Erle Robotics: C++ Gitbook

Table of Contents

Erle Robotics: C++ Gitbook

Book

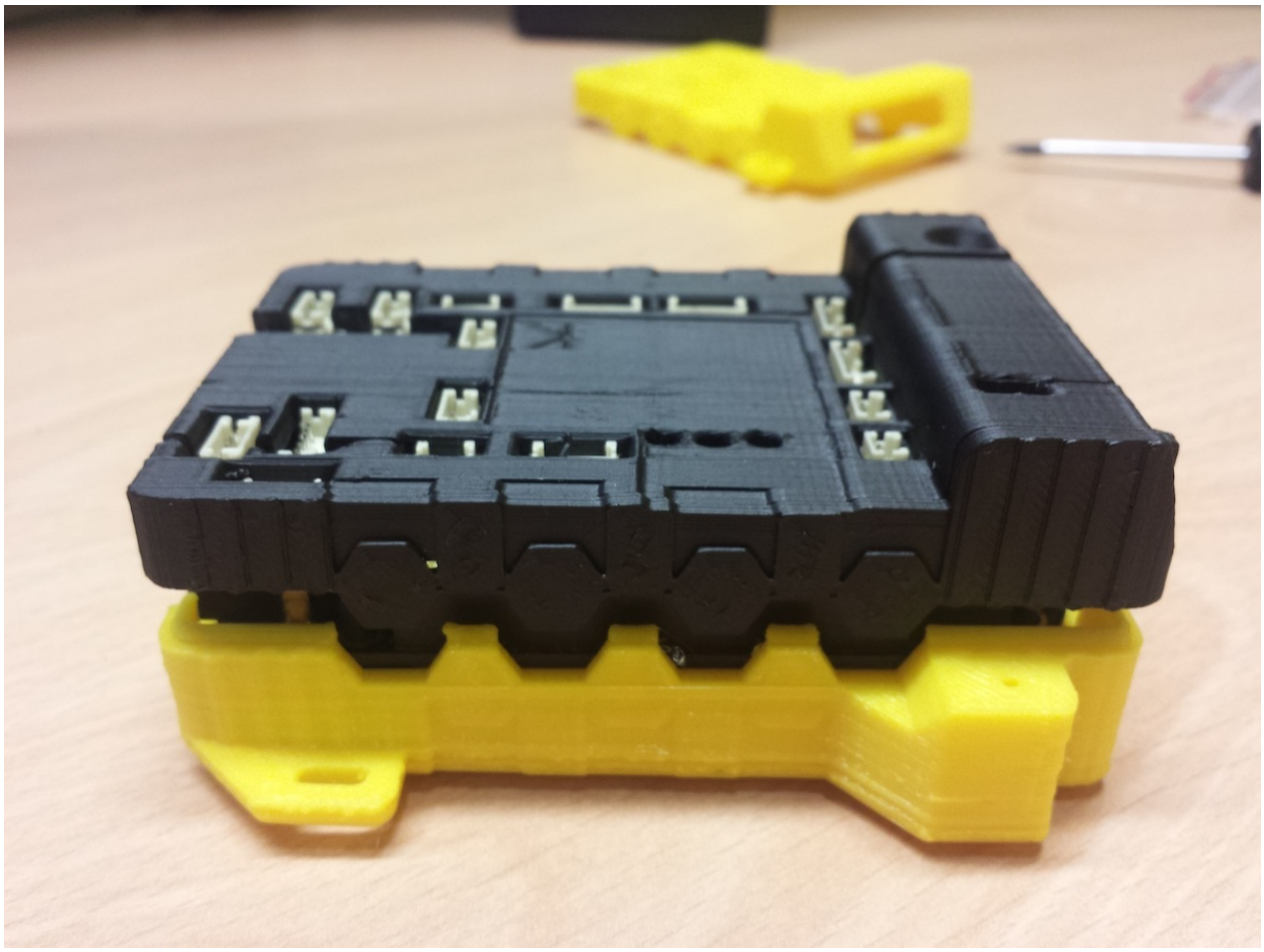
This book is a **C++ Programming Language** tutorial using Erle Robotics technology. **Erle-brain** is **a small-size Linux computer for making drones.**

Throught this tutorial you will find examples and explanations of how to use C++ sintaxis, variables, fuctions... Definitely, you will learn to program in C++ language. In the last chapter, you can also find GDB debugger basic concepts.

About

For years we've been working in the robotics field, particularly with drones. We have passed through different Universities and research centers and in all these places we actually found that most of the **drones are black boxes** (check out our 60s pitch). Not meant to be used for learning, research. The software they use is in most of the cases unknown, closed source or **not documented**. Given these conditions, how are we going to educate the next generations on this technologies? How do you get started programming drones if you don't have $1000+ budget? Which platform allows me to get started with drones without risking a hand?

We are coming up with an answer to all these questions, our technology: **Erle**.



Inspired by the BeagleBone development board, we have designed a small computer with about 36+ sensors, plenty of I/O and processing power for real-time analysis. Erle is the enabling technology for the next generation of aerial and terrestrial robots that will be used in cities solving tasks such as surveillance, enviromental monitoring or even providing aid at catastrophes.

Our small-size Linux computer is bringing robotics to the people and businesses.

License

This book is an adaptation of:

- Introduction to C++: *MIT Course Number 6.096* under Creative Commons License
- C++ exercises an example from: http://www.programmr.com/zone/cpp © All rights reserved.
- C++ official Documentation
- Simple session with GDB
- GDB docs

For any questions, concerns, or issues submit them to support [at] erlerobot.com.

C++ programming language

Through this tutorial you will learn to programm in C++.

C++ is immensely popular, particularly for applications that require speed and/or access to some low-level features. It was created in 1979 by Bjarne Stroustrup, at first as a set of extensions to the C programming language.

For more information visit: http://www.cplusplus.com/

NOTE:

All the compilable code of the book is available in code folder. In code folder you can find the sample codes organised by chapters and the solutions to the exercises.

Why to use C++?

The advantages in regard to write processor instructions are the followings:

- **Conciseness**: programming languages allow us to express common sequences of com mands more concisely. C++ provides some especially powerful shorthands.
- **Maintainability**: modifying code is easier when it entails just a few text edits, instead of rearranging hundreds of processor instructions. C++ is object oriented, which further improves maintainability.
- **Portability**: different processors make different instructions available. Programs writ ten as text can be translated into instructions for many different processors; one of C++'s strengths is that it can be used to write programs for nearly any processor.

C++ is a high-level language: when you write a program in it, the shorthands are sufficiently expressive that you don't need to worry about the details of processor instructions. C++ does give access to some lower-level functionality than other languages (e.g. memory addresses).

Compiling C++

When using C++ you will need two things: an editor and a compiler.

Editor

As editor we recommend you these two:

- *Sublime Text*: is a sophisticated text editor for code, markup and prose.You can download it for free in this link.
- *Vim Text Editor*: Vim is a highly configurable text editor built to enable efficient text editing. It is an improved version of the vi editor distributed with most UNIX systems.You can download it here. When using vim you will also be capable of writing and runing programs; if you install this Plugin: http://www.thegeekstuff.com/2009/01/tutorial-make-vim-as-your-cc-ide-using-cvim-plugin/

Compilers

About the compiler, as we have see you can use the vim text editor itselfs. Also, you need to install gcc and g++:

- Unix/Linux: http://www.cyberciti.biz/faq/howto-compile-and-run-c-cplusplus-code-in-linux/
- Macintosh: Install Xcode and command line tools
- Windows:
  - Microsoft Visual C++ 2008 Express
  - Mingw

Extensions

When you create a C++ file to be able to execute it, the file should have the **correct extension**:

Historically, the first extensions used for C++ were `.c` and `.h`, exactly like for C. This caused practical problems, especially the .c which didn't allow build systems to easily differentiate C++ and C files.

Unix, on which C++ has been developed, has case sensitive file systems. So some used `.c` for C++ files. Other used `.c++`, `.cc` and `.cxx`. `.C` and `.c++` have the problem that they aren't available on other file systems and their use quickly dropped. DOS and Windows C++ compilers tended to use `.cpp`, and some of them make the choice difficult, if not impossible, to configure. Portability consideration made that choice the most common, even outside MS-Windows.

Headers have used the corresponding `.H`, `.h++`, `.hh`, `.hxx` and `.hpp`. But unlike the main files, .h remains to this day a popular choice for C++ even with the disadvantage that it doesn't allow to know if the header can be included in C context or not. Standard headers now have no extension at all.

Additionally, some are using `.ii`, `.ixx`, `.ipp`, `.inl` for headers providing inline definitions and `.txx`, `.tpp` and `.tpl` for template definitions. Those are either included in the headers providing the definition, or manually in the contexts where they are needed.

Compilers and tools usually don't care about what extensions are used, but using an extension that they associate with C++ prevents the need to track out how to configure them so they correctly recognise the language used.

You don't need to know all this extensions, we have mentioned them here, in case you ever find yourself faced with one of them. Through this tutorial we will use mostly `.cpp` extension.

C++ and Erle-Brain

You should be able to run C++ programs in erleboard:

- Create a file with the correcte extension `.cpp`, you can use *vim text editor*.
- Use gcc to compile it.The syntax is the following:

  ```
  gcc filename.cpp -o executablename
  ```

  or

  ```
  g++ filename.cpp -o executablename
  ```

- Now you have an executable file `executablename`. Do `./executablename` to compile it.

Example:

```
root@erlerobot:~/C++# ls
hello.cpp
root@erlerobot:~/C++# gcc hello.cpp -o Hello
root@erlerobot:~/C++# ls
Hola  hello.cpp
root@erlerobot:~/C++#
root@erlerobot:~/C++# ./Hello
Hello World!
```

C++ basics

In this chapter you will learn the basics about C++ programming language.

Hello World

In the tradition of programmers everywhere, we'll use a "Hello, world!" program as an entry point into the basic features of C++.

```
#include <iostream>

int main () {
    std :: cout << " Hello , world !\n ";
    return 0;
}
```

There is also another possibilities, like this one:

```
#include <stdio.h>

int main(){
    printf("Hola Mundo! \n ");
    return 0;
}
```

Take into account the following:

- A statement is a unit of code that does something – a basic building block of a program.
- An expression is a statement that has a value – for instance, a number, a string, the sum of two numbers, etc. 4 + 2, x - 1, and "Hello, world!\n" are all expressions. Not every statement is an expression. It makes no sense to talk about the value of an `#include` statement, for instance.

Tokens

Tokens are the minimals chunk of program that have meaning to the compiler – the smallest meaningful symbols in the language. Our code displays all 6 kinds of tokens, though the usual use of operators is not present here:

Token type	Description/Purpose	Examples
Keywords	Words with special meaning to the compiler	int, double, for, auto
Identifiers	Names of things that are not built into the language	cout, std, x, myFunction
Literals	Basic constant values whose value is specified directly in the source code	"Hello, world!",24.3, 0, ’c’
Operators	Mathematical or logical operations	+, -, &&, %, <<
Punctuation/Separators	Punctuation defining the structure of a program	{ } ( ) , ;
Whitespace	Spaces of various sorts; ignored by the compiler	Spaces, tabs, newlines, comments

Line-By-Line Explanation

- `//` indicates that everything following it until the end of the line is a comment: it is ignored by the compiler. Another way to write a comment is to put it between `/*` and `*/` (e.g. x = 1 + /*sneaky comment here*/ 1;). A comment of this form may span multiple lines. Comments exist to explain non-obvious things going on in the code.

- Lines beginning with `#` are preprocessor commands, which usually change what code is actually being compiled. `#include` tells the preprocessor to dump in the contents of another file, for example `#include <stdio.h>`.

- `int main() {...}` defines the code that should execute when the program starts up. The curly braces represent grouping of multiple commands into a block.

- `cout <<` : This is the syntax for outputting some piece of text to the screen.

- *Namespaces*: In C++, identifiers can be defined within a context – sort of a directory of names – called a namespace. When we want to access an identifier defined in a namespace, we tell the compiler to look for it in that namespace using the scope resolution operator (::). Here, we're telling the compiler to look for cout in the std namespace, in which many standard C++ identifiers are defined. A cleaner alternative is to add the following line below line 2:

  ```
  using namespace std ;
  ```

This line tells the compiler that it should look in the std namespace for any identifier we haven't defined. If we do this, we can omit the std:: prefix when writing cout. This is the recommended practice.

- *Strings*: A sequence of characters such as Hello, world is known as a string. A string that is specified explicitly in a program is a string literal.

- *Escape sequences*: The `\n` indicates a newline character. It is an example of an escape sequence – a symbol used to represent a special character in a text literal. Here are all the C++ escape sequences which you can include in strings:

Escape Sequence	Represented Character
\a	System bell (beep sound)
\b	Backspace
\f	Formfeed (page break)
\n	Newline (line break)
\r	"Carriage return" (returns cursor to start of line)
\t	Tab
\	Backslash
\'	Single quote character
\"	Double quote character
\some integer x	The character represented by x

- `return 0` indicates that the program should tell the operating system it has completed successfully. This syntax will be explained in the context of functions; for now, just include it as the last line in the main block.

Note that every statement ends with a semicolon (except preprocessor commands and blocks using {}).

Operators

We can perform arithmetic calculations with operators. Operators act on expressions to form a new expression. For example, we could replace "Hello, world!\n" with (4 + 2) / 3, which would cause the program to print the number 2. In this case, the + operator acts on the expressions 4 and 2 (its operands).

Operator types:

- Mathematical: +, -, *, /, and parentheses have their usual mathematical meanings, including using - for negation. % (the modulus operator) takes the remainder of two numbers: 6 % 5 evaluates to 1.
- Logical: used for “and,” “or,” and so on. More on those in the next lecture.
- Bitwise: used to manipulate the binary representations of numbers.

Data Types

Every expression has a type – a formal description of what kind of data its value is. For instance, 0 is an integer, 3.142 is a floating-point (decimal) number, and "Hello, world!\n" is a string value (a sequence of characters).

