ENGR101: Lecture 3

How to use C++, variables, arrays

2024

What we cover this time?

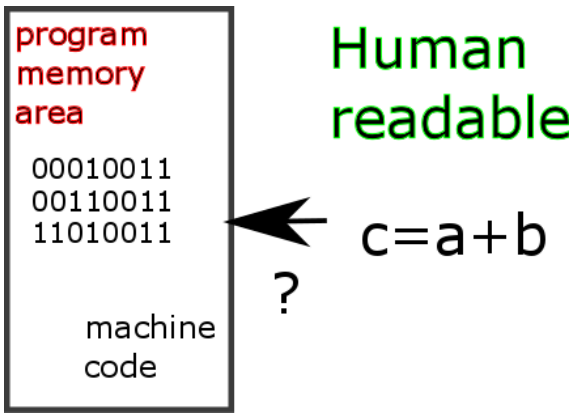
- What is computer program and how you make one
- Variables - and **another go at the memory**
- Variable types
- Grouping variables together

Everything in the computer, including programs, is a number in binary format

- Everything is stored and processed as binary numbers.
- Binary numbers are broken into groups of 8 bits, called bytes.
- Processor executes **machine codes** which are not human-readable

Humans don't understand binary. Processors understand binary only.

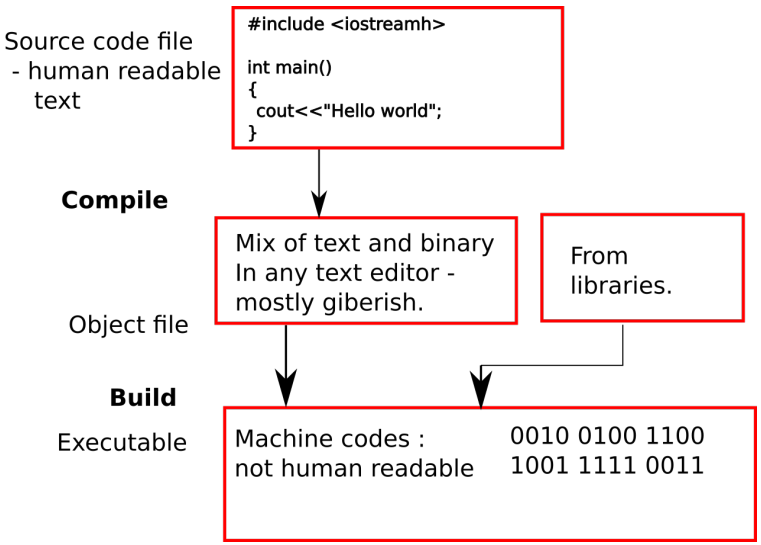
Machine language vs Human-readable language



- All CPU instructions are stored in memory as numbers.
- Set of such a numbers is called **machine code** and that what processor is running.
- Nobody writes machine codes any more.
- Programs are created in **human-readable** form and automatically translated into machine codes.

Now lets have a look how this translation is done.

Making program to run. Ignore what program does for now.



1. Write/edit text of the program. Human readable - if you are specific human
2. Translate text into machine codes
   It is done in two stages:
   - **Compiler** checks that there are no errors in this text and produces **object** file
   - **Linker** (sometimes called builder) takes object file, adds services and produces **executable** file
3. Run the executable

Optional step: Watch computer to go up in flames, say "It is funny" and go back to 1

We will be using C++ programming language

- Very old - basic C written in 1972 by Dennis Ritchie in ATT.
- There are two versions: C - development frozen
  C++ - C with additions. Keeps growing and expanding.
- As close to hardware as it can get (with the exception of assembler)
- Why do we have to use the tool which is almost 50 years old?

Why indeed?

- It is fastest of all languages.
- And most dangerous one. There is no built-in safeguards. Gives you good programming habits.
- Used in areas where speed is of utmost concern: gaming, graphics, networking, operating systems (Windows written C++, Linux written in C, Mac - modified C++)
- Used when hardware is limited: **robotics**, network gear.

C++ programming tools

We will be trying things as we go along. To do it we need some tools, at least:

- C++ compiler (C++20 recommended)
- code editor (we use **Geany**)

Instructions on how to install C++ and code editror (Geany):
`https://ecs.wgtn.ac.nz/Courses/ENGR101_2024T1/InstallC`

It is not compulsory to use Geany. But you are on your own if you use Sublime, VSCode, CodeBlocks, Atom or any of the others.
When programming robot we will use Command Line Interface.

