suitable to signify that it is a function without parameters or arguments and not something else. What you must always be aware of is that the format for calling a function includes specifing its name and enclosing the arguments between parenthesis. The non-existence of arguments does not exempt us from the obligation to use parenthesis. For that reason the call to **dummy function** is **dummyfunction ();**

This clearly indicates that it is a call to a function and not the name of a variable or anything else.

SOAL !

1. Perhitungan mencari:

MENU:

A. LUAS/KELILING PERSEGI PANJANG
B. LUAS/KELILING LINKARAN
C. LUAS/KELILING BUJUR SANGKAR

PILIH MENU:

Dengan menggunakan fungsi dengan pengembalian

PERTEMUAN VIII

FUNCTION LANJUT

Tujuan :

- Membantu dalam memahami fungsi dengan membawa nilai serta peenrapan fungsi dalam rekursif

***Arguments passed* by value *and* by reference.**

Until now, in all the functions we have seen, the parameters passed to the functions have been passed *by value*. This means that when calling a function with parameters, what we have passed to the function were values but never the specified variables themselves. For example, suppose that we called our first function **addition** using the following code :

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

What we did in this case was to call function **addition** passing the values of **x** and **y**, that means **5** and **3** respectively, not the variables themselves.

This way, when function **addition** is being called the value of its variables **a** and **b** become **5** and **3** respectively, but any modification of **a** or **b** within the function **addition** will not affect the values of **x** and **y** outside it, because variables **x** and **y** were not passed themselves to the the function, only their values.

But there might be some cases where you need to manipulate from inside a function the value of an external variable. For that purpose we have to use *arguments passed by reference*, as in the function **duplicate** of the following example:

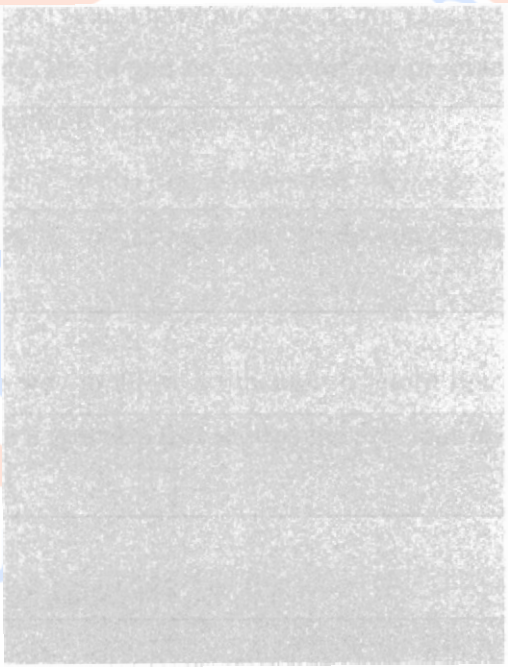
```
// passing parameters by reference
#include <iostream.h>

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

```
x=2, y=6, z=14
```

```
  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;
}
```



The first thing that should call your attention is that in the declaration of **duplicate** the type of each argument was followed by an *ampersand* sign (**&**), that serves to specify that the variable has to be passed *by reference* instead of *by value*, as usual. When passing a variable *by reference* we are passing the variable itself and any modification that we do to that parameter within the function will have effect in the passed variable outside it.

To express it another way, we have associated **a**, **b** and **c** with the parameters used when calling the function (**x**, **y** and **z**) and any change that we do on **a** within the function will affect the value of **x** outside. Any change that we do on **b** will affect **y**, and the same with **c** and **z**. That is why our program's output, that shows the values stored in **x**, **y** and **z** after the call to **duplicate**, shows the values of the three variables of **main** doubled. If when declaring the following function:

void duplicate (int& a, int& b, int& c)

we had declared it thus:

void duplicate (int a, int b, int c)

that is, without the *ampersand* (**&**) signs, we would have not passed the variables *by reference*, but their values, and therefore, the output on screen for our program would have been the values of **x**, **y** and **z** without having been modified.

This type of declaration "*by reference*" using the *ampersand* (**&**) sign is exclusive of C++. In C language we had to use pointers to do something equivalent.

Passing by reference is an effective way to allow a function to return more than one single value. For example, here is a function that returns the previous and next numbers of the first parameter passed.

```
// 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;
}
```

Previous=99, Next=101

Default values in arguments.

When declaring a function we can specify a default value for each parameter. This value will be used if that parameter is left blank when calling to the function. To do that we simply have to assign a value to the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is stepped on and the passed value is used. For example:

```
// 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;
}
```

```
6
5
```

As we can see in the body of the program there are two calls to the function **divide**. In the first one:

divide (12)

We have only specified one argument, but the function **divide** allows up to two. So the function **divide** has assumed that the second parameter is **2** since that is what we have specified to happen if this parameter is lacking (notice the function

declaration, which finishes with **int b=2**). Therefore the result of this function call is **6** (**12/2**). In the second call :

divide (20,4)

there are two parameters, so the default assignation (int b=2) is stepped on by the passed parameter, that is **4**, making the result equal to **5** (**20/4**).

Overloaded functions.

Two different functions can have the same name if the prototype of their arguments are different, that means that you can give the same name to more than one function if they have either a different number of arguments or different types in their arguments. For example,

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