Data of different types take a different amounts of memory to store. Here are the built-in datatypes we will use most often:

Type Names	Description	Size	Range
char	Single text character or small integer.Indicated with single quotes ('a', '3').	1 byte	signed: -128 to 127 unsigned: 0 to 255
int	Larger integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean (true/false). Indicated with the keywords true and false.	1 byte	Just true (1) or false (0).
double	"Doubly" precise floating point number.	8 bytes	+/- 1.7e +/- 308 (15 digits)

Notes on this table:

- A signed integer is one that can represent a negative number; an unsigned integer will never be interpreted as negative, so it can represent a wider range of positive numbers. Most compilers assume signed if unspecified.
- There are actually 3 integer types: short, int, and long, in non-decreasing order of size (int is usually a synonym for one of the other two). You generally don't need to worry about which kind to use unless you're worried about memory usage or you're using really huge numbers. The same goes for the 3 floating point types, float, double, and long double, which are in non-decreasing order of precision (there is usually some imprecision in representing real numbers on a computer).
- The sizes/ranges for each type are not fully standardized; those shown above are the ones used on most 32-bit computers. An operation can only be performed on compatible types. You can add 34 and 3, but you can't take the remainder of an integer and a floating-point number.

An operator also normally produces a value of the same type as its operands; thus, 1 / 4 evaluates to 0 because with two integer operands, / truncates the result to an integer. To get 0.25, you'd need to write something like 1 / 4.0. A text string, has the type char *.

Variables

We might want to give a value a name so we can refer to it later. We do this using variables. A variable is a named location in memory. For example, say we wanted to use the value 4 + 2 multiple times. We might call it x and use it as follows:

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

int main () {
    int x ;
    x = 4 + 2;
    cout << x / 3 << ' ' << x * 2;

    return 0;
}
```

(Note how we can print a sequence of values by “chaining” the << symbol.) The name of a variable is an identifier token. Identifiers may contain numbers, letters, and underscores (_), and may not start with a number. Line 5 is the declaration of the variable x. We must tell the compiler what type x will be so that it knows how much memory to reserve for it and what kinds of operations may be performed on it. Line 6 is the initialization of x, where we specify an initial value for it. This introduces a new operator: =, the assignment operator. We can also change the value of x later on in the code using this operator. We could replace lines 5 and 6 with a single statement that does both declaration and initialization: int x = 4 + 2;

This form of declaration/initialization is cleaner, so it is to be preferred.

Input

Now that we know how to give names to values, we can have the user of the program input values. This is demonstrated in line 6 below:

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

int main () {
    int x ;
    cin >> x ;
    cout << x / 3 << ' ' << x * 2;

    return 0;
}
```

Just as `cout <<` is the syntax for outputting values, `cin >>` (line 6) is the syntax for inputting values.

Exercises:Basics

Exercise 1

Write a program that adds two inputed numbers.

Solution

Exercise 2

Write a program that ask for a name and say hello.

Solution

Exercise 3

Write a program that evaluates the following things for two int numbers:

Sum,Diff,Product,Quotient,Remainder,Increment first num., Decrement Second num. in num3 =0.

Solution

Flow control

Sometimes is necessary to alter the order in which a program's statements are executed, the control flow.

In this chapter we will discuss how to do it.

Operators

Conditionals (*if statements*) use two kinds of special operators: relational and logical. These are used to determine whether some condition is true or false.

Relational Operators

The relational operators are used to test a relation between two expressions:

Operator	Meaning
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to

They work the same as the arithmetic operators (e.g., a > b) but return a Boolean value of either true or false, indicating whether the relation tested for holds. (An expression that returns this kind of value is called a Boolean expression.) For example, if the variables x and y have been set to 6 and 2, respectively, then x > y returns true. Similarly, x < 5 returns false.

Logical Operators

The logical operators are often used to combine relational expressions into more complicated Boolean expressions:

Operator	Meaning
&&	and
double pipe	or
!	not

The `and` and `or` operators return true or false, according to the rules of logic:

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

The `not` operator is a unary operator, taking only one argument and negating its value:

a	!a
true	false
false	true

Of course, Boolean variables can be used directly in these expressions, since they hold true and false values. In fact, any kind of value can be used in a Boolean expression due to a quirk C++ has: false is represented by a value of 0 and anything that is not 0 is true. So, “Hello, world!” is true, 2 is true, and any int variable holding a non-zero value is true. This means !x returns false and x && y returns true.

Conditionals

The if conditional has the form:

```
if(condition){

    statement1

    statement2
    …
}
```

The condition is some expression whose value is being tested. If the condition resolves to a value of true, then the statements are executed before the program continues on. Otherwise, the statements are ignored. If there is only one statement, the curly braces may be omitted, giving the form:

```
if(condition)

    statement
```

The if-else form is used to decide between two sequences of statements referred to as blocks:

```
if(condition){

    statementA1

    statementA2

    …
}else{

    statementB1

    statementB2

    …
}
```

If the condition is met, the block corresponding to the if is executed. Otherwise, the block corresponding to the else is executed. Because the condition is either satisfied or not, one of the blocks in an if-else must execute. If there is only one statement for any of the blocks, the curly braces for that block may be omitted:

```
if(condition)

    statementA1

else

    statementB1
```

The else if is used to decide between two or more blocks based on multiple conditions:

```
if(condition1){

    statementA1

    statementA2

    …

} else if(condition2){
```

```
    statementB1

    statementB2

    …
}
```

If condition1 is met, the block corresponding to the if is executed. If not, then only if condition2 is met is the block corresponding to the else if executed. There may be more than one else if, each with its own condition. Once a block whose condition was met is executed, any else ifs after it are ignored. Therefore, in an if-else-if structure, either one or no block is executed.

An else should be included at the end. If none of the previous conditions is met the else is executed.

Here you have a simple example:

```
#include <iostream>

using namespace std;

int main() {

  int x = 6;
  int y = 2;

  if(x > y)

    cout << "x is greater than y\n";

  else if(y > x)

    cout << "y is greater than x\n";

  else

    cout << "x and y are equal\n";

  return 0;

 }
```

The output of this program is x is greater than y. If we replace lines 5 and 6 with

```
int x = 2;

int y = 6;
```

then the output is y is greater than x. If we replace the lines with:

```
int x = 2;

int y = 2;
```

then the output is x and y are equal.

switch-case

The switch-case is another conditional structure that may or may not execute certain statements. However, the switch-case has peculiar syntax and behavior:

```
switch(expression){
  case constant1:

    statementA1

    statementA2

    ...

    break;

  case constant2:

    statementB1

    statementB2

    ...

    break;

  ...

  default:

    statementZ1

    statementZ2

    ...

}
```

The switch evaluates expression and, if expression is equal to constant1, then the statements beneath case constant 1: are executed until a break is encountered. If expression is not equal to constant1, then it is compared to constant2. If these are equal, then the statements beneath case constant 2: are executed until a break is encountered. If not, then the same process repeats for each of the constants, in turn. If none of the constants match, then the statements beneath default: are executed. Due to the peculiar behavior of switch-cases, curly braces are not necessary for cases where there is more than one statement (but they are necessary to enclose the entire switch-case). switch-cases generally have if-else equivalents but can often be a cleaner way of expressing the same behavior. Here is an example using switch-case:

```
#include <iostream>

using namespace std;

int main() {

    int x = 6;

    switch(x){
        case 1:
            cout << "x is 1\n";
            break;
        case 2:
        case 3:
            cout << "x is 2 or 3";
            break;
        default:
            cout << "x is not 1, 2, or 3";
    }

    return 0;
}
```

This program will print x is not 1, 2, or 3. If we replace line 5 with int x = 2; then the program will print x is 2 or 3.

Loops

Conditionals execute certain statements if certain conditions are met; loops execute certain statements while certain conditions are met. C++ has three kinds of loops: while, do-while, and for.

while and do-while

The while loop has a form similar to the if conditional:

```
while(condition){

    statement1

    statement2

    …
}
```

As long as condition holds, the block of statements will be repeatedly executed. If there is only one statement, the curly braces may be omitted. Here is an example:

```
#include <iostream>

using namespace std;

int main() {
    int x = 0;

    while(x < 10){
      x = x + 1;
      cout << "x is " << x << "\n";
    }
    return 0;
}
```

This program will print x is 10. The do-while loop is a variation that guarantees the block of statements will be executed at least once:

```
do{
    statement1

    statement2

    …
}while(condition);
```

The block of statements is executed and then, if the condition holds, the program returns to the top of the block. Curly braces are always required. Also note the semicolon after the while condition.

for

The for loop works like the while loop but with some change in syntax:

```
for(initialization; condition; incrementation){

  statement1

  statement2

  …

}
```

The for loop is designed to allow a counter variable that is initialized at the beginning of the loop and incremented (or decremented) on each iteration of the loop. Curly braces may be omitted if there is only one statement. Here is an example:

```
#include <iostream>

using namespace std;

int main() {

  for(int x = 0; x < 10; x = x + 1)
    cout << x << "\n";

  return 0;

}
```

This program will print out the values 0 through 9, each on its own line. If the counter variable is already defined, there is no need to define a new one in the initialization portion of the for loop. Therefore, it is valid to have the following:

```
 #include <iostream>

using namespace std;

int main() {

    int x = 0;

    for(; x < 10; x = x + 1)

        cout << x << "\n";
    return 0;
 }
```

Note that the first semicolon inside the for loop's parentheses is still required. A for loop can be expressed as a while loop and vice-versa. Recalling that a for loop has the form

```
for(initialization; condition; incrementation){

  statement1

  statement2

  …

}
```

we can write an equivalent while loop as initialization

```
while(condition){

  statement1

  statement2

  …

  incrementation

}
```

Using our example above,

```
#include <iostream>

using namespace std;

int main() {

  for(int x = 0; x < 10; x = x + 1)

     cout << x << "\n";

  return 0;
}
```

is converted to

```
#include <iostream>

using namespace std;

int main() {

  int x = 0;

  while(x < 10){
    cout << x << "\n";
    x = x + 1;
  }
  return 0;
 }
```

The incrementation step can technically be anywhere inside the statement block, but it is good practice to place it as the last step, particularly if the previous statements use the current value of the counter variable.

Exercises: Flow Control

Exercise 1

Write a program that checks if the angles given can make a triangle or not (their sum must be 180 degrees).

Solution

Exercise 2

Write a program that prints on the screen all the even numbers up to 10.

Solution

Exercise 3

Initialize y=0 and x=6. Then write the appropiate staments to print *x is greater than y* on one line and then *x is equal to 6* on the next line. Do this using if statement.

Solution

Exercise 4

Write a program that tells you if a number is even or odd.

Solution

Functions

Functions are a set of instructions that perform a specific task.This chapter will focus on the declaration an use of the functions in C++.

Built-in functions

Simple examples

For now you know how to calculate 3^4 using for loop, like this example:

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

int main() {
      int threeExpFour = 1;

     for (int i = 0; i < 4; i = i + 1){
           threeExpFour = threeExpFour * 3;
     }
     cout << "3^4 is " << threeExpFour << endl;
     return 0;
}
```

This example shows the result of 3^4. Now we want to do the same but using functions.That is, define the function `threeExpFour` which calculates the value of 3^4.This is very simple if we use a built-in function: `pow(base,exp)` from the `cmath` library.

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

int main() {
    int threeExpFour = pow(3, 4);
    cout << "3^4 is " << threeExpFour << endl;

    return 0;
}
```

In the `cmath` module you can find many others mathematical functions. Here you will find a list.

This link leads you to the official web site library. Here you can find all the built-in functions.

Finally, here is another list of built-in functions, with use examples for each of them.

Define your own function

Continuing with the example of the previus section, what about defining our own fuction to do 3^4, like this:

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

// some code which raises an arbitrary integer
// to an arbitrary power

int main() {

 int threeExpFour = raiseToPower(3, 4);

 cout << "3^4 is " << threeExpFour << endl;

 return 0;
}
```

The function `raiseToPower` must be declared, like this:

```
int raiseToPower(int base, int exponent){
 int result = 1;
 for (int i = 0; i < exponent; i = i + 1){
  result = result * base;
 }
 return result;
}
```

So, the function declaration syntax is the following:

```
[return type][functioname](arguments)
{
statement1;
statement2;
...
}
```

Note that arguments are declared with type: `int base` .They are separated by ','.

So the final result is this:

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

int raiseToPower(int base, int exponent) {
    int result = 1;
    for (int i = 0; i < exponent; i = i + 1){
        result = result * base;
    }
    return result;
}

int main() {
    int threeExpFour = raiseToPower(3, 4);
    cout << "3^4 is " << threeExpFour << endl;

    return 0;
}
```

Note that the function must be declared before it is called.

Key ideas when declaring functions

The key ideas when declaring the function are the following:

- Up to one value may be returned; it must be the same type as the return type.
- If no values are returned, give the function a void return type. (e.g.: `return 0;` or `return;`)
- If there are many functions with the same name, but different arguments, the called function is the one whose arguments match the invocation.
- `void` type is used when the function has no return value:

```
void printOnNewLine(int x)
{
  cout << "1 Integer: " << x << endl;
}
```

- Function declarations need to occur before invocations. You can use a function prototype to inform the compiler you'll implement it later(Function prototypes should match the signature of the method, though argument names don't matter):

```
int square(int); //This is the prototype

int cube(int x)
{
    return x*square(x);
}

int square(int x) //Here is the declaration
{
    return x*x;
}
```

- Function prototypes are generally put into separate header files – Separates specification of the function from its implementation :

```
// myLib.h - header
// contains prototypes

int square(int);
int cube (int);
```

Libraries are generally distributed as the header file containing the prototypes.

```
// myLib.cpp - implementation
#include "myLib.h"

int cube(int x){
    return x*square(x);
}

int square(int x){
    return x*x;
}
```

Library user only needs to know the function prototypes (in the header file), not the implementation source code (in the .cpp file) . The Linker (part of the compiler) takes care of locating the implementation of functions in the .dll file at compile time .

- Functions can call themselves.

```
int fibonacci(int n){
    if (n == 0 || n == 1) {
```

```
        return 1;
    }else {
        return fibonacci(n-2) + fibonacci(n-1);
    }
}
```

Global Variables

They are vaiables that can be accessed from any function. For example, how many times is function foo() called?

```
#include <iostream>

int numCalls = 0;

void foo() {
 ++numCalls;
}

int main() {
 foo(); foo(); foo();
 std::cout << numCalls << endl; // 3

 return 0;
}
```

`numCalls` is a global variable. Where a variable was declared, determines where it can be accessed from.