Program text to machine codes: Compile, build and run

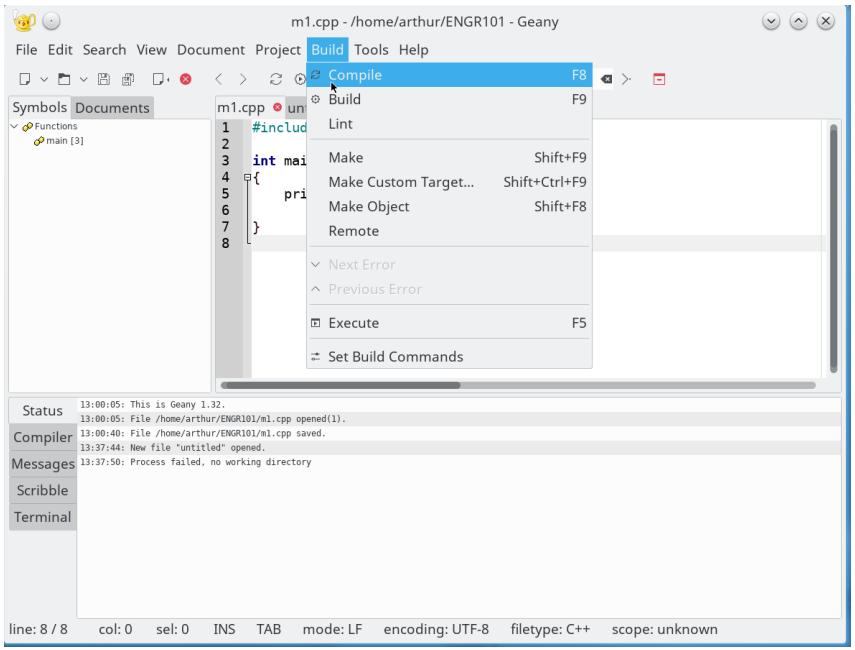


Figure: Compile, build and run in Geany

Using *Build* menu option you can convert text of your program into machine codes (executable file) and run it. It is two stages process:

- Compile - convert some program text into **object code**. Object code can not execute.
- Build (sometimes called link) - takes object codes (usually from several programs) and converts object code into true executable.
- Execute - it runs the program.

Hello, world

Following time-honored tradition we start with the program which prints **Hello, world** on the screen.

Listing 1: Prints "Hello world" and looks Greek

```
#include <iostream>
int main(){
  std::cout<<"Hello, world"<<std::endl;
  return 0;
}
```

File should be saved with **cpp** extension, so the Geany knows that C++ compiler should be engaged. Compile(F8), build (F9), run (F5). Now lets go through what is in listing.

std::cout ?

Listing 2: Prints "Hello world" and looks Greek

```
std::cout<<"Hello, world"<<std::endl;
```

- Standard console out:: **std::cout**.
- Prints whatever is after <<.
- std::endl moves screen cursor to next line.
- if you need more output: add << to the end followed by more output

main() ?

What is **main()** ? Lets make program without one.

- Compile - OK
- Build - (.text+0x20): undefined reference to ‘main’

It needs **main()** - which is entry point of the program.
After that comes block of code enclosed in curly braces. Program runs from opening brace until closing one.
OK, let us put in main()...

Listing 3: "Does nothing and does not compile either"

```
main()
{
}
```

Still complaining but runs:
me1.cpp:2:6: warning: ISO C++ forbids declaration of ‘main’ with no type [-Wreturn-type]

Quiz time...

Can I have two **main()** functions? Let's vote ...

1. Yes
2. No

Let's check...

include ?

If there is *std* :: *cout* in the program and no $\#include < iostream >$ then compiler fails.
When describing how to produce machine codes, we mentioned **...adding srevices...** . Thats what happens here:
$\#include < iostream >$ instructs compiler find and, er, include services provided by **iostream** program(library). This library provides Input-Output services for printing on the screen ans reading keyboard.