```
2
2.5
```

In this case we have defined two functions with the same name, but one of them accepts two arguments of type **int** and the other accepts them of type **float**. The compiler knows which one to call in each case by examining the types when the function is called. If it is called with two **int**s as arguments it calls to the function that has two **int** arguments in the prototype and if it is called with two **float**s it will call to the one which has two **float**s in its prototype. For simplicity I have included the same code within both functions, but this is not compulsory. You can make two function with the same name but with completely different behaviors.

inline ***functions.***

The *inline* directive can be included before a function declaration to specify that the function must be compiled as code at the same point where it is called. This is equivalent to declaring a macro. Its advantage is only appreciated in very short functions, in which the resulting code from compiling the program may be faster if the overhead of calling a function (stacking of arguments) is avoided. The format for its declaration is:

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

and the call is just like the call to any other function. It is not necessary to include the *inline* keyword before each call, only in the declaration.

Recursivity.

Recursivity is the property that functions have to be called by themselves. It is useful for tasks such as some sorting methods or to calculate the factorial of a number. For example, to obtain the factorial of a number (n) its mathematical formula is:

$n! = n * (n-1) * (n-2) * (n-3) ... * 1$

more concretely, 5! (factorial of 5) would be:

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

and a recursive function to do that could be this:

```
// 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;
}
```

```
Type a number: 9
!9 = 362880
```

Notice how in function **factorial** we included a call to itself, but only if the argument is greater than **1**, since otherwise the function would perform *an infinite recursive loop* in which once it arrived at **0** it would continue multiplying by all the negative numbers (probably provoking a stack overflow error on runtime). This function has a limitation because of the *data type* used in its design (long) for more simplicity. In a standard system, the type **long** would not allow storing factorials greater than **12!**.

Prototyping functions.

Until now, we have defined the all of the functions before the first appearance of calls to them, that generally was in **main**, leaving the function **main** for the end.

If you try to repeat some of the examples of functions described so far, but placing the function **main** before any other function that is called from within it, you will most likely obtain an error. The reason is that to be able to call a function it must have been declared previously (it must be known), like we have done in all our examples.

But there is an alternative way to avoid writing all the code of all functions before they can be used in **main** or in another function. It is by *prototyping functions*. This consists in making a previous shorter, but quite significant, declaration of the complete definition so that the compiler can know the arguments and the return type needed. Its form is:

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

It is identical to the header of a function definition, except:

- It does not include a *statement* for the function. That means that it does not include the body with all the instructions that are usually enclose within curly brackets { }.
- It ends with a semicolon sign (;).
- In the argument enumeration it is enough to put the type of each argument. The inclusion of a name for each argument as in the definition of a standard function is optional, although recommended.

For example:

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

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

int main ()
{
  int i;
  do {
```

```
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.
```

```
    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);
}
```

This example is indeed not an example of effectiveness, I am sure that at this point you can already make a program with the same result using only half of the code lines. But this example ilustrates how protyping works. Moreover, in this concrete case the prototyping of -at least- one of the two functions is necessary. The first things that we see are the prototypes of functions **odd** and **even**:

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

that allows these functions to be used before they are completely defined, for example, in **main**, which now is located in a more logical place: the beginning of the program's code.

Nevertheless, the specific reason why this program needs at least one of the functions prototyped is because in **odd** there is a call to **even** and in **even** there is a call to **odd**. If none of the two functions had been previously declared, an error would have happened, since either **odd** would not be visible from **even** (because it has not still been declared), or **even** would not be visible from **odd**.

Many programmers recommend that all functions be prototyped. It is also my recommendation, mainly in case that there are many functions or in case that they are very long. Having the prototype of all the functions in the same place can spare us some time when determining how to call it or even ease the creation of a header file.

PERTEMUAN IX

ARRAY

Tujuan :

- Memahami mengenai data komposit dalam ARRAY

Arrays are a series of elements (variables) of the same type placed consecutively in memory that can be individually referenced by adding an index to a unique name. That means that, for example, we can store 5 values of type **int** without having to declare 5 different variables each with a different identifier. Instead, using an *array* we can store 5 different values of the same type, **int** for example, with a unique identifier. For example, an array to contain 5 integer values of type **int** called *billy* could be represented this way is where each blank panel represents an *element* of the array, that in this case are integer values of type **int**. These are numbered from **0** to **4** since in arrays the first index is always **0**, independently of its length. Like any other variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

type name [elements];

where *type* is a valid object type (**int**, **float**...), *name* is a valid variable *identifier* and the *elements* field, that is enclosed within brackets **[]**, specifies how many of these elements the array contains. Therefore, to declare *billy* as shown above it is as simple as the following sentence : int billy [5];

NOTE: The *elements* field within brackets [] when declaring an array must be a constant value, since arrays are blocks of static memory of a given size and the compiler must be able to determine exactly how much memory it must assign to the array before any instruction is considered.

Initializing arrays.

When declaring an array of local scope (within a function), if we do not specify otherwise, it will not be initialized, so its content is undetermined until we store some values in it. If we declare a global array (outside any function) its content will be initialized with all its elements filled with zeros. Thus, if in the global scope we declare : int billy [5]; every element of *billy* will be set initialy to **0**. But additionally, when we declare an Array, we have the possibility to assign initial values to each one of its elements using curly brackets { }. For example:

int billy [5] = { 16, 2, 77, 40, 12071 }; this declaration would have created an array like the following one.

The number of elements in the array that we initialized within curly brackets { } must match the length in elements that we declared for the array enclosed within square brackets [ ]. For example, in the example of the *billy* array we have declared that it had 5 elements and in the list of initial values within curly brackets { } we have set 5 different values, one for each element.

Because this can be considered useless repetition, C++ includes the possibility of leaving the brackets empty [ ] and the size of the Array will be defined by the number of values included between curly brackets { }:

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

Access to the values of an Array.