Pass by reference

So far we've been passing everything by value – makes a copy of the variable; changes to the variable within the function don't occur outside the function . For example:

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

// pass-by-value

void increment(int a) {
 a = a + 1;
 cout << "a in increment " << a << endl;
}

int main() {
 int q = 3;
 increment(q); // does nothing
 cout << "q in main " << q << endl;
}
```

The result of this program is:

```
a in increment 4
q in main 3
```

As you can see the original variable has not been modified. If you want to modify the original variable as opposed to making a copy, pass the variable by reference (`int &a` instead of `int a`).

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

//pass by-reference
void increment(int &a) {
 a = a + 1;
 cout << "a in increment " << a << endl;
}

int main() {
 int q = 3;
 increment(q); // works
 co
```

The output is:

```
a in increment 4
q in main 4
```

As you can see the q value (in the main program) has been updated too.

Returning multile values

The return statement only allows you to return 1 value. Passing output variables by reference overcomes this limitation. For example:

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

int divide(int numerator, int denominator, int &remainder)
{
    remainder = numerator % denominator;
    return numerator / denominator;
}

int main() {
    int num = 14;
    int den = 4;
    int rem;

    int result = divide(num, den, rem);

    cout << result << "*" << den << "+" << rem << "=" << num << endl;
    //The print on the screen is: 3*4+2=12
}
```

Here we obtain the cotient=`numerator/denominator` stored in the result value.(The cotient is the return of the function divide) We also obtain the value of the remainder, which has been passed by reference, that means its value in the main program has been updated when passed to the function.

Exercises: Functions

Exercise 1

Write a program that calculates 6^5. Declare your own function to do this.

Solution

Exercise 2

Write a program that asks a name say hello. Use your own function, that recives a string of characters (name) and prints on screen the hello message. (Doesn't returns anything- `void` type)

Solution

Exercise 3

Write a program that ask for two numbers, compare them and show the maximun.Declare a function called `max_two` that compares the numbers and returns the maximun.

Solution

Exercise 4

Write a program that asks the user for an integer number and find the sum of all natural numbers upto that number.

Solution

Arrays and Strings

In this chapter we explore a means to store multiple values together as one unit, the array. A string is simply a character array and can be manipulated as such.

Arrays basics

Unidimensional arrays

An array is a fixed number of elements of the same type stored sequentially in memory. Therefore, an integer array holds some number of integers, a character array holds some number of characters, and so on. The size of the array is referred to as its dimension. To declare an array in C++, we write the following:

```
type arrayName[dimension];
```

example:

```
int arr[4];
```

The elements of an array can be accessed by using an index into the array. Arrays in C++ are zero-indexed, so **the first element has an index of 0**. So, to access the third element in arr, we write `arr[2]`; The value returned can then be used just like any other integer.

Like normal variables, the elements of an array must be initialized before they can be used; otherwise we will almost certainly get unexpected results in our program. There are several ways to initialize the array. One way is to declare the array and then initialize some or all of the elements:

```
int arr[4];

arr[0] = 6;

arr[1] = 0;

arr[2] = 9;

arr[3] = 6;
```

Another way is to initialize some or all of the values at the time of declaration:

```
int arr[4] = { 6, 0, 9, 6 };
```

Sometimes it is more convenient to leave out the size of the array and **let the compiler determine the array's size** for us, based on how many elements we give it:

```
int arr[] = { 6, 0, 9, 6, 2, 0, 1, 1 };
```

Here, the compiler will create an integer array of dimension 8.

The array can also be initialized with values that are not known beforehand:

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

int main() {

 int arr[4];

 cout << "Please enter 4 integers:" << endl;
```

```
  for(int i = 0; i < 4; i++)
    cin >> arr[i];\\Fill in the array

  cout << "Values in array are now:";

  for(int i = 0; i < 4; i++)
    cout << " " << arr[i];\\ Show the content of the array

  cout << endl;

  return 0;
}
```

Note that when accessing an array the index given must be a positive integer from 0 to n-1, where n is the dimension of the array. The index itself may be directly provided, derived from a variable, or computed from an expression:

```
arr[5];

arr[i];

arr[i+3];
```

Arrays can also be passed as arguments to functions. When declaring the function, simply specify the array as a parameter, without a dimension. The array can then be used as normal within the function. For example:

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

int sum(const int array[], const int length) {
  long sum = 0;
  for(int i = 0; i < length; sum += array[i++]);
  return sum;
}

int main() {

  int arr[] = {1, 2, 3, 4, 5, 6, 7};
  cout << "Sum: " << sum(arr, 7) << endl;

  return 0;
}
```

The function `sum` takes a constant integer array and a constant integer length as its arguments and adds up length elements in the array. It then returns the `sum`, and the program prints out `Sum: 28`.

It is important to note that arrays are passed by reference and so any changes made to the array within the function will be observed in the calling scope.

Multidimensional arrays

C++ also supports the creation of multidimensional arrays, through the addition of more than one set of brackets. Thus, a two-dimensional array may be created by the following:

```
type arrayName[dimension1][dimension2];
```

The array will have dimension1 x dimension2 elements of the same type and can be thought of as an array of arrays. The first index indicates which of dimension1 subarrays to access, and then the second index accesses one of dimension2 elements within that subarray. Initialization and access thus work similarly to the one-dimensional case:

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

int main() {
```

```
  int twoDimArray[2][4];

  twoDimArray[0][0] = 6;

  twoDimArray[0][1] = 0;

  twoDimArray[0][2] = 9;

  twoDimArray[0][3] = 6;

  twoDimArray[1][0] = 2;

  twoDimArray[1][1] = 0;

  twoDimArray[1][2] = 1;

  twoDimArray[1][3] = 1;

  for(int i = 0; i < 2; i++)

    for(int j = 0; j < 4; j++)

      cout << twoDimArray[i][j];
      out << endl;

  return 0;

}
```

The array can also be initialized at declaration in the following ways:

```
int twoDimArray[2][4] = { 6, 0, 9, 6, 2, 0, 1, 1 };

int twoDimArray[2][4] = { { 6, 0, 9, 6 } , { 2, 0, 1, 1 } };
```

Note that **dimensions must always be provided** when initializing multidimensional arrays, as it is otherwise impossible for the compiler to determine what the intended element partitioning is. For the same reason, when multidimensional arrays are specified as arguments to functions, all dimensions but the first must be provided (the first dimension is optional), as in the following:

```
int aFunction(int arr[][4]) { … }
```

Multidimensional arrays are merely an abstraction for programmers, as all of the elements in the array are sequential in memory. Declaring `int arr[2][4];` is the same thing as declaring `int arr[8];`.

Strings basics

String literals such as "Hello, world!" are actually represented by C++ as a sequence of characters in memory. In other words, a string is simply a character array and can be manipulated as such. Consider the following program:

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

int main() {

  char helloworld[] = { 'H', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!', '\0' };
  cout << helloworld << endl;
  return 0;

}
```

This program prints *Hello, world!*. Note that the character array helloworld ends with a special character known as the **null character**. This character is used to indicate the end of the string.

Character arrays can also be initialized using string literals. In this case, no null character is needed, as the compiler will automatically insert one:

```
char helloworld[] = "Hello, world!";
```

The individual characters in a string can be manipulated either directly by the programmer or by using special functions provided by the C/C++ libraries. These can be included in a program through the use of the `#include` directive. Of particular note are the following:

- `cctype (ctype.h)` : character handling
- `cstdio (stdio.h)` : input/output operations
- `cstdlib (stdlib.h)` : general utilities
- `cstring (string.h)` : string manipulation

Here is an example to illustrate the cctype library:

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

int main() {

  char messyString[] = "t6H0I9s6.iS.999a9.STRING";
  char current = messyString[0];

  for(int i = 0; current != '\0'; current = messyString[++i]) {

    if(isalpha(current))

      cout << (char)(isupper(current) ? tolower(current) : current);

    else if(ispunct(current))

      cout << ' ';

  }
  cout << endl;

  return 0;

 }
```

This example uses the `isalpha` , `isupper` , `ispunct` , and `tolowe` r functions from the cctype library. The `is-` functions

check whether a given character is an alphabetic character, an uppercase letter, or a punctuation character, respectively. These functions return a Boolean value of either true or false. The `tolower` function converts a given character to lowercase.

The for loop beginning at line 9 takes each successive character from messyString until it reaches the null character. On each iteration, if the current character is alphabetic and uppercase, it is converted to lowercase and then displayed. If it is already lowercase it is simply displayed. If the character is a punctuation mark, a space is displayed. All other characters are ignored. The resulting output is this is a string. For now, ignore the `(char)` on line 11; we will cover later.

Here is an example to illustrate the cstring library:

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

int main() {

  char fragment1[] = "I'm a s";

  char fragment2[] = "tring!";

  char fragment3[20];

  char finalString[20] = "";

  strcpy(fragment3, fragment1);

  strcat(finalString, fragment3);

  strcat(finalString, fragment2);

  cout << finalString;

  return 0;

}
```

This example creates and initializes two strings, `fragment1` and `fragment2`. `fragment3` is declared but not initialized. `finalString` is partially initialized (with just the null character). `fragment1` is copied into `fragment3` using `strcpy`, in effect initializing `fragment3` to "I'm a s". `strcat` is then used to concatenate `fragment3` onto `finalString` (the function overwrites the existing null character), thereby giving `finalString` the same contents as `fragment3`. Then `strcat` is used again to concatenate `fragment2` onto `finalString`. `finalString` is displayed, giving "I'm a string!".

Exercises: Arrays

Exercise 1

Create an array that can hold ten integers, and get input from user. Display those values on the screen, and then prompt the user for an integer. Search through the array, and count the number of times the item is found.

Solution

Exercise 2

Write a program that asks for an index and a number. Then it includes the number at the indicated index of the array = {1,2,3,4,5,6} and moves a position forward (from u to u+1) each element after the selected index.

Solution

Exercise 3

Write a program that reverses a string and prints it on the screen.

Solution

Pointers

Memory addresses, or pointers, allow us to manipulate data much more flexibly; manipulating the memory addresses of data can be more efficient than manipulating the data itself.In this chapter we will learn how to use them.

Pointers and their behavior

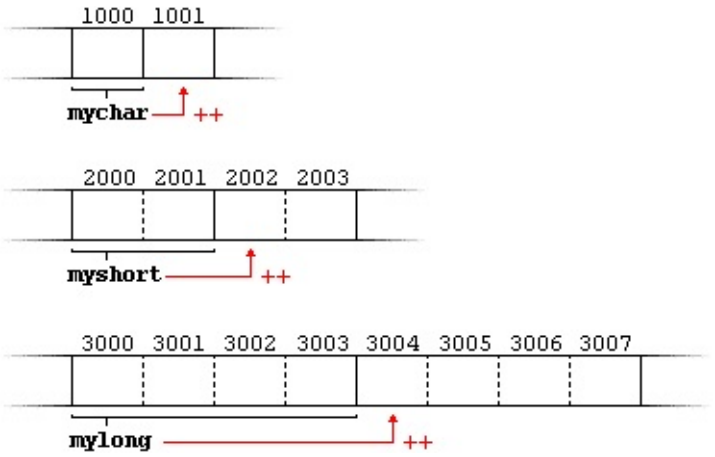
Variables are explained as locations in the computer's memory which can be accessed by their identifier (their name). This way, the program does not need to care about the physical address of the data in memory; it simply uses the identifier whenever it needs to refer to the variable.

For a C++ program, the memory of a computer is like a succession of memory cells, each one byte in size, and each with a unique address. These single-byte memory cells are ordered in a way that allows data representations larger than one byte to occupy memory cells that have consecutive addresses.

This way, each cell can be easily located in the memory by means of its unique address. For example, the memory cell with the address 1776 always follows immediately after the cell with address 1775 and precedes the one with 1777, and is exactly one thousand cells after 776 and exactly one thousand cells before 2776.

When a variable is declared, the memory needed to store its value is assigned a specific location in memory (its memory address). Generally, C++ programs do not actively decide the exact memory addresses where its variables are stored. It may be useful for a program to be able to obtain the address of a variable during runtime in order to access data cells that are at a certain position relative to it.

Pointers are just variables storing integers, that happend to be memory addresses. We can acces the value of a variable deferencing the pointer.



Pointers usage/syntaxis

Declaring Pointers

To declare a pointer variable named `ptr` that points to an integer variable named x:

```
int *ptr = &x;
```

`int *ptr` declares the pointer to an integer value, which we are initializing to the address of x.

We can have pointers to values of any type. The general scheme for declaring pointers is:

```
data_type *pointer_name; // Add "= initial_value " if applicable
```

Pointer name is then a variable of type data type * – a "pointer to a data type value."

Using Pointer Values

Once a pointer is declared, we can dereference it with the * operator to access its value:

```
cout << *ptr;
// Prints the value pointed to by ptr.

// which in the above example would be x’s value
```

We can use deferenced pointers as l-values:

```
*ptr = 5; // Sets the value of x
```

Without the * operator, the identifier x refers to the pointer itself, not the value it points to:

```
cout << ptr; // Outputs the memory address of x in base 16
```

Just like any other data type, we can pass pointers as arguments to functions. The same way we’d say void func(int x) {...}, we can say void func(int *x){...}. Here is an example of using pointers to square a number in a similar fashion to pass-by-reference:

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

void squareByPtr ( int * numPtr ) {
    * numPtr = * numPtr * * numPtr ;
}

int main () {
    int x = 5;
    squareByPtr (& x);
    cout << x; // Prints 25

    return 0;
}
```

Note that in line 2 * has different uses.

const Pointer

There are two places the `const` keyword can be placed within a pointer variable declaration. This is because there are two different variables whose values you might want to forbid changing: the pointer itself and the value it points to.

```
const int *ptr;
```

declares a changeable pointer to a constant integer. The integer value cannot be changed through this pointer, but the pointer may be changed to point to a different constant integer.

```
int * const ptr;
```

declares a constant pointer to changeable integer data. The integer value can be changed through this pointer, but the pointer may not be changed to point to a different constant integer.

```
const int * const ptr;
```

forbids changing either the address ptr contains or the value it points to.