Listing 4: Prints

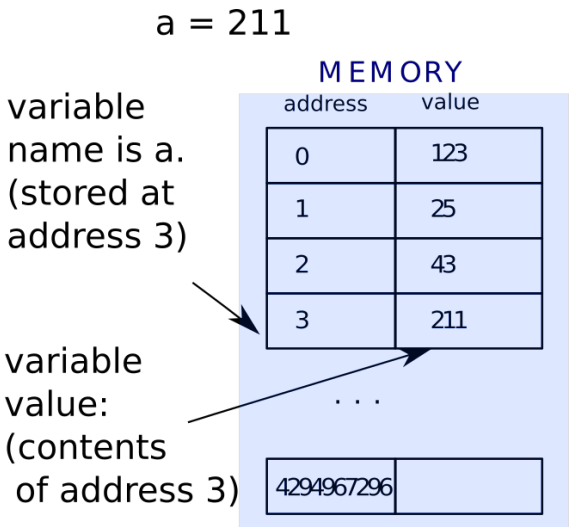
```
#include <iostream>
int main(){
  std::cout<<"something"<<std::endl;
  return 0;
}
```

Demos

- What happens if we try to read **object** and **executable** files?
- Not all languages go through this process in this order.
  - Compiled languages (C++, Java) - take whole file and convert it into machine codes. Then file with machine codes (binary) can be executed, or copied to another computer and run there
  - Interpreted languages (Python) - take one human-readable instruction, convert into code, execute. Take next instruction... Such an approach requires Python to run on the machine where you want program executed.

Variables - numbers stored in memory

Now that we know how to run the code, let us start trying to write some code.
When program runs values in memory are modified.



- Values are stored in memory by addresses
- To remember addresses of specific values is too cumbersome
- Lets give them some meaningful labels
- Even **a** is better than address 23,567,788
- Compiler assigns addresses to names automatically

Variables

Let's try to make variable, called **a**.
$a = 211$ - we want value of **a** to be 211.

Listing 5: " Attempt to make the variable"

```
int main(){
  a=5;
}
```

Compile?

Variable types

We got error message:

error: ‘a’ was not declared in this scope

what means that compiler did not understand what the **a** is and **a** was not recognized as a variable.

Error happens because of the way variables are stored in memory in C++ (Java as well).

C is **strongly typed** language. All variables should be of certain type.

int a = 0;, for example, works just fine. **int** is a **type**.

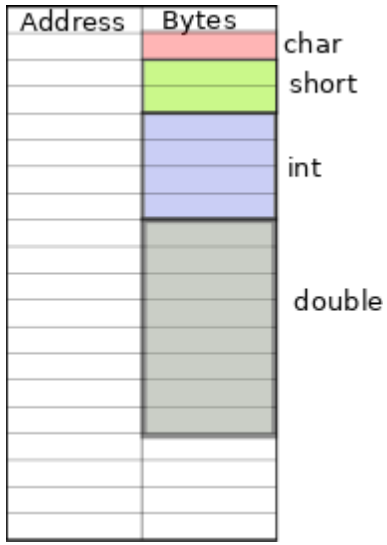
What are types?

Type specifies how many bytes of memory are used to store the variable.

Some variable types in C:

Some of C++ variable types:

- **char**: variable is 8 bits (1 Byte) long. Maximum value is $1111111_2 = 255_{10}$.
- **short**: variable is 2 Bytes long. Max = 11111111 $11111111_2 = 65535$
- **int**: 4 Bytes. Max = 4 294 967 295
- **double**: 8 bytes - can store fractional parts



There is an operator to figure out number of bytes variable takes:

Listing 6: Prints size of variable in Bytes

```
int a;
std::cout<<"Size␣of␣a␣is:<<sizeof(a)<<endl
```

Can we see what address variable is stored at?

Yes.
C is almost unique when it allows the programmer to work with memory addresses.
Programmer can read and modify contents of any memory cell.
If it happens to be wrong memory cell - bad luck.
If programmer modified memory cell inside program memory - program is ruined.

Can we see what address variable is stored at?

To see variable address use ampersand (**&**) in front of variable name.

Listing 7: "Address"

```
#include <iostream>
int main(){
  int a=5;
  std::cout<<"␣Value␣of␣a␣is␣"<<a<<std::endl;
  std::cout<<"␣Address␣of␣a␣is␣"<<&a<<std::endl;
}
```

