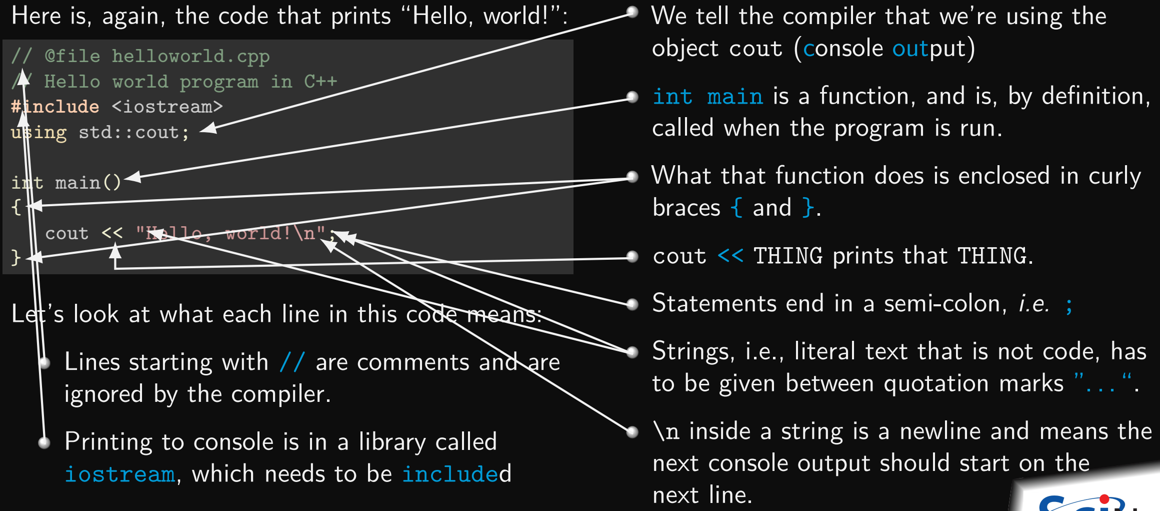
# C++ by Example

# Back to the C++ example



Here is, again, the code that prints “Hello, world!”:

```cpp
// @file helloworld.cpp
// Hello world program in C++
#include <iostream>
using std::cout;

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

Let’s look at what each line in this code means:

- Lines starting with // are comments and are ignored by the compiler.
- Printing to console is in a library called iostream, which needs to be included

- We tell the compiler that we’re using the object cout (console output)
- int main is a function, and is, by definition, called when the program is run.
- What that function does is enclosed in curly braces { and }.
- cout << THING prints that THING.
- Statements end in a semi-colon, *i.e.* ;
- Strings, i.e., literal text that is not code, has to be given between quotation marks "...“.
- \n inside a string is a newline and means the next console output should start on the next line.

# Another C++ Example: Input and variables

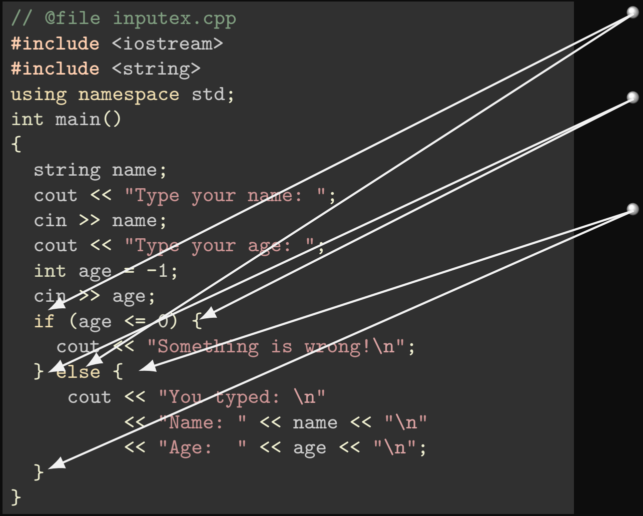
```cpp
// @file inputex.cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string name;
  cout << "Type your name: ";
  cin >> name;
  cout << "Type your age: ";
  int age;
  cin >> age;
  cout << "You typed: \n"
       << "Name: " << name << "\n"
       << "Age:  " << age << "\n";
}
```

- This program uses many `std::` objects, so we import all of that namespace.

  (not generally a good idea)
- `int main` starts by defining a variable named **name** of **type** `string`.
  All variables have a **type** in C++
- It reads from `cin` (console in, *i.e.*, keyboard) into the existing variable `name`
- It also defines and reads an `age` variable, which is of type `int`.
- And it reports what was typed by the user.

Note that variables and their types must be defined before they can be used!

# Let's add a conditional statement



```cpp
// @file inputex.cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string name;
  cout << "Type your name: ";
  cin >> name;
  cout << "Type your age: ";
  int age = -1;
  cin >> age;
  if (age <= 0) {
    cout << "Something is wrong!\n";
  } else {
     cout << "You typed: \n"
          << "Name: " << name << "\n"
          << "Age:  " << age << "\n";
  }
}
```

- Depending on the age variable, the program prints one thing or another, using `if/else`.
- Note that the code for the "one thing" has to be in a code block, delineated by curly braces, *i.e.* {...} .
- Similarly, the else code block is delineated by braces.

# Let's add a return value

```cpp
// @file inputex.cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string name;
  cout << "Type your name: ";
  cin >> name;
  cout << "Type your age: ";
  int age = -1;
  cin >> age;
  if (age <= 0) {
    cout << "Something is wrong!\n";
    return 1;
  } else {
    cout << "You typed: \n"
         << "Name: " << name << "\n"
         << "Age:  " << age << "\n";
    return 0;
  }
}
```

- In addition to errors writing to console, we return an exit code to the shell indicating success (0) or failure (non-zero).
- The value returned by main must be an `int`.