Pointers:Runtime Error

Some pointers do not point to valid data; dereferencing such a pointer is a runtime error.

Any pointerset to 0 is called a null pointer, and since there is no memory location 0, it is an *invalid pointer*. One should generally check whether a pointer is null before dereferencing it. Pointers are often set to 0 to signal that they are not currently valid. Dereferencing pointers(*unlocated pointers*) to data that has been erased from memory also usually causes runtime errors. Example:

```
int * myFunc () {
    int phantom = 4;
    return & phantom ;
}
```

phantom is deallocated when myFunc exits, so the pointer the function returns is invalid. As with any other variable, the value of a pointer is *undefined* until it is initialized, so it may be invalid.

References

When we write void f(int &x) {...} and call f(y), the reference variable x becomes another name – an alias – for the value of y in memory. We can declare a reference variable locally, as well:

```
int y;

int &x = y; // Makes x a reference to, or alias of, y
```

After these declarations, changing x will change y and vice versa, because they are two names for the same thing.

References are just pointers that are dereferenced every time they are used. Just like pointers, you can pass them around, return them, set other references to them, etc. The only differences between using pointers and using references are:

- References are sort of pre-dereferenced – you do not dereference them explicitly.
- You cannot change the location to which a reference points, whereas you can change the location to which a pointer points. Because of this, references must always be initialized when they are declared.
- When writing the value that you want to make a reference to, you do not put an & before it to take its address, whereas you do need to do this for pointers.

We will talk about this topic later, in classes chapter.

Pointers and arrays

The name of an array is actually a pointer to the first element in the array. Writing `myArray[3]` tells the compiler to return the element that is 3 away from the starting element of myArray. This explains why arrays are always passed by reference: passing an array is really passing a pointer.

This also explains why array indices start at 0: the first element of an array is the element that is 0 away from the start of the array

Pointer arithmetic

pointer arithmetic is a way of using subtraction and addition of pointers to move around between locations in memory, typically between array elements. Adding an integer n to a pointer produces a new pointer pointing to n positions further down in memory.

Take the following code snippet:

```
long arr [] = {6 ,0,9,6};
long * ptr = arr ;
ptr ++;
long *ptr2 = arr + 3;
```

When we add 1 to ptr in line 3, we don't just want to move to the next byte in memory, since each array element takes up multiple bytes; we want to move to the next element in the array. The C++ compiler automatically takes care of this, using the appropriate step size for adding to and subtracting from pointers. Thus, line 3 moves ptr to point to the second element of the array.

Similarly, we can add/subtracttwo pointers: ptr2 - ptr givesthe number of array elements between ptr2 and ptr (2). All addition and subtraction operations on pointers use the appropriate step size.

Arrays acces notation

Because of the interchangeability of pointers and array names, array-subscript notation (the form `myArray[3]`) can be used with pointers as well as arrays. When used with pointers, it is referred to as pointer-subscript notation.

An alternative is pointer-offset notation, in which you explicitly add your offsetto the pointer and dereference the resulting address. For instance, an alternate and functionally identical way to express `myArray[3]` is `*(myArray + 3)`.

char * Strings

You should now be able to see why the type of a string value is char *: a string is actually an array of characters. When you set a char* to a string, you are really setting a pointer to point to the first character in the array that holds the string.

You cannot modify string literals; to do so is either a syntax error or a runtime error, depending on how you try to do it. (String literals are loaded into read-only program memory at program startup.) You can, however, modify the contents of an array of characters. Consider the following example:

```
char courseName1[] = {'6 ', '. ', '0 ', '9 ', '6 ', '\0 ' };
char *courseName2 = "6.096 ";
```

Attempting to modify one of the elements courseName1 is permitted, but attempting to modify one of the characters in courseName2 will generate a runtime error, causing the program to crash.

Exercises: Pointers

Exercise 1

Write a program that asks the user to enter integers as inputs to be stored in the variables 'a' and 'b' respectively. There are also two integer pointers named ptrA and ptrB. Assign the values of 'a' and 'b' to ptrA and ptrB respectively, and display them.

Solution

Exercise 2

Write a C++ program to find the max of an integral data set. The program will ask the user to input the number of data values in the set and each value. The program prints on screen a pointer that points to the max value.

Solution

Exercise 3

Take input in variable and display same value by pointer.

Solution

Exercise 4

Given the string "A string." Print on one line the letter on the index 0, the pointer position and the letter t. undate the pointer to pointer +2. Then, in another line print the pointer and the letters r and g of the string (using the pointer).

Solution

Classes and structs

In C++, the concept of structure has been generalized in an object-oriented sense.

- Structures are a way of storing many different values in variables of potentially different types under the same name.
- Classes define types of data structures and the functions that operate on those data structures.

In this chapter we will learn about classes and structures in C++.

Classes

A user-defined datatype which groups together related pieces of information is a class. In other words, classes are types representing groups of similar instances; each instance has certain fields that define it (*instance variables*). Instances also have functions that can be applied to them (represented as function fields and called *methods*).Also offers the possibility to the programmer to limit access to parts of the class (to only those functions that need to know about the internals).

Let's see an example:*Representing a (Geometric) Vector*.

- In the context of geometry, a vector consists of 2 points: a start and a finish.
- Each point itself has an x and y coordinate.
- Our representation will use 4 doubles (startx, starty, endx, endy).
- We need to pass all 4 doubles to functions.

So for now if we want to define the vector and print it we will do something like this.

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



void printVector(double x0, double x1, double y0, double y1) {
 cout << "(" << x0 << "," << y0 << ") -> ("
 << x1 << "," << y1 << ")" << endl;
}


int main() {
 double xStart = 1.2;
 double xEnd = 2.0;
 double yStart = 0.4;
 double yEnd = 1.6;

printVector(xStart,xEnd,yStart,yEnd);

return 0;
}
```

Now we are going to define a vector as a class:

```


class Vector {
public:
 double xStart;
 double xEnd;
 double yStart;
 double yEnd;
};
```

So with this, we can see the syntaxis of a class:

```
class ClassName {
  access_modifier1:
    field11_definition;
    field12_definition;
    …
  access_modifer2:
    field21_definition;
    field22_definition;
  …
};
```

ClassName is a new type.(In the example above Vector). The possible *access modifiers* are: public,private and protected. *Fields* indicate what related pieces of information our datatype consists of (They are also called members). Fields can be variables or functions and can have different types.

```
class MITStudent {
public:
 char *name;
 int studentID;
};
```

Structures

Structs are a carry-over from the C; in C++, classes are generally used. In C++, they're essentially the same as classes, except structs' default access modifier is public.

Class example:

```
class Point {
public:
 double x;
 double y;
};
```

Struct example:

```
struct Point {
 double x;
 double y;
};
```

In C++ struct and class can be used interchangeably to create a class with one exception:

- struct: public by default.
- class: private by default.

Access Modifiers

Acces Modifiers define where your fields/methods can be accessed from.

- Fields marked as `public` can be accessed by anyone.
- Fields marked as `private` can only be accessed by functions that are part of that class.

Classes are `private` by default, this means that:

```
class Point {
 double x, y;
};
```

is the same as:

```
class Point {
private:
 double x, y;
};
```

Now, let's analyze this example:

```
class Robot {
  public:
    float getX() { return locX; }
    float getY() { return locY; }
    float getFacing() { return facing; }
    void setFacing(float f) { facing = f; }
    void setLocation(float x, float y);
  private:
    float locX;
    float locY;
    float facing;
};
```

Here is eveidenced that you can use getters to allow read-only access to private fields.

With `protected`, C++ allows users to create classes based on other classes:

- For example: FordCar class based on a general Car class.
- The general Car class has fields (variables and functions that describe all cars). The FordCar class then uses the fields from the general Car class and adds fields specific to FordCars done with inheritance. Public fields are inherited (as public) and private fields are not inherited. protected fields are like private, but can be inherited.

Instances

An instance is an occurrence of a class. Different instances can have their own set of values in their fields. If we continue the example shown before:

```
class MITStudent {
public:
 char *name;
 int studentID;
};
```

Now, if you wanted to represent 2 different students (who can have different names and IDs), you would use 2 instances of MITStudent. So we define 2 instances of MITStudent: one called student1, the other called student2.

```
class MITStudent {
public:
 char *name;
 int studentID;
};
int main() {
 MITStudent student1;
MITStudent student2;
}
```

To access fields of instances, use `variable.fieldName`.

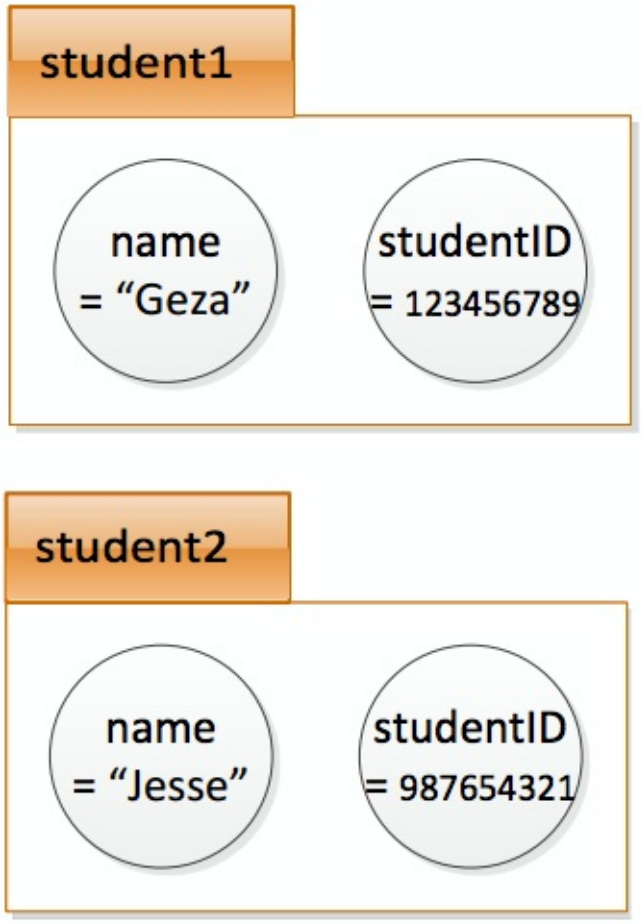
```
#include <iostream>
using namespace std;

class MITStudent {
 public:
  char *name;
  int studentID;
};

int main() {
   MITStudent student1;
   MITStudent student2;
    student1.name = "Geza";
    student1.studentID = 123456789;
    student2.name = "Jesse";
    student2.studentID = 987654321;

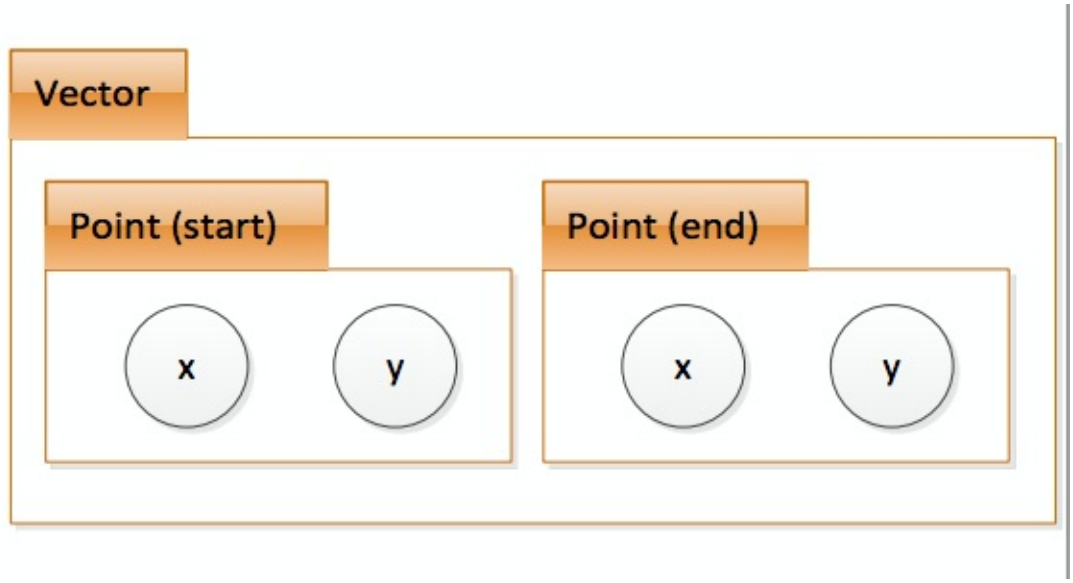
 cout << "student1 name is" << student1.name << endl;
 cout << "student1 id is" << student1.studentID << endl;
 cout << "student2 name is" << student2.name << endl;
 cout << "student2 id is" << student2.studentID << endl;


return 0;
}
```


student1
name
= “Geza”
studentID
= 123456789
student2
name
= “Jesse”
studentID
= 987654321

Classes of classes

As we have explained in the beginning example:a point consists of an x and y coordinate and a vector consists of 2 points: a start and a finish. So the scheme is the following:




```
class Point {
public:
 double x;
 double y;
};
class Vector {
public:
 Point start;
 Point end;
};
```

So fields can be classes. Note the order in which classes are defined. To access the fields we should do the following:

```
class Point {
public:
 double x, y;
};

class Vector {
public:
 Point start, end;
};

int main() {
 Vector vec1;// define vec1 of Vector class
 vec1.start.x = 3.0;
 vec1.start.y = 4.0;
 vec1.end.x = 5.0;
 vec1.end.y = 6.0;
}
```

You can also assign one instance to another, that means copy all fields.

```
vec2.start = vec1.start;
```

Passing classes to functions

Passing by value passes a copy of the class instance to the function; changes aren't preserved.