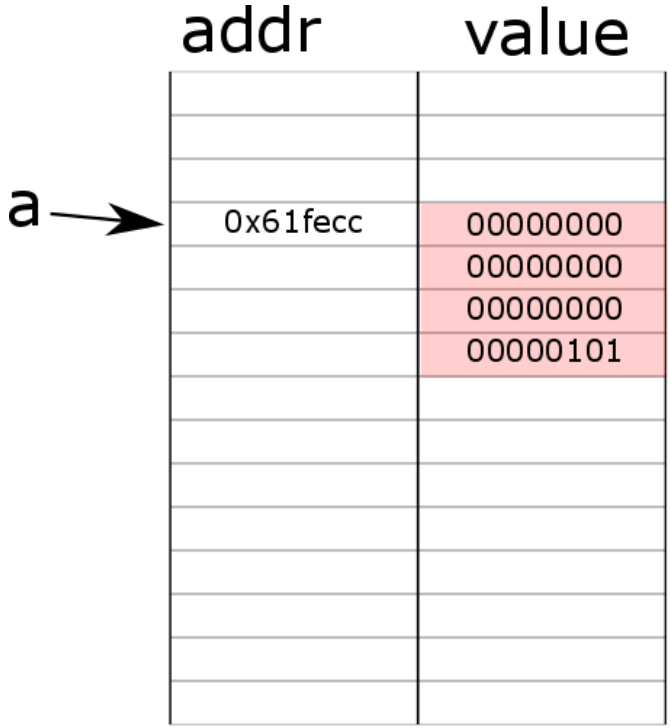
Output of the program is (from run to run address can be different):

Listing 8: "Pointer value"

```
 Value of a is 5
Address of a is 0x61fecc
```

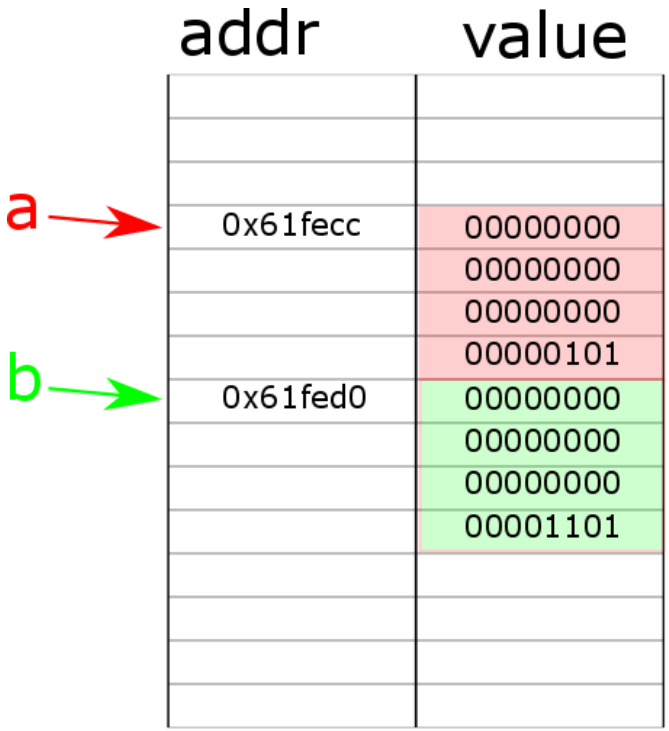
Address is printed in **hexadecimal**

What is happening under the hood when variable is declared?



- Declare variable name **a** of type **int**
- **int** is 4 bytes long
- Compiler find free memory block and chooses memory address for the variable
- From now on **a** means: 4 bytes starting from address 0x61fecc

What is happening under the hood when variable is declared?



Listing 9: "Two variables"

```
int a;
int b;
```

- We declare another variable
- C reserves memory in order of variables declaration

Variables scope

When variable is declared compiler associates variable name with memory address. How long compiler keeps this information?
Simple answer - from line with declaration statement until end of **code block**. Code block is set of statements between pair of curly brackets {....}.

Listing 10: "Variable scope"

```
#include <iostream>

int main()
{
 {
   int s=254;
   std::cout<<s<<std::endl;
 }
 std::cout<<"s="<<s<<std::endl;
}
```

Question:
Will that compile?

1. Yes
2. No

Variable scope

Listing 11: "Scope"

```
#include <iostream>
int a; //global variable
int main(){
 a = 5;
 std::cout<<"a="<<a<<std::endl;
}
```

- If variable was declared before code block - it can be used inside this block.
- If variable was declared before **main()** then it is **global** variable and can be used anywhere in the program. BAD IDEA.

Limits of variables

Listing 12: Hm..

```
#include <iostream>

int main(){
 unsigned char a = 255;
 std::cout<<"before: a="<<(int)a<<std::endl;
 a = a + 1;
 std::cout<<"after: a="<<(int)a<<std::endl;
}
```

What will happen?