```
$ g++ -std=c++17 -o inputex inputex.cpp
$ echo Alex -1 | ./inputex
Something is wrong
$ echo $?
1
$ echo Alex 48 | ./inputex
You typed:
Name: Alex
Age: 48
$ echo $?
0
```

- In *bash*, the exit code of the last executed command is stored in the variable $?.
- Here, *bash* types input with "echo" and "pipes" that into "inputex".

# How to ask again: Repetition

```cpp
#include <iostream>
#include <string>
using namespace std;
int main()
{
  string name;
  cout << "Type your name: ";
  cin >> name;
  cout << "Type your age: ";
  int age = -1;
  cin >> age;
  while (age <= 0) {
    cout << "Something is wrong!\n";
    cout << "Type your age again: ";
    cin >> age;
  }
  cout << "You typed: \n";
  cout << "Name: " << name << "\n";
  cout << "Age:  " << age << "\n";
}
```

- The idea here is to keep asking numbers for the age variable until a positive one is given.
- The `while` construct is good for this.
- But this can fail if we do not give an integer. (will fix later)

# Arrays

```
#include <iostream>
#include <string>

using namespace std;
int main() {
  string name;
  cout << "Type your name: ";
  cin >> name;
  int nmax = 10;
  int numbers[nmax] = {0,0,0,0,0,0,0,0,0,0};
  int n;
  for (n = 0; n < nmax; n++) {
    string word;
    cout << "Type a number (-1 to stop): ";
    cin >> word;
    numbers[n] = stoi(word);
    if (numbers[n] == -1)
      break;
  }
  cout << "You typed: \n";
  cout << "Name: " << name << "\n";
```

```
  cout << "Numbers:";
  for (int i = 0; i < n; i++) {
    cout << " " << numbers[i];
  }
  cout << "\n";
}
```

- Purpose of this code is get several numbers and store them.
- C++ supports "C-style automatic arrays". `numbers` is defined as an array by putting the number of elements in square brackets.
- Also use square brackets for element access.
- The first element is element `[0]`
- The `for` loop is suitable for iterating over such an array.

# Vectors

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;
int main() {
  string name;
  cout << "Type your name: ";
  cin >> name;
  int nmax = 10;
  vector<int> numbers;
  int n;
  for (n = 0; n < nmax; n++) {
    string word;
    cout << "Type a number (-1 to stop): ";
    cin >> word;
    numbers.push_back(stoi(word));
    if (numbers[n] == -1)
      break;
  }
  cout << "You typed: \n";
  cout << "Name: " << name << "\n";
  cout << "Numbers:";
  for (int number : numbers) {
    cout << " " << number;
  }
  cout << "\n";
}
```

- Here again we want to get several numbers and store them.
- But we're using the C++ standard `vector`.
- These have variable sizes.
- Can use square brackets are used for indexing, with the first element begin `[0]`.
- But they also support range-based for loop.

# Functions

The code is starting to look a bit messy; we can make it clearer with some functions.

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;
string getword(const string& prompt) {
  string result;
  cout << prompt;
  cin >> result;
  return result;
}
int getint(const string& prompt) {
  while (true) {
    string word = getword(prompt);
    try {
      return stoi(word);
    } catch (invalid_argument& e) {
      cout << "Error: invalid input\n";
      if (cin.eof()) return -1;
    }
  }
}
```

```cpp
int main() {
  string name = getword("Type your name: ");
  int nmax = 10;
  vector<int> numbers;
  while (true) {
    int x = getint("Type a number (-1 to stop): ");
    if (x != -1)
      numbers.push_back(x);
    if (numbers.size() == nmax or x == -1)
      break;
  }
  cout << "You typed: \n";
  cout << "Name: " << name << "\n";
  cout << "Numbers:";
  for (int number : numbers) {
    cout << " " << number;
  }
  cout << "\n";
}
```

# Dealing with input errors

You may have noticed thet the `getint` function does something interesting to catch errors.

We could just have

```
int getint(const string& prompt) {
  string word = getword(prompt);
  return stoi(word);
}
```

but this would crash when the `word` does not contain an integer.

This code can handle that:

```
int getint(const string& prompt) {
  while (true) {
    string word = getword(prompt);
    try {
      return stoi(word);
    } catch (invalid_argument& e) {
      cout << "Error: invalid input\n";
      if (cin.eof()) return -1;
    }
  }
}
```

## Catching errors using exceptions

- Exceptions can be used to catch unexpected events, like entering a non-number for age.
- This goes via the `try`/`catch` construct.
- If `stoi` encounters an error, an exception is “thrown”.
- The exception is caught by the catch clause (in fact of a specific type).

# C++ Details

# C++ Details: Variable definition

```
type name [=value];
```

Here, type may be a:

- floating point type:

  `float, double, long double, std::complex<float>, ...`

- integer type:

  `[unsigned] short, int, long, long long`

- character or string of characters:

  `char, char*, std::string`

- boolean i.e., truth value: `bool`

- array, pointer, class, structure, . . .

Examples:

```
int a;
int b;
a = 4;
b = a + 2;
```

```
float f = 4.0f;
double d = 4.0;
d += f;
```

```
char* str = "Hello There!";
```

```
bool itis2018 = false;
```

**Non-initialized variables are not 0, but have random values!**

### const

The type can be proceeded by `const` to make it immutable.

# C++ Details: Functions

Function = a piece of code that can be reused.

A function has:

1. a name
2. a set of arguments of specific type
3. and returns a value of some specfic type

These three properties are called the function's signature.

- To write a piece of code that uses (”calls”) the functions, we only need to know its signature or interface;

  To make the signature known, one has to place a function declaration before the piece of code that is to use the function.

- The actual code (function definition) can be in a different file or in a library.

# C++ function example

```cpp
// funcexample.cpp

// external function declarations:
#include <iostream>
#include <cmath>

// function declaration:
double geometric_mean(double a, double b);

// main function to call when program starts:
int main() {
    double x = 16.3;
    double y = 102.4;
    std::cout << geometric_mean(x,y) << "\n";
}

// function definition:
double geometric_mean(double a, double b) {
    return sqrt(a*b);
}
```

```
$ ssh USERNAME@teach.scinet.utoronto.ca

$ module load gcc

$ g++ -std=c++17 -o funcexample funcexample.cpp

$ ./funcexample
40.8549

$
```

# C++ Details: Functions

- Function declaration (prototype/signature/interface)

```
returntype name(argument-spec);
```

  `argument-spec` = comma separated list of variable definitions

- Function definition (code/implementation)

```
returntype name(argument-spec) {
    statements
    return expression-of-type-returntype ;
}
```

  Functions which do not return anything have to be declared with a `returntype` of `void`.
  Functions which have a non-void return type must have a `return` statement (except main).
  The function definition can double as the declaration if it preceeds all uses of it in the same source file.

- Function call

```
var = name(argument-list);
f(name(argument-list));
name(argument-list);
```

  `argument-list` = comma separated list of values

# C++ Details: Scope

*Variables do not live forever, they have well-defined scopes in which they exist. These are the rules:*

If you define a variable inside a code block, it exists only until the code hits the closing curly brace (}) that correspond to the opening curly brace ({) that started the block. This is its local scope.

The variable will only be known in that code block and its subblocks.

If you call a function from a code block, variables from that block will not be known in the body of the function.

It is possible to define variables outside of any code block; these are global variables. **Avoid those.**

When a variable goes out of scope, the memory associated with it is returned to the system, except for memory that was dynamically allocated.

# C++ Details: Arguments by value or by reference

### Passing function arguments by value

```cpp
// passval.cpp
#include <iostream>

void inc(int i) {
    i = i + 1;
}

int main() {
    int j = 10;
    inc(j);
    std::cout << j << "\n";
}
```

```
$ g++ -std=c++17 -o passval passval.cpp
$ ./passval
10
$
```

- j is set to 10.
- j is passed to inc,
- where it is copied into a variable called i.
- i is increased by one,
- but the original j is not changed.

# C++ Details: Arguments by value or by reference

## Passing function arguments by reference

```
// passref.cpp
#include <iostream>

void inc(int &i) {
    i = i + 1;
}

int main() {
    int j = 10;
    inc(j);
    std::cout << j << "\n";
}
```

```
$ g++ -std=c++17 -o passref passref.cpp
$ ./passref
11
$
```

- j is set to 10.
- j is passed to inc,
- where it referred to as i (but it's still j).
- i is increased by one,
- because i is just an alias for j, j reflects this change.

# C++ Details: Operators

### Arithmetic

a+b Add a and b

a-b Subtract a and b

a*b Multiply a and b

a/b Divide a and b

a%b Remainder of a over b

### Assignment

a=b Assign a expression b to the variable b

a+=b Add b to a (result stored in a)

a-=b Substract b from a (result stored in a)

a*=b Multiply a with b (result stored in a)

a/=b Divide a by b (result stored in a)

a++ Increase value of a by one

### Logic

a==b a equals b

a!=b a does not equal b

!a a is not true (also: not a)

a&&b both a and b are true (also: a and b)

a||b either a or b is true (also: a or b)

### Logic/Numeric

a<b is a less than b

a>b is a greater than b

a<=b is a less then or equal to b

a>=b is a greater than or equal to b