```
class Point { public: double x, y; };
void offsetPoint(Point p, double x, double y) { // does nothing
 p.x += x;
 p.y += y;
}
int main() {
 Point p;
 p.x = 3.0;
 p.y = 4.0;
 offsetPoint(p, 1.0, 2.0); // does nothing
 cout << "(" << p.x << "," << p.y << ")"; // (3.0,4.0)
}
```

When a class instance is passed by reference, changes are reflected in the original:

```
class Point { public: double x, y; };
void offsetPoint(Point &p, double x, double y) { // works
 p.x += x;
 p.y += y;
}
int main() {
 Point p;
 p.x = 3.0;
 p.y = 4.0;
 offsetPoint(p, 1.0, 2.0); // works
 cout << "(" << p.x << "," << p.y << ")"; // (4.0,6.0)
}
```

Passing by reference

References are perfectly valid types, just like pointers. For instance, just like `int *` is the “pointer to an integer” type, int & is the “reference to an integer” type. References can be passed as arguments to functions, returned from functions, and otherwise manipulated just like any other type.

References are just pointers internally; when you declare a reference variable, a pointer to the value being referenced is created, and it’s just dereferenced each time the reference variable is used.

The syntax for setting a reference variable to become an alias for another variable is just like regular assignment:

```
int &x = y; // x and y are now two names for the same variable
```

Similarly, when we want to pass arguments to a function using references, we just call the function with the arguments as usual, and put the & in the function definiton, where the argument variables are being set to the arguments actually passed:

```
void sq( int &x) { // & is part of the type of x
// - x is an int reference
x *= x;
}
sq(y);
```

Note that on the last line, where we specify what variable x will be a reference to, we just write the name of that variable; we don’t need to take an address with & here. References can also be returned from functions, as in this contrived example:

```
int g; // Global variable
int & getG () { // Return type is int reference
 return g; // As before , the value we ’re making a
 // reference *to* doesn ’t get an & in front of it
 }

 // ... Somewhere in main
int & gRef = getG (); // gRef is now an alias for g
gRef = 7; // Modifies g
```

If you’re writing a class method that needs to return some internal object, it’s often best to return it by reference, since that avoids copying over the entire object. You could also then use your method to do something like:

```
vector <Card > & cardList= deck . getList (); // getList declared to return a reference
 cardList . pop_back ();
```

The second line here modifies the original list in deck, because cardList was declared as a reference and getList returns a reference.

Class Methods

Methods are functions which are part of a class. Functions associated with a class are declared in one of two ways:

```
ReturnType FuncName(params) { code }
function is both declared and defined (code provided)
ReturnType FuncName(params);
```

function is merely declared, we must still define the body of the function separately To call a method we use the `.` form:

```
classinstance.FuncName(args);
```

FuncName is a field just like any other field in the structured variable classinstance.

```
Vector vec;
vec.start.x = 1.2; vec.end.x = 2.0;
vec.start.y = 0.4; vec.end.y = 1.6;
vec.print();\\Method
vec.offset(1.0, 1.5);\\ Arguments can be passed
to methods.
```

Note: Implicitly pass the current instance.

To be able to call this methods the definietion of the Vector class should be:

```
class Vector {
public:
 Point start;
 Point end;
 void offset(double offsetX, double offsetY) {
 start.x += offsetX;
 end.x += offsetX;
 start.y += offsetY;
 end.y += offsetY;
 }

 void print() {
 cout << "(" << start.x << "," << start.y << ") -> (" << end.x <<
"," << end.y << ")" << endl;
 }
};
```

Implementing Methods Separately

Recall that function prototypes allowed us to declare that functions will be implemented later.

```
// vector.h - header file
class Point {
public:
 double x, y;
 void offset(double offsetX, double offsetY);
 void print();
};
class Vector {
public:
 Point start, end;
 void offset(double offsetX, double offsetY);
 void print();
};
```

```
#include "vector.h"
// vector.cpp - method implementation
void Point::offset(double offsetX, double offsetY) {
 x += offsetX; y += offsetY;
}
void Point::print() {
 cout << "(" << x << "," << y << ")";
}
void Vector::offset(double offsetX, double offsetY) {
 start.offset(offsetX, offsetY);
 end.offset(offsetX, offsetY);
}
void Vector::print() {\\:: indicates which class' method is being implemented.
 start.print();
 cout << " -> ";
 end.print();
 cout << endl;
}
```

Constructors

Manually initializing your fields can get tedious.Can we initialize them when we create an instance?

A constructor is a method that is called when an instance is created.

```
class Point {
public:
 double x, y;
 Point() {
 x = 0.0; y = 0.0; cout << "Point instance created" << endl;
 }
};

int main() {
 Point p; // Point instance created
         // p.x is 0.0, p.y is 0.0
}
```

When using constructors you also can use parameters:

```
class Point {
public:
 double x, y;
 Point(double nx, double ny) {
 x = nx; y = ny; cout << "2-parameter constructor" << endl;
 }
};

int main() {
 Point p(2.0, 3.0); // 2-parameter constructor
                    // p.x is 2.0, p.y is 3.0
}
```

You can also use multiple constructors.You can, also, recall that assigning one class instance to another copies all fields (default copy constructor):

```
Point q(1.0, 2.0); // 2-parameter constructor
Point r = q;// r.x is 1.0, r.y is 2.0)
```

Sometimes it is desirable to define your own copy constructor:

```
Point(Point &o) {
 x = o.x; y = o.y; cout << "custom copy constructor" << endl;
 }
```

Exercises: Classes

Exercise 1

Write a class having two private variables and one member function which will return the area of the rectangle.

Solution

Exercise 2

Write a program and input two integers in main and pass them to default constructor of the class. Show the result of the additon of two numbers.

Solution

Exercise 3

Write a c++ class called 'student' with

```
Data members:

name(char type),

marks1,marks2 (integer type)
```

The program asks the user to enter name and marks. Then calc_media() calculates the media note and disp() display name and total media mark on screen in different lines.

Solution

Exercise 4

Perform addition operation on complex data using class and object. The program should ask for real and imaginary part of two complex numbers, and display the real and imaginary parts of their sum.

Solution

Object-Oriented Programming

Along this chapter we'll take a step back and consider the programming philosophy underlying classes, known as object-oriented programming (OOP).

The basic Ideas of OOP

The way you structure a program C++ is:

1. Split it up into a set of tasks and subtasks .
2. Make functions for the tasks .
3. Instruct the computer to perform them in sequence.

With large amounts of data and/or large numbers of tasks, this makes for complex and unmaintainable programs.

To manage this complexity, it’s nicer to package up self-sufficient, modular pieces of code.OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more abstractly and focus on the interactions between them.

There are lots of definitions for OOP, but 3 primary features of it are:

- Encapsulation: grouping related data and functions together as objects and defining an interface to those objects.
- Inheritance: allowing code to be reused between related types .
- Polymorphism: allowing a value to be one of several types, and determining at runtime which functions to call on it based on its type .

Encapsulation

Encapsulation just refers to packaging related stuff together. We've already seen how to package up data and the operations it supports in C++: with classes.

If someone hands us a class, we do not need to know how it actually works to use it; all we need to know about is its public methods/data – its interface.

Interfaces abstract away the details of how all the operations are actually performed, allowing the programmer to focus on how objects will use each other's interfaces – how they interact.

This is why C++ makes you specify public and private access specifiers: by default, it assumes that the things you define in a class are internal details which someone using your code should not have to worry about. The practice of hiding away these details from client code is called "data hiding," or making your class a "black box."

One way to think about what happens in an object-oriented program is that we define what objects exist and what each one knows, and then the objects send messages to each other (by calling each other's methods) to exchange information and tell each other what to do.

Inheritance

Inheritance allows us to define hierarchies of related classes.

Imagine we're writing an inventory program for vehicles, including cars and trucks. We could write one class for representing cars and an unrelated one for representing trucks, but we'd have to duplicate the functionality that all vehicles have in common. Instead, C++ allows us to specify the common code in a Vehicle class, and then specify that the Car and Truck classes share this code.

```
class Vehicle {
protected :
string license ;
int year ;


 public :
 Vehicle ( const string & myLicense , const int myYear )
 : license ( myLicense ), year ( myYear ) {}
  const string getDesc () const
  { return license + " from " + stringify ( year );}
const string & getLicense () const { return license ;}
const int getYear () const { return year ;}
};
```

A few notes on this code, by line:

As we have seen protected is largely equivalent to private, bu tit is used when we want to crean classes based on classes.

Line 8 demonstrates member initializer syntax. When defining a constructor, you sometimes want to initialize certain members, particularly const members, even before the constructor body. You simply put a colon before the function body, followed by a comma-separated list of items of the form dataMember(initialValue).

Line 10 assumes the existence of some function stringify for converting numbers to strings.

Now we want to specify that Car will inherit the Vehicle code, but with some additions. This is accomplished in line 1 below:

```
class Car : public Vehicle { // Makes Car inherit from Vehicle
 string style ;

 public :
 Car( const string & myLicense , const int myYear , const string
& myStyle )
 : Vehicle ( myLicense , myYear ), style ( myStyle ) {}
 const string & getStyle () { return style ;}
 };
```

Now class Car has all the data members and methods of Vehicle, as well as a style data member and a getStyle method. Class Car inherits from class Vehicle. This is equivalent to saying that Car is a derived class, while Vehicle is its base class. You may also hear the terms subclass and superclass instead. Notes on the code:

Note how we use member initializer syntax to call the base-class constructor. We need to have a complete Vehicle object constructed before we construct the components added in the Car. If you do not explicitly call a base-class constructor using this syntax, the default base-class constructor will be called.

There are two ways we could describe some class A as depending on some other class B:

- Every A object has a B object. For instance, every Vehicle has a string object (called license).
- Every instance of A is a B instance. For instance, every Car is a Vehicle, as well.

Inheritance allows us to define “is-a” relationships, but it should not be used to implement “has-a” relationships. It would be a design error to make Vehicle inherit from string because every Vehicle has a license; a Vehicle is not a string. “Has-a” relationships should be implemented by declaring data members, not by inheritance.

We might want to generate the description for Cars in a different way from generic Vehicles. To accomplish this, we can simply redefine the getDesc method in Car, as below. Then, when we call getDesc on a Car object, it will use the redefined function. Redefining in this manner is called overriding the function.

```
class Car : public
string style ;

public :
Car( const string & myLicense , const int myYear , const string
& myStyle )
: Vehicle ( myLicense , myYear ), style ( myStyle ) {}

const string getDesc () // Overriding this member function
{ return stringify ( year ) + ' ' + style + ": " + license
;}
const string & getStyle () { return style ;}
 };
```

Polymorphism

Polymorphism means “many shapes.” It refers to the ability of one object to have many types. If we have a function that expects a Vehicle object, we can safely pass it a Car object, because every Car is also a Vehicle. Likewise for references and pointers: anywhere you can use a Vehicle , *you can use a Car* .

virtual Functions

A virtual function or virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature. This concept is an important part of the polymorphism.

Take the following example:

```
Car c(" VANITY ", 2003) ;
Vehicle * vPtr = &c;
cout << vPtr -> getDesc ();// is the same as cout << *vPtr.getDesc ();
```

(The -> notation on line 3 just dereferences and gets a member. ptr->member is equivalent to (*ptr).member.) In the first line, we define an instance of class Car. Because vPtr is declared as a Vehicle , this will call the Vehicle version of getDesc, even though the object pointed to is actually a Car, as Car inherits from Vehicle.

Note: `*vPtr.getDesc () == vPtr->getDesc ()` In other words the `->` is equivalent to do an indirection(`*`) and add the dot (`.`) to access a class member.

Usually we'd want the program to select the correct function at runtime based on which kind of object is pointed to. We can get this behavior by adding the keyword virtual before the method definition:

```
class Vehicle {
...
virtual const string getDesc () {...}
};
```

With this definition, the code above would correctly select the Car version of getDesc. Selecting the correct function at runtime is called dynamic dispatch. This matches the whole OOP idea – we're sending a message to the object and letting it figure out for itself what actions that message actually means it should take. Because references are implicitly using pointers, the same issues apply to references:

```
Car c(" VANITY ", 2003) ;
Vehicle &v = c;
cout << v. getDesc ();
```

This will only call the Car version of getDesc if getDesc is declared as virtual. Once a method is declared virtual in some class C, it is virtual in every derived class of C, even if not explicitly declared as such. However, it is a good idea to declare it as virtual in the derived classes anyway for clarity.

Arguably, there is no reasonable way to define getDesc for a generic Vehicle – only derived classes really need a definition of it, since there is no such thing as a generic vehicle that isn't also a car, truck, or the like. Still, we do want to require every derived class of Vehicle to have this function.

We can omit the definition of getDesc from Vehicle by making the function pure virtual via the following odd syntax:

```
class Vehicle {
 ...
 virtual const string getDesc () = 0; // Pure virtual
 };
```

The = 0 indicates that no definition will be given. This implies that one can no longer create an instance of Vehicle; one can only create instances of Cars, Trucks, and other derived classes which do implement the getDesc method. Vehicle is then an abstract class – one which defines only an interface, but doesn't actually implement it, and therefore cannot be instantiated.

Multiple Inheritance

C++ allows a class to have multiple based classes.

```
class Car : public Vehicle , public InsuredItem {
 ...
 };
```

This specifies that Car should have all the members of both the Vehicle and the InsuredItem classes.

Multiple inheritance is tricky and potentially dangerous:

- If both Vehicle and InsuredItem define a member x this can cause problems.
- If both Vehicle and InsuredItem inherited from the same base class, you’d end up with two instances of the base class within each Car.

Exercises: OOP

Exercises 1

Write a program that defines a shape class with a constructor that gives value to width and height. The define two sub-classes triangle and rectangle, that calculate the area of the shape `area ()`. In the main, define two variables a triangle and a rectangle and then call the area() function in this two varibles.

Solution

Exercise 2

Write a program with a mother class and an inherited daugther class.Both of them should have a method `void display ()` that prints a message (different for mother and daugther).In the main define a daughter and call the display() method on it.

Solution

Exercise 3

Write a probram with a mother class animal. Inside it define a name and an age variables, and set_value() function.Then create two bases variables Zebra and Dolphin which write a message telling the age, the name and giving some extra information (e.g. place of origin).

Solution

Memory Management

Till now all memory needs were determined before program execution by defining the variables needed. But there may be cases where the memory needs of a program can only be determined during runtime. For example, when the memory needed depends on user input. On these cases, programs need to dynamically allocate memory, for which the C++ language integrates the operators `new` and `delete`.We will discuss it along this chapter.

Scoping and Memory

As we have seen previously, whenever we declare a new variable (int x), memory is allocated. But, when can this memory be freed up (so it can be used to store other variables)?The answer is simply, when the variable goes out of scope.

When a variable goes out of scope, that memory is no longer guaranteed to store the variable's value.

Let's analyze an example:

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

int main() {

 if (true) {
  int x = 5;
  }
 // x now out of scope, memory it used to occupy can be reused
}
```

Now, p will become a dangling pointer (points to memory whose contents are undefined):

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

int main() {

 int *p;
 if (true) {
  int x = 5;
  p = &x;
  }
 cout << *p << endl; // ???
}
```

So, taking this in to account, think about implementing a function which returns a pointer to some memory containing the integer 5.You may think about doing it like this:

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

int* getPtrToFive() {
 int x = 5;
 return &x;
}

int main() {
 int *p = getPtrToFive();
 cout << *p << endl; // ???
}
```

This is an incorrect implementation, why?: – x is declared in the function scope.