In any point of the program in which the array is visible we can access individually anyone of its values for reading or modifying as if it was a normal variable. The format is the following: *name*[*index*]. Following the previous examples in which *billy* had 5 elements and each of those elements was of type **int**, the name which we can use to refer to each element is the following:

For example, to store the value 75 in the third *element* of *billy* a suitable sentence would be : **billy[2] = 75;** and, for example, to pass the value of the third element of billy to the variable **a**, we could write : **a = billy[2];** Therefore, for all purposes, the expression **billy[2]** is like any other variable of type **int**.

Notice that the third element of **billy** is specified **billy[2]**, since first is **billy[0]**, the second is **billy[1]**, and therefore, third is **billy[2]**. By this same reason, its last element is **billy[4]**. Since if we wrote **billy[5]**, we would be acceding to the sixth element of *billy* and therefore exceeding the size of the array. In C++ it is perfectly valid to exceed the valid range of indices for an Array, which can create problems since they do not cause compilation errors but they can cause unexpected results or serious errors during execution. The reason why this is allowed will be seen farther ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets [ ] have related to arrays. They perform two differt tasks: one is to set the size of arrays when declaring them; and second is to specify indices for a concrete array element when referring to it. We must simply take care not to confuse these two possible uses of brackets [ ] with arrays:

```
int billy[5];       // declaration of a new Array (begins with a type name)
billy[2] = 75;      // access to an element of the Array.
```

Other valid operations with arrays:

```
billy [0] = a;
billy [a] = 75;
b = billy [a+2];
billy[billy[a]] = billy[2] + 5;
```

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

int billy [] = {16, 2, 77, 40, 12071};
int n, result=0;

int main ()
{
  for ( n=0 ; n<5 ; n++ )
  {
```

12206

```
    result += billy[n];
  }
  cout << result;
  return 0;
}
```


Multidimensional Arrays

Multidimensional arrays can be described as arrays of arrays. For example, a bidimensional array can be imagined as a bidimensional table of a uniform concrete data *type*. Jimmy represents a bidimensional array of 3 per 5 values of type **int**. The way to declare this array would be:

int jimmy [3][5];

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

jimmy[1][3]

(remember that array indices always begin by **0**).

Multidimensional arrays are not limited to two indices (two dimensions). They can contain as many indices as needed, although it is rare to have to represent more than 3 dimensions. Just consider the amount of memory that an array with many indices may need. For example:

char century [100][365][24][60][60];

assigns a **char** for each second contained in a century, that is more than 3 billion **chars**! This would consume about 3000 *megabytes* of RAM memory if we could declare it.

Multidimensional arrays are nothing more than an abstraction, since we can obtain the same results with a simple array just by putting a factor between its indices:

int jimmy [3][5] ; is equivalent to

int jimmy [15] ; (3 * 5 = 15)

with the only difference that the compiler remembers for us the depth of each imaginary dimension. Serve as example these two pieces of code, with exactly the same result, one using bidimensional arrays and the other using only simple arrays:

```
// 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;
}
```

none of the programs above produce any output on the screen, but both assign values to the memory block called **jimmy** in the following way:

We have used defined constants (**#define**) to simplify possible future modifications of the program, for example, in case that we decided to enlarge the array to a height of **4** instead of **3** it could be done by changing the line:

```
#define HEIGHT 3
```

to

```
#define HEIGHT 4
```

with no need to make any other modifications to the program.

Arrays as parameters

At some moment we may need to pass an array to a function as a parameter. In C++ is not possible to pass by value a complete block of memory as a parameter to a function, even if it is ordered as an array, but it is allowed to pass its address. This has almost the same practical effect and it is a much faster and more efficient operation. In order to admit arrays as parameters the only thing that we must do when declaring the function is to specify in the argument the base **type** for the array, an identifier and a pair of void brackets **[]**. For example, the following function:

void procedure (int arg[])

admits a parameter of type "Array of **int**" called **arg**. In order to pass to this function an array declared as:

int myarray [40];

it would be enough to write a call like this : procedure (myarray);

Here you have a complete example:

```
// 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;
}
```

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

As you can see, the first argument (**int arg[]**) admits any array of type **int**, wathever its length is. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as the first parameter. This allows the **for** loop that prints out the array to know the range to check in the passed array. In a function declaration is also possible to include multidimensional arrays. The format for a tridimensional array is:

base_type[][*depth*][*depth*]

for example, a function with a multidimensional array as argument could be:

void procedure (int myarray[][3][4])

notice that the first brackets [] are void and the following ones are not. This must always be thus because the compiler must be able to determine within the function which is the depth of each additional dimension.

Arrays, both simple or multidimensional, passed as function parameters are a quite common source of errors for less experienced programmers. I recommend the reading of, **Pointers** for a better understanding of how *arrays* operate.

SOAL !

1. MENU DERET :
   RATA_RATA :
   MAKSIMUM :
   MINIMUM :
   SUKU :

   Dengan memasukkan suku maka akan keluar output sesuai pilihan

PERTEMUAN X

STRINGS OF CHARACTERS

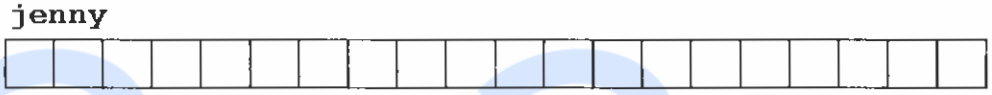
Tujuan ;

- Untuk mempelajari mengenai aplikasi penggunaan string, serta implementasi dalam program C++.

In all programs seen until now, we have used only numerical variables, used to express numbers exclusively. But in addition to numerical variables there also exist strings of characters, that allow us to represent successions of characters, like words, sentences, names, texts, et cetera. Until now we have only used them as constants, but we have never considered variables able to contain them. In C++ there is no specific *elemental* variable type to store strings of characters. In order to fulfill this feature we can use arrays of type **char**, which are successions of **char** elements. Remember that this data type (**char**) is the one used to store a single character, for that reason arrays of them are generally used to make strings of single characters. For example, the following array (or string of characters):

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

can store a string up to 20 characters long. You may imagine it thus:



This maximum size of 20 characters is not required to always be fully used. For example, **jenny** could store at some moment in a program either the string of characters "Hello" or the string "Merry christmas". Therefore, since the array of characters can store shorter strings than its total length, a convention has been reached to end the valid content of a string with a null character, whose constant can be written **0** or **'\0'**. We could represent **jenny** (an array of 20 elements of type **char**) storing the strings of characters "Hello" and "Merry Christmas" in the following way:

Notice how after the valid content a null character ('**\0**') it is included in order to indicate the end of the string. The panels in gray color represent indeterminate values.

Initialization of strings

Because strings of characters are ordinary arrays they fulfill all their same rules. For example, if we want to initialize a string of characters with predetermined values we can do it just like any other array:

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

In this case we would have declared a string of characters (array) of 6 elements of type **char** initialized with the characters that compose **Hello** plus a null character '**\0**'. Nevertheless, strings of characters have an additional way to initialize their values: using constant strings. In the expressions we have used in examples in previous chapters constants that represented entire strings of characters have already appeared several times. These are specified enclosed between double quotes ("), for example:

```
"the result is: "
```

is a constant string that we have probably used on some occasion.

Unlike single quotes (') which specify single character constants, double quotes (") are constants that specify a succession of characters. Strings enclosed between double quotes always have a null character ('\0') automatically appended at the end. Therefore we could initialize the string **mystring** with values by either of these two ways:

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

```
char mystring [] = "Hello";
```

In both cases the array or string of characters **mystring** is declared with a size of 6 characters (elements of type **char**): the 5 characters that compose **Hello** plus a final null character ('\0') which specifies the end of the string and that, in the second case, when using double quotes (") it is automatically appended. Before going further, notice that the assignation of multiple constants like double-quoted constants (") to arrays are only valid when initializing the array, that is, at the moment when declared. Expressions within the code like:

```
mystring = "Hello";
mystring[] = "Hello";
```

are not valid for arrays, like neither would be:

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

So remember: We can "assign" a multiple constant to an Array only at the moment of initializing it. The reason will be more comprehensible when you know a bit more about pointers, since then it will be clarified that an array is simply a *constant pointer* pointing to an allocated block of memory. And because of this constantnes, the array itself can not be assigned any value, but we can assing values to each of the elements of the array.The moment of initializing an Array it is a special case, since it is not an assignation, although the same equal sign (=) is used. Anyway, always have the rule previously underlined present.

Assigning values to strings

Since the *lvalue* of an assignation can only be an element of an array and not the entire array, it would be valid to assign a string of characters to an array of **char** using a method like this:

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

But as you may think, this does not seem to be a very practical method. Generally for assigning values to an array, and more specifically to a string of

characters, a series of functions like **strcpy** are used. **strcpy** (**str**ing **copy**) is defined in the **cstring** (string.h) library and can be called the following way:

strcpy (*string1***,** *string2***);** This does copy the content of *string2* into *string1*. *string2* can be either an array, a pointer, or a constant string, so the following line would be a valid way to assign the constant string **"Hello"** to **mystring**:

strcpy (mystring, "Hello");

For example:

```
// 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

Notice that we needed to include **<string.h>** header in order to be able to use function **strcpy**. Although we can always write a simple function like the following **setstring** with the same operation as cstring's **strcpy**:

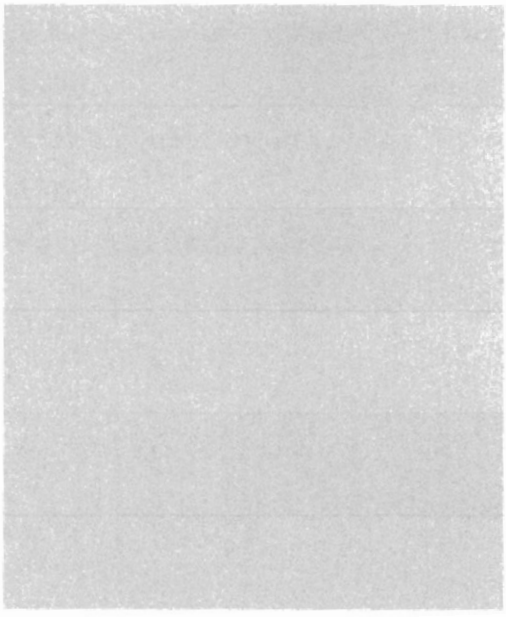
```
// setting value to string
#include <iostream.h>

void setstring (char szOut [], char szIn
[])
{
  int n=0;
  do {
```

J. Soulie

```
    szOut[n] = szIn[n];
  } while (szIn[n++] != '\0');
}

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



Another frequently used method to assign values to an array is by directly using the input stream (**cin**). In this case the value of the string is assigned by the user during program execution. When **cin** is used with strings of characters it is usually used with its **getline** method, that can be called following this prototype: **cin.getline (char** *buffer*[], **int** *length*, **char** *delimiter* = ' \n'); where ***buffer*** is the address of where to store the input (like an array, for example), ***length*** is the maximum length of the buffer (the size of the array) and ***delimiter*** is the character used to determine the end of the user input, which by default - if we do not include that parameter - will be the newline character ('\n').

The following example repeats whatever you type on your keyboard. It is quite simple but serves as an example of how you can use **cin.getline** with strings:

```
// 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;
}
```

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

Notice how in both calls to **cin.getline** we used the same string identifier (**mybuffer**). What the program does in the second call is simply step on the previous content of **buffer** with the new one that is introduced. If you remember the section about communication through the console, you will remember that we used the extraction operator (>>) to receive data directly from the standard input. This method can also be used instead of **cin.getline** with strings of characters. For example, in our program, when we requested an input from the user we could have written:

```
cin >> mybuffer;
```

this would work, but this method has the following limitations that **cin.getline** has not:

- It can only receive single words (no complete sentences) since this method uses as a delimiter any occurrence of a blank character, including spaces, tabulators, newlines and carriage returns.
- It is not allowed to specify a size for the buffer. That makes your program unstable in case the user input is longer than the array that will host it.

For these reasons it is recommended that whenever you require strings of characters coming from **cin** you use **cin.getline** instead of **cin >>**.

Converting strings to other types

Due to that a string may contain representations of other data types like numbers, it might be useful to translate that content to a variable of a numeric type. For example, a string may contain **"1977"**, but this is a sequence of 5 chars not so easily convertable to a single integer data type. The **cstdlib (stdlib.h)** library provides three useful functions for this purpose:

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

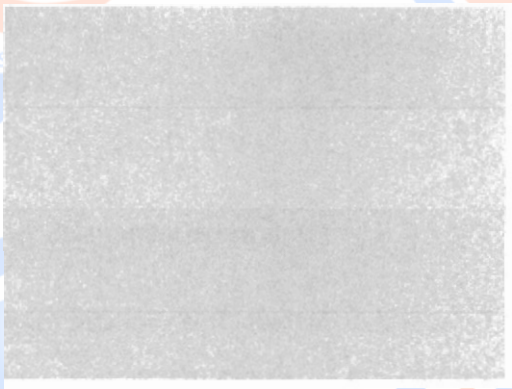
All of these functions admit one parameter and return a value of the requested type (int, long or float). These functions combined with **getline** method of **cin** are a more reliable way to get the user input when requesting a number than the classic **cin>>** method:

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

int main ()
{
 char mybuffer [100];
 float price;
 int quantity;
 cout << "Enter price: ";
```

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

```
    cin.getline (mybuffer,100);
    price = atof (mybuffer);
    cout << "Enter quantity: ";
    cin.getline (mybuffer,100);
    quantity = atoi (mybuffer);
    cout << "Total price: " << price*quantity;
    return 0;
}
```



Functions to manipulate strings

The **cstring** library (string.h) defines many functions to perform manipulation operations with C-like strings (like already explained strcpy). Here you have a brief look at the most usual:

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*.

NOTE: **char*** is the same as **char[]**

Check the C++ Reference for extended information about these and other functions of this library.

Soal !

1. Pencetakan nama, alamat, dan NIM dengan penginputan:
2. Penggabungan penginputan Nama, NIM, Nilai1, Nilai 2, dan Nilai2.

PERTEMUAN XI
POINTER

Tujuan:

- Mengerti lebih lanjut tentang pointer dalam penerapan di C++

We have already seen how variables are memory cells that we can access by an identifier. But these variables are stored in concrete places of the computer memory. For our programs, the computer memory is only a succession of 1 *byte* cells (the minimum size for a datum), each one with a unique address. A good simile for the computer memory can be a street in a city. On a street all houses are consecutively numbered with an unique identifier so if we talk about 27th of Sesame Street we will be able to find that place without trouble, since there must be only one house with that number and, in addition, we know that the house will be between houses 26 and 28.

In the same way in which houses in a street are numbered, the operating system organizes the memory with unique and consecutive numbers, so if we talk about location 1776 in the memory, we know that there is only one location with that address and also that is between addresses 1775 and 1777.

Address (dereference) operator (&).

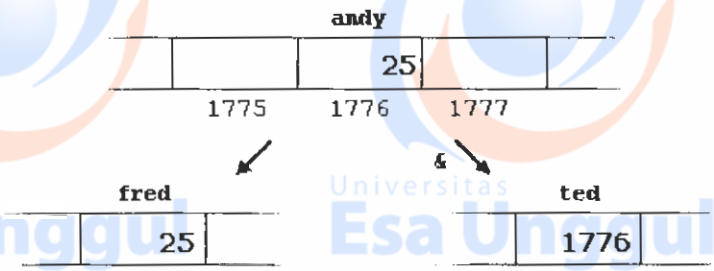
At the moment in which we declare a variable it must be stored in a concrete location in this succession of cells (the memory). We generally do not decide where the variable is to be placed - fortunately that is something automatically done by the compiler and the operating system at runtime, but once the operating system has assigned an address there are some cases in which we may be interested in knowing where the variable is stored. This can be done by preceding the variable identifier by an *ampersand sign* (&), which literally means ***"address of"***.

For example: ted = &andy; would assign to variable **ted** the address of variable **andy**, since when preceding the name of the variable **andy** with the *ampersand* (&) character we are no longer talking about the content of the variable, but about its address in memory.