1. before a = 255 after a = 256
2. before a = 255 after a = 255
3. before a = 255 after a = 0
4. Computer explodes

Reminder - char is 8 bits(1 Byte) long.
Maximum value:
$11111111_2 = 255_{10}$

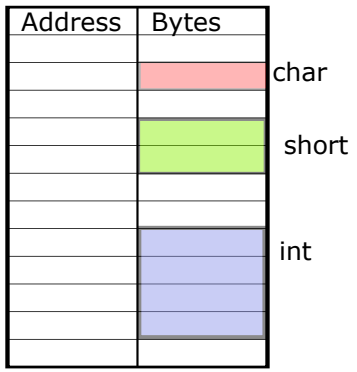


Figure:

Overflow

- Why?

		8 bits = 1 Byte							
a=255		1	1	1	1	1	1	1	1
+1									
b=256 as it should be	1	0	0	0	0	0	0	0	0
b=0 as it is in memory	x	0	0	0	0	0	0	0	0

- Worst type of programming error - **logical** program **error**. It is not detected until program runs

Arrays: why use them and what are arrays anyway?

Example: You have an image. Image is made out of pixels (dots). There can be thousands of them. Each pixel contains three numbers: reg, green and blue. To specify red levels across the image we can use type **char**. If we have say 2000 pixels then to describe level od red in all pixels we can declare 2000 variables.

Listing 13: "Naive way to describe level of red"

```
char red_pix0;
cahr red_pix1;
// many more of them
char red_pix1999;
```

Not impossible but it is a lot of typying. All elements of the array should be of the same type.

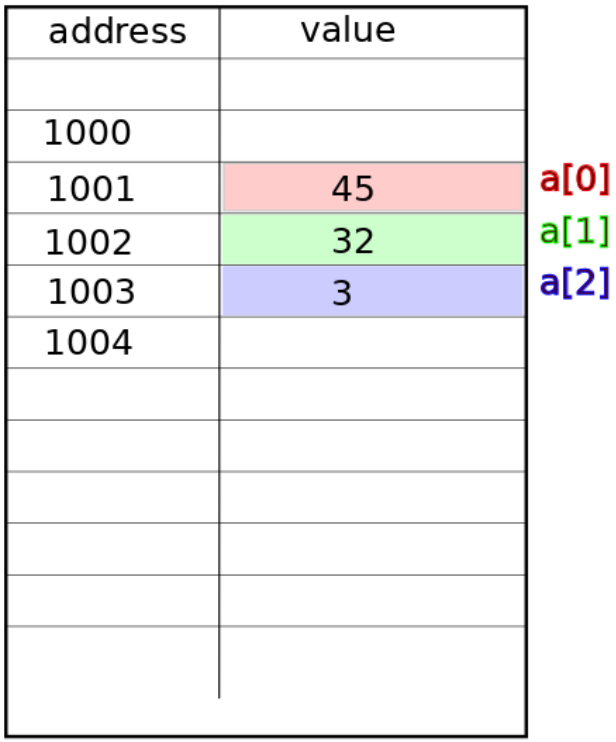
Arrays

There is a better way. It is possible to declare many instances of similar variables using only one line of code.

Listing 14: " Much less typing"

```
char red_pix[2000];
```

Array of **char**s in memory

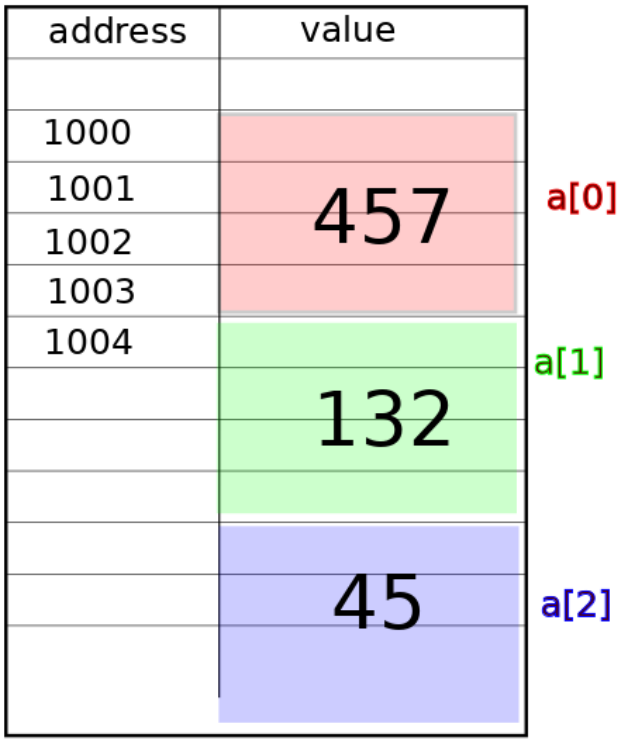


Memory address of the array elements is decided automatically.

Listing 15: "Array of char"

```
char a[3];
a[0] = 45;
a[1] = 32;
a[2] = 3;
```

Array of **int**s in memory



Memory address of the array elements is decided automatically.

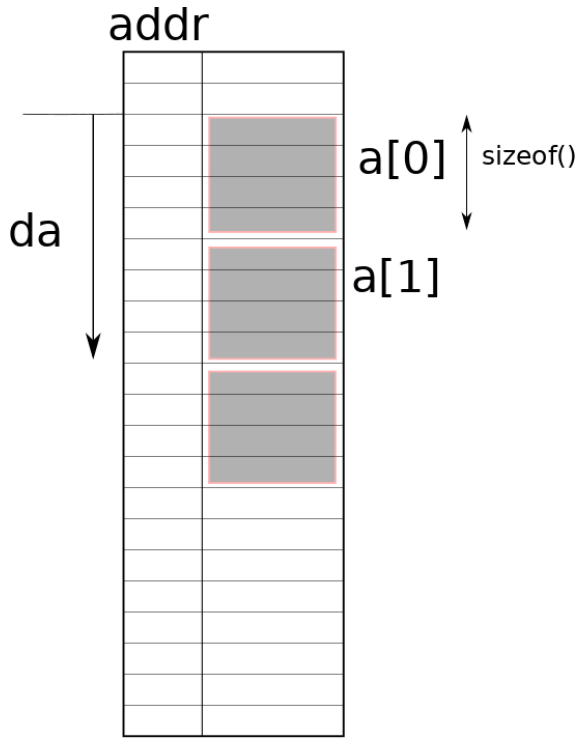
Listing 16: "Array of int"

```
int a[3];
a[0] = 457;
a[1] = 132;
a[2] = 45;
```

Memory reserved is without gaps - next **int** starts right after previous one.

Indexing

Indexes are **zero-based**, i.e. first element can be accessed using array_name[0].



To calculate position in memory element number **index**

$$da = a[0] + index \cdot sizeof(element) \tag{1}$$

This equation gives memory address of array element.
It is very fast to calculate: one multiplication and one addition.

Listing 17: "Logical error"

```
#include <iostream>

int main(){
 int a[5];
 a[0] = 100;
 a[1] = 10;
 a[2] = 1000;
 a[3] = 121;
 a[4] = 10;
 std::cout<<"a[0]="<<a[0]<<std::endl;
 std::cout<<"a[1]="<<a[1]<<std::endl;
 std::cout<<"a[2]="<<a[2]<<std::endl;
 std::cout<<"a[3]="<<a[3]<<std::endl;
 std::cout<<"a[4]="<<a[4]<<std::endl;
 // hm..
 std::cout<<"a[50]="<<a[50]<<std::endl;
 return 0;
}
```

No boundary checking

Each program is given certain range of memory addresses. If your C program asks for memory address which is outside this range - **segmentation fault** occurs. Decision is made by operating system.
C itself has no safeguards.
Memory is allocated sequentially for all variables.

Array of non-fixed size

How to reserve the memory for an array size of which we don't know?
Some of forgiving C++ compilers will allow something like that

Listing 18: Descriptive Caption Text

```
int n = 9;   // declaration of the variable
int a[n];    // declaration of the array
```

and some compilers will not.
Microsoft compiler certainly does not.

Array of non-fixed size

Better (more compatible) way is to write:

Listing 19: Proper way

```
#include <iostream>
int main(){
 int* a;      //pointer, address of 1st byte of 1st element
 int n = 45;  //variable
 a = new int[n]; //make memory for the array of int, size 45
 a[34] = 98;
 std::cout<<a[34]<<std::endl;
 delete(a); // gives memory back to Operating System
 return 0;
}
```

even if it is quite a lot type in. **delete()** is marking reserved memory as free for re-use.
Memory leaks.

Questions?