– As `getPtrToFive()` returns, x goes out of scope. So a dangling pointer is returned.

The new operator

Dynamic memory is allocated using operator new. new is followed by a data type specifier and, if a sequence of more than one element is required, the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. That is, another way to allocate memory, where the memory will remain allocated until you manually de-allocate it.

```
int *x = new int;
```

If using int x; the allocation occurs on a region of memory called the stack. If using new int; the allocation occurs on a region of memory called the heap. For example:

```
int * foo;
foo = new int [5];
``

In this case, the system dynamically allocates space for five elements of type int and returns a pointer to the first e

New can also be used to allocate a class instance:
```cpp
class Point {
public:
 int x, y;
 Point(int nx, int ny) {
 x = ny; x = ny; cout << "2-arg constructor" << endl;
 }
};

int main() {
 Point *p = new Point;
delete p;
}
```

Destructor is called when the class instance gets de-allocated:

```
class Point {
public:
 int x, y;
Point() {
 cout << "constructor invoked" << endl;
 }
 ~Point() {
 cout << "destructor invoked" << endl;
 }
}
```
```

The delete operator

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory.That is, de-allocates memory that was previously allocated using new.The delete operatot takes a pointer to the memory location.

The syntaxis is the following:

```
delete pointer;
delete[] pointer;
```

The first statement releases the memory of a single element allocated using new, and the second one releases the memory allocated for arrays of elements using new and a size in brackets ([]).

The value passed as argument to delete shall be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

Let's see one example step by step: We are going to implement a function which returns a pointer to some memory containing the integer 5.

- Allocate memory using new to ensure it remains allocated.

  ```
  int *getPtrToFive() {
  int *x = new int;
  *x = 5;
  return x;
  }
  ```

- When done, de-allocate the memory using delete. ```cpp

  # include

  using spacename std;

int *getPtrToFive() { int* x = new int; *x = 5; return x; }

int main() { int *p = getPtrToFive(); cout << p << endl; // 5 delete p; }

```
If you don’t use de-allocate memory using
delete, your application will waste memory.When your program allocates memory but is
unable to de-allocate it, this is a memory leak.

So the final searched result is:
``cpp
#include <iostream>
using namestapce std;

int *getPtrToFive() {
 int *x = new int;
 *x = 5;
 return x;
}

int main() {
 int *p;
 for (int i = 0; i < 3; ++i) {
  p = getPtrToFive();
  cout << *p << endl;
  delete p;
 }
```

```
  return 0;
}
```

Note that to fix the memory leak, de-allocate memory within the loop. Remember to only delete if memory was allocated by new.

Allocating Arrays

When allocating arrays on the stack (using “int arr[SIZE]”), size must be a constant.As commented before, if we use new[] to allocate arrays, they can have variable size.To de-allocate arrays use delete[].

```
int numItems;
 cout << "how many items?";
 cin >> numItems;
 int *arr = new int[numItems];
 delete[] arr;
```

When representing an array, often pass around both the pointer to the first element and the number of elements.Let's see an example:

```
class IntegerArray {
public:
 int *data;
 int size;
};

int main() {
 IntegerArray arr;
 arr.size = 2;
 arr.data = new int[arr.size];//This can be moved to a consructor.
 arr.data[0] = 4; arr.data[1] = 5;
 delete[] a.data;//This can be moved to a destructor.
}
```

Including the constructor/destructor idea:

```
class IntegerArray {
public:
 int *data;
 int size;
 IntegerArray(int size) {
 data = new int[size];
 this->size = size;
 }
 ~IntegerArray () {
 delete[] data;
 }
};

};
int main() {
IntegerArray arr(2);
 arr.data[0] = 4; arr.data[1] = 5;

}
```

Exercises: Memory

Exercise 1

Given the following code add a destructor for Foo class.

```
#include <iostream>

class Foo {
public:
    Foo( void )
    { std::cout << "Foo constructor 1 called" << std::endl; }


int main( )
{
    Foo foo_1;
    return 0;
}
```

Solution

Exercise 2

Write program that ask for the numbers of elements in an array and then fill it in. Use new and delete.

Solution

Exercise 3

Write a program containing a constructor and a destructor for an array.The body shuld be :

```
int main() {
 IntegerArray a(2);//This call the constructor
 a.data[0] = 4; a.data[1] = 2;
 if (true) {
 IntegerArray b = a;
 }
 cout << a.data[0] << endl; // The result is 4
}
```

Note: Pass the data to the data_construtor by reference.

Solution

Advanced topics I

In this chapter we will focus on templates (very useful to parameterize these classes to suit any type of data), overloading operators and other advanced topics when sorking with classes.

Templates

We have seen that functions can take arguments of specific types and have a specific return type. We now consider templates, which allow us to work with generic types. Through templates, rather than repeating function code for each new type we wish to accommodate, we can create functions that are capable of using the same code for different types. For example:

```
int sum(const int x, const int y) {
return x + y;
}
```

For this function to work with doubles, it must be modified to the following:

```
double sum (const double x, const double y) {
return x + y;
}
```

For a simple function such as this, it may be a small matter to just make the change as shown, but if the code were much more complicated, copying the entire function for each new type can quickly become problematic. To overcome this we rewrite sum as a function template. The format for declaring a function template is:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

Both forms are equivalent to one another, regardless of what type identifier ends up being. We can then use identifier to replace all occurrences of the type we wish to generalize. So, we rewrite our sum function:

```
template <typename T>
T sum(const T a, const T b) {
return a + b;
}
```

Now, when sum is called, it is called with a particular type, which will replace all Ts in the code. To invoke a function template, we use:

```
function_name <type> (parameters);
```

Here is an example main function using the above sum function template:

```
int main() {
cout << sum<int>(1, 2) << endl;
cout << sum<float>(1.21, 2.43) << endl;
return 0;
}
```

This program prints out 3 and 3.64 on separate lines. So, as you can see, **templates allow us to operate with generic types**.

The identifier can be used in any way inside the function template, as long as the code makes sense after identifier is replaced with some type. It is also possible to invoke a function template without giving an explicit type, in cases where the generic type identifier is used as the type for a parameter for the function. In the above example, the following would also have been valid:

```
int main() {
 cout << sum(1, 2) << endl;
 cout << sum(1.21, 2.43) << endl;
 return 0;
 }
```

Templates can also specify more than one type parameter. For example:

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

template <typename T, typename U>
U sum(const T a, const U b) {
return a + b;
}

int main() {
cout << sum<int, float>(1, 2.5) << endl;

return 0;
}
```

This program prints out 3.5. In this case we can also call sum by writing sum(1, 2.5).

Class templates are also possible, in much the same way we have written function templates:

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

 template <typename T>
 class Point {
 private:
 T x, y;
 public:
 Point(const T u, const T v) : x(u), y(v) {}
 T getX() { return x; }
 T getY() { return y; }
 };

 int main() {
 Point<float> fpoint(2.5, 3.5);
 cout << fpoint.getX() << ", " << fpoint.getY() << endl;
 return 0;
 }
```

The program prints out 2.5, 3.5.

To declare member functions externally, we use the following syntax:

```
template <typename T>
T classname<T>::function_name()
```

So, for example, getX could have been declared in the following way:

```
template <typename T>
T Point<T>::getX() { return x; }
```

assuming a prototype of T getX(); inside the class definition. We can also define different implementations for a single template by using template specialization. Consider the following example:

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

 template <typename T>
 class Container {
 private:
 T elt;
 public:
 Container(const T arg) : elt(arg) {}
T inc() { return elt+1; }
 };

 template <>
 class Container <char> {
 private:
 char elt;
 public:
 Container(const char arg) : elt(arg) {}
 char uppercase() { return toupper(elt); }
 };

 int main() {

   Container<int> icont(5);
   Container<char> ccont('r');
   cout << icont.inc() << endl;
   cout << ccont.uppercase() << endl;

   return 0;
 }
```

This program prints out 6 and R on separate lines. Here, the class Container is given two implementations: a generic one and one specifically tailored to the char type. Notice the syntax at lines 14 and 15 when declaring a specialization. Finally, it is possible to parametrize templates on regular types:

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

 template <typename T, int N>
 class ArrayContainer {
 private:
 T elts[N];
 public:
 T set(const int i, const T val) { elts[i] = val; }
 T get(const int i) { return elts[i]; }
 };

 int main() {
 ArrayContainer <int, 5> intac;
 ArrayContainer <float, 10> floatac;
 intac.set(2, 3);
 floatac.set(3, 3.5);
 cout << intac.get(2) << endl;
 cout << floatac.get(3) << endl;
 return 0;
 }
```

This program prints out 3 and 3.5 on separate lines. Here, one instance of the ArrayContainer class works on a 5-element array of ints whereas the other instance works on a 10-element array of floats.

Default values can be set for template parameters. For example, the previous template definition could have been:

```
template <typename T=int, int N=5> class ArrayContainer { ... }
```

and we could have created an ArrayContainer using the default parameters by writing:

```
ArrayContainer<> identifier;
```

Standard Template Library

Part of the C++ Standard Library, the Standard Template Library (STL) contains many useful container classes and algorithms. As you might imagine, these various parts of the library are written using templates and so are generic in type. The containers found in the STL are lists, maps, queues, sets, stacks, and vectors. The algorithms include sequence operations, sorts, searches, merges, heap operations, and min/max operations. We will explore how to use some of these through example here:

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


int main() {
   set<int> iset;
   iset.insert(5);
   iset.insert(9);
   iset.insert(1);
   iset.insert(8);
   iset.insert(3);

  cout << "iset contains:";
  set<int>::iterator it;
  for(it=iset.begin(); it != iset.end(); it++)
   cout << " " << *it;
   cout << endl;

  int searchFor;
  cin >> searchFor;
  if(binary_search(iset.begin(), iset.end(), searchFor))
    cout << "Found " << searchFor << endl;
  else

   cout << "Did not find " << searchFor << endl;
return 0;
}
```

In this example, we create an integer set and insert several integers into it. We then create an iterator corresponding to the set at lines 14 and 15. An iterator is basically a pointer that provides a view of the set. (Most of the other containers also provide iterators.) By using this iterator, we display all the elements in the set and print out iset contains: 1 3 5 8 9. Note that the set automatically sorts its own items. Finally, we ask the user for an integer, search for that integer in the set, and display the result.

Here is another example:

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

 void printArray(const int arr[], const int len) {
 for(int i=0; i < len; i++)
 cout << " " << arr[i];
 cout << endl;
 }

int main() {
 int a[] = {5, 7, 2, 1, 4, 3, 6};

 sort(a, a+7);
 printArray(a, 7);
 rotate(a,a+3,a+7);
 printArray(a, 7);
 reverse(a, a+7);
 printArray(a, 7);

 return 0;
  }
```

This program prints out:

```
1 2 3 4 5 6 7
4 5 6 7 1 2 3
3 2 1 7 6 5 4
```

Static and const members

Static members

A class can contain static members, either data or functions.

A static data member of a class is also known as a "class variable", because there is only one common variable for all the objects of that same class, sharing the same value: i.e., its value is not different from one object of this class to another.

For example, it may be used for a variable within a class that can contain a counter with the number of objects of that class that are currently allocated, as in the following example:

```
// static members in classes
#include <iostream>
using namespace std;

class Dummy {
  public:
    static int n;
    Dummy () { n++; };
    ~Dummy () { n--; };
};

int Dummy::n=0;

int main () {
  Dummy a;
  Dummy b[5];
  Dummy * c = new Dummy;
  cout << a.n << '\n';
  delete c;
  cout << Dummy::n << '\n';
  return 0;
  }
```

Const members

When an object of a class is qualified as a const object:

```
const MyClass myobject;
```

The access to its data members from outside the class is restricted to read-only, as if all its data members were const for those accessing them from outside the class. Note though, that the constructor is still called and is allowed to initialize and modify these data members:

```
// constructor on const object
#include <iostream>
using namespace std;

class MyClass {
  public:
    int x;
    MyClass(int val) : x(val) {}
    int get() {return x;}
};

int main() {
  const MyClass foo(10);
// foo.x = 20;            // not valid: x cannot be modified
  cout << foo.x << '\n';  // ok: data member x can be read
  return 0;
}
```

Note: If a variable is passed as a const to a function it won't modify it.(See exercise 2)

Converting between const and non-const

You can always provide a non-const value where a const one was expected. For instance, you can pass non-const variables to a function that takes a const argument. The const-ness of the argument just means the function promises not to change it, whether or not you require that promise. The other direction can be a problem: you cannot provide a const reference or pointer where a non-const one was expected. Setting a non-const pointer/reference to a const one would violate the latter's requirement that it not be changeable. The following, for instance, does not work:

```
int g; // Global variable
 const int & getG () { return g; }

 // ... Somewhere in main
 int & gRef = getG ();
```

This fails because gRef is a non-const reference, yet we are trying to set it to a const reference (the reference returned by getG). In short, the compiler will not let you convert a const value into a non-const value unless you're just making a copy (which leaves the original const value safe).

For simple values like ints, the concept of const variables is simple: a const int can't be modified. It gets a little more complicated when we start talking about const objects. Clearly, no fields on a const object should be modifiable, but what methods should be available? It turns out that the compiler can't always tell for itself which methods are safe to call on const objects, so it assumes by default that none are. To signal that a method is safe to call on a const object, you must put the const keyword at the end of its signature, e.g. int getX() const;. const methods that return pointers/references to internal class data should always return const pointers/references.