We are going to suppose that **andy** has been placed in the memory address **1776** and that we write the following:

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

the result is shown in the following diagram:



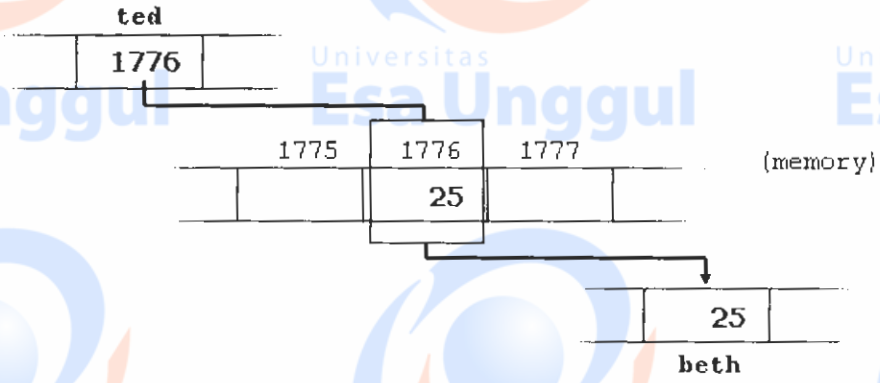
We have assigned to **fred** the content of variable **andy** as we have done in many other occasions in previous sections of this tutorial, but to **ted** we have assigned the address in memory where the operating system stores the value of **andy**, that we have imagined was **1776** (it can be any address, I have just invented this one). The reason is that in the allocation of **ted** we have preceded **andy** with an *ampersand* (&) character. The variable that stores the address of another variable (like **ted** in the previous example) is what we call a **pointer**. In C++ pointers have certain virtues and they are used very often. Farther ahead we will see how this type of variable is declared.

Reference operator (*)

Using a pointer we can directly access the value stored in the variable pointed by it just by preceding the pointer identifier with the reference operator *asterisk* (*), that can be literally translated to ***"value pointed by"***. Therefore, following with the values of the previous example, if we write:

```
beth = *ted;
```

(that we could read as: "beth equal to value pointed by ted") **beth** would take the value **25**, since **ted** is **1776**, and *the value pointed by* **1776** is **25**.



You must clearly differenciate that **ted** stores **1776**, but ***ted** (with an asterisk * before) refers to the value stored in the address **1776**, that is **25**. Notice the difference of including or not including the reference asterisk (I have included an explanatory commentary of how each expression could be read):

beth = ted; *// beth equal to ted (1776)*

beth = *ted; *// beth equal to **value pointed by** ted (25)*

Operator of address or dereference (&)

It is used as a variable prefix and can be translated as **"address of"**, thus: &variable1 can be read as *"address of variable1"*

Operator of reference (*)

It indicates that what has to be evaluated is the content pointed by the expression considered as an address. It can be translated by **"value pointed by"**. * mypointer can be read as *"value pointed by mypointer"*.

At this point, and following with the same example initiated above where:

andy = 25;

ted = &andy;

you should be able to clearly see that all the following expressions are true:

andy == 25

&andy == 1776

ted == 1776

*ted == 25

The first expression is quite clear considering that its assignation was **andy=25;**. The second one uses the address (or derefence) operator (&) that returns the address of the variable **andy**, that we imagined to be **1776**. The third one is quite obvious since the second was true and the assignation of **ted** was **ted = &andy;**. The fourth expression uses the reference operator (*) that, as we have just seen, is equivalent to the value contained in the address pointed by **ted**, that is **25**. So, after all that, you may also infer that while the address pointed by **ted** remains unchanged the following expression will also be true:

*ted == andy

Declaring variables of type pointer

Due to the ability of a pointer to directly reference the value that it point to, it becomes necessary to specify which data type a pointer points to when declaring it. It is not the same to point to a **char** as it is to point to an **int** or a **float** type. Therefore, the declaration of pointers follows this form:

***type * pointer_name*;**

where ***type*** is the type of data pointed, not the type of the pointer itself. For example:

int * number;

char * character;

float * greatnumber;

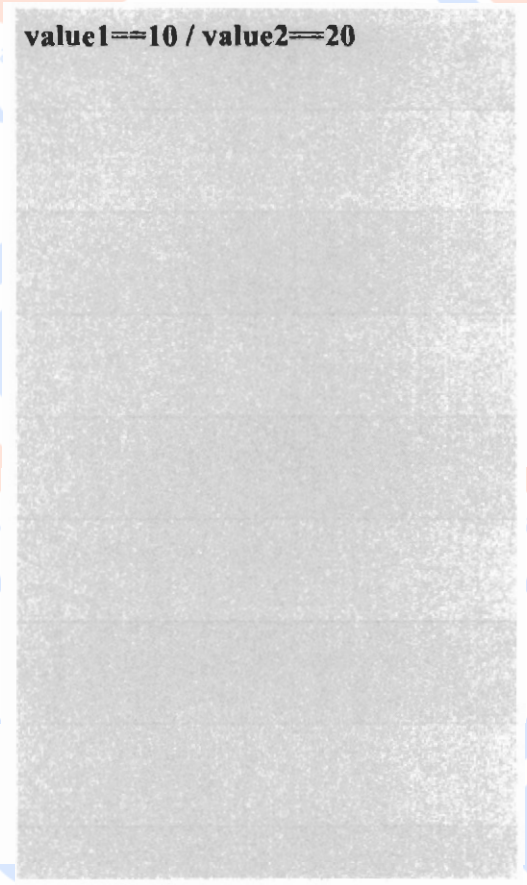
they are three declarations of pointers. Each one points to a different data type, but the three are pointers and in fact the three occupy the same amount of space in memory (the size of a pointer depends on the operating system), but the data to which they point do not occupy the same amount of space nor are of the same type, one is **int**, another one is **char** and the other one **float**.

I emphasize that the asterisk (*) that we use when declaring a pointer means only *that it is a pointer*, and should not be confused with the reference operator that we have seen a bit earlier which is also written with an asterisk (*). They are simply two different tasks represented with the same sign.

```
// 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;
}
```

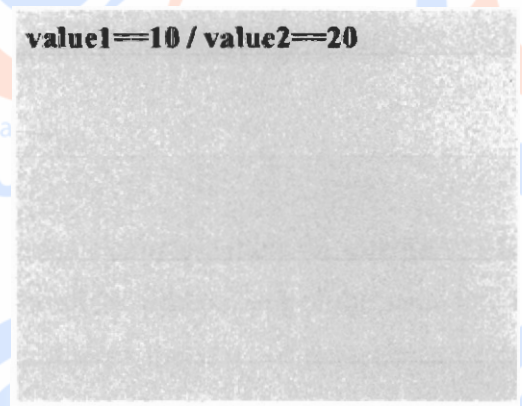


Notice how the values of **value1** and **value2** have changed indirectly. First we have assigned to **mypointer** the address of **value1** using the deference ampersand sign (&). Then we have assigned **10** to the value pointed by **mypointer**, which is pointing to the address of **value1**, so we have modified **value1** indirectly. In order that you can see that a pointer may take several different values during the same program we have repeated the process with **value2** and the same pointer.

Here is an example a bit more complicated:

```
// 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;
}
```

I have included as comments on each line how the code can be read: ampersand (&) as "address of" and asterisk (*) as "value pointed by". Notice that there are expressions with pointers **p1** and **p2** with and without the asterisk. The meaning of using or not using a reference asterisk is very different: An asterisk (*) followed by the pointer refers to the place pointed by the pointer, whereas a pointer without an asterisk (*) refers to the value of the pointer itself, that is, the address of where it is pointing. Another thing that can call your attention is the line:

```
int *p1, *p2;
```

that declares the two pointers of the previous example putting an asterisk (*) for each pointer. The reason is that the type for all the declarations of the same line is **int** (and not **int***). The explanation is because of the level of precedence of the

reference operator asterisk (*) that is the same as the declaration of types, therefore, because they are associative operators from the right, the asterisks are evaluated first than the type. We have talked about this in section 1.3: Operators, although it is enough that you know clearly that -unless you include parenthesis- you will have to put an asterisk (*) before each pointer that you declare.

Pointers and arrays

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so in fact they are the same thing. For example, supposing these two declarations:

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

the following allocation would be valid:

```
p = numbers;
```

At this point **p** and **numbers** are equivalent and they have the same properties, the only difference is that we could assign another value to the pointer **p** whereas **numbers** will always point to the first of the 20 integer numbers of type int with which it was defined. So, unlike **p**, that is an ordinary *variable pointer*, **numbers** is a *constant pointer* (indeed an array name is a constant pointer). Therefore, although the previous expression was valid, the following allocation is not:

```
numbers = p;
```

because **numbers** is an array (constant pointer), and no values can be assigned to constant identifiers.

Due to the character of *variables* all the expressions that include pointers in the following example are perfectly valid:

```
// more pointers
#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;
}
```

```
10, 20, 30, 40, 50,
```

In chapter "*Arrays*" we used bracket signs [] several times in order to specify the index of the element of the Array to which we wanted to refer. Well, the bracket signs operator [] are known as *offset* operators and they are equivalent to adding the number within brackets to the address of a pointer. For example, both following expressions:

```
a[5] = 0;      // a [offset of 5] = 0
*(a+5) = 0;    // pointed by (a+5) = 0
```

are equivalent and valid either if **a** is a pointer or if it is an array.

Pointer initialization

When declaring pointers we may want to explicitly specify to which variable we want them to point,

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

this is equivalent to:

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

When a pointer assignation takes place we are always assigning the address where it points to, never the value pointed. You must consider that at the moment of declaring a pointer, the asterisk (*) indicates only that it is a pointer, it in no case indicates the reference operator (*). Remember, they are two different operators, although they are written with the same sign. Thus, we must take care not to confuse the previous with:

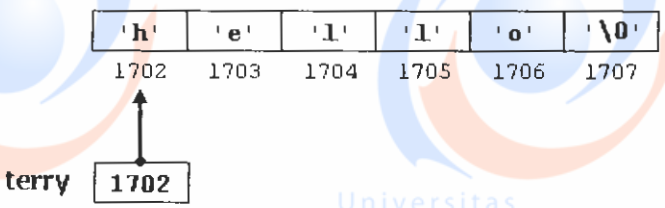
```
int number;
int *tommy;
*tommy = &number;
```

that anyway would not have much sense in this case.

As in the case of arrays, the compiler allows the special case that we want to initialize the content at which the pointer points with constants at the same moment as declaring the variable pointer:

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

in this case static storage is reserved for containing "hello" and a pointer to the first **char** of this memory block (that corresponds to 'h') is assigned to **terry**. If we imagine that "hello" is stored at addresses 1702 and following, the previous declaration could be outlined thus:



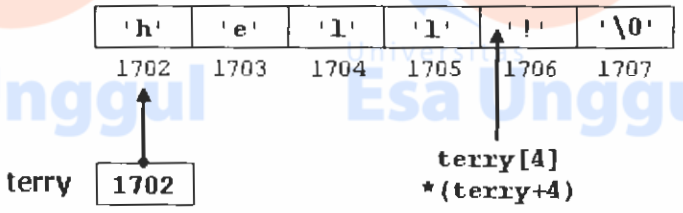
it is important to indicate that **terry** contains the value **1702** and not **'h'** nor **"hello"**, although **1702** points to these characters.

The pointer **terry** points to a string of characters and can be used exactly as if it was an Array (remember that an array is just *a constant pointer*). For example,

if our temper changed and we wanted to replace the 'o' by a '!' sign in the content pointed by **terry**, we could do it by any of the following two ways:

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

remember that to write **terry[4]** is just the same as to write ***(terry+4)**, although the most usual expression is the first one. With either of those two expressions something like this would happen:



Arithmetic of pointers

To conduct arithmetical operations on pointers is a little different than to conduct them on other integer data types. To begin with, only addition and subtraction operations are allowed to be conducted, the others make no sense in the world of pointers. But both addition and subtraction have a different behavior with pointers according to the size of the data type to which they point. When we saw the different data types that exist, we saw that some occupy more or less space than others in the memory. For example, in the case of integer numbers, *char* occupies 1 byte, *short* occupies 2 bytes and *long* occupies 4.