Overloading operators

Classes, essentially, define new types to be used in C++ code. And types in C++ not only interact with code by means of constructions and assignments. They also interact by means of operators. For now, we have been using operators on primitives, but sometimes it makes sense to use them on user-defined datatypes. For example, take the following operation on fundamental types:

```
int a, b, c;
a = b + c;
```

Here, different variables of a fundamental type (int) are applied the addition operator, and then the assignment operator. For a fundamental arithmetic type, the meaning of such operations is generally obvious and unambiguous, but it may not be so for certain class types. For example:

```
struct myclass {
  string product;
  float price;
} a, b, c;
a = b + c;
```

Here, it is not obvious what the result of the addition operation on b and c does. In fact, this code alone would cause a compilation error, since the type myclass has no defined behavior for additions. However, C++ allows most operators to be overloaded so that their behavior can be defined for just about any type, including classes.Operators are overloaded by means of operator functions, which are regular functions with special names: their name begins by the operator keyword followed by the operator sign that is overloaded. The syntax is:

```
type operator sign (parameters) { /*... body ...*/ }
```

Let's analyze a pair of example. First, the addition operation of two cartesian vectors is defined as the addition both x coordinates together, and both y coordinates together. For example, adding the cartesian vectors (3,1) and (1,2) together would result in (3+1,1+2) = (4,3). This could be implemented in C++ with the following code:

```
// overloading operators example
#include <iostream>
using namespace std;

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int a,int b) : x(a), y(b) {}
    CVector operator + (const CVector&);
};

CVector CVector::operator+ (const CVector& param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return temp;
}

int main () {
  CVector foo (3,1);
  CVector bar (1,2);
  CVector result;
  result = foo + bar;
  cout << result.x << ',' << result.y << '\n';
  return 0;
}
```

Secondly, consider the following example, starting with this struct:

```
struct USCurrency {
int dollars;
int cents;
};
```

Perhaps we would like to add two USCurrency objects together and get a new one as a result, just like in normal addition:

```
USCurrency a = {2, 50};
USCurrency b = {1, 75};
USCurrency c = a + b;
```

This of course gives a compiler error, but we can define behavior that our datatype should have when used with the addition operator by overloading the addition operator. This can be done either inside the class as part of its definition (the addition from the point of view of the object on the left side of the +):

```
 USCurrency operator+(const USCurrency o) {
 USCurrency tmp = {0, 0};
 tmp.cents = cents + o.cents;
 tmp.dollars = dollars + o.dollars;
 if(tmp.cents >= 100) {
 tmp.dollars += 1;
 tmp.cents -= 100;
 }
 return tmp;
}
```

or outside the class as a function independent of the class (the addition from the point of view of the +):

```
USCurrency operator+(const USCurrency m, const USCurrency o) {
USCurrency tmp = {0, 0};
tmp.cents = m.cents + o.cents;
tmp.dollars = m.dollars + o.dollars;

if(tmp.cents >= 100) {
tmp.dollars += 1;
tmp.cents -= 100;
}
 return tmp;
}
```

Similarly, we can overload the << operator to display the result:

```
 ostream& operator<<(ostream &output, const USCurrency
 output << "$" << o.dollars << "." << o.cents;
 return output;
 }
```

Assuming the above definitions, we can run the following program:

```
int main() {
USCurrency a = {2, 50};
USCurrency b = {1, 75};
USCurrency c = a + b;
cout << c << endl;
return 0;
}
```

and get the printout $4.25. The list of overloadable operators:

```
+ - * / += -= *= /= % %= ++ --
= == < > <= >= ! != && ||
<< >> <<= >>= & ^ | &= ^= |= ~
[] () , ->* -> new new[] delete delete[]
```

Exercises: Advaced I

Exercise 1

You have done this exercise before, but now you are ask to do this exercise usinh overloaded operators. Perform addition operation on complex data using class and object. The program should ask for real and imaginary part of two complex numbers, and display the real and imaginary parts of their sum.

Solution

Exercise 2

Define a const array and pass it to a `reader ()` function and a modifier () function which cahnger every character for an 'O'.Analyze what happends.

Solution

Exercise 3

Define a table that can be filled with both: type integer or float. Use templates.

Solution

- You can find here more exercise about templates: http://deekle.net/Jones/FallAndWinter2005-6/COP/14.pdf

Advanced topics II

This chapter will introduce you some possibilities when using C ++ language, in order to make it easier what your programs do.For example: files handling, some comands...

File handling

File handling in C++ works almost identically to terminal input/output. To use files, you write `#include <fstream>` at the top of your source file. Then you can access two classes from the std namespace:

- `ifstream` : allows reading input from files
- `ofstream` : allows outputting to files

Each open file is represented by a separate ifstream or an ofstream object. You can use ifstream objects in excatly the same way as cin and ofstream objects in the same way as cout, except that you need to declare new objects and specify what files to open. For example:

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

int main () {

 ifstream source (" source - file . txt ");
 ofstream destination ("dest - file . txt ");
 int x;

 source >> x; // Reads one int from source - file . txt
 source . close (); // close file as soon as we 're done using it
 destination << x; // Writes x to dest - file . txt

 return 0;
 } // close () called on destination by its destructor
```

As an alternative to passing the filename to the constructor, you can use an existing ifstream or ofstream object to open a file by calling the open method on it: `source.open("other-file.txt");`. Remember to close your files using the `close()` method when you're done using them.

You can specify a second argument to the constructor or the open method to specify what "mode" you want to access the file in – read-only, overwrite, write by appending, etc. Check documentation online for details.

Reading strings

You'll likely find that you want to read some text input from the user. We've so far seen only how to do ints, chars, etc. It's usually easiest to manage text using the C++ string class. You can read in a string from cin like other variables:

```
string mobileCarrier ;
cin >> mobileCarrier ;
```

However, this method only reads up to the first whitespace; it stops at any tab, space, newline, etc. If you want to read multiple words, you can use the getline function, which reads everything up until the user presses enter:

```
string sentence ;
getline (cin , sentence );
```

enum

In many cases you'll find that you'll want to have a variable that represents one of a discrete set of values. For instance, you might be writing a card game and want to store the suit of a card, which can only be one of clubs, diamonds, hearts, and spades. One way you might do this is declaring a bunch of const ints, each of which is an ID for a particular suit. If you wanted to print the suit name for a particular ID, you might write this:

```
const int CLUBS = 0, DIAMONDS = 1, HEARTS = 2, SPADES = 3;
void print_suit ( const int suit ) {
const char * names [] = {" Clubs ", " Diamonds ",
" Hearts ", " Spades "};
return names [ suit ];
}
```

The problem with this is that suit could be integer, not just one of the set of values we know it should be restricted to. We'd have to check in our function whether suit is too big. Also, there's no indication in the code that these const ints are related. Instead, C++ allows us to use enums. An enum just provides a set of named integer values which are the only legal values for some new type. For instance:

```
enum suit_t {CLUBS , DIAMONDS , HEARTS , SPADES };
 void print_suit ( const suit_t suit ) {
 const char * names [] = {" Clubs ", " Diamonds ",
 " Hearts ", " Spades "};
 return names [ suit ];
 }
```

Now, it is illegal to pass anything but CLUBS, DIAMODNS, HEARTS, or SPADES into print suit. However, internally the suit t values are still just integers, and we can use them as such (as in line 5). You can specify which integers you want them to be:

```
enum suit_t { CLUBS =18 , DIAMONDS =91 , HEARTS =241 , SPADES =13};
```

The result is something like this:

```
#include <iostream>

enum suit_t { CLUBS =18 , DIAMONDS =91 , HEARTS =241 , SPADES =13};

void print_suit ( const suit_t suit ) {
 const char * names [] = {" Clubs ", " Diamonds ",
 " Hearts ", " Spades "};
 std::cout << names[suit];
 }

int main(){
     suit_t card;
     card=DIAMONDS;
 print_suit(card);

 return 0;
 }
```

The following rules are used by default to determine the values of the enum constants:

- The first item defaults to 0.
- Every other item defaults to the previous item plus 1.For instance, by the rule "next enumerator value is previous + 1", the value of "Lexus" enumerator is 46:

```
enum e_acomany {
    Audi=4,
    BMW=5,
    Cadillac=11,
    Ford=44,
    Jaguar=45,
    Lexus,
    Maybach=55,
    RollsRoyce=65,
    Saab=111
};
```

Just like any other type, an enum type such as suit_t can be used for any arguments, variables, return types, etc.

Exceptions

Sometimes functions encounter errors that make it impossible to continue normally. For instance, a getFirst function that was called on an empty Array object would have no reasonable course of action, since there is no first element to return.

A functions can signal such an error to its caller by throwing an exception. This causes the function to exit immediately with no return value. The calling function has the opportunity to catch the exception – to specify how it should be handled. If it does not do so, it exits immediately as well, the exception passes up to the next function, and so on up the call stack (the chain of function calls that got us to the exception).

An example:

```
const int DIV_BY_0 = 0;
int divide ( const int x, const int y) {
if(y == 0)
throw DIV_BY_0 ;
return x / y;
}

void f( int x, int ** arrPtr ) {
try {
* arrPtr = new int [ divide (5, x)];
} catch ( int error ) {
// cerr is like cout but for error messages
cerr << " Caught error : " << error ;
}
// ... Some other code ...
}
```

The code in main is executing a function (divide) that might throw an exception. In anticipation, the potentially problematic code is wrapped in a try block. If an exception is thrown from divide, divide immediately exits, passing control back to main. Next, the exception's type is checked against the type specified in the catch block (line 11). If it matches (as it does in this case), the code in the catch block is executed; otherwise, f will exit as well as though it had thrown the exception. The exception will then be passed up to f's caller to see if it has a catch block defined for the exception's type.

You can have an arbitrary number of catch blocks after a try block:

```
int divide ( const int x, const int y) {
if(y == 0)
throw std :: runtime_exception (" Divide
return x / y;
by 0!");
}

void f( int x, int ** arrPtr ) {
try {
 * arrPtr = new int [ divide (5, x)];
 }
 catch ( bad_alloc & error ) {// new throws exceptions  this type.
 cerr << " new failed to allocate memory ";
 }
 catch ( runtime_exception & error ) {
 // cerr is like cout but for error messages
 cerr << " Caught error : " << error . what ();
 }
 // ...
}
```

In such a case, the exception's type is checked against each of the catch blocks' argument types in the order specified. If line 2 causes an exception, the program will first check whether the exception is a bad alloc object. Failing that, it checks whether it was a runtime exception object. If the exception is neither, the function exits and the exception continues propagating up the call stack. The destructors of all local variables in a function are called before the function exits due to an exception.

Exception usage notes:

- Though C++ allows us to throw values of any type, typically we throw exception objects. Most exception classes inherit from class std::exception in header file `<stdexcept>`.
- The standard exception classes all have a constructor taking a string that describes the problem. That description can be accessed by calling the what method on an exception object.
- You should always use references when specifying the type a catch block should match (as in lines 11 and 14). This prevents excessive copying and allows virtual functions to be executed properly on the exception object.

friend Functions/Classes

Occasionally you'll want to allow a function that is not a member of a given class to access the private fields/methods of that class. (This is particularly common in operator overloading.)

We can specify that a given external function gets full access rights by placing the signature of the function inside the class, preceded by the word friend.So, a non-member function can access the private and protected members of a class if it is declared a friend of that class. That is done by including a declaration of this external function within the class, and preceding it with the keyword friend.The same happend with classes, a friend class is a class whose members have access to the private or protected members of another class:

```
// friend class
#include <iostream>
using namespace std;

class Square;

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

class Square {
  friend class Rectangle;
  private:
    int side;
  public:
    Square (int a) : side(a) {}
};

void Rectangle::convert (Square a) {
  width = a.side;
  height = a.side;
}

int main () {
  Rectangle rect;
  Square sqr (4);
  rect.convert(sqr);
  cout << rect.area();


  return 0;
}
```

In this example, class Rectangle is a friend of class Square allowing Rectangle's member functions to access private and protected members of Square. More concretely, Rectangle accesses the member variable Square::side, which describes the side of the square.

Preprocessor Macros

We've seen how to define constants using the preprocessor command #define. We can also define macros, small snippets of code that depend on arguments. For instance, we can write:

```
# define sum (x, y) (x + y)
```

Now, every time sum(a, b) appears in the code, for any arguments a and b, it will be replaced with (a + b). Macros are like small functions that are not type-checked; they are implemented by simple textual substitution. Because they are not type-checked, they are considered less robust than functions.

Casting

Casting is the process of converting a value between types. All C++-style casts are of the form cast `type<type>(value)`, where type is the type you're casting to. The possible cast types to replace cast type with are:

- `static cast` : This is by far the most commonly used cast. It creates a simple copy of the value of the specified type. Example:

  ```
  static cast<float>(x)
  ```

  where x is an int, gives a float copy of x.

- `dynamic_cas` t can only be used with pointers and references to classes (or with void*). Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type. `dynamic cast` can also be used with references: if v is a Vehicle & variable, dynamic cast `<Car &>(v)` will return a valid reference if v is actually a reference to a Car, and will throw a bad cast exception otherwise.

- `reinterpret cast` :Does no conversion; just treats the memory containing value as though it were of type type.

- `const cast` : Used for changing const modifiers of a value. You can use this to tell the compiler that you really do know what you're doing and should be allowed to modify a const variable. You could also use it to add a const modifier to an object so you can force use of the const version of a member function.

Exercises: Advanced II

Exercise 1

Write a program that enumerate some auto's brands. Then create a variable called `my_car_brand` and make it equal to one of them. Print a message saying hello to the user.

Solution

Exercise 2

Write a simple program:

- read() is a function that ask for a name and returns a string containing the name.
- in the main() you should ask for the name and write a hello message.

Solution

Exercise 3

Create a clas called Rectangle.It should contain a constructor for width and height, a function that returns the area and a friend fucntion that duplicate the rectangle's dimensions. In the main, creaate a rectangle instance, duplicate it and print out the area.

Solution

Exercise 4

Create a class called Box with a variable: width of type double. Inside the class define a constructor and a friend that prints the width value(printWidth). In the `main()` define a Box instance, set values and call printWidth.

Solution

Exercises: miscellaneous

Exercise 1

Write a program that asks for txo 3x3 Matrix.Then the porgram sums the matrix and stores the result in the first one(A=A+B).Then prints on the screen the resulting matrix.

Solution

Exercise 2

Write a program that takes input array daily temperatures, as floats. and findout the maximum and minimum values.

Solution

Exercise 3

Write a program which calculates the sum of all odd numbers upto a particular limit. The limit will be an input to the program.

Solution

Exercise 4

Write a program which takes input a string and a character then remove this character in this string.

Solution

Exercise 5

Write a program that clasifies angle: acute, straight and obtuse.

Solution

Exercise 6

Write a program that input a number and calculates its factorial(use a function called `factorial`).

Solution

GDB

In this chapter we will discuss what is GDB and what are the possibilities GDB debugger offers.

We will introduce basic concepts so, for more information you can take a look at:

- Simple session with GDB
- GDB docs

What is GDB?

GDB is a debugger.

A debugger is a program that is used to run other programs. In a debugger, a program may be executed, debugged, or a combination of both. Further, debuggers provide commands in order that data may be examined. There are many operations available in most debuggers.

The purpose of a debugger such as GDB is to allow you to see what is going on “inside” another program while it executes —or what another program was doing at the moment it crashed.

GDB can do four main kinds of things (plus other things in support of these) to help you catch bugs in the act:

- *Start your program* specifying anything that might affect its behavior.
- *Make your program stop* on specified conditions.
- *Examine* what has happened, when your program has stopped.
- *Change things* in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

First steps

In this section you can find the first steps to follow when using GDB.

Erle-Brain and GDB

By default you should have GDB installed in your Erle-Brain. You can start a gdb debugging session by typing `gdb` and close it using the command `quit` as shown below.You can always ask GDB itself for information on its commands, using the command `help`.

```
root@erlerobot:~/GDB# gdb
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
(gdb)
(gdb)
(gdb)
(gdb)
(gdb) quit
```

If you don't have it installed you can dowload it from the source here. (Dowload directly in erleboard if you have internet-connection through a wireless network card- or download it to your PC and follow the suitable steps here.)

Then from the command line do the following.Decompress the file and configure it:

```
root@erlerobot:~# tar -xzvf gdb-7.7.tar.gz
root@erlerobot:~# cd gdb-7.7
root@erlerobot:~# ./configure --disable-intl
...
```

Compile the source:

```
root@erlerobot:~# make
...
```

And install it:

```
root@erlerobot:~# sudo make install
...
```

Enabling and Starting GDB

To enable gdb to be able to display the statements of your program, you must use the –g flag during compilation and linking of all files comprising your program. The –g flag causes the compiler and linker to maintain variable names and untranslated C++ statements, in order that variables can be examined and C++ statements can be displayed and followed.

To start a stand-alone gdb session, simply type `gdb <executable>` at the system prompt. For example, to debug `a.out`, issue the command

```
root@erlerobot:~#  >gdb a.out
```

gdb will start and load `a.out`.

Alternatively, gdb can be started by simply issuing the command gdb at the Unix prompt. To then load an executable, issue a file command `(gdb)file a.out`. For example:

```
root@erlerobot:~/GDB# ls
hello.cpp
root@erlerobot:~/GDB# g++ hello.cpp
root@erlerobot:~/GDB# ls
a.out  hello.cpp
root@erlerobot:~/GDB# ./a.out
 Hello , world !
root@erlerobot:~/GDB#
root@erlerobot:~/GDB#
root@erlerobot:~/GDB# gdb
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>.
(gdb)
(gdb) file a.out
Reading symbols from /root/GDB/a.out...(no debugging symbols found)...done.
(gdb)
```

Here you can find more information about files options: https://sourceware.org/gdb/current/onlinedocs/gdb/File-Options.html#File-Options

Another important task is to have the correct **working directory**.You can change it by using `cd`.

```
(gdb) file Hello
Hello: No such file or directory.
(gdb)
Hello: No such file or directory.
(gdb) cd /root/GDB
Working directory /root/GDB.
(gdb)
(gdb) file a.out
Reading symbols from /root/GDB/a.out...(no debugging symbols found)...done.
(gdb)
```

GDB commands

When GDB starts, the loaded executable is not running; there are many commands that can be issued to prepare debugging. A GDB command is a single line of input. There is no limit on how long it can be. It starts with a command name, which is followed by arguments whose meaning depends on the command name.

A glossary of commands can be displayed by issuing the command.

```
(gdb)help
```

There will be a list of command areas, each of which has its own help list. For example,

```
(gdb)help running
```

will list commands available for running a program. If you issue a help command followed by and actual command, the help information for that command will be displayed.

Commands can often be issued without typing the entire command. Here are some commonly used commands; many of them can be invoked using only the first letter:

```
 (gdb) quit – exit the debugger
 (gdb) file – load an executable file
 (gdb) break line-number/function name -- Set a break-point on a line/at start of function
 (gdb) run <args> -- start running the program; if there are command-line arguments, put them after the run invocation
 (gdb) cont -- continue running, after a break
 (gdb) next -- Next program line (step over function calls)
 (gdb) step -- Step into function calls.
 (gdb) finish - Step out of the present function
 (gdb) print expression -- Show value of a variable or expression
 (gdb) list – List 10 lines of the program being debugged. The sixth line is the preset statement. Subsequent, consecut
 (gdb) where – obtain a backtrace showing all function calls before the current statement
 (gdb) up – Move to the function that called the present function. Useful if your program crashes in a library functior
 (gdb) down – Reverses the action of up
 (gdb) delete – Removes breakpoint by number (see example following). If no number, all deleted.
 (gdb) kill – Terminates the program.
 (gdb) info- This command (abbreviated i) is for describing the state of your program. For example, you can show the ar
(gdb) set-You can assign the result of an expression to an environment variable with set. For example, you can set the
(gdb)show-In contrast to info, show is for describing the state of GDB itself. You can change most of the things you ca
```

Other interesting commands

`Break` : To set a breakpoint on a line, issue the break command with a line number. For example, to set a breakpoint on line 43, issue b 43 . To set a breakpoint at the start of a function, issue the break command with the function name. For example, to suspend execution and enter debugging mode at the start of function foo, issue b foo . To debug from the start of the program, issue b main In projects with multiple files, a breakpoint can be set in an associated file using `<file name>:line number` . A breakpoint can be set at the beginning of an object's function using `<class>::<function>` .

Note: In IDEs a breakpoint is set explicitly within the program code in a window.

`Print` : The print command has great functionality. You can enter just about any valid expression, and it will be evaluated an printed. You can print the values of pointers, dereference a pointer, and print entire arrays and objects. See the emacs example on the next page for sample uses of the print command.

######Logging output

You may want to save the output of GDB commands to a file. There are several commands to control GDB's logging.

`set logging on` : Enable logging.

`set logging off` Disable logging.

`set logging file file` Change the name of the current logfile. The default logfile is gdb.txt.

`set logging overwrite [on|off]` By default, GDB will append to the logfile. Set overwrite if you want set logging on to overwrite the logfile instead.

`set logging redirect [on|off]` By default, GDB output will go to both the terminal and the logfile. Set redirect if you want output to go only to the log file.

`show logging` Show the current values of the logging settings.

######Comands to specify files

`file filename` Use filename as the program to be debugged. It is read for its symbols and for the contents of pure memory. It is also the program executed when you use the run command. If you do not specify a directory and the file is not found in the GDB working directory, GDB uses the environment variable PATH as a list of directories to search, just as the shell does when looking for a program to run. You can change the value of this variable, for both GDB and your program, using the path command.

You can load unlinked object .o files into GDB using the file command. You will not be able to “run” an object file, but you can disassemble functions and inspect variables. Also, if the underlying BFD functionality supports it, you could use gdb -write to patch object files using this technique. Note that GDB can neither interpret nor modify relocations in this case, so branches and some initialized variables will appear to go to the wrong place. But this feature is still handy from time to time.

`file` file with no argument makes GDB discard any information it has on both executable file and the symbol table.

`exec-file [ filename ]` Specify that the program to be run (but not the symbol table) is found in filename. GDB searches the environment variable PATH if necessary to locate your program. Omitting filename means to discard information on the executable file.

Running programs

Use the run command to start your program under GDB. You must first specify the program name with an argument to GDB, for instance by using the `file` or `exec-file` command.

Take into account that when compiling the program in g++ you should add the appropiate options, for later debugging.For example `-g` let you produce debugging information - Debugging options.

If you are running your program in an execution environment that supports processes, run creates an inferior process and makes that process run your program. In some environments without processes, run jumps to the start of your program.

The execution of a program is affected by certain information it receives from its superior. GDB provides ways to specify this information, which you must do **before starting** your program. (You can change it after starting your program, but such changes only affect your program the next time you start it.) This information may be divided into four categories:

- *The arguments.*

Specify the arguments to give your program as the arguments of the run command. If a shell is available on your target, the shell is used to pass the arguments, so that you may use normal conventions (such as wildcard expansion or variable substitution) in describing the arguments. In Unix systems, you can control which shell is used with the SHELL environment variable. If you do not define SHELL, GDB uses the default shell (/bin/sh). You can disable use of any shell with the set startup-with-shell command (see below for details).

- *The environment.*

Your program normally inherits its environment from GDB, but you can use the GDB commands set environment and unset environment to change parts of the environment that affect your program. See Your Program's Environment.

- *The working directory.*

Your program inherits its working directory from GDB. As we have coment previously, you can set the GDB working directory with the `cd` command in GDB.

- *The standard input and output.*

Your program normally uses the same device for standard input and standard output as GDB is using. You can redirect input and output in the run command line, or you can use the tty command to set a different device for your program. See Your Program's Input and Output.

Warning: While input and output redirection work, you cannot use pipes to pass the output of the program you are debugging to another program; if you attempt this, GDB is likely to wind up debugging the wrong program.

For more information visit: https://sourceware.org/gdb/current/onlinedocs/gdb/Starting.html#Starting

GDB Examples

EXAMPLE 1

First create a .cpp file with the content of the sample file `crash.cpp` (You can copy it to another .cpp file or downoad it). You can find the file stored here.

The `crash.cpp` program which generate a core dump. Frist , to enable debugging, the program must be compiled with the `-g` option(`-g` option is used to request debugging information).

```
root@erlerobot:~/GDB# g++ -g crash.cpp -o crash
root@erlerobot:~/GDB# ls
a.out  crash  crash.cpp  hello.cpp
root@erlerobot:~/GDB#
```

Now, when run this program on your linux machine, it produces the result:

```
root@erlerobot:~/GDB# ./crash
Floating point exception
```

Now to debug the problem, start gdb debugger at command prompt:

```
root@erlerobot:~/GDB# gdb crash
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /root/GDB/crash...done.
(gdb)
(gdb)
```

The following steps are:

```
    (gdb) r
    Program received signal SIGFPE, Arithmetic exception.
    0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
    21        return a / b;
    # 'r' runs the program inside the debugger
    # In this case the program crashed and gdb prints out some
    # relevant information.  In particular, it crashed trying
    # to execute line 21 of crash.cc.  The function parameters
    # 'a' and 'b' had values 3 and 0 respectively.

    (gdb) l
    # l is short for 'list'.  Useful for seeing the context of
    # the crash, lists code lines near around 21 of crash.cc

    (gdb) where
    #0  0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
    #1  0x08048654 in main () at crash.cc:13
    # Equivalent to 'bt' or backtrace.  Produces what is known
    # as a 'stack trace'.  Read this as follows:  The crash occurred
    # in the function divint at line 21 of crash.cc.  This, in turn,
    # was called from the function main at line 13 of crash.cc

    (gdb) up
    # Move from the default level '0' of the stack trace up one level
    # to level 1.
```

```
    (gdb) list
    # list now lists the code lines near line 13 of crash.cc

    (gdb) p x
    # print the value of the local (to main) variable x
```

In this example, it is fairly obvious that the crash occurs because of the attempt to divide an integer by 0.

That's one of he reasonns why gdb is so useful, we are able to identify errors using GDB.

EXAMPLE 2

Let us write another program which will cause a core dump due to non initialized memory: `crash1.cpp` .Find the complete file here

When compiling and running it, the following error is produced:

```
root@erlerobot:~/GDB# g++ -g crash1.cpp -o crash1
root@erlerobot:~/GDB# ls
a.out  crash  crash.cpp  crash1  crash1.cpp  hello.cpp
root@erlerobot:~/GDB# ./crash1
10
Segmentation fault
root@erlerobot:~/GDB#
```

Now we want to identify the error:

```
root@erlerobot:~/GDB# gdb crash1
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "arm-linux-gnueabihf".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /root/GDB/crash1...done.
(gdb) r
Starting program: /root/GDB/crash1
10

Program received signal SIGSEGV, Segmentation fault.
0x0000867a in setint (ip=0x0, i=10) at crash1.cpp:16
16          *ip = i;
(gdb) where
#0  0x0000867a in setint (ip=0x0, i=10) at crash1.cpp:16
#1  0x0000863c in main () at crash1.cpp:11
```

Unfortunately, the program did not crash in either of the user-defined functions main or setint, so there is no useful trace or local variable information. In this case, it may be more useful to single-step through the program.

```
(gdb) b main
Breakpoint 1 at 0x8606: file crash1.cpp, line 8.
# Set a breakpoint at the beginning of the function main.
(gdb)
(gdb) r
# Run the program, but break immediately due to the breakpoint.
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/GDB/crash1

Breakpoint 1, main () at crash1.cpp:8
8           setint(&a, 10);
(gdb) n
# n = next, runs one line of the program.
9           cout << a << endl;
```

```
(gdb) n
10
11          setint(b, 10);
(gdb) s
# s = step, is like next, but it will step into functions.
# In this case the function stepped into is setint.
setint (ip=0x0, i=10) at crash1.cpp:16
16          *ip = i;
(gdb) p ip
# p print the value on the screen.
$1 = (int *) 0x0
(gdb) p *ip
Cannot access memory at address 0x0
(gdb)
```

The value of `*ip` is the value of the integer pointed to by `ip`. In this case it is an unusual value and is strong evidence that there is a problem. The problem in this case is that the pointer was never properly initialized, so it is pointing to some random area in memory. By pure luck, the process of assiging a value to `*ip` does not crash the program, but it creates some problem that crashes the program when it finishes.