Let's suppose that we have 3 pointers:

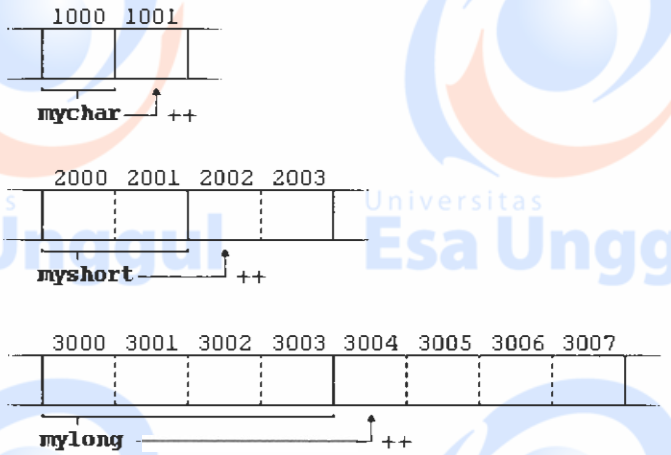
```
char *mychar;
short *myshort;
long *mylong;
```

and that we know that they point to memory locations **1000**, **2000** and **3000** respectively. So if we write:

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

mychar, as you may expect, would contain the value 1001.

Nevertheless, myshort would contain the value 2002, and mylong would contain 3004. The reason is that when adding 1 to a pointer we are making it to point to the following element of the same type with which it has been defined, and therefore the size in *bytes* of the *type* pointed is added to the pointer.



This is applicable both when adding and subtracting any number to a pointer. It would happen exactly the same if we write:

```
mychar = mychar + 1;
myshort = myshort + 1;
mylong = mylong + 1;
```

It is important to warn you that both increase (++) and decrease (--) operators have a greater priority than the reference operator asterisk (*), therefore the following expressions may lead to confussion:

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

The first one is equivalent to *(p++) and what it does is to increase **p** (the address where it points to - not the value that contains). In the second, because both increase operators (++) are after the expressions to be

evaluated and not before, first the value of *q is assigned to *p and then both q and p are increased by one. It is equivalent to:

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

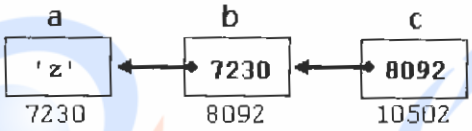
Like always, I recommend you use parenthesis () in order to avoid unexpected results.

Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data. In order to do that we only need to add an asterisk (*) for each level of reference:

```
char a;
char * b;
char ** c;
a = 'z';
b = &a;
c = &b;
```

this, supposing the randomly chosen memory locations of **7230**, **8092** and **10502**, could be described thus:



(inside the cells there is the content of the variable; under the cells its location)

The new thing in this example is variable **c**, which we can talk about in three different ways, each one of them would correspond to a different value:

c is a variable of type (char **) with a value of 8092

***c is a variable of type (char*) with a value of 7230**

****c is a variable of type (char) with a value of'z'**

void *pointers*

The type of pointer *void* is a special type of pointer. *void* pointers can point to any data type, from an integer value or a float to a string of characters. Its sole limitation is that the pointed data cannot be referenced directly (we can not use reference asterisk * operator on them), since its length is always undetermined, and for that reason we will always have to resort to *type casting* or assignations to turn our *void* pointer to a pointer of a concrete data type to which we can refer.

One of its utilities may be for passing generic parameters to a function:

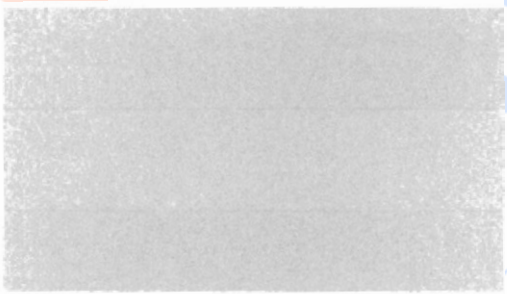
```
// integer increaser
#include <iostream.h>

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));
```

6, 10, 13

```
  increase (&c,sizeof(c));
  cout << (int) a << ", " << b << ", " <<
c;
  return 0;
}
```



sizeof is an operator integrated in the C++ language that returns a constant value with the size in bytes of its parameter, so, for example, **sizeof(char)** is **1**, because **char** type is 1 byte long.

Pointers to functions

C++ allows operations with pointers to functions. The greatest use of this is for passing a function as a parameter to another function, since these cannot be passed dereferenced. In order to declare a pointer to a function we must declare it like the prototype of the function except the name of the function is enclosed between parenthesis () and a pointer asterisk (*) is inserted before the name. It might not be a very handsome syntax, but that is how it is done in C++:

8

```
// 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;
}
```

In the example, **minus** is a global pointer to a function that has two parameters of type **int**, it is immediately assigned to point to the function **subtraction**, all in a single line:

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

DAFTAR PUSTAKA

Ditdit N Utama & Riya Widayanti, "Algoritma & Pemrograman dengan Borland C++", 2005.

Rinaldi Munir, "Algoritma & Pemrograman dalam Bahasa Pascal dan C Edisi - 3,2004

Andri Kristanto, "Algoritma dan Pemrograman dengan C++